cse141: Introduction to Computer Architecture

Steven Swanson
Hung-Wei Tseng

Today's Agenda

- What is architecture?
- Why is it important?
- At the highest level, where is architecture today? Where is it going?
- What's in this class?

What is architecture?

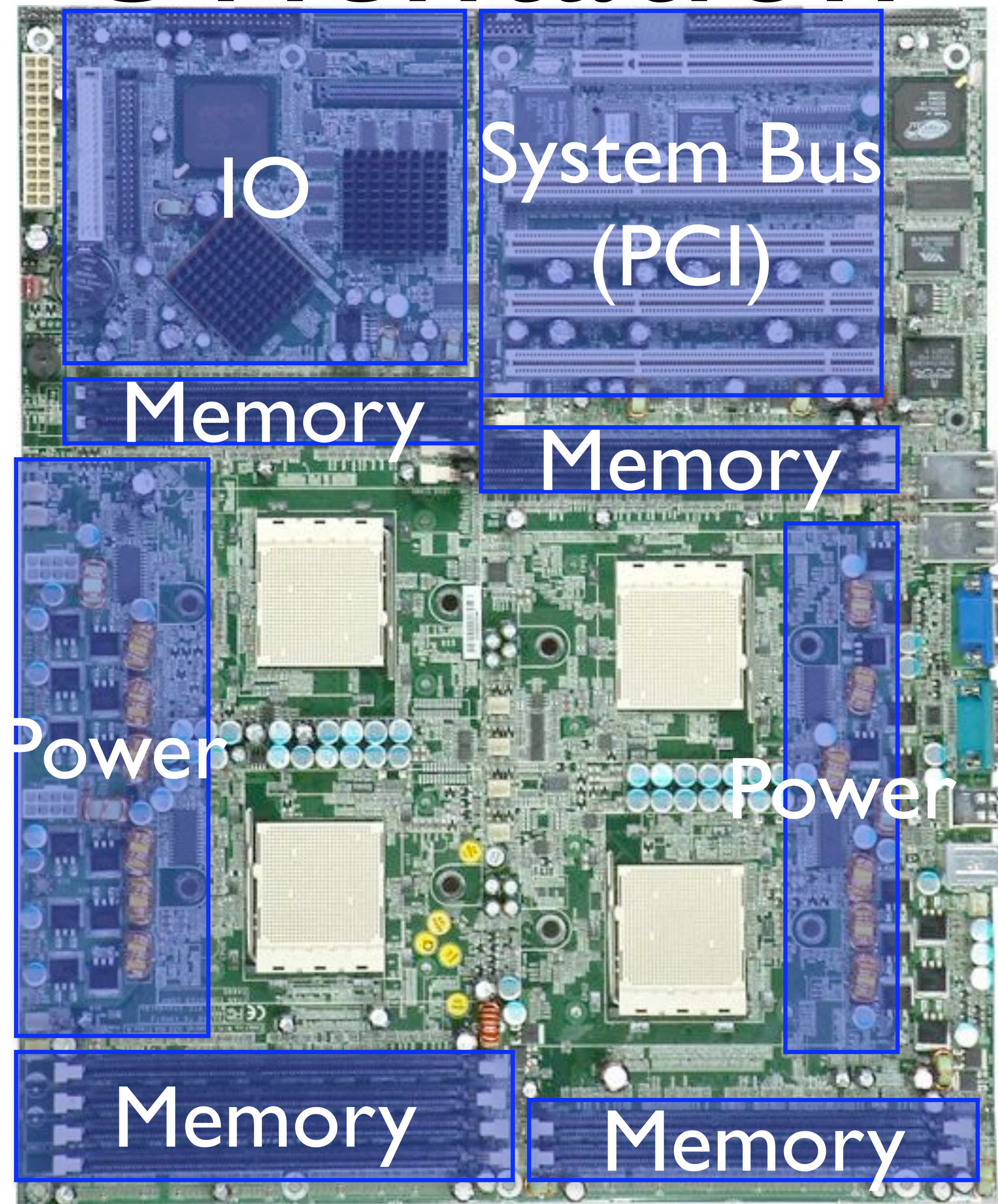
- How do you build a machine that computes?
  - Quickly, safely, cheaply, efficiently, in technology X, for application Y, etc.

Civilization advances by extending
the number of important
operations which we can perform
without thinking about them.
-- *Alfred North Whitehead*

Orientation

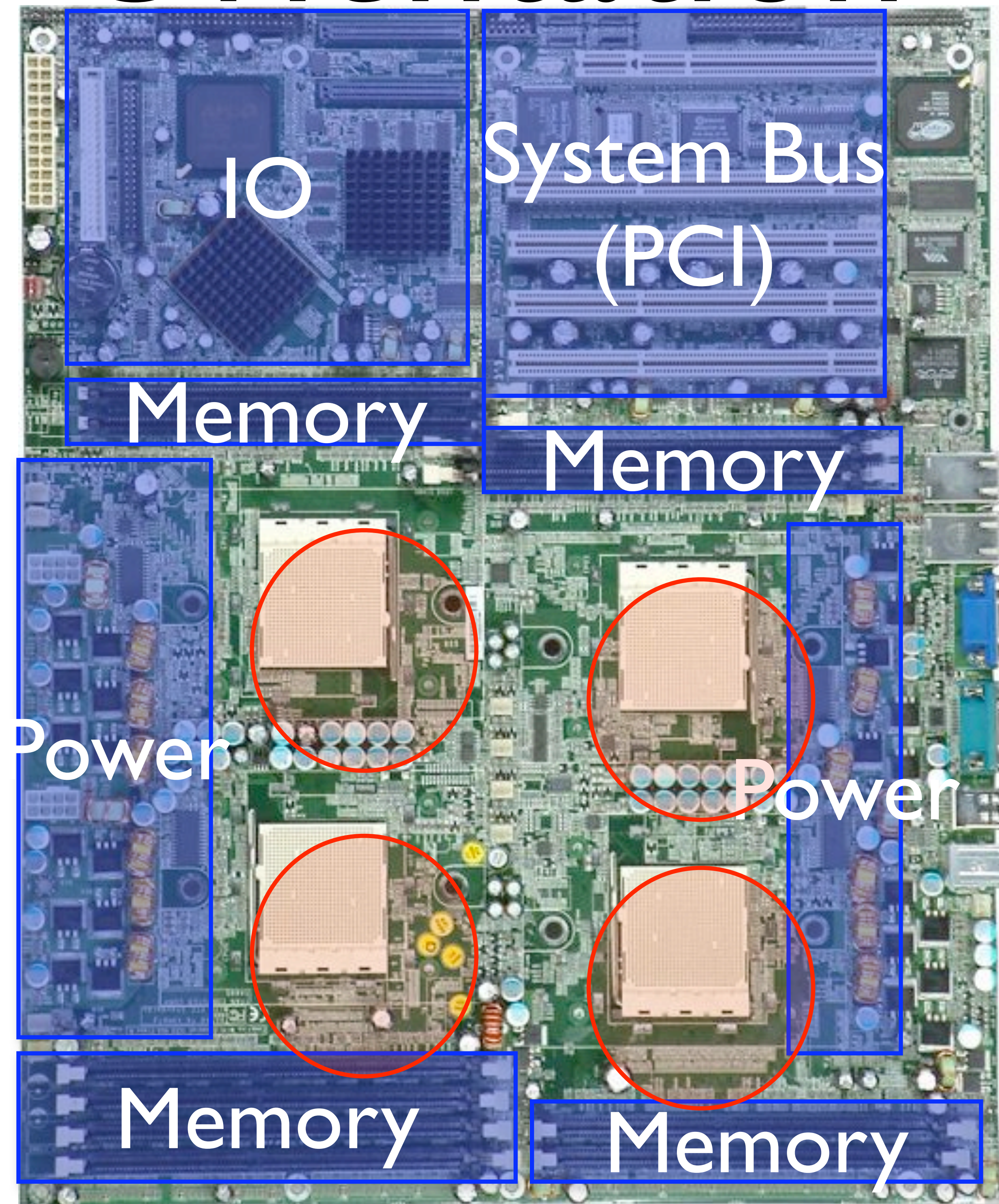
Orientation

Orientation

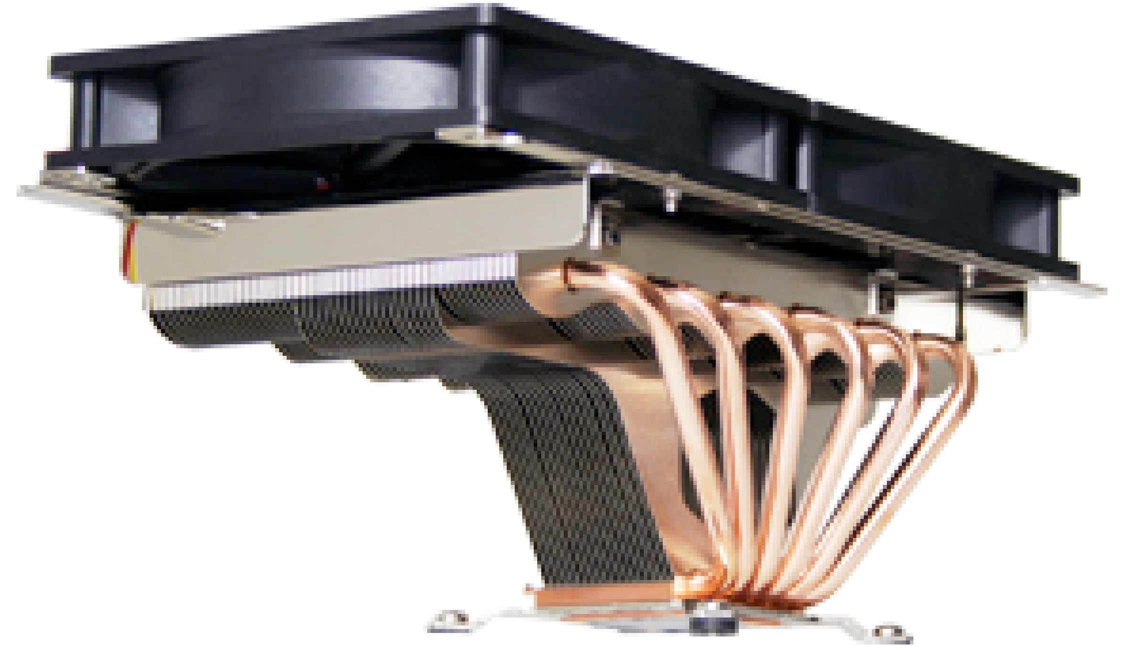


Architecture begins about here.

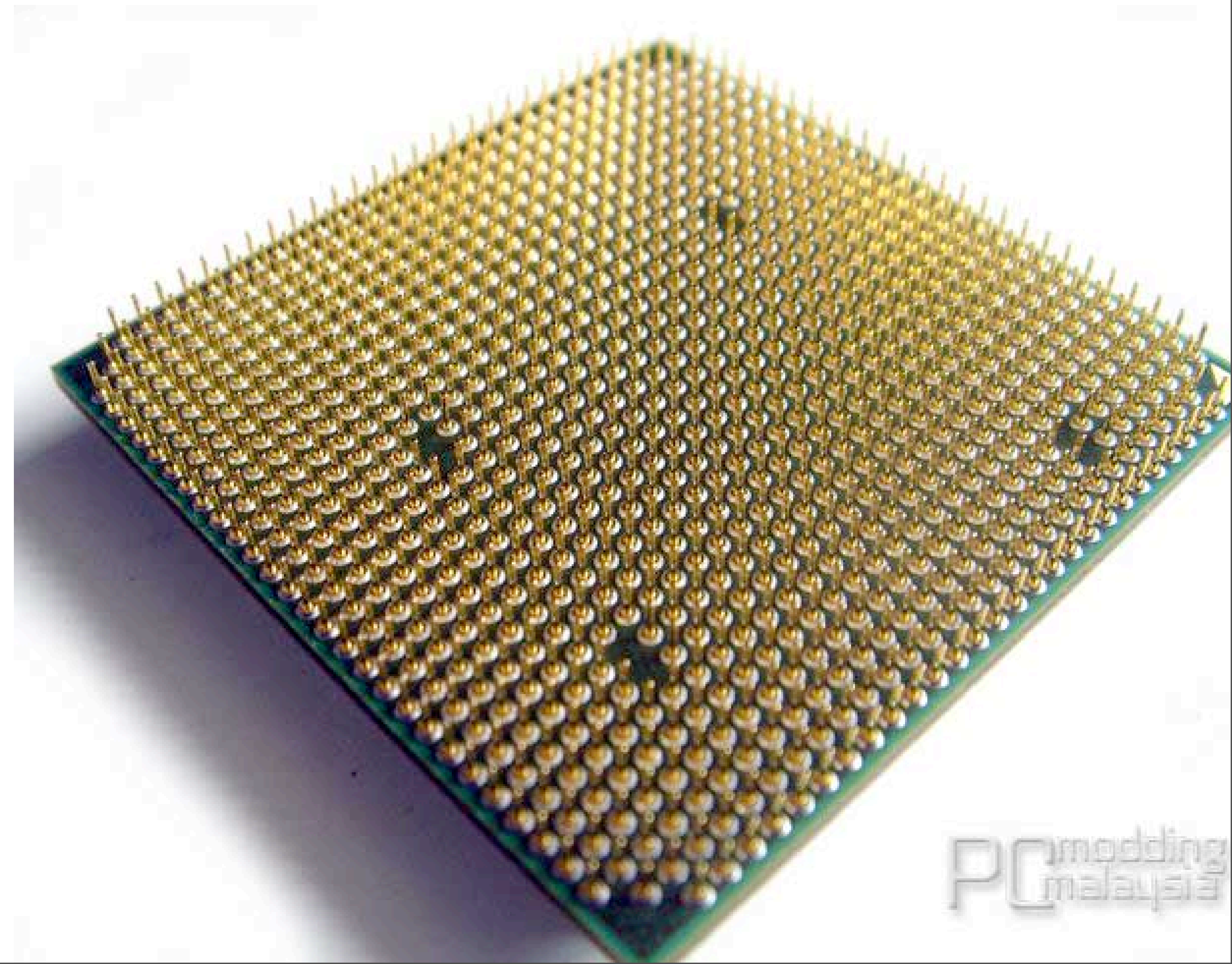
Orientation

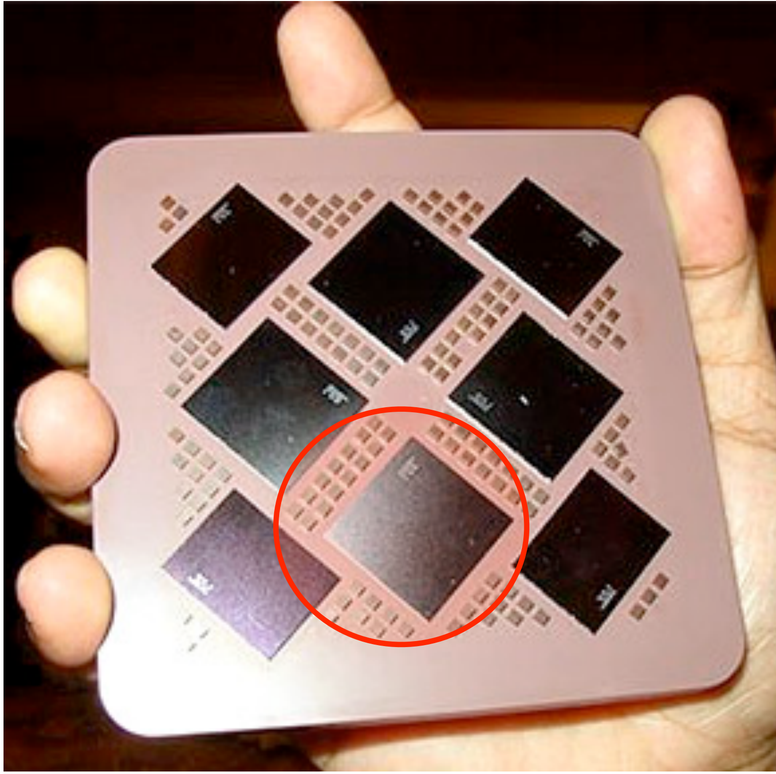


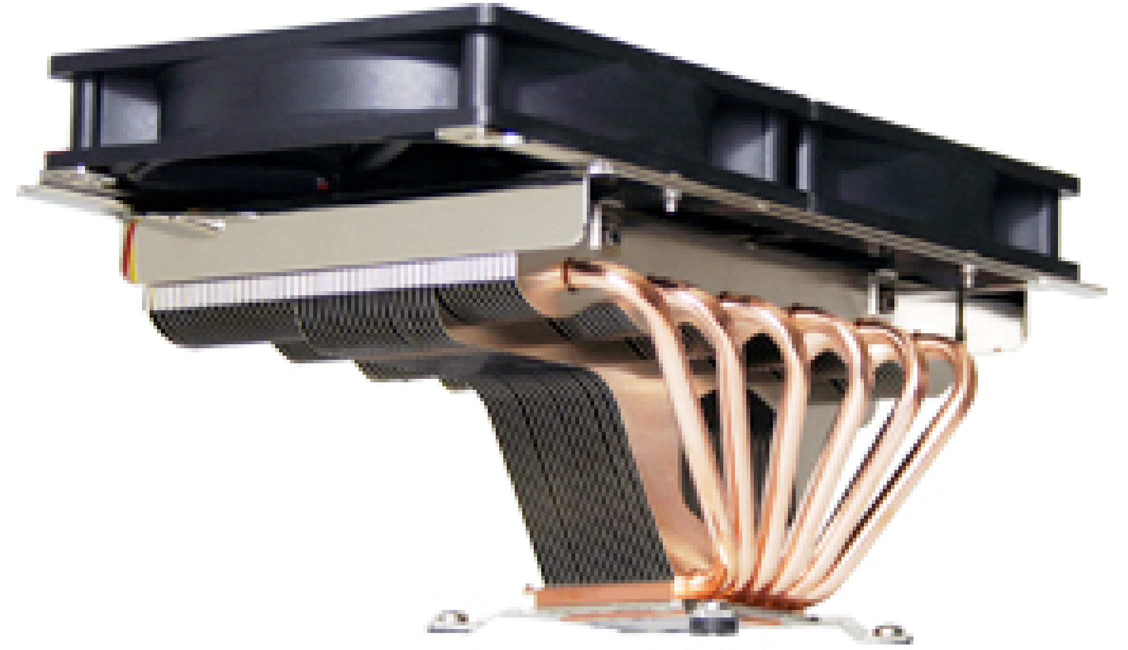
Architecture begins about here.

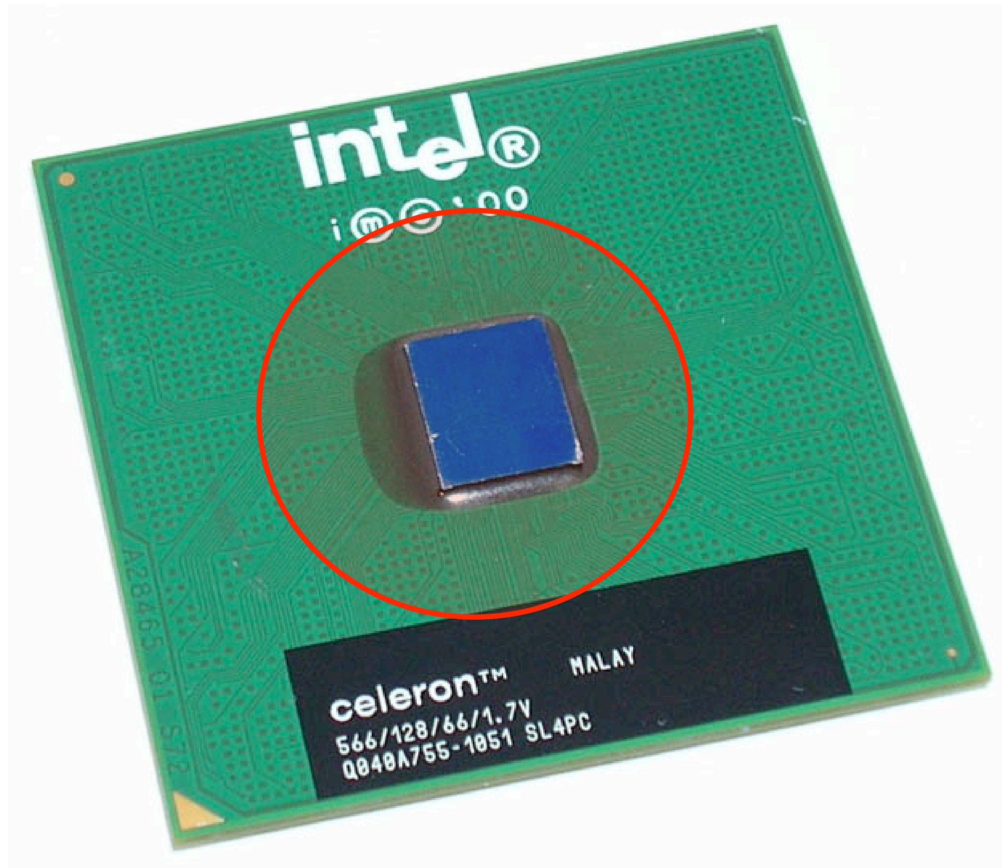


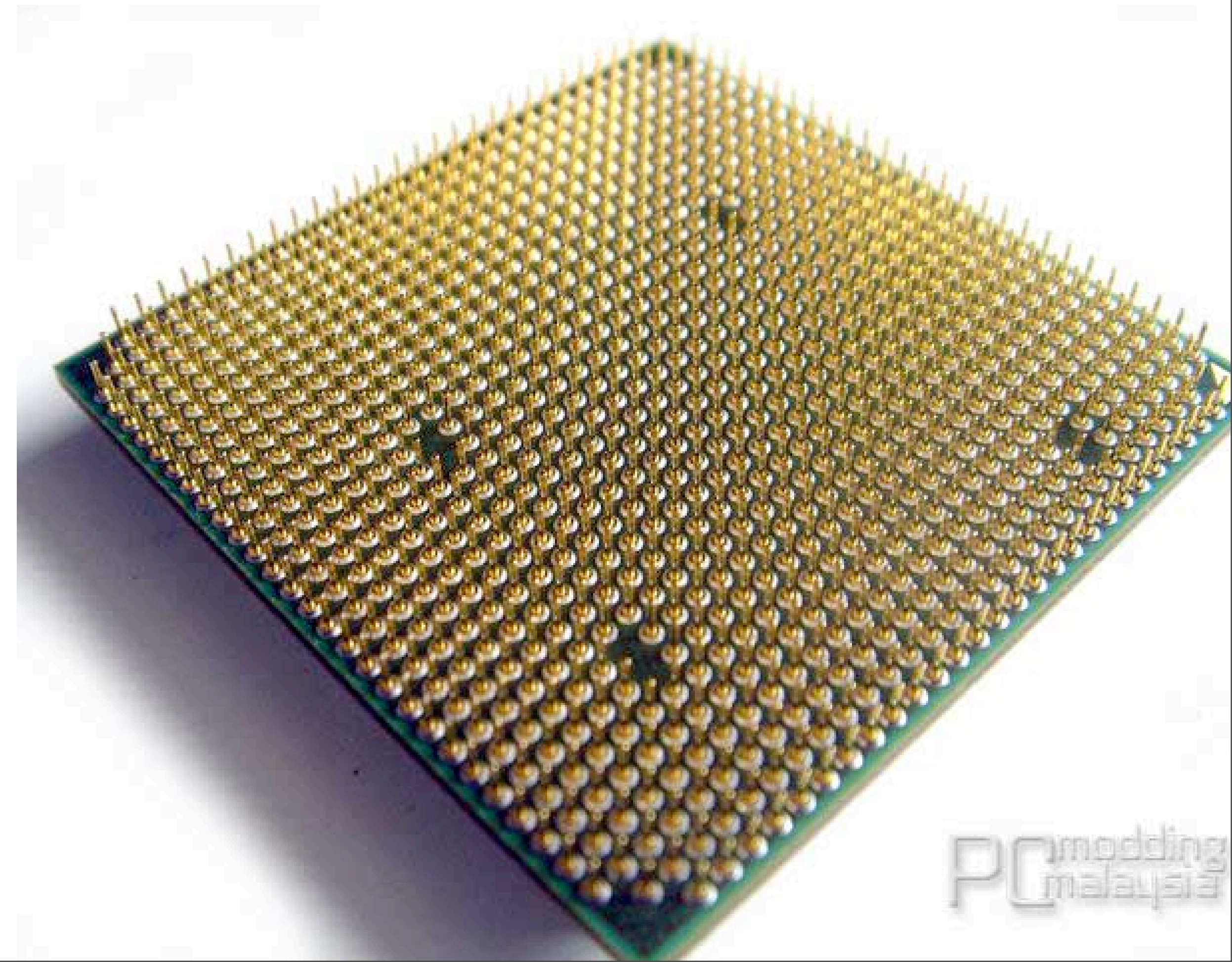
intel®
celeron™
MALAY
566/128/66/1.7V
Q040A755-1051 SL4PC


PC modding malaysia

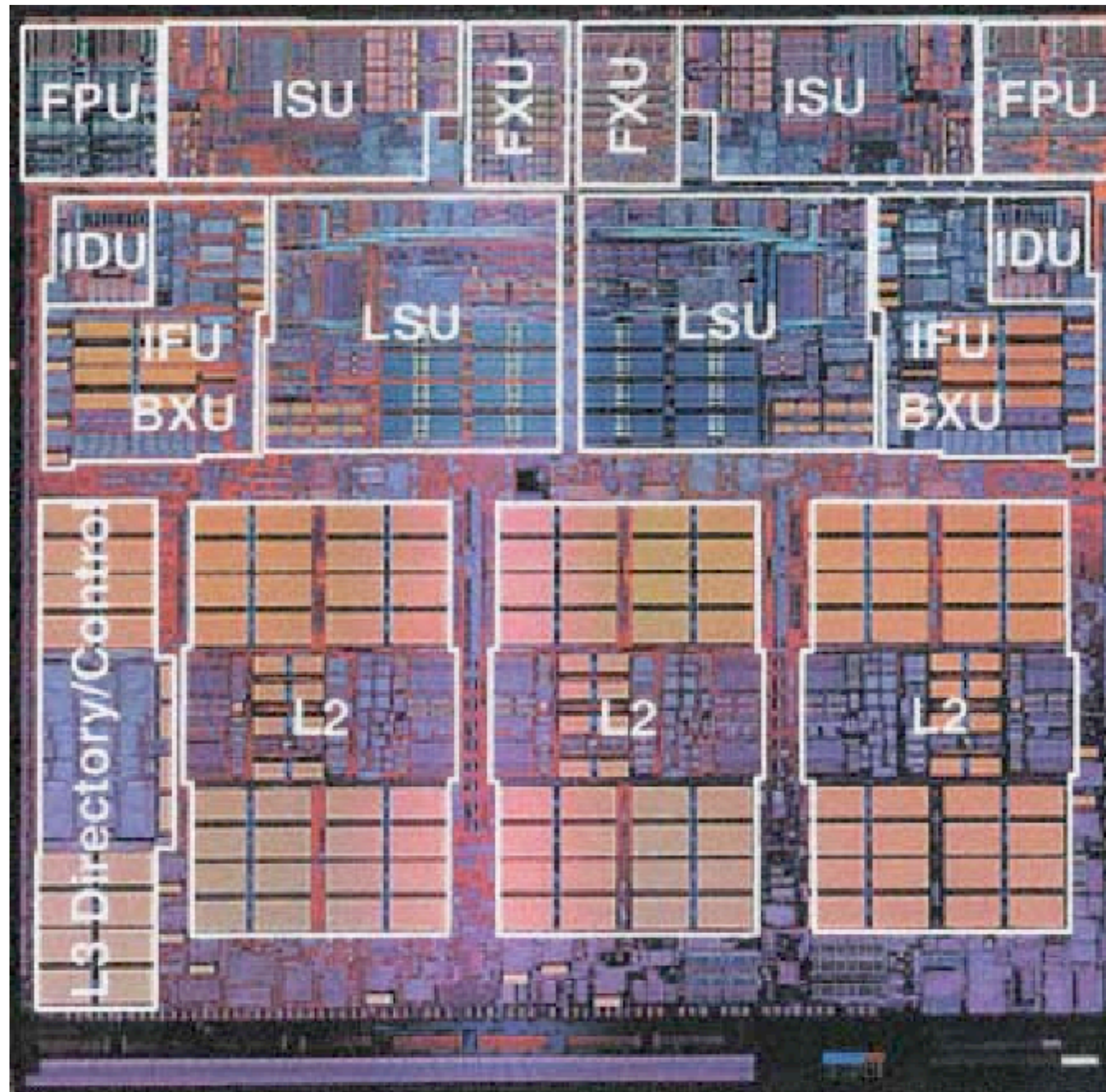




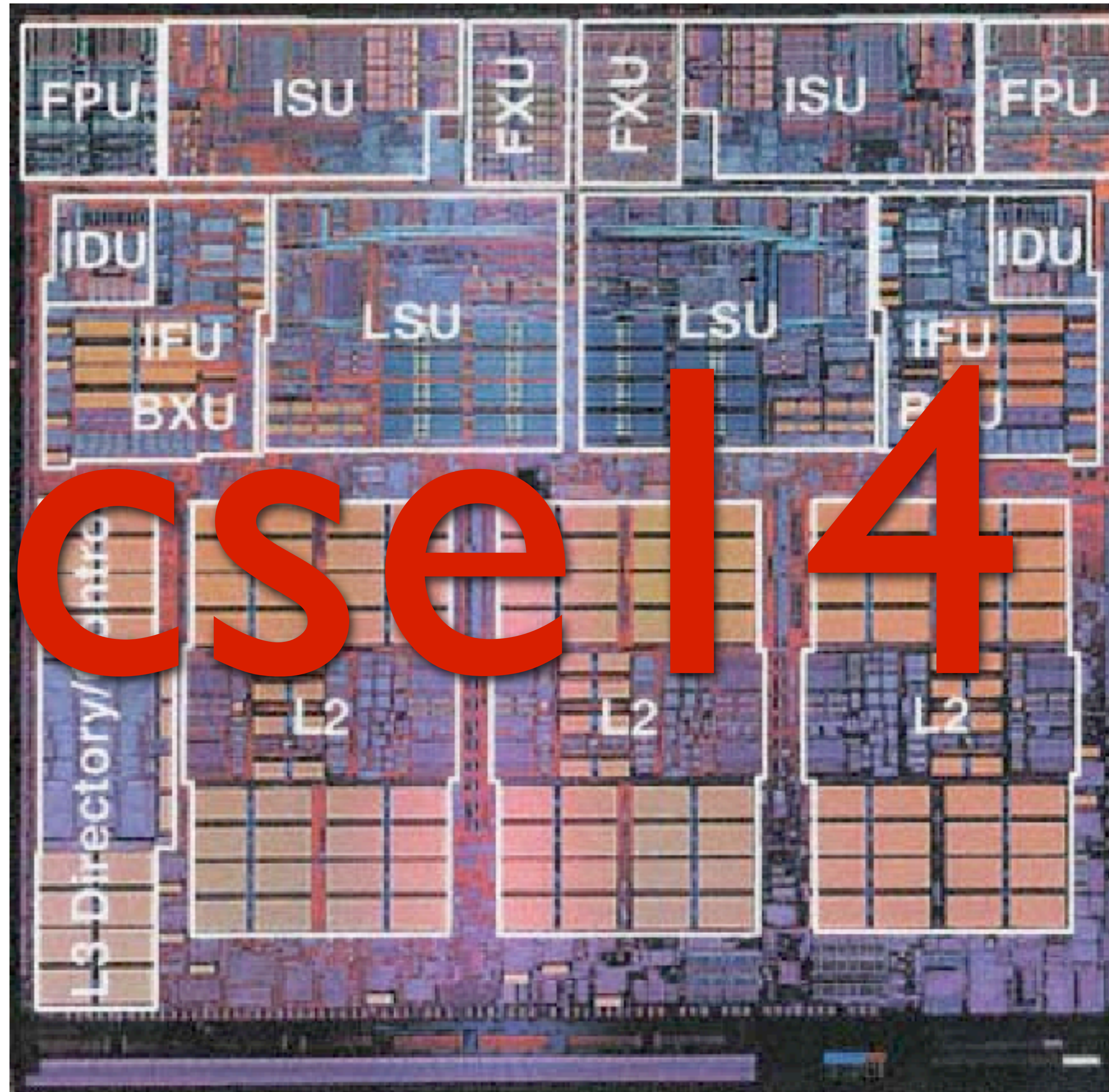

intel®
celeron™ MALAY
566/128/66/1.7V SL4PC
Q040A755-1051


PC modding malaysia

You are here



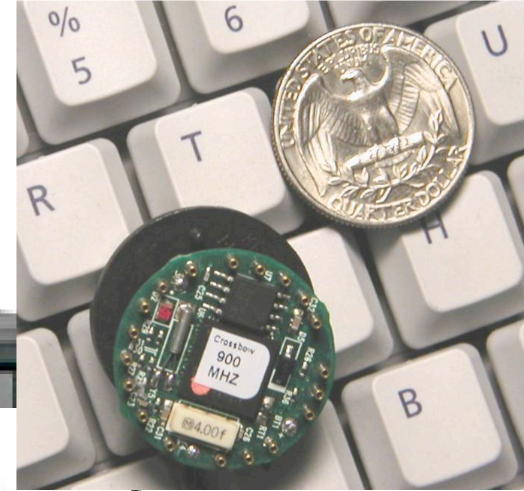
You are here



The processors go here...

The processors go here…

Google™



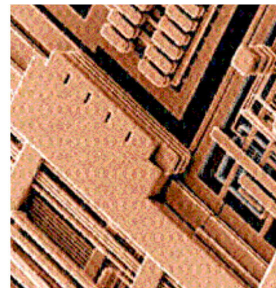
Abstractions of the Physical World…

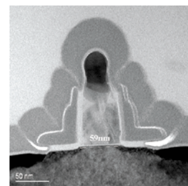
$$\oint H \cdot dl = I + \varepsilon \frac{d}{dt} \iint E \cdot ds$$

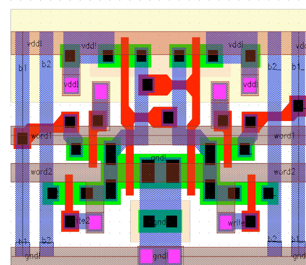
$$\oint E \cdot dl = -\mu \frac{d}{dt} \iint H \cdot ds$$

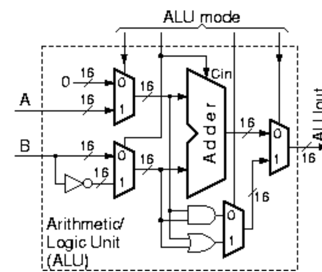
$$\mu \oiint H \cdot ds = 0$$

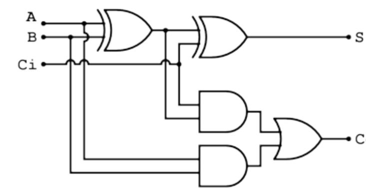
$$\varepsilon \oiint E \cdot ds = \iiint q_v dv$$

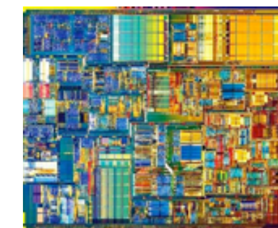


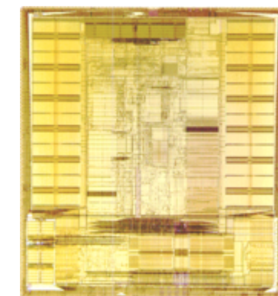


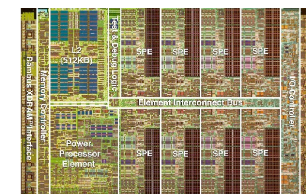


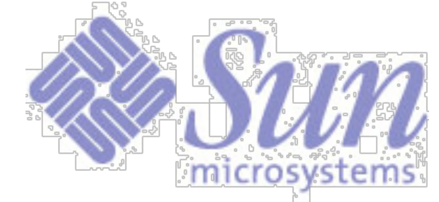












Physics/Materials Devices Micro-architecture Processors Architectures

Abstractions of the Physical World…

Physics/
Chemistry/
Material science

$$\oint H \cdot dl = I + \varepsilon \frac{d}{dt} \iint E \cdot ds$$

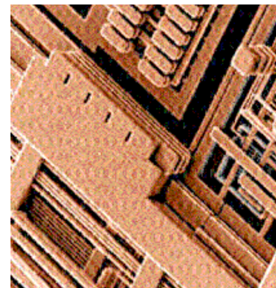
$$\oint E \cdot dl = -\mu \frac{d}{dt} \iint H \cdot ds$$

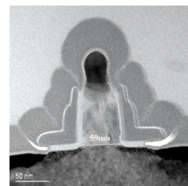
$$\mu \oiint H \cdot ds = 0$$

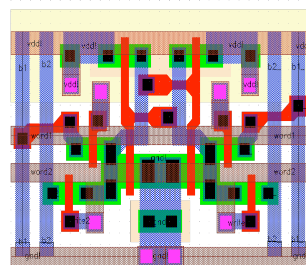
$$\varepsilon \oiint E \cdot ds = \iiint q_v dv$$

Physics/Materials

cse241a/
ECE dept

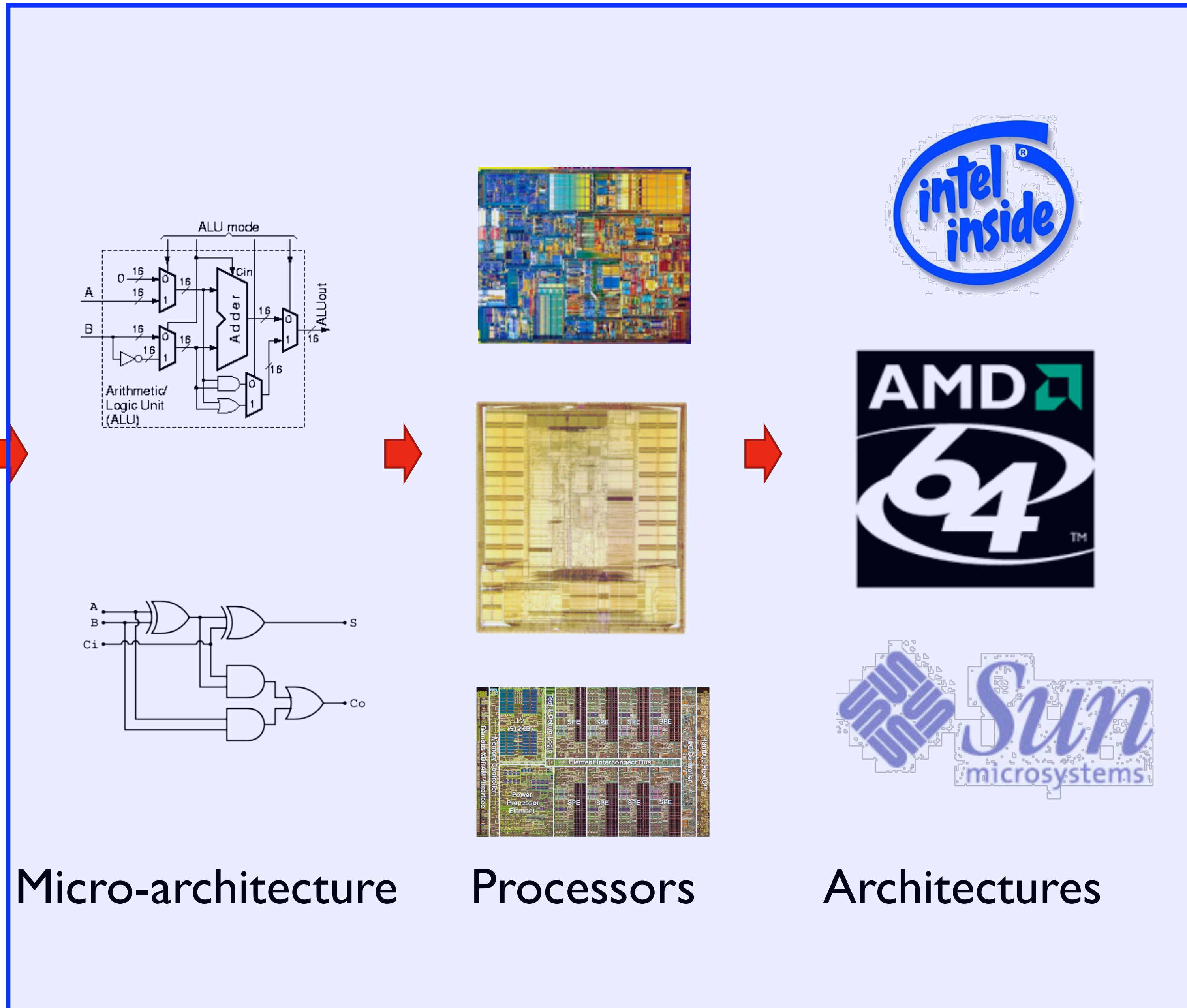






Devices

This Course

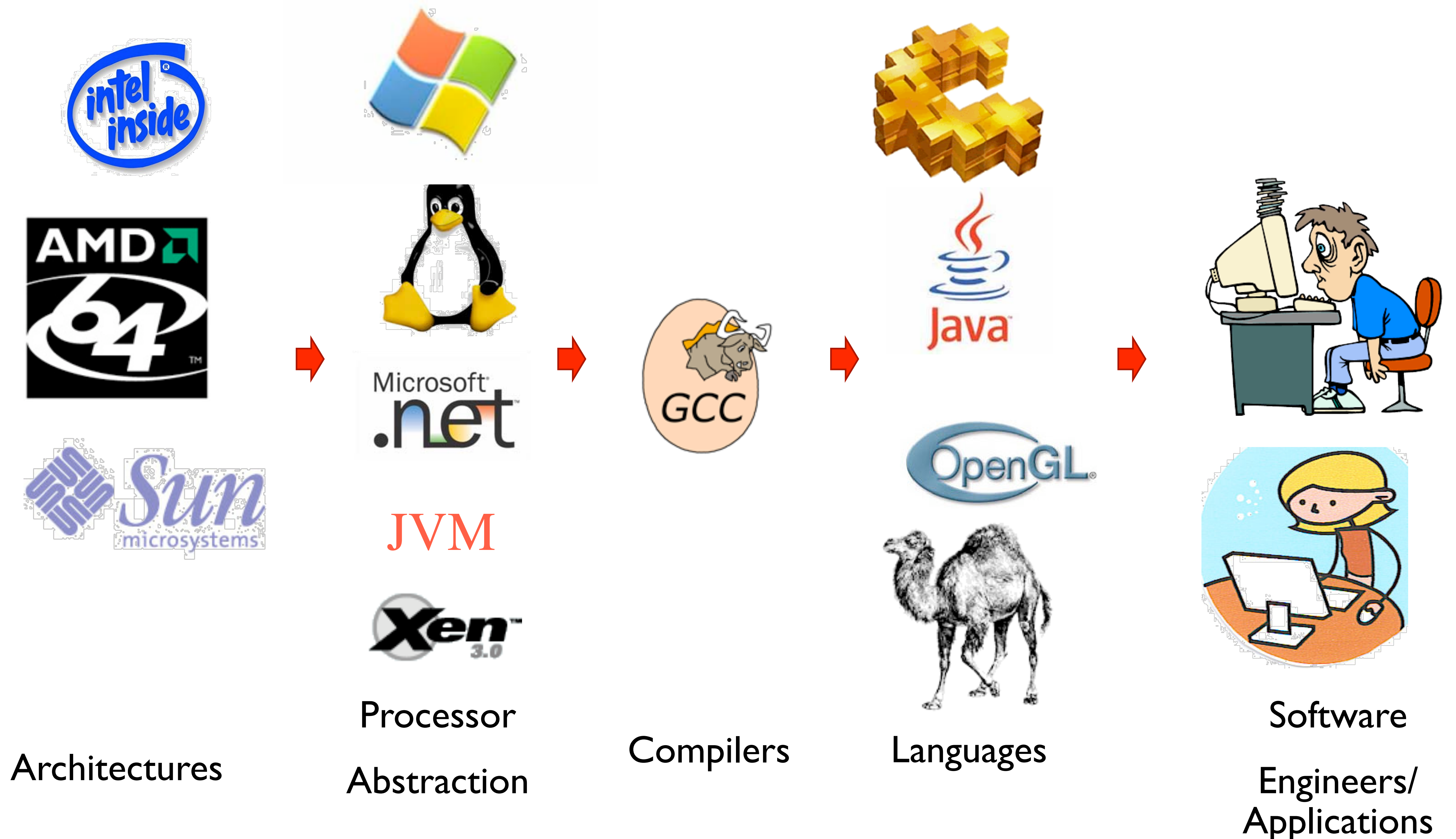


Micro-architecture

Processors

Architectures

…for the Rest of the System



…for the Rest of the System

cse121 cse131 cse130 cseEverythingElse

Architectures

Microsoft .net

JVM

Xen 3.0

Processor Abstraction

GCC

Compilers

Java

OpenGL

Languages

Software Engineers/ Applications

Why study architecture?

- As CEs or CSs you should understand how computers work
  - Processors are the basis for everything in CS (except theory)
  - They are where the rubber meets the road.
- Performance is important
  - Faster machines make applications cheaper
  - Understanding hardware is essential to understanding how systems behave
- It's cool!
  - Microprocessors are among the most sophisticated devices manufactured by people
  - How they work (and even that they work) as reliably and as quickly as they do is amazing.
- Architecture is undergoing a revolution
  - The future is uncertain
  - Opportunities for innovation abound.

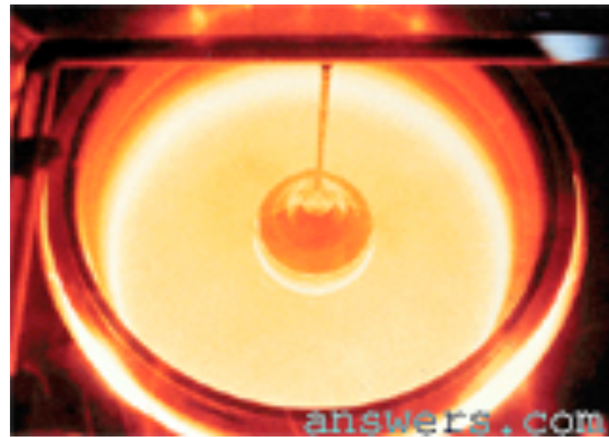
Performance and You!

- Live Demo

Processor are Cool!

- Chips are made of silicon
  - Aka “sand”
  - The most adundant element in the earth’s crust.
  - Extremely pure (<1 part per billion)
  - This is the purest stuff people make

Building Chips



Building Chips

- Photolithography

Silicon Wafer

Building Chips

- Photolithography

Silicon Wafer

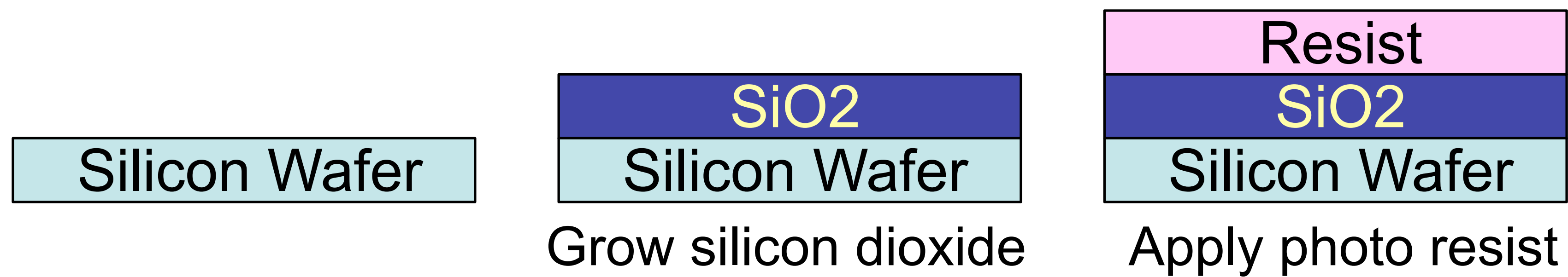
SiO_2

Silicon Wafer

Grow silicon dioxide

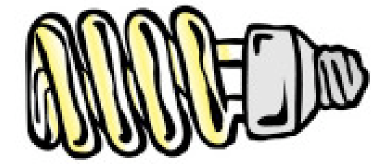
Building Chips

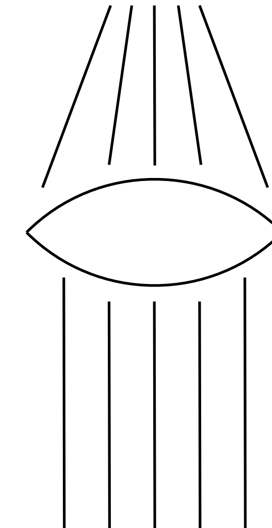
- Photolithography



Building Chips

- Photolithography



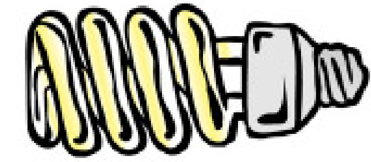


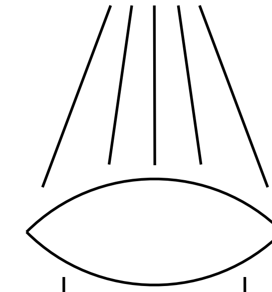
Mask Mask

		Resist	Resist
	SiO2	SiO2	SiO2
Silicon Wafer	Silicon Wafer	Silicon Wafer	Silicon Wafer
	Grow silicon dioxide	Apply photo resist	Expose to UV

Building Chips

- Photolithography





Mask Mask

Silicon Wafer

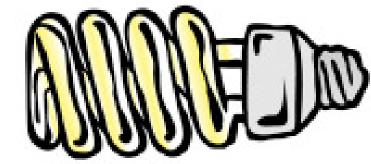
SiO2
Silicon Wafer
Grow silicon dioxide

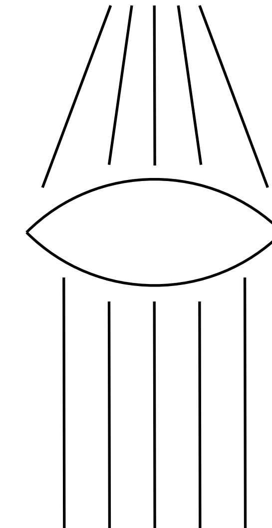
Resist
SiO2
Silicon Wafer
Apply photo resist

Resist
SiO2
Silicon Wafer
Expose to UV

SiO2
Silicon Wafer
Patterned resist

Building Chips





- Photolithography

Mask Mask

Resist	Resist

SiO2 Silicon Wafer

Silicon Wafer

Grow silicon dioxide

Resist
SiO2
Silicon Wafer

Apply photo resist

Resist
SiO2
Silicon Wafer

Expose to UV

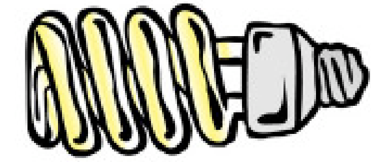
SiO2
Silicon Wafer

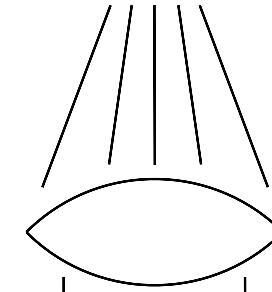
Patterned resist

Silicon Wafer

Etch SiO2

Building Chips





- Photolithography

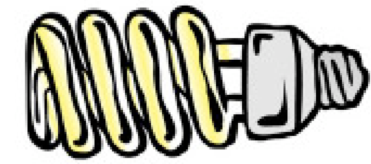
Mask Mask

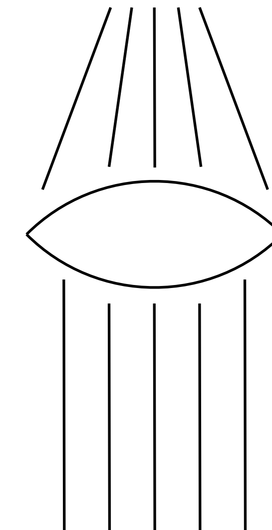
		Resist	Resist
	SiO2	SiO2	SiO2
Silicon Wafer	Silicon Wafer	Silicon Wafer	Silicon Wafer
	Grow silicon dioxide	Apply photo resist	Expose to UV

SiO2		Met
Silicon Wafer	Silicon Wafer	Silicon Wafer
Patterned resist	Etch SiO2	Deposit metal

Building Chips

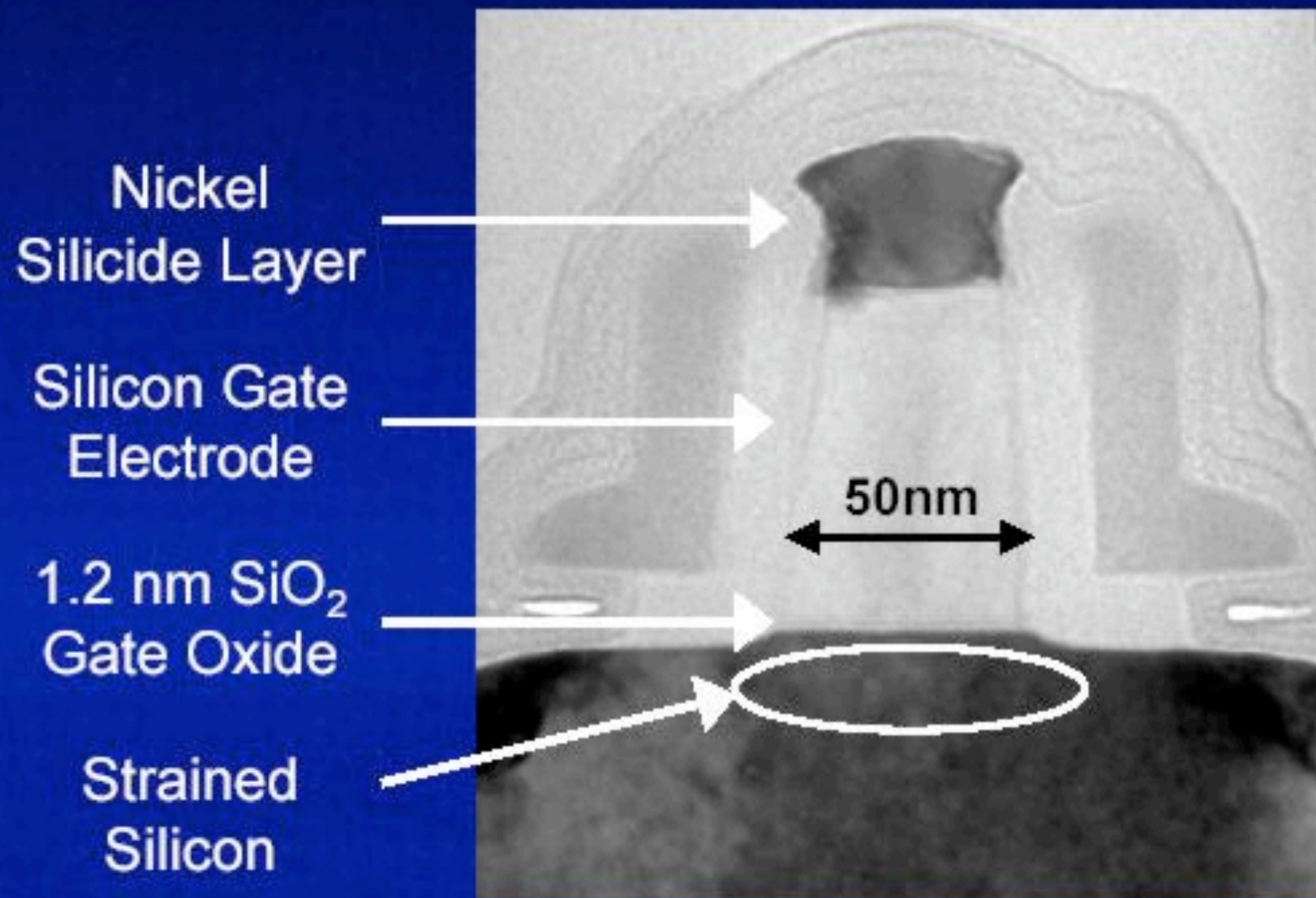
- Photolithography





Building Blocks: Transistors

90 nm Generation Transistor

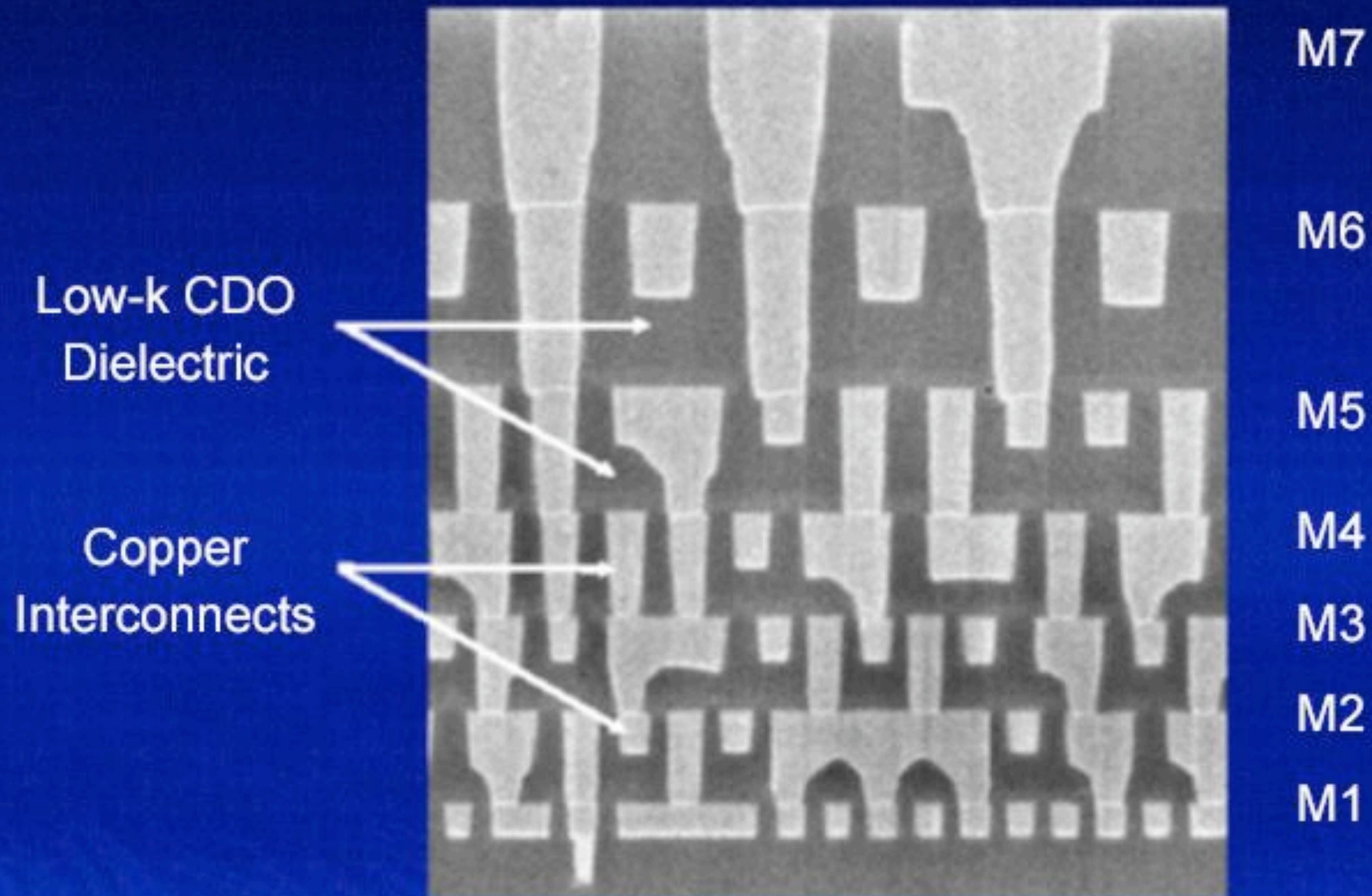


No other company combines these transistor features at the 90 nm generation

Intel

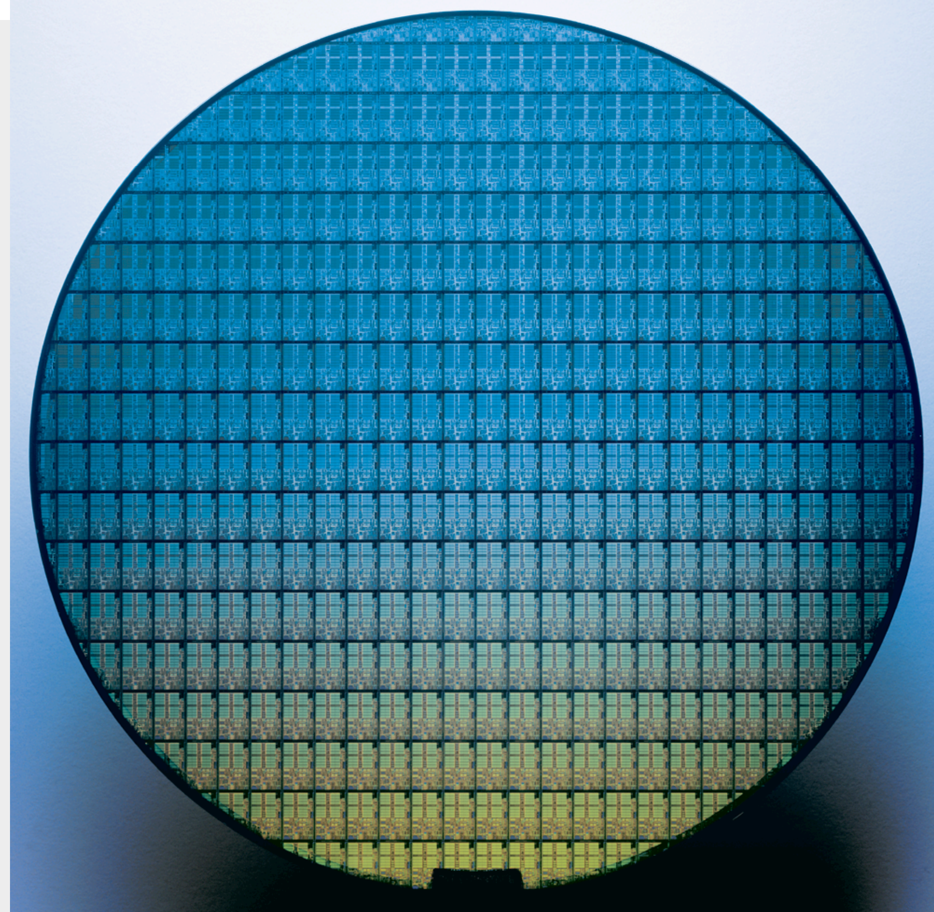
3

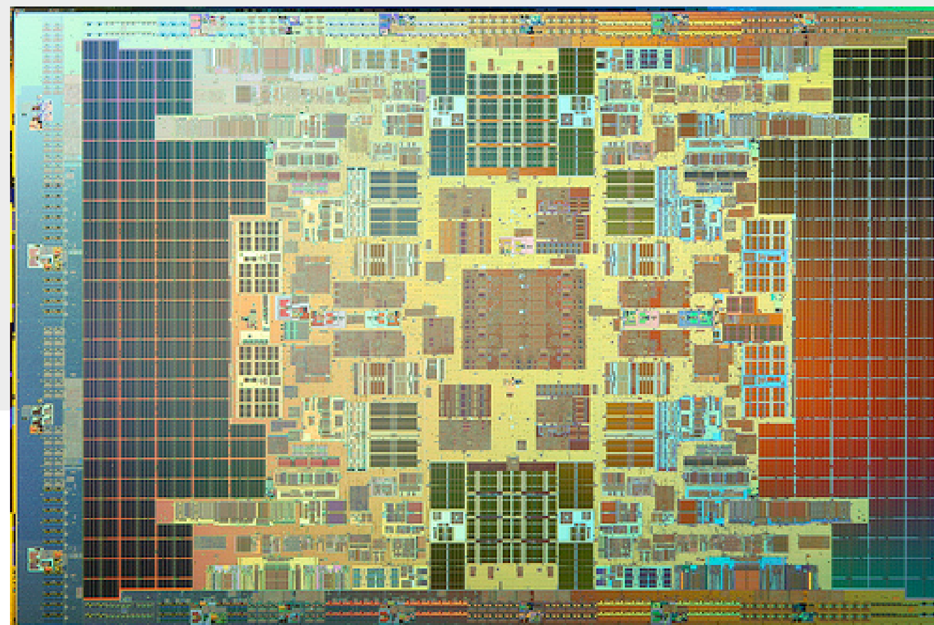
Building Blocks: Wires



State of the art CPU

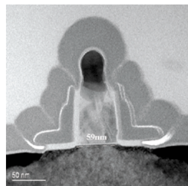
- 1-2 Billion xtrs
- 45nm features
- 3-4Ghz
- Several 100 designers
- >5 years
- $3Billion fab
- 70 GFLOPS

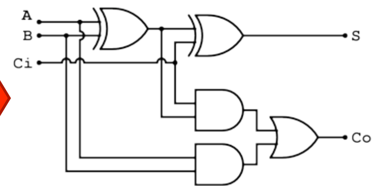


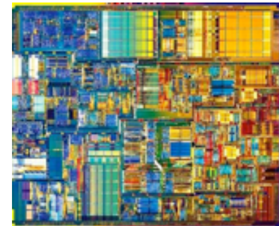


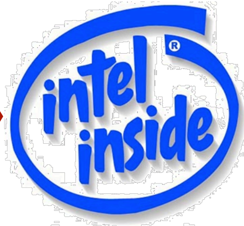
Current state of architecture

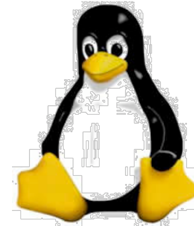
Since 1940



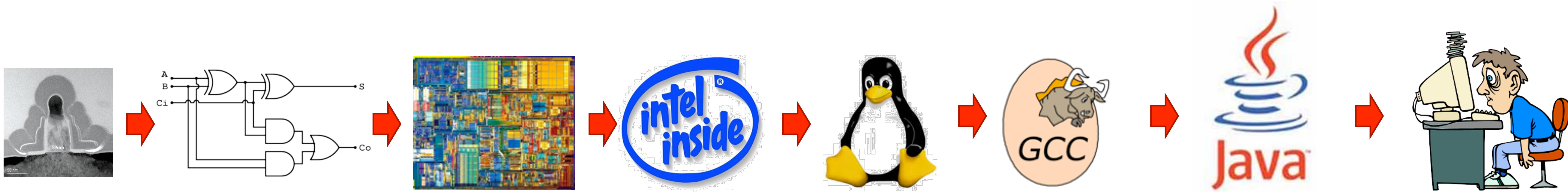






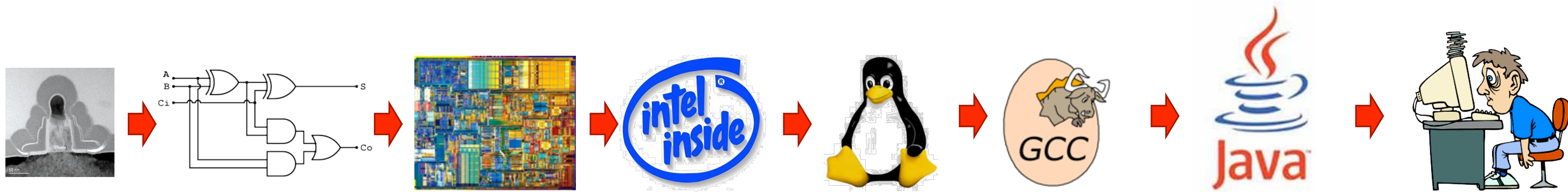


Since 1940



- Plug boards -> Java
- Hand assembling -> GCC
- No OS -> Windows Vista

Since 1940

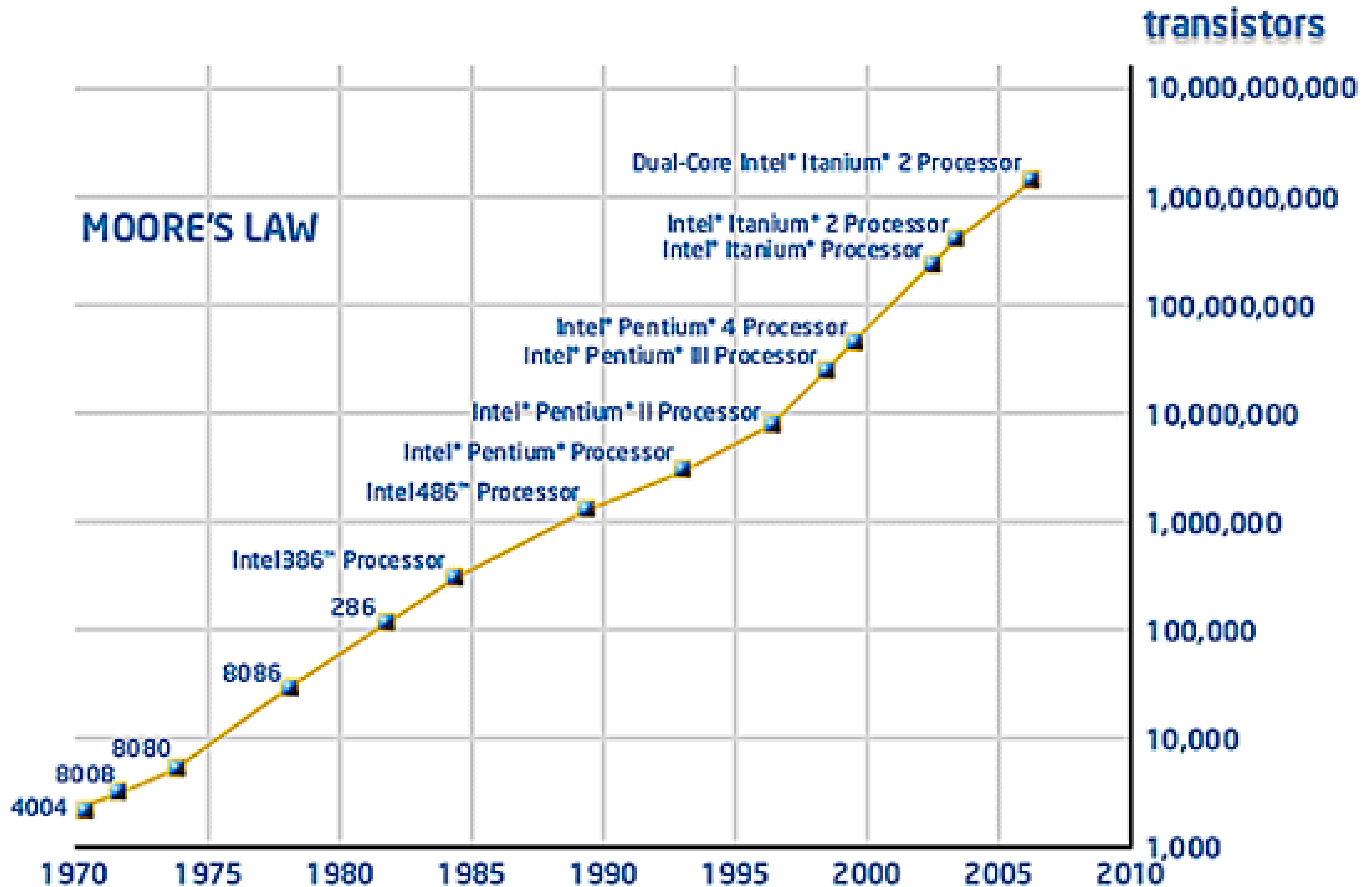


- 50,000 x speedup
- >1,000,000,000 x density (Moore's Law)

- Plug boards -> Java
- Hand assembling -> GCC
- No OS -> Windows Vista

Flexible performance is a liquid asset

Moore's Law: Raw transistors

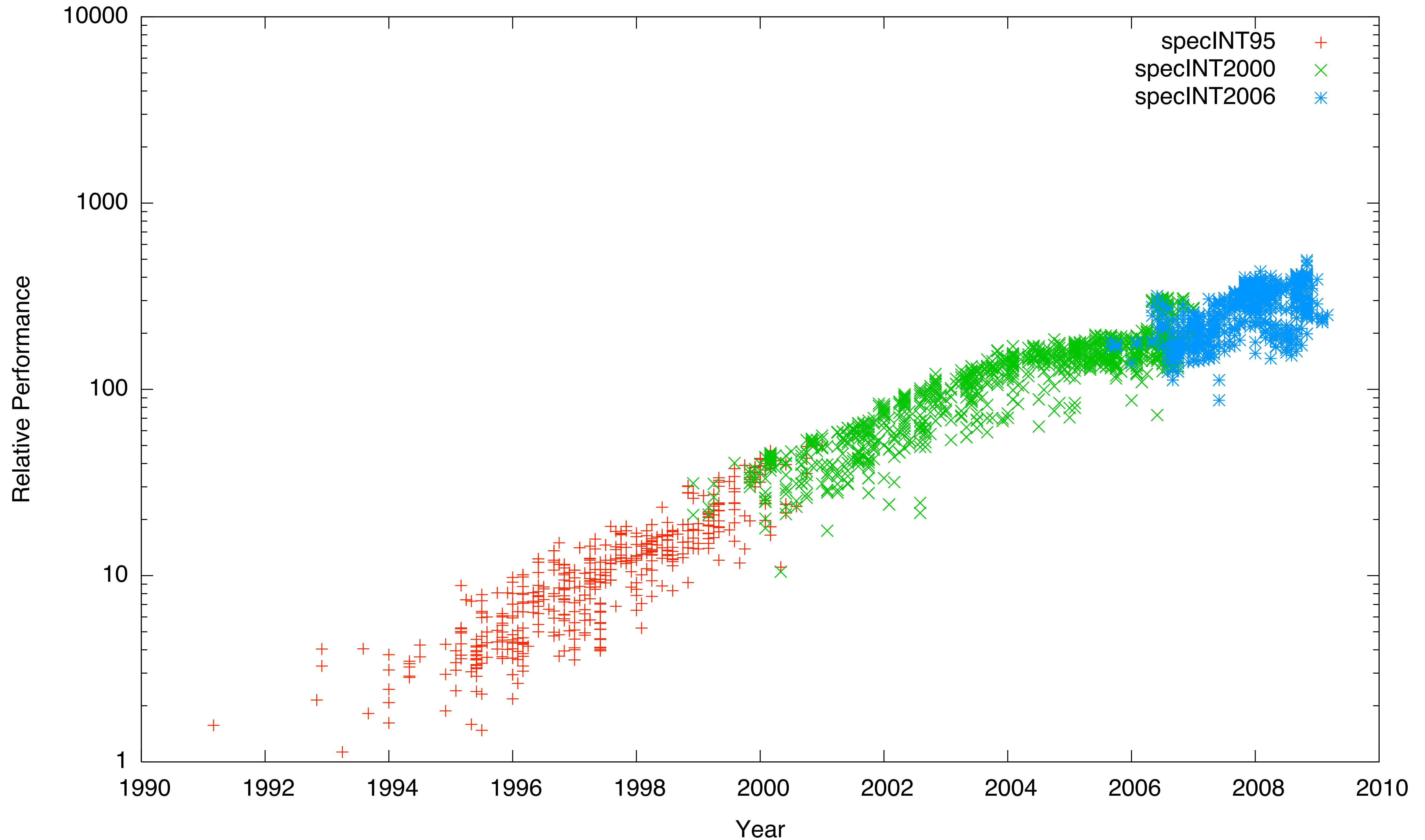


The Importance of Architecture

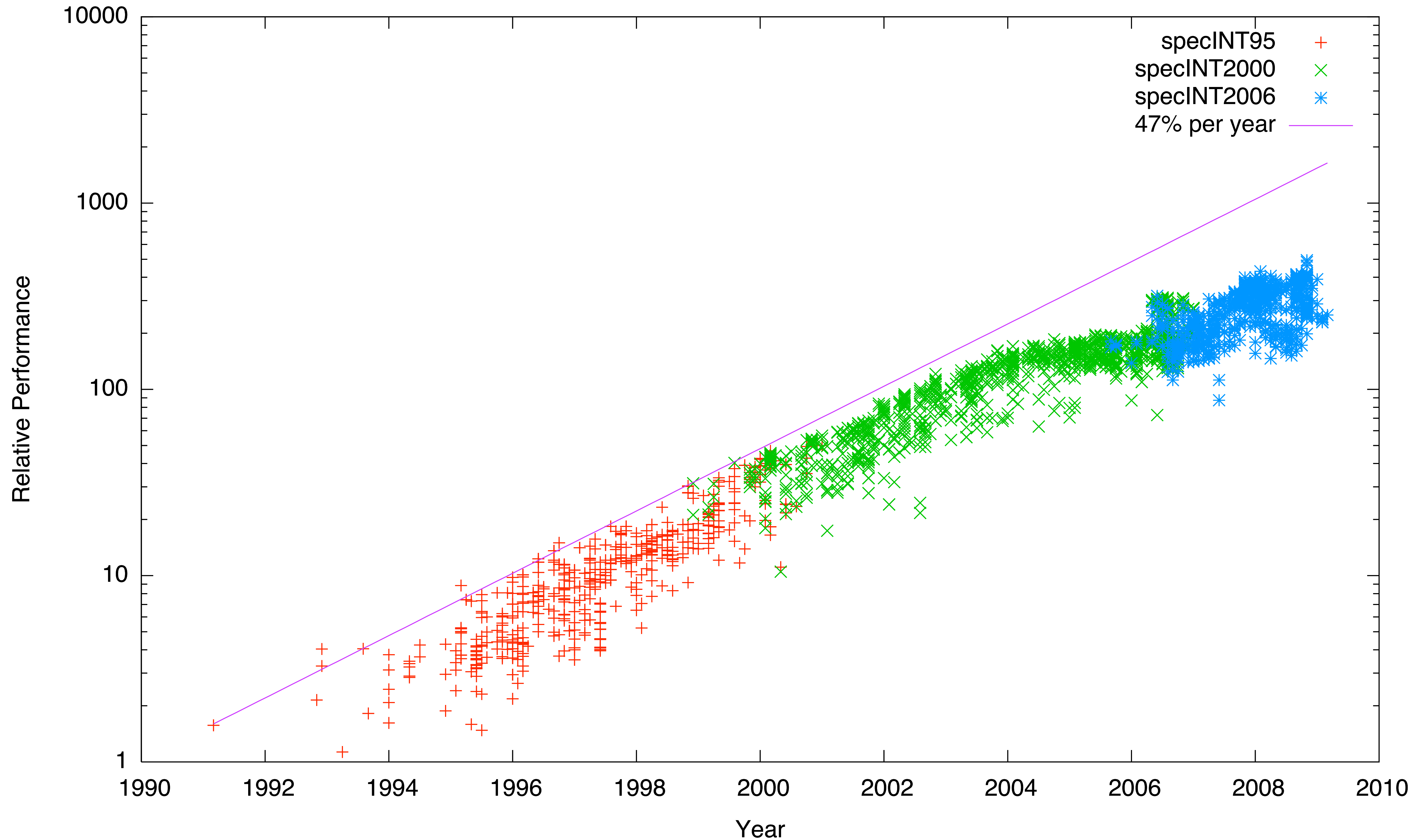
- We design smarter and smarter processors
  - Process technology gives us about 20% performance improvement per year
  - Until 2004, performance grew at about 40% per year.
- The gap is due to architecture! (and compilers)

Computer Performance

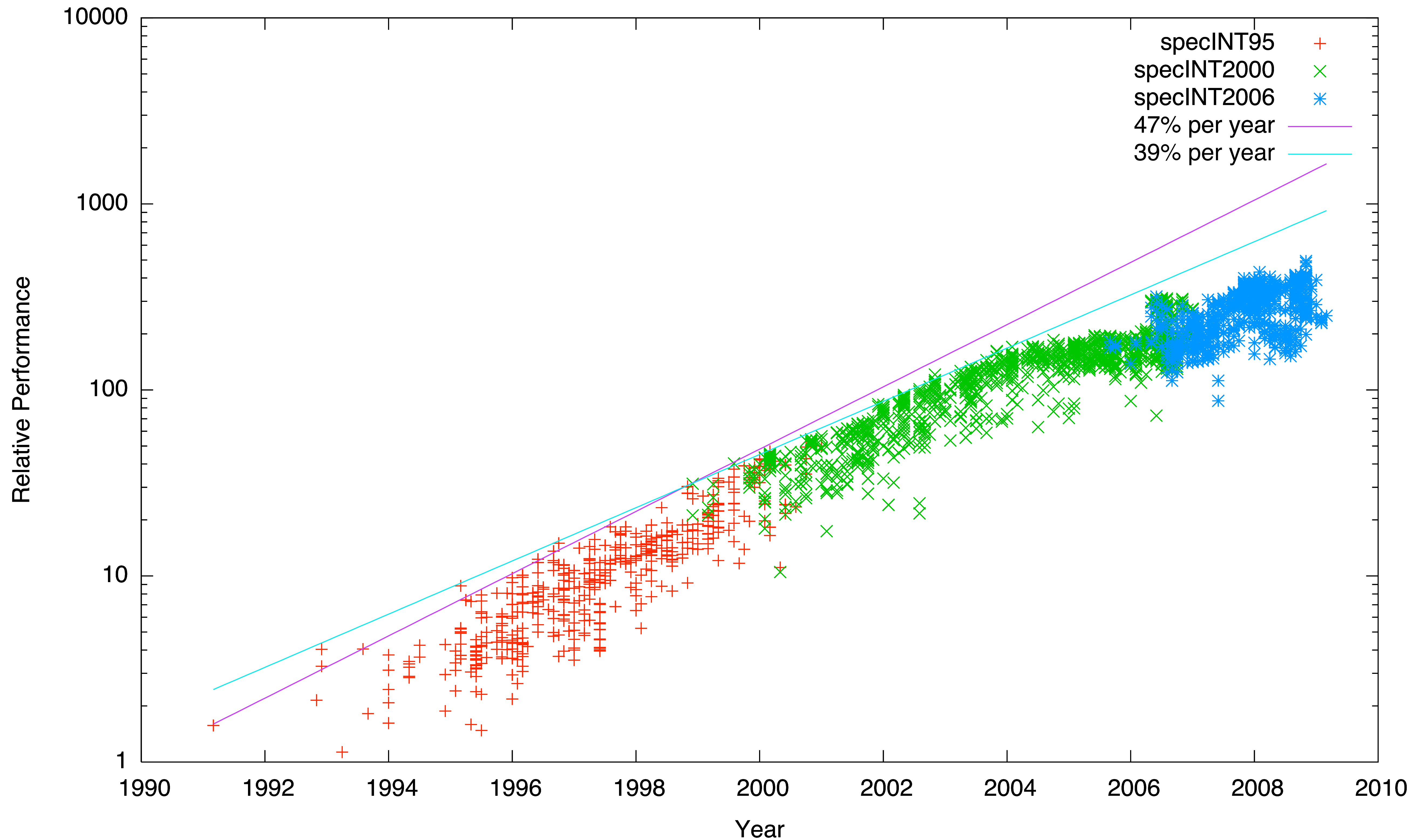
Computer Performance



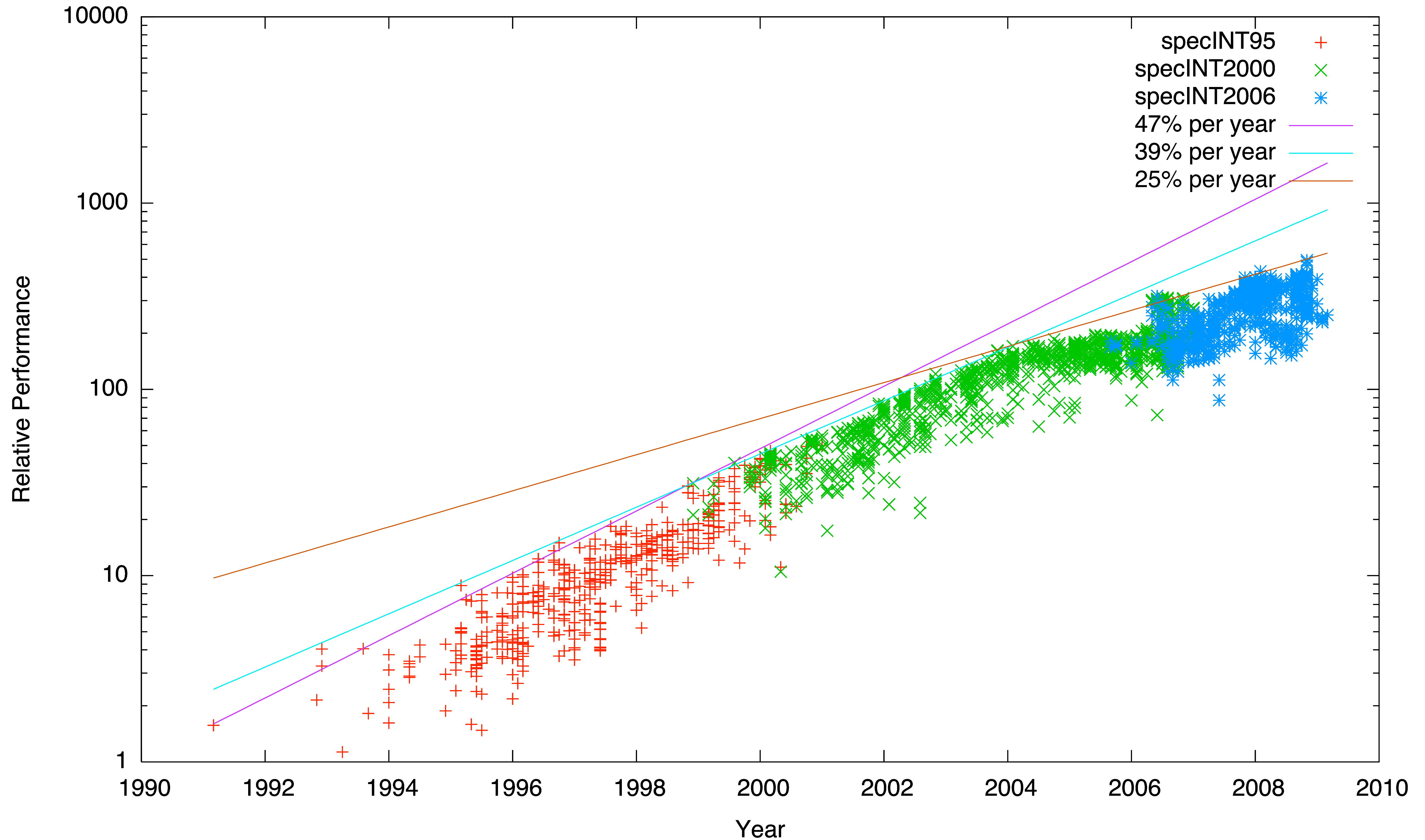
Computer Performance



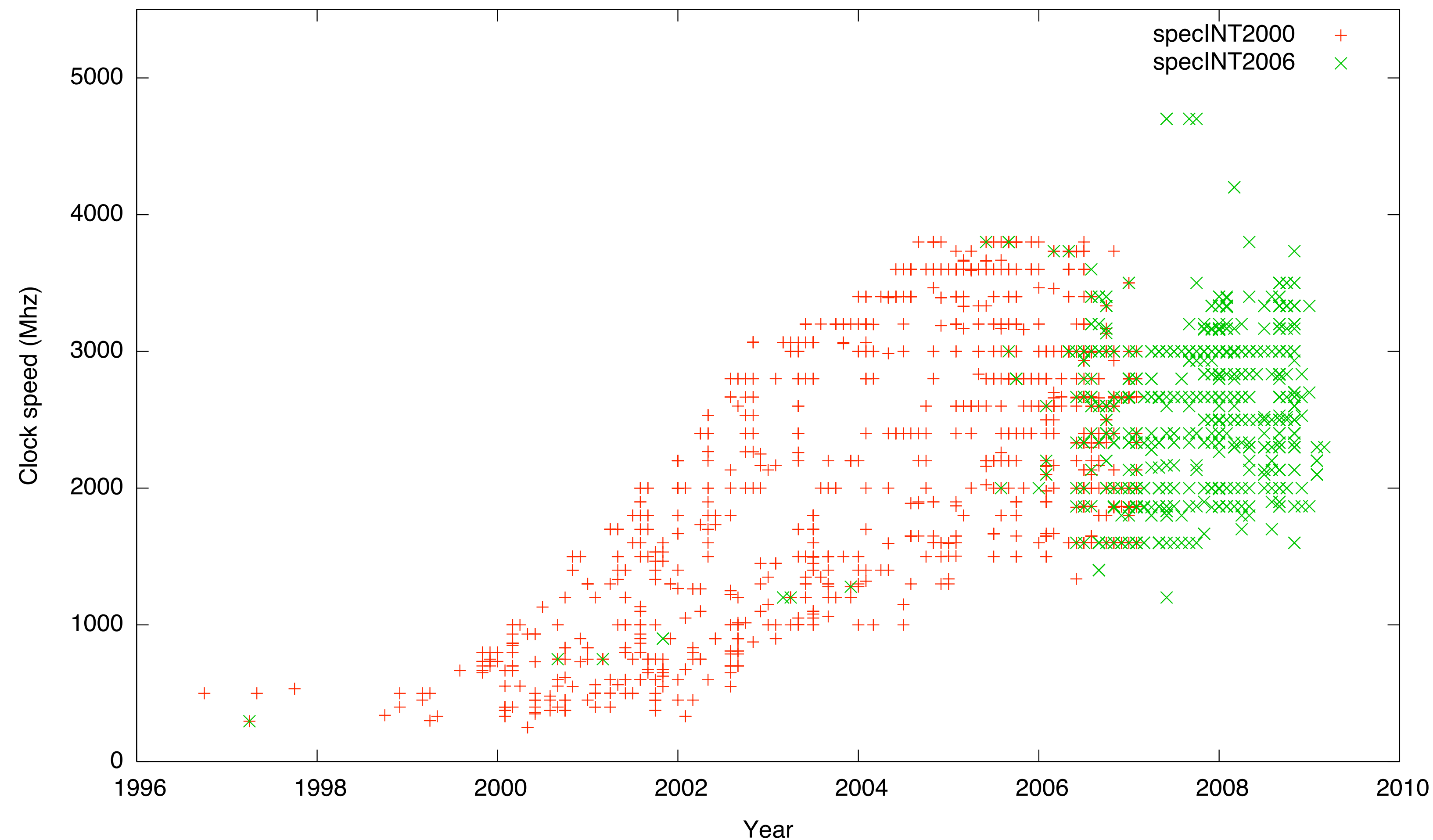
Computer Performance



Computer Performance

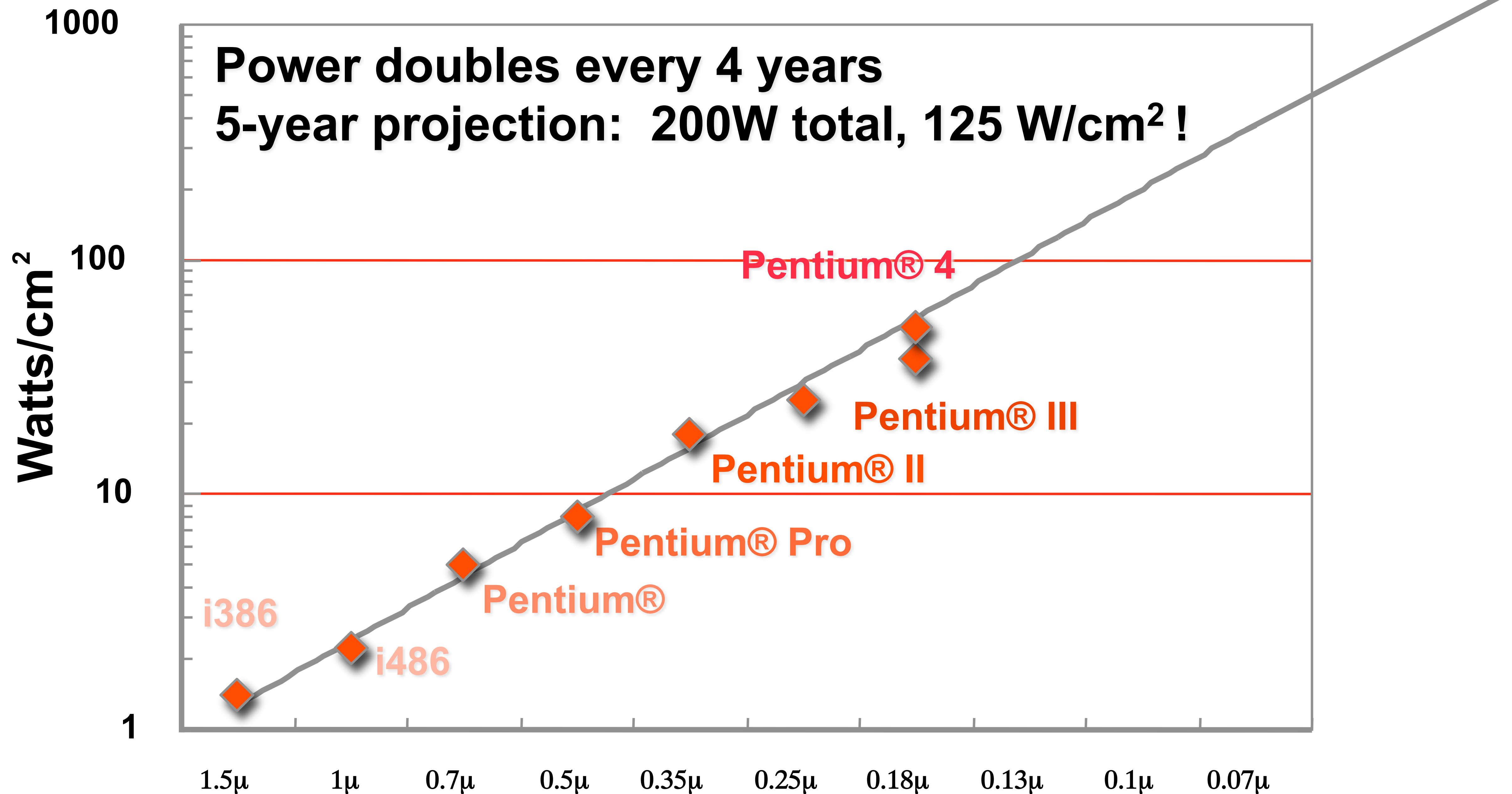


The clock speed addiction



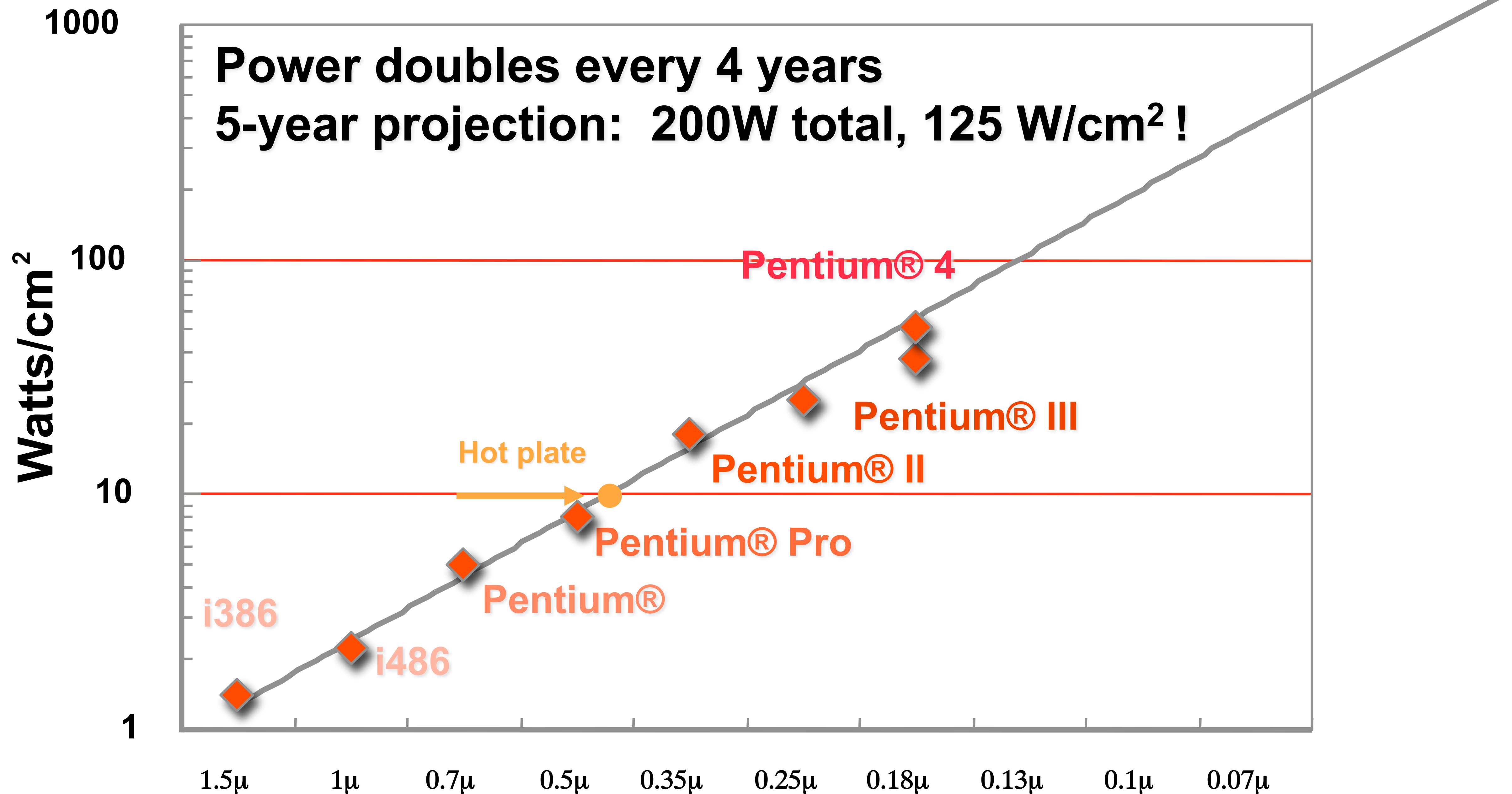
- Clock speed is the biggest contributor to power
  - Chip manufactures (Intel, esp.) pushed clock speeds very hard in the 90s and early 2000s.
  - Doubling the clock speed increases power by 2-8x
  - Clock speed scaling is essentially finished.

Power



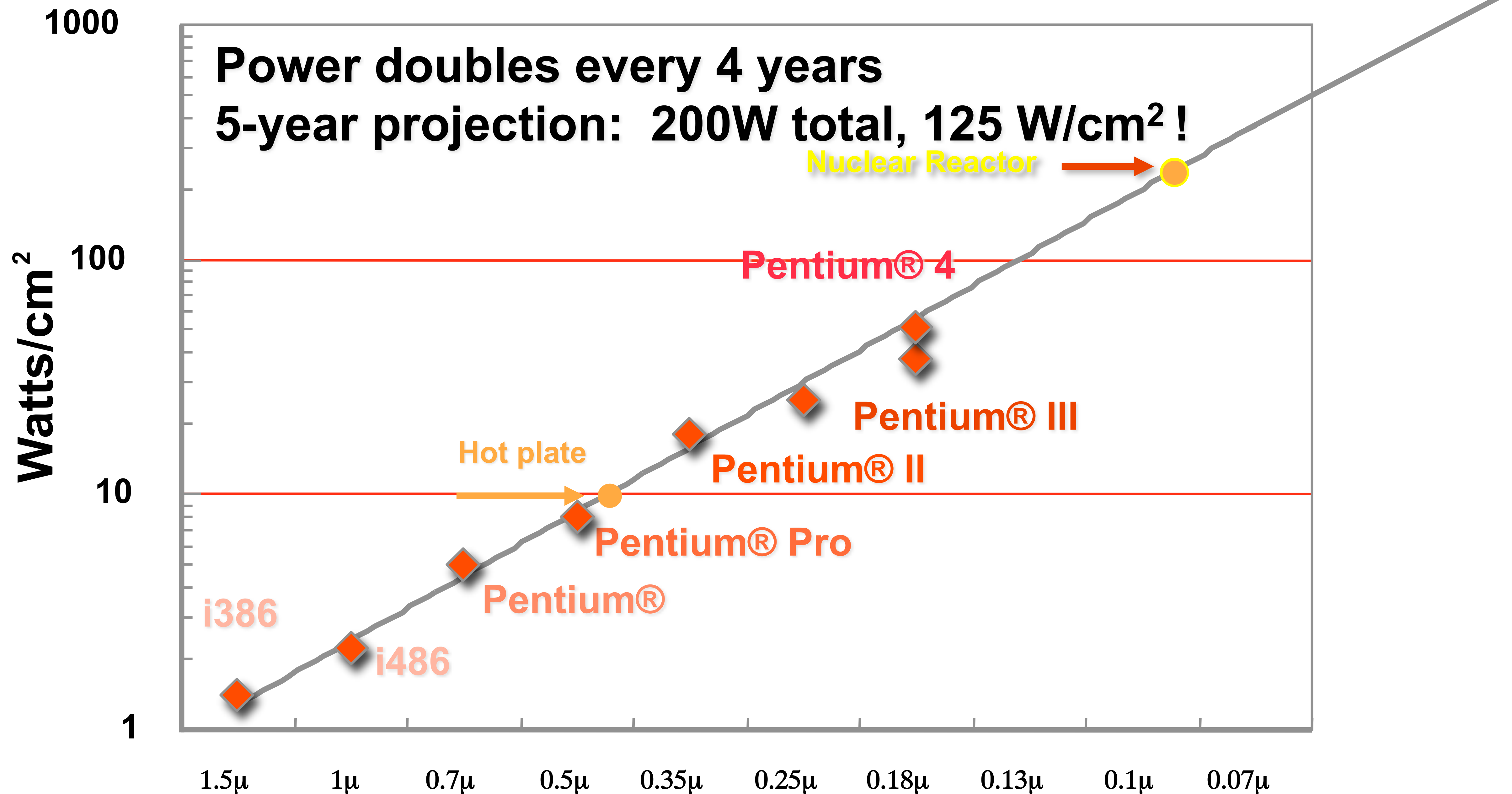
From "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies" – Fred Pollack, Intel Corp. Micro32 conference key note - 1999.

Power



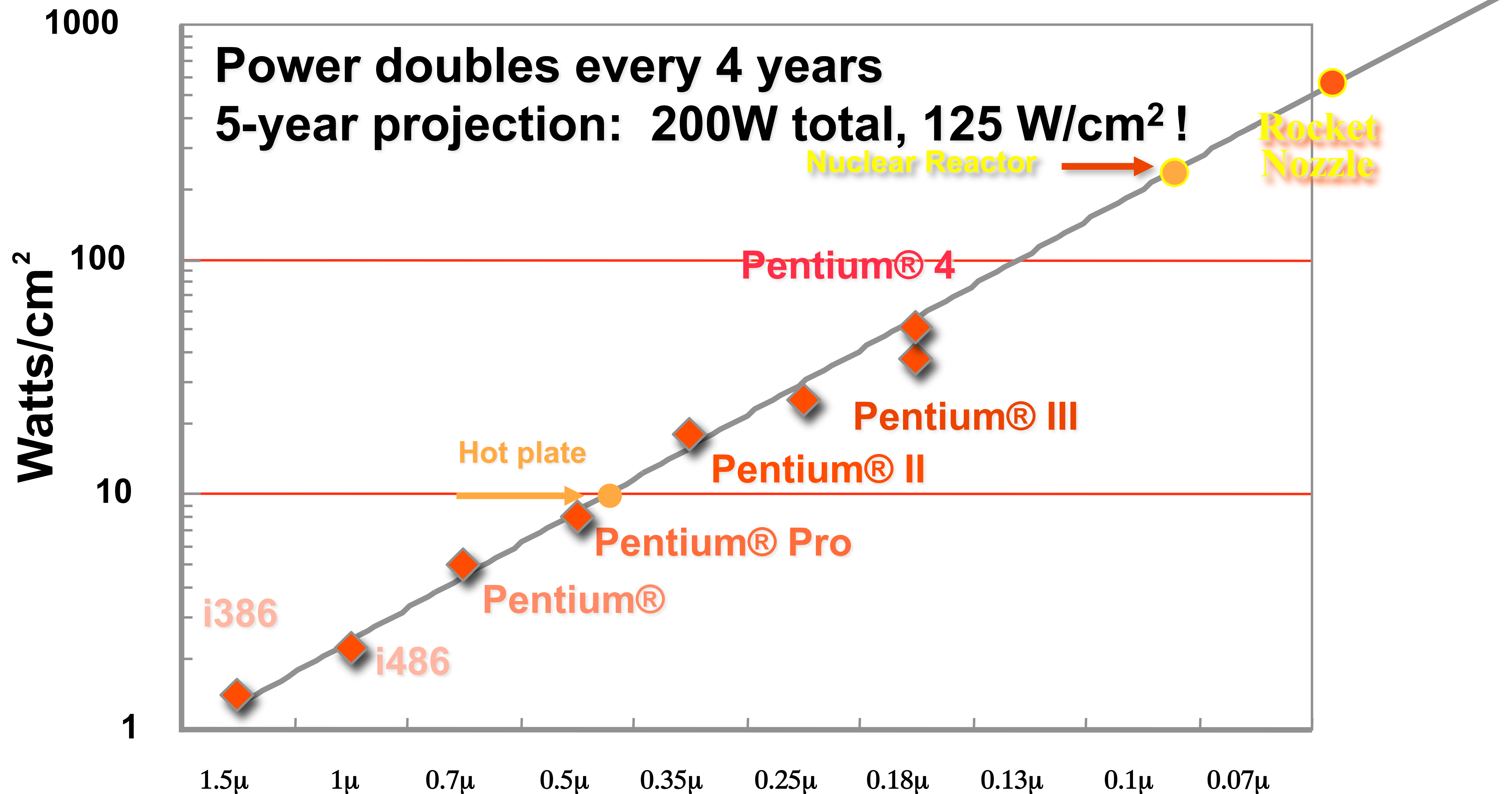
From "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies" – Fred Pollack, Intel Corp. Micro32 conference key note - 1999.

Power



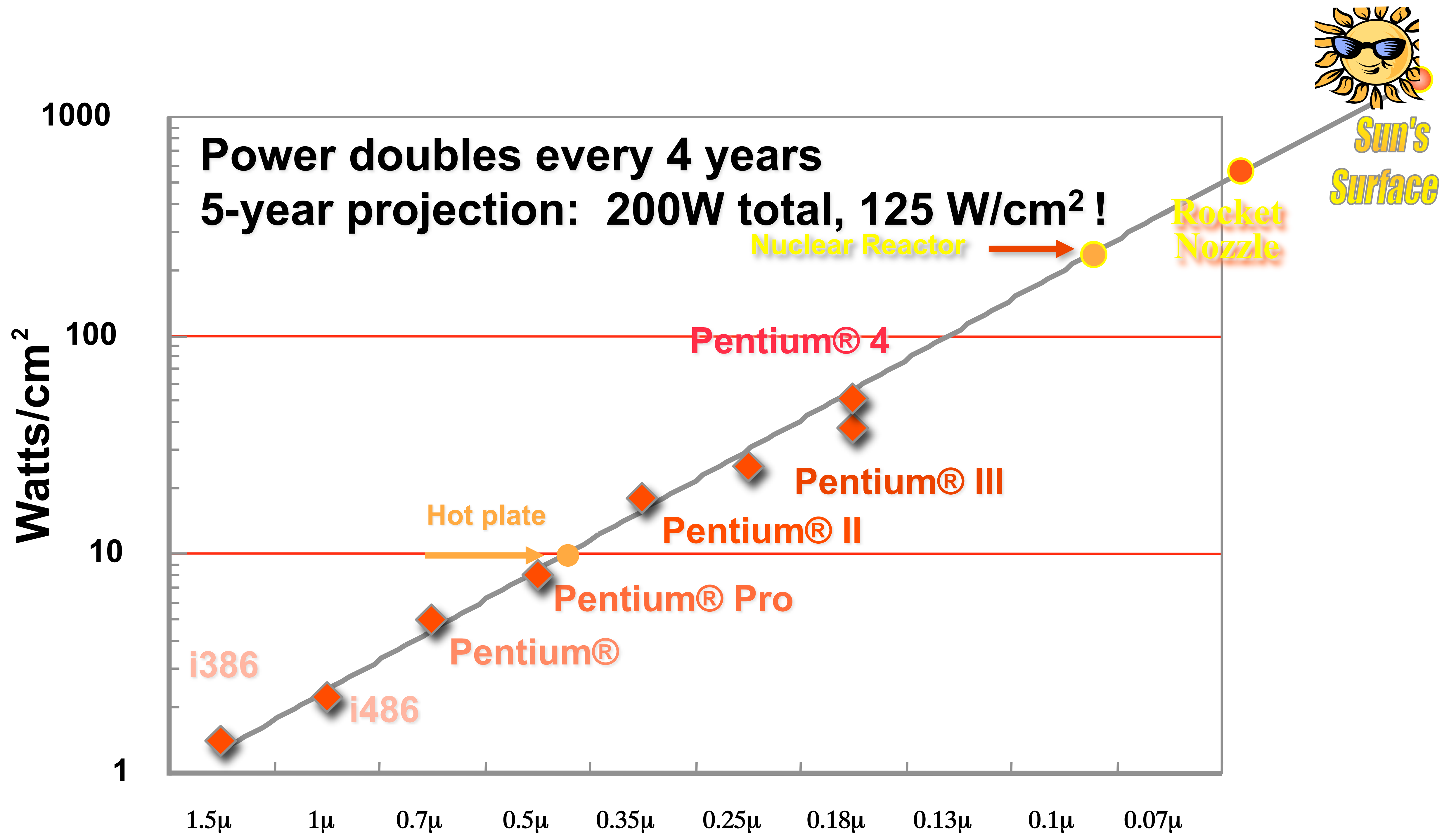
From "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies" – Fred Pollack, Intel Corp. Micro32 conference key note - 1999.

Power



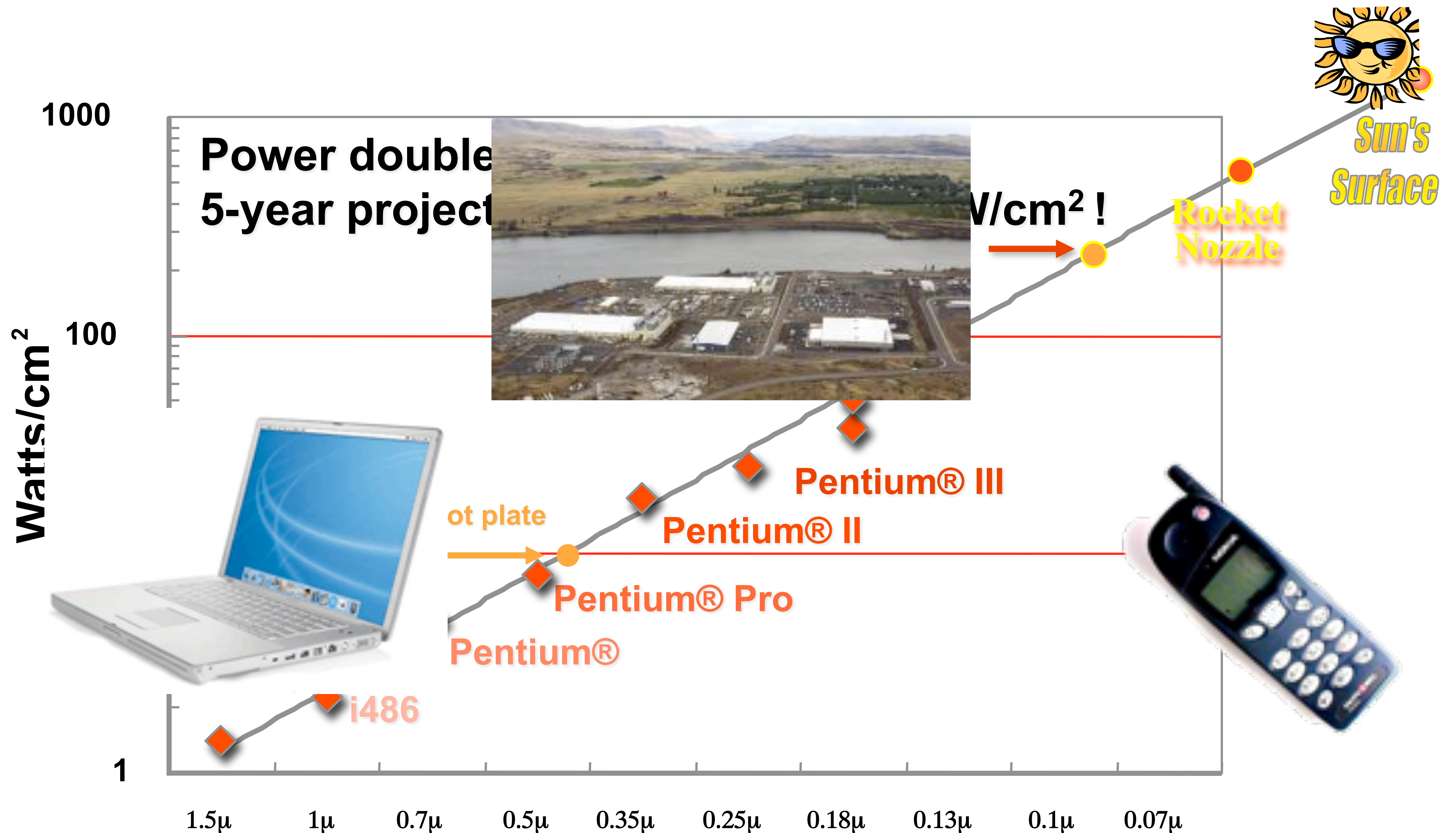
From "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies" – Fred Pollack, Intel Corp. Micro32 conference key note - 1999.

Power



From "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies" – Fred Pollack, Intel Corp. Micro32 conference key note - 1999.

Power



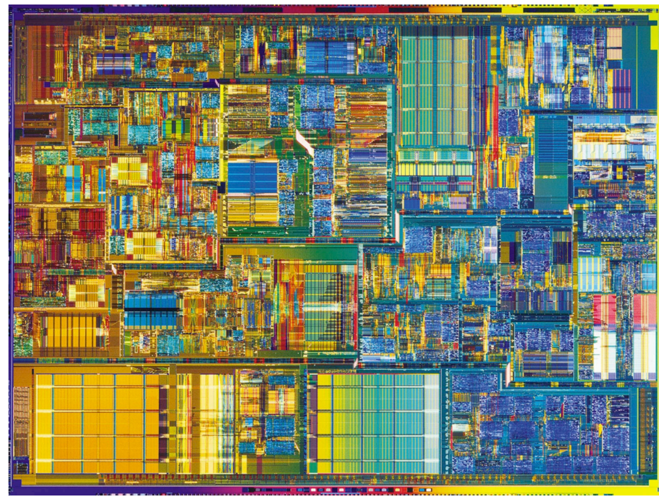
From "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies" – Fred Pollack, Intel Corp. Micro32 conference key note - 1999.

What's Next: Brainiacs

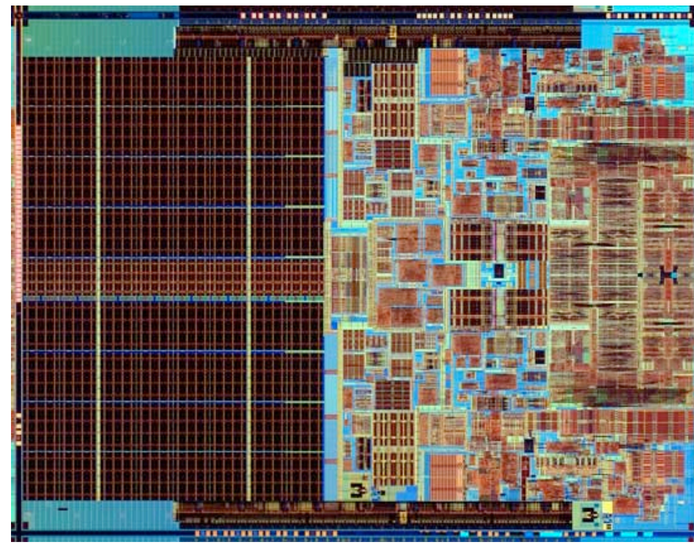
- Hold the clock rate steady.
- Be smarter in silicon
  - More sophisticated processors
  - More clever algorithms
  - This continues to deliver about 25% per year.
  - But for how long?

What's Next: Parallelism

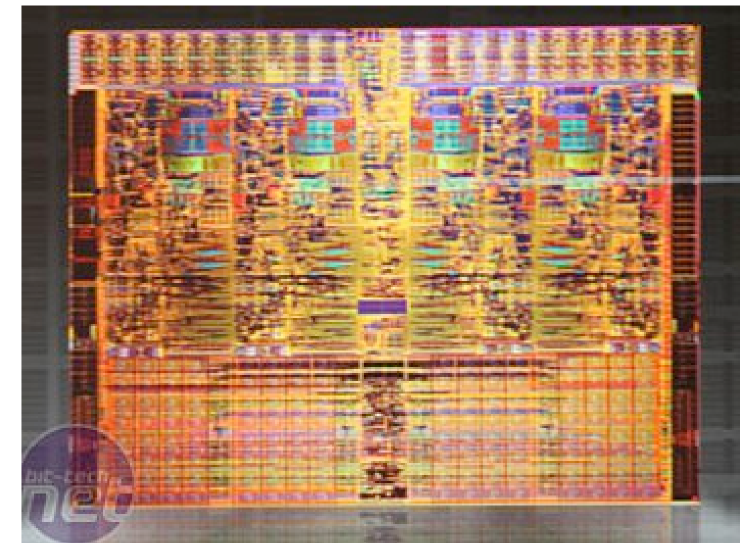
- This is all the rage right now
- You probably own a multi-processor, they used to be pretty exotic.
- They provide some performance, but it's hard to use.
  - There aren't that many threads
  - Remember, *flexible* performance is a liquid asset
  - Remember or look forward to cse121



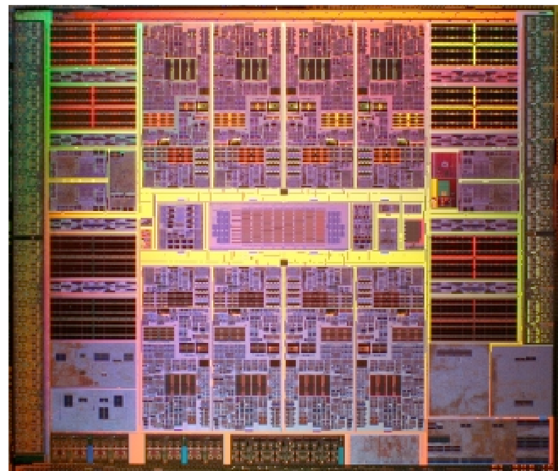
Intel P4
1 core



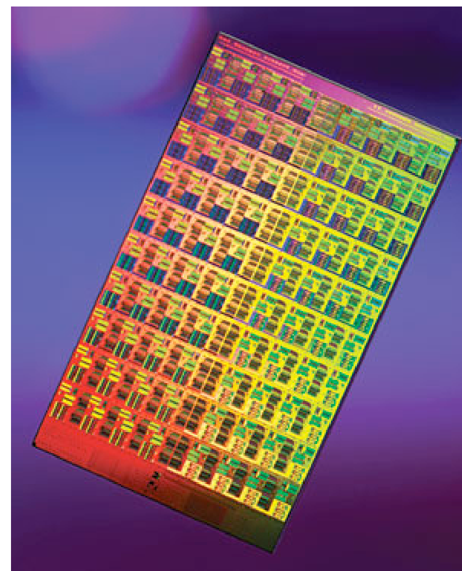
Intel Core 2 Duo
2 cores



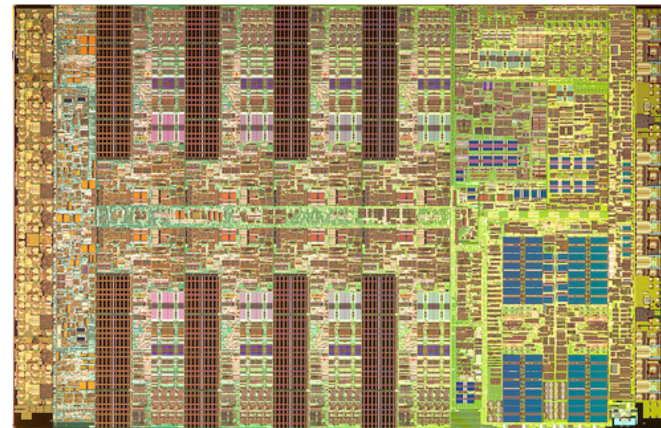
Intel Nahalem
4 cores



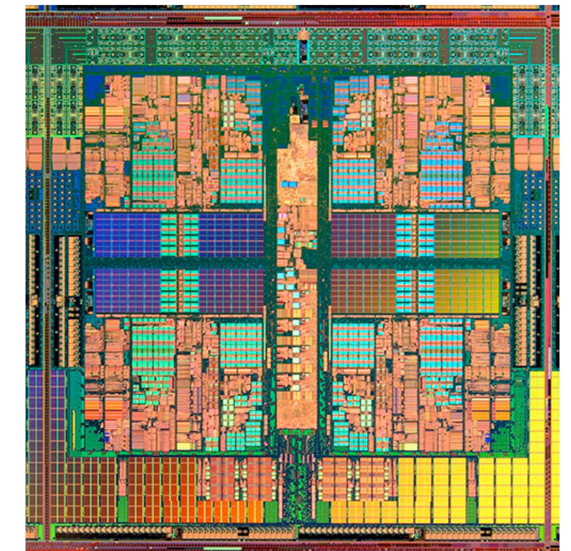
SPARC T1
8 cores



Intel Prototype
80 cores



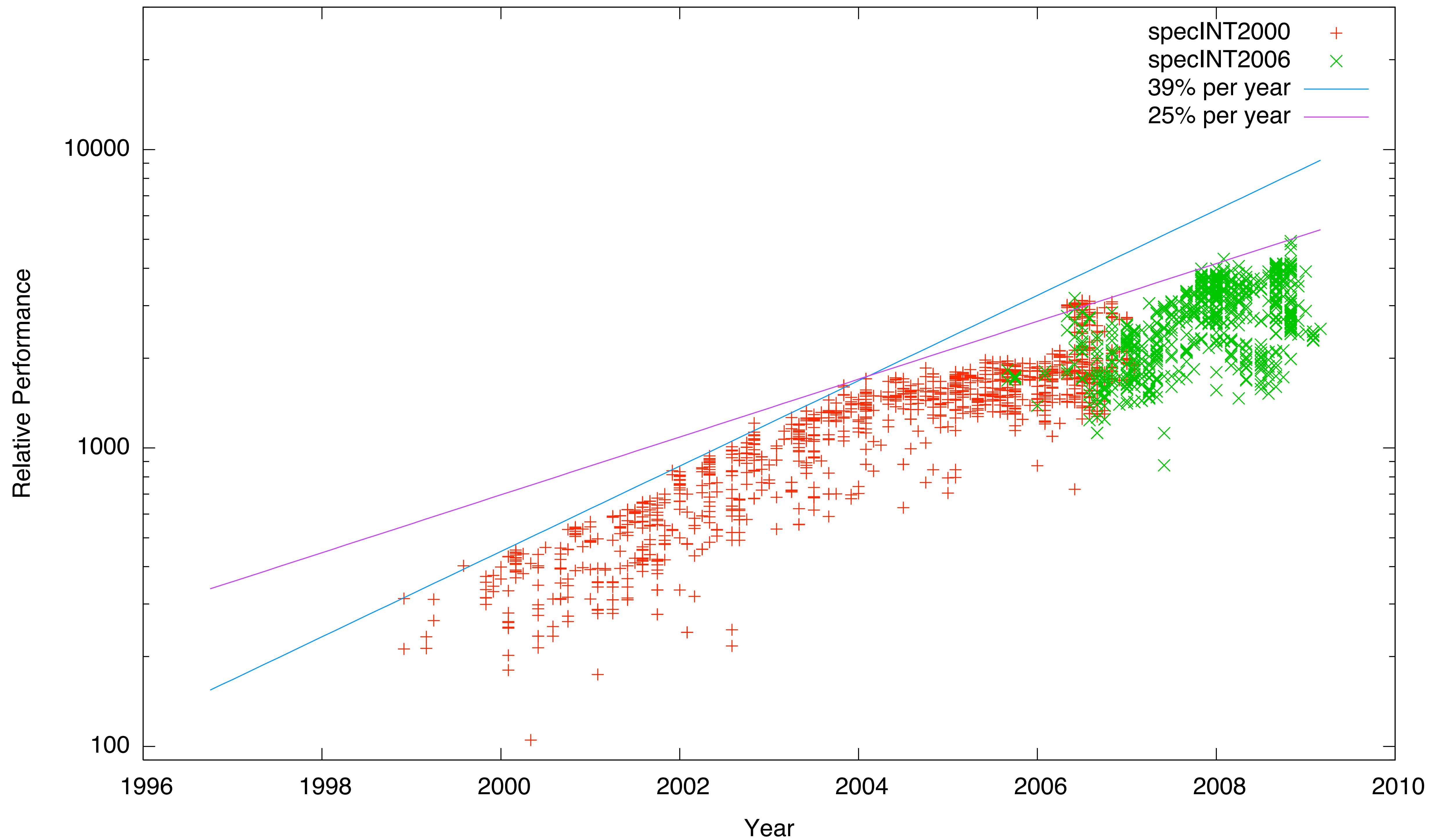
Cell BE
8 + 1 cores



AMD Barcelona
4 cores

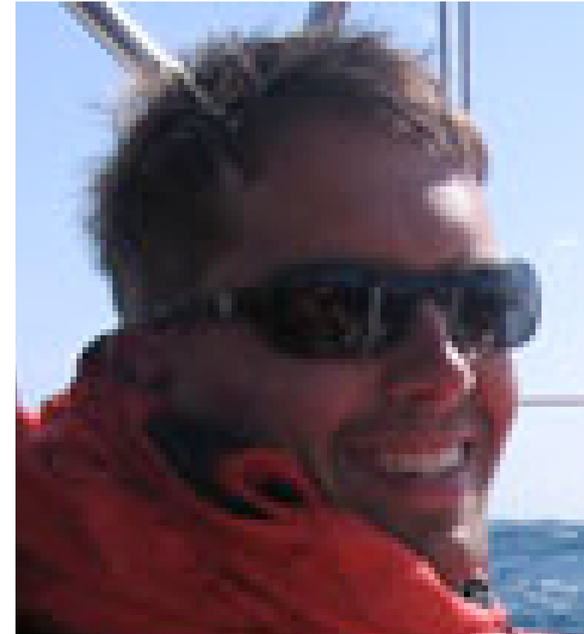
Computer Performance

Computer Performance



Course Staff

- Instructor: Steven Swanson
  - Lectures Tues + Thurs
- TA: Hung-Wei Tseng
  - Discussion sec: Wed.
  - (but not this week)
- See the course web page for contact information and office hours.



What's in this Class

- Course outline
  - Instruction sets
  - The basics of silicon technology
  - Measuring performance
  - How processors work
    - Basic pipelining
    - Data and control hazards
    - Branch prediction and speculation
  - The memory system
  - Introduction to multiprocessors
- Weekly technology digressions
  - How various technologies actually work.

Your Tasks

- Read the text!
  - Computer Organization and Design: The Hardware/Software Interface (4th Edition) -- previous editions are not supported
  - I'm not going to cover everything in class, but you are responsible for all the assigned text.
- Come to class!
  - I will cover things not in the book. You are responsible for that too.
  - Class participation (5%)
- Homeworks throughout the course. (10%)
- Weekly quizzes on Thursdays (10%)
- One midterm. (25%)
- One cumulative final. (35%)
- One project (15%)
  - Design your own ISA!

The Link to 141L

- You do not need to take 141L along with 141, but you may need both to get your degree.
- The classes are mostly independent, except
  - The results of the project will be used in 141L
  - You can earn extra credit by licensing your ISA groups in 141L who are not in 141

Grading

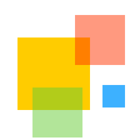
- Grading is on a 13 point scale -- F through A+
  - You will get a letter grade on each assignment
  - Your final grade is the weighted average of the assignment grades.
- An excel spreadsheet calculates your grades
  - We will post a sanitized version online once a week.
  - It will tell you exactly where you stand.
  - It specifies the curves used for each assignment etc.
- OpenOffice doesn't run it properly.

Academic Honesty

- Don't cheat.
  - Cheating on a test will get you an F in the class and no option to drop, and a visit with your college dean.
  - Cheating on homeworks means you don't have to turn them in any more, but you don't get points either. You will also take at least 25% penalty on the exam grades.
- Copying solutions of the internet or a solutions manual is cheating.
- Review the UCSD student handbook
- When in doubt, ask. Honest mistakes will be forgiven.

CpE 252
Computer Organization & Design

Central Processing Unit

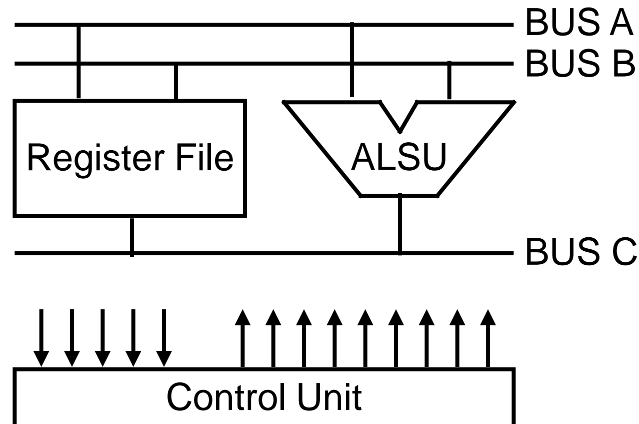


Central Processing Unit (CPU)

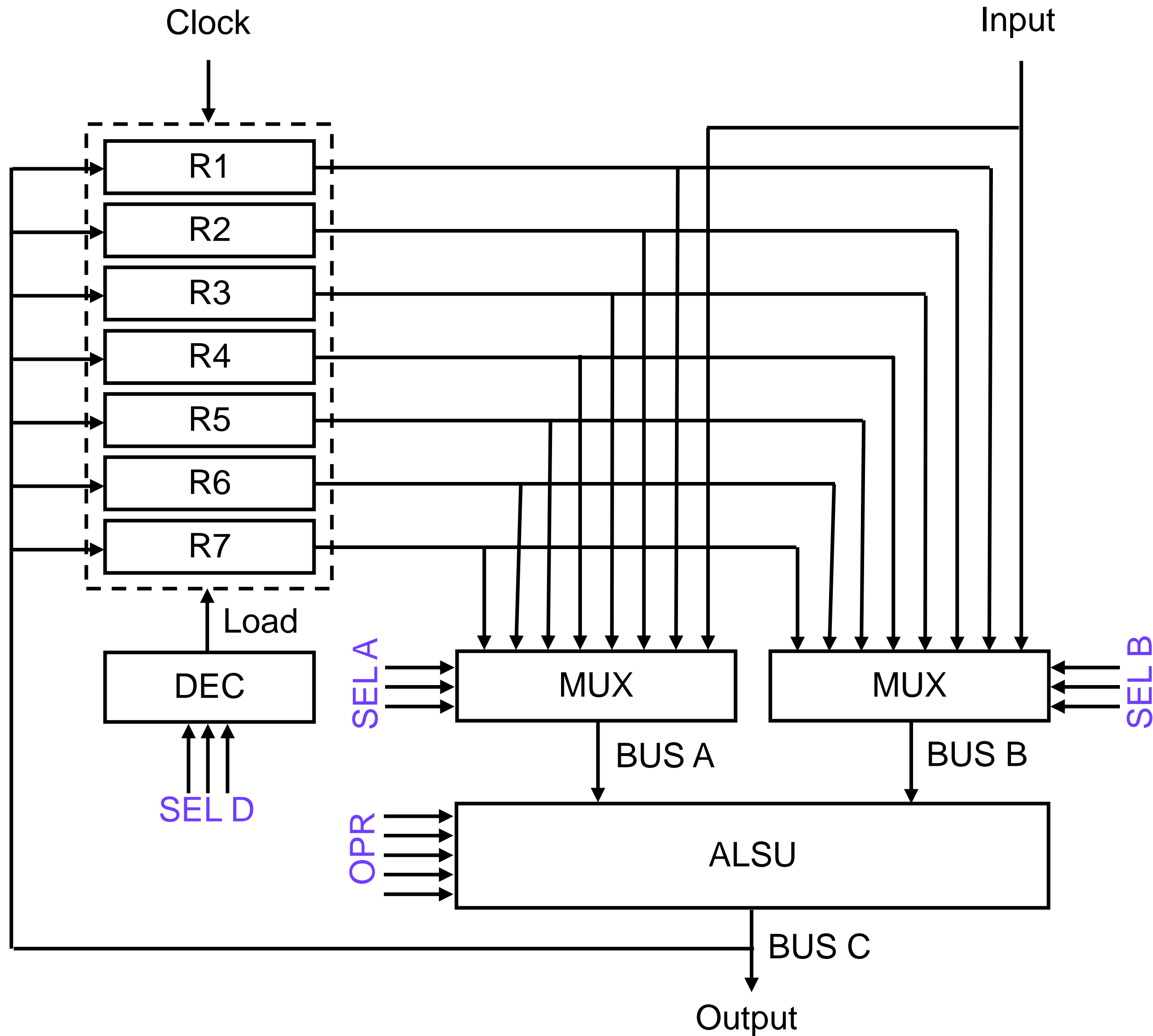
- Introduction
- General Register Organization
- Stack Organization
- Instruction Formats
- Addressing Modes
- Data Transfer and Manipulation
- Program Control
- Reduced Instruction Set Computer

CPU: Major Components

- Datapath
  - Storage Components; Registers & Flags
  - Processing Components; Arithmetic, Logic, Shift Unit (ALSU)
  - Transfer Components; Bus
- Control
  - Control Unit



General Register Organization - Datapath



Operation of Control Unit

- Control Unit, directs the information flow through ALU by
  - Selecting various Components in the system
  - Selecting the Function of ALU
- Example: R1 ← R2 + R3
  - SELA: BUS A ← R2
  - SELB: BUS B ← R3
  - OPR: ALU instruction to ADD
  - SELD: R1 ← BUS C

Control Word

Field Encoding

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

ALU Control

Encoding of ALU operations

OPR	Operation	Outcome	Symbol
00000	Transfer	A	TSFA
00001	Increment	A+1	INCA
00010	Add	A + B	ADD
00101	Subtract	A – B	SUB
00110	Decrement	A – 1	DECA
01000	And	$A \wedge B$	AND
01010	Or	$A \vee B$	OR
01100	Xor	$X \oplus B$	XOR
01110	Complement	A’	COMA
10000	Shift Right		SHRA
10001	Shift Left		SHLA

ALU Microoperations

- Example of ALU Microoperation using the 3-address format
- Unary operation like Increment Register needs a source and destination (can be the same, too)

ALU Microoperations: Example

Microoperation	SELA	SELB	SELD	OPR	Control Word
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \wedge R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	-	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	-	R7	TSFA	001 000 111 00000
Output $\leftarrow$ R2	R2	-	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	-	None	TSFA	000 000 000 00000
$R4 \leftarrow$ shl R4	R4	-	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

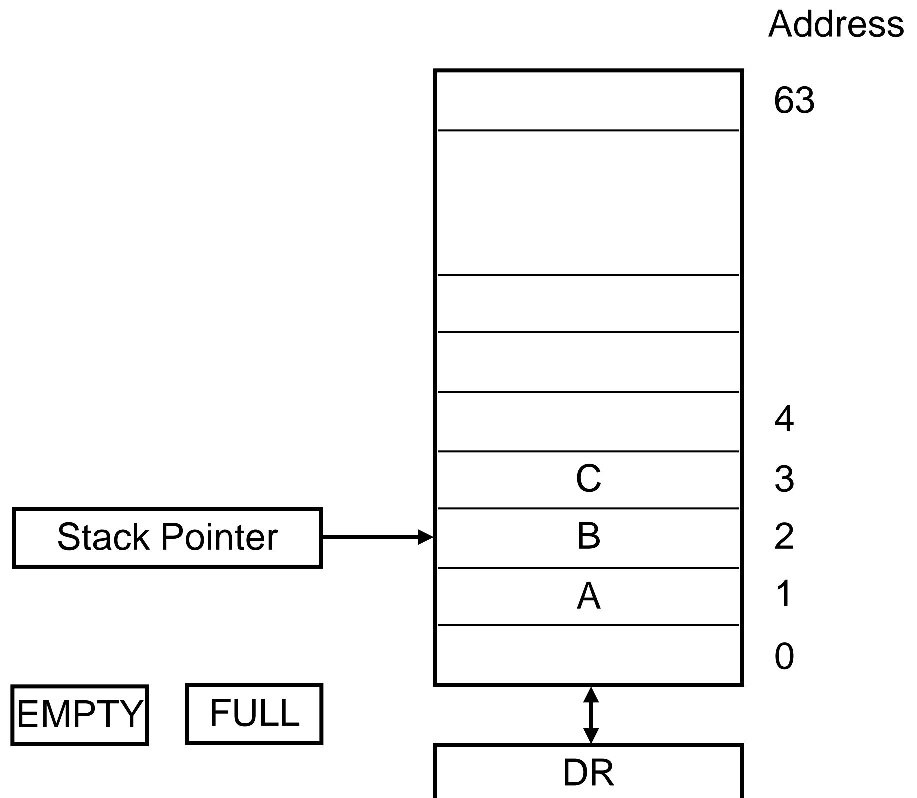
Register Stack Organization

- Useful in nested subroutines and nested loops control
- Efficient for arithmetic expression evaluation
- LIFO; only PUSH and POP operations are applicable

Initially, SP = 0, EMPTY = 1, FULL = 0

PUSH:
SP ← SP + 1
M[SP] ← DR
EMPTY ← 0, SP = 0: FULL ← 1

POP:
DR ← M[SP]
SP ← SP – 1
FULL ← 0, SP = 0: EMPTY ← 1



Memory Stack Organization

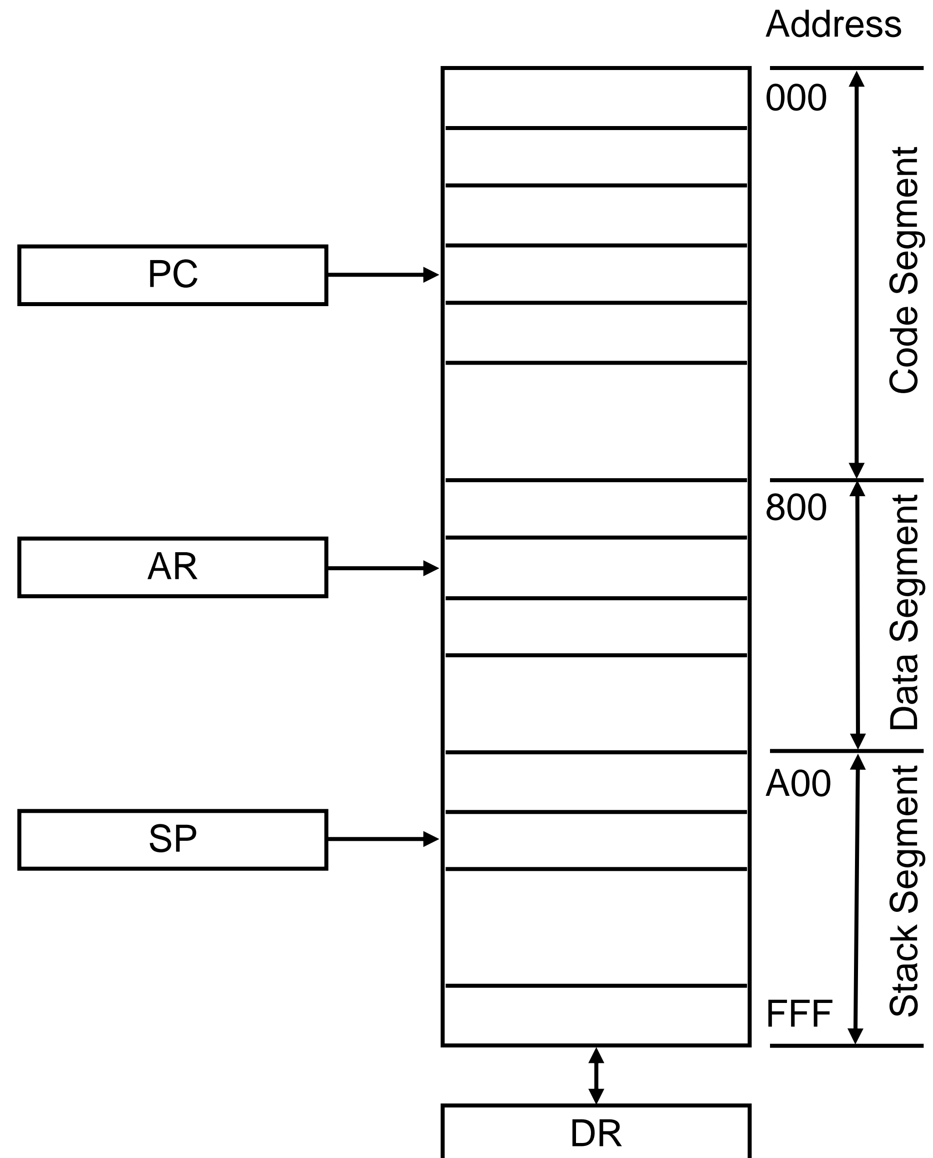
- Memory with Program, Data, and Stack Segments
- Portion of memory used as stack with a register as a stack pointer
- Check overflow & underflow
- Initially, SP is set to the end of memory (FFF)
- Stack overflows if it exceeds some limit
- Operations:

PUSH: $SP \leftarrow SP - 1$

$M[SP] \leftarrow DR$

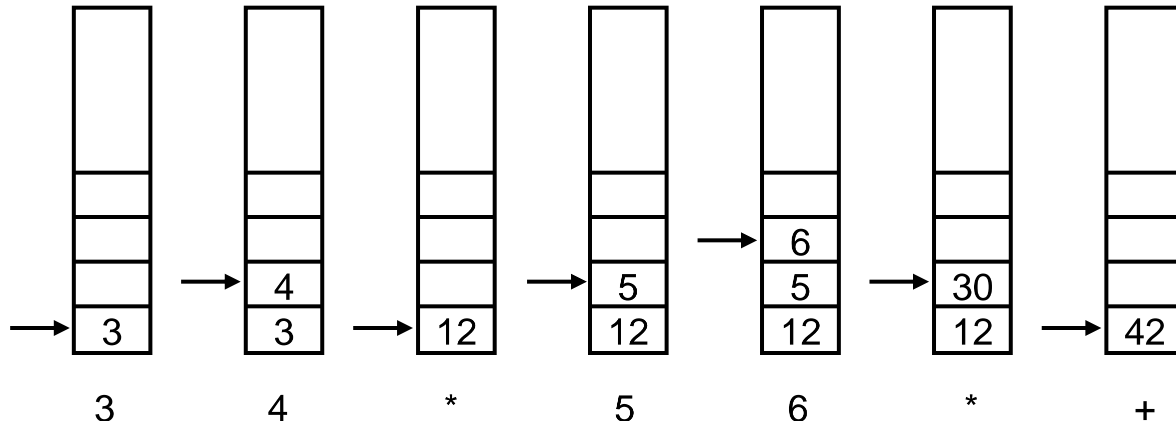
POP: $DR \leftarrow M[SP]$

$SP \leftarrow SP + 1$



Reverse Polish Notation

- Consider the arithmetic expressions: A + B
  - A + B   Infix notation
  - + A B   Prefix or Polish notation
  - A B +   Postfix or Reverse Polish notation
- Reverse Polish Notation is suitable for evaluation using stack
- Any arithmetic expression can be expressed in parenthesis-free
- Example: RPN of expression (3 * 4) + (5 * 6) is 3 4 * 5 6 * +
- Arrow stands for the top of the stack



Instruction Format

- Instruction Fields
  - OP-code; specifies the operation to be performed
  - Address; designates memory addresses or a processor registers
  - Mode; specifies the way operand or effective address is determined
- Number of address fields in the instruction format depends on the internal organization. Most common organizations are:
  - Single accumulator organization:
    - ADD X ; $AC \leftarrow AC + M[X]$
  - Register organization:
    - ADD R1, R2, R3 ; $R1 \leftarrow R2 + R3$
    - ADD R1, R2 ; $R1 \leftarrow R1 + R2$
    - ADD R1, X ; $R1 \leftarrow R1 + M[X]$
  - Stack organization:
    - PUSH X ; $TOS \leftarrow M[X]$
    - ADD

0-address & 1-address instructions

- 0-address, used in a stack computers; evaluate X = (A + B) * (C + D) :

```
PUSH    A     ; TOS ← A
PUSH    B     ; TOS ← B
ADD           ; TOS ← (A + B)
PUSH    C     ; TOS ← C
PUSH    D     ; TOS ← D
ADD           ; TOS ← (C + D)
MUL           ; TOS ← (C + D) * (A + B)
POP     X     ; M[X] ← TOS
```

- 1-address, implies AC for manipulation; evaluate X = (A + B) * (C + D) :

```
LOAD    A     ; AC ← M[A]
ADD     B     ; AC ← AC + M[B]
STORE   T     ; M[T] ← AC
LOAD    C     ; AC ← M[C]
ADD     D     ; AC ← AC + M[D]
MUL     T     ; AC ← AC * M[T]
STORE   X     ; M[X] ← AC
```

2-address & 3-address Instruction

- 2-address, evaluate X = (A + B) * (C + D)

```
MOV    R1, A        ; R1 ← M[A]
ADD    R1, B        ; R1 ← R1 + M[B]
MOV    R2, C        ; R2 ← M[C]
ADD    R2, D        ; R2 ← R2 + M[D]
MUL    R1, R2       ; R1 ← R1 * R2
MOV    X, R1        ; M[X] ← R1
```

- 3-address, evaluate X = (A + B) * (C + D)

```
ADD    R1, A, B     ; R1 ← M[A] + M[B]
ADD    R2, C, D     ; R2 ← M[C] + M[D]
MUL    X, R1, R2    ; M[X] ← R1 * R2
```

- Compared to the 2-address instructions, 3-address results in short programs but instruction becomes long (many bits)

Addressing Modes

- Techniques, methods or ways by which the instruction accesses its operand
- Specifies a rule for interpreting or modifying the address field of the instruction (before the operand is actually referenced)
- Many addressing modes for flexibility and efficient use of bits
- Types are:
  - Implied
  - Immediate
  - Absolute (Direct Address)
  - Register Direct
  - Register Indirect
  - Register Indirect with Autoincrement and Autodecrement
  - Relative:
    - PC relative
    - Index relative
    - Base relative

Addressing Modes

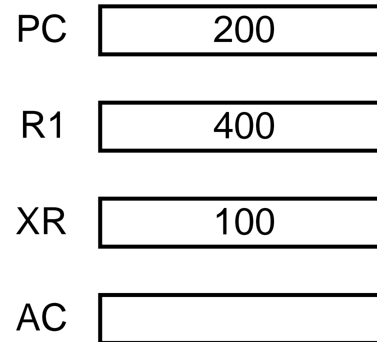
- Implied Mode
  - Address of operand specified implicitly in the instruction
    - No need to specify address in the instruction
    - Example: effective address of CMA is AC and that of POP is stack pointer
- Immediate Mode
  - Instead of specifying the address, operand itself is specified
    - No need to specify address, operand itself needs to be specified
    - Sometimes, require more bits than the address
    - Fast to acquire an operand
    - Example: LD #129, R1, source is immediate
- Register Mode
  - Address specified in the instruction is the register address
    - Designated operand need to be in a register
    - Shorter address than the memory address
    - Saving address field in the instruction
    - Faster to acquire an operand than the memory addressing
    - Example: MOV R1, R2, source and destinations are register direct

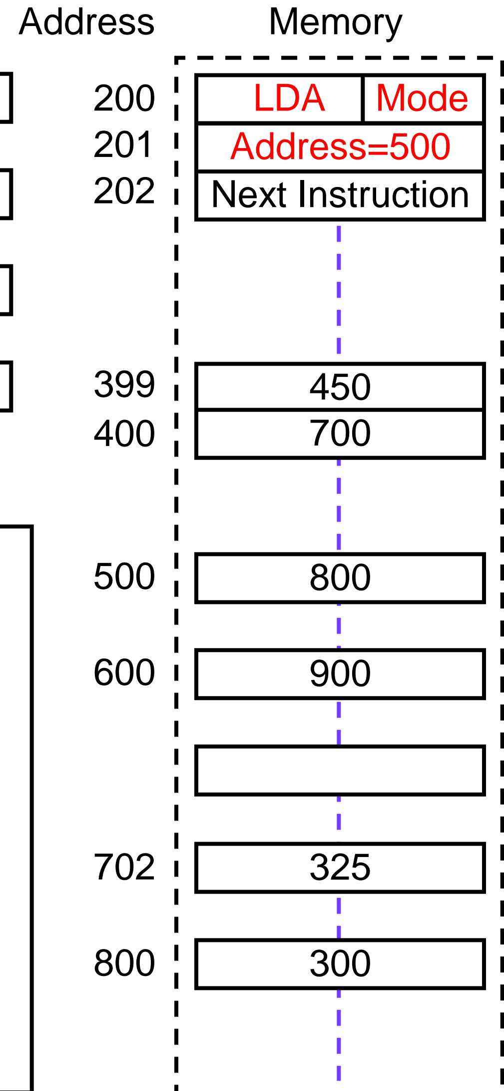
Addressing Modes

- Register Indirect Mode
  - Instruction specifies a register containing address of operand
    - Saving instruction bits since register address is shorter than memory address
    - Slower to acquire an operand than both register and memory addressing
    - Example: MOV R1, (R4), destination is memory whose address is in R4
- Register Indirect with Autoincrement/Autodecrement
  - Register based addressing is automatically adjusted by incrementing or decrementing [some processor restrict the use to ()+ and –()]
    - Automatically adjust the pointers by adding proper offset
    - Example: ST R1, (R4)+
- Absolute or Direct Address Mode
  - Instruction specifies directly the memory address of the operand
    - Faster than the other memory addressing modes
    - Too many bits are needed to specify the address for a large memory space
    - Example: ST #$12, 124400, destination is operand at address 124400

- Indirect Addressing Mode
  - Address field specifies address to address of operand in memory
    - Abbreviated address is used to address operand using small number of bits
    - Slow to acquire an operand because of an additional memory access
    - Example: ST R1, (124400), destination address is at address 124400
- Relative Addressing Modes
  - Address fields of an instruction specifies part of the address which can be used along with a designated register to calculate operand address
    - Address field of the instruction is short
    - Large physical memory can be accessed with a small number of address bits
    - Example: Operand is at address formed modifying a special purpose register
  - 3 different Relative Addressing Modes depending on register:
    - PC Relative, effective address is PC + offset, like BRA Loop
    - Indexed
    - Base Register

Addressing Modes: Example





Load Accumulator using variety of Modes

Addressing Modes Example: LDA [Mode] 500

Mode	EA	Operation	AC content
Direct	500	; AC ← (500)	800
Immediate	-	; AC ← 500	500
Indirect	800	; AC ← ((500))	300
Relative	702	; AC ← (PC+500)	325
Indexed	600	; AC ← (XR+500)	900
Register	-	; AC ← R1	400
Register Indirect	400	; AC ← (R1)	700
Autoincrement	400	; AC ← (R1)+	700
Autodecrement	399	; AC ← -(R)	450

Instruction Types: Data Transfer

- Highest frequency
- Transfer data between registers and memory or input-output devices

Typical Data Transfer Instructions

Mnemonic	Name
LD	Load
ST	Store
MOV	Move
EX	Exchange
EXX	Exchange All
SWP	Swap
IN	Input
OUT	Output
PUSH	Push
POP	Pop

Addressing Modes for Data Transfer

Data Transfer Instructions with Different Addressing Modes

Addressing Mode	Assembler Convention	Register Transfer
Direct address	LD ADR	AC ← M[ADR]
Indirect address	LD @ADR	AC ← M[M[ADR]]
Relative address	LD $ADR	AC ← M[PC + ADR]
Immediate operand	LD #NBR	AC ← NBR
Index addressing	LD ADR(X)	AC ← M[XR+ADR]
Register	LD R1	AC ← R1
Register indirect	LD (R1)	AC ← M[R1]
Autoincrement	LD (R1)+	AC ← M[R1], R1 ← R1 + 1
Autodecrement	LD -(R1)	R1 ← R1 - 1, AC ← M[R1]

Instruction Types: Arithmetic Operations

- Some processors include FP and/or BCD arithmetic

Arithmetic Instructions

Mnemonic	Name
INC	Increment
DEC	Decrement
ADD	Add
SUB	Subtract
MUL	Multiply
DIV	Divide
ADDC	Add with Carry
SUBB	Subtract with Borrow
NEG	Negate (2’s Complement)

Instruction Types: Logical and Bit Operations

Logical and Bit Manipulation Instructions

Mnemonic	Name	Mnemonic
CLR	Clear	
COM	Complement	
AND	AND	
OR	OR	
XOR	Exclusive-OR	
CLRC	Clear carry	CLRC
SETC	Set Carry	
COMC	Complement Carry	
EI	Enable Interrupt	
DI	Disable Interrupt	

Instruction Types: Shift Operations

Shift Instructions

Mnemonic	Name Mnemonic
SHR	Logical Shift Right
SHL	Logical Shift Left
SHRA	Shift Right Arithmetic
SHLA	Shift Left Arithmetic
ROR	Rotate Right
ROL	Rotate Left
RORC	Rotate Right thru Carry
ROLC	Rotate Left thru Carry

Instruction Types: Program Control

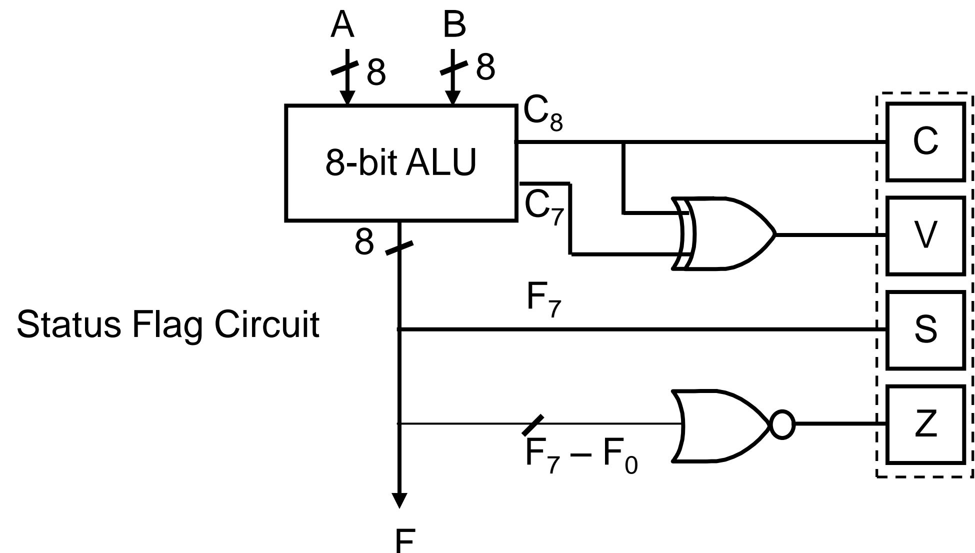
- PC is updated in two major ways;
  - By incrementing
    - Fetch from the fall through path
    - Skip the next instruction to the next
  - By loading
    - Jump to an address
    - Branch to an address
    - Call subroutine
    - Return to the calling part of the program

Program Control Instructions

Mnemonic	Name
BR	Branch
JMP	Jump
SKP	Skip
CALL	Call Subroutine
RTN	Return
CMP	Compare (using SUB)
TST	Test (using AND)

Condition Codes

- The recent state of the machine is expressed in flip-flops collectively called:
  - Flags
  - Status register
  - Condition Code Register
- The S (sometimes called N) for Sign or Negative, it reflects the sign of the outcome, that is why it is a copy of the MSB of the ALU
- The Z (Zero) flag is set when all the bits of the result are 0's
- The C (Carry) and V (oVerflow) flags are meant to reflect the carry for the unsigned and signed



Conditional Branch Instructions

Simple Compare Condition

Branch Condition	Tested condition	Mnemonic
Branch if Zero	Z = 1	BZ
Branch if Not Zero	Z = 0	BNZ
Branch if Carry	C = 1	BC
Branch if No Carry	C = 0	BNC
Branch if Plus	S = 0	BP
Branch if Minus	S = 1	BM
Branch if oVerflow	V = 1	BV
Branch if No oVerflow	V = 0	BNV

Arithmetic Compare Conditions

Unsigned Compare conditions (A - B)

Branch Condition	Tested	Condition	Mnemonic
Branch if Higher	$A > B$	$C' . Z'$	BHI
Branch if Higher or Equal	$A \geq B$	C'	BHE or BNC
Branch if Lower	$A < B$	C	BLO or BC
Branch if Lower or Equal	$A \leq B$	$C + Z$	BLOE

Signed Compare Conditions (A - B)

Branch Condition	Tested	Condition	Mnemonic
Branch if Greater Than	$A > B$	$(N \oplus V)' . Z'$	BGT
Branch if Greater or Equal	$A \geq B$	$(N \oplus V)'$	BGE
Branch if Less Than	$A < B$	$(N \oplus V)$	BLT
Branch if Less or Equal	$A \leq B$	$(N \oplus V) + Z$	BLE

Signed & Unsigned Compare conditions (A - B)

Branch Condition	Tested	Condition	Mnemonic
Branch if Equal	$A = B$	Z	BEQ or BZ
Branch if Not Equal	$A \neq B$	Z'	BNE or BNZ

Subroutine Call & Return

- Subroutine calls have flavors: Call, Jump and Branch
- Two Most Important Operations are Implied:
  - Save Return Address (current value of PC) for proper operation
    - Locations for storing Return Address
      - Fixed Location in the subroutine area
      - Fixed Location in memory
      - Special register within the processor
      - Memory stack, which is the most efficient way
  - Branch to the beginning of the subroutine by placing effective address into the PC
- Stack Based Microoperations:
  - CALL $SP \leftarrow SP - 1$
    $M[SP] \leftarrow PC$
    $PC \leftarrow EA$
  - RTN $PC \leftarrow M[SP]$
    $SP \leftarrow SP + 1$

Program Interrupt - Types

- External interrupts; initiated from external devices
  - Input-Output Device; data transfer start & stop
  - Timing Device
  - Power Failure
  - Operator; a pushbutton
- Internal interrupts (traps); caused by a running program
  - Register Check
  - Stack Overflow
  - Divide by zero
  - OP-code Violation (illegal instruction)
  - Protection Violation
- Software Interrupts; initiated by executing an instruction
  - Supervisor Call; to switch from the user mode to the supervisor mode

Interrupt Procedure

- Interrupts are two types:
  - Hardware, or external interrupt, usually initiated by an external event
  - Software, or internal interrupt, due to instruction execution
- The address of the interrupt service program is determined by:
  - Hardware, requesting device send a vector
  - Software, fixed address for each type
- An interrupt procedure usually stores all the information necessary to define the state of processor rather than storing only the PC
  - The state of the processor is determined from;
    - Content of the PC
    - Content of all processor registers
    - Content of status bits
  - Saving the state of processor depends on the architecture

- The instruction set is an important aspect of computer design
- The instruction set determines how the machine language programs are constructed
- Early 80's, stuck by marginal improvement in performance through technology, directed the effort towards the organization instead
- Ideas for improving the organization included:
  - Adding Registers, to localize the variables for high speed access
  - Adding Caches, to keep data and instructions handy
  - Adding Functional units, to overlap executions
- This would have increased the performance, but not with the limited integration level of 10^5 transistors/chip of that time
- Conventionally, control units used to eat up good deal of that, nearly 50%, but could reach 70% in some implementations
- The idea was to make the control unit less complex, taking only a fraction of that; say 5%

Major Trends

- Trends were then called:
  - Reduced Instruction Set Computers (RISC) &
  - Complex Instruction Set Computers (CISC)
- Both have good reasons to stay
- Differences are becoming less and less, as the number of transistors per package is no longer an issue;
  - Registers are in abundance in both
  - Cache is now in abundance in both and implemented in many levels
  - Functional units are also many in both
- To stay backwardly compatible, a new trend is now in existence; RISC core with CISC shell, P4 is just like that

- Instruction set size < 100
- Addressing modes < 4
- Control unit logic < 10%
- Memory access is restricted to Load/Store instructions
- Fixed length instructions
- Single cycle execution
- Most of the instructions are used
- Hardwired control unit
- Highly complier dependent for efficient code
- Because of less complex control unit, it has:
  - Large number of registers > 128
  - Larger caches
  - More functional units

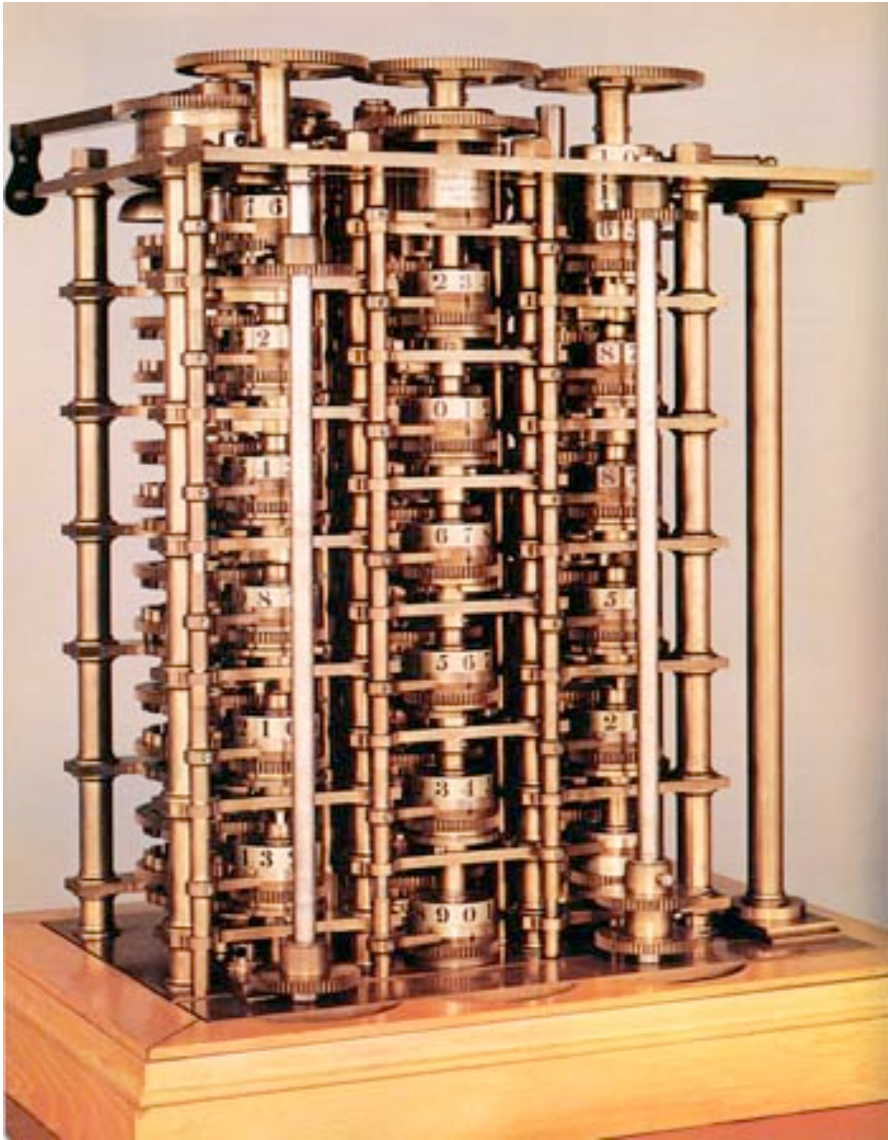
- Instructions set size, > 200
- Addressing modes > 8
- Control unit logic > 50%
- Memory access is allowed for data manipulation instructions
- Variable length instructions
- Multiple cycle execution
- Some instructions are rarely used
- Microprogrammed control unit
- Compiler dependence is not as much
- Because of more complex control unit, it has:
  - Small number of registers < 32
  - Smaller caches
  - Less functional units

Instruction Set Architectures: Talking to the Machine

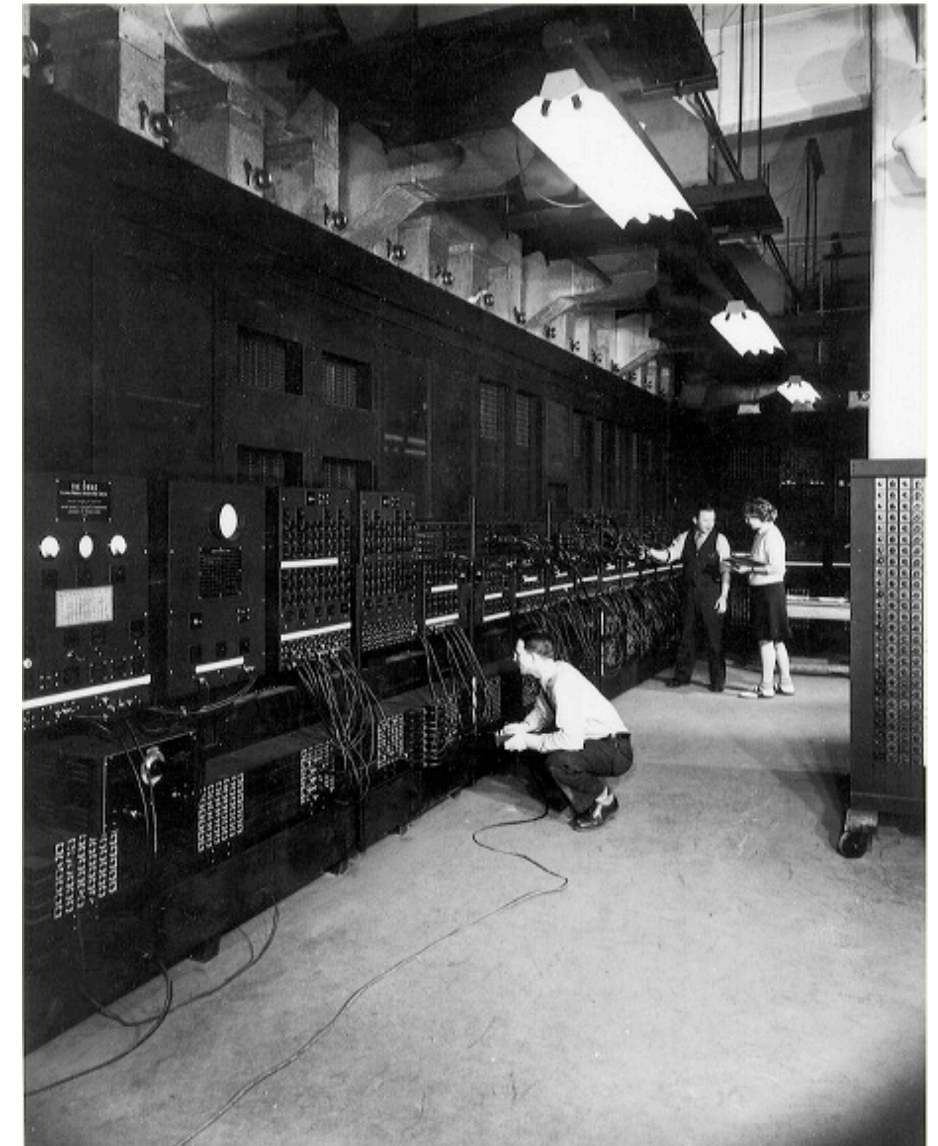
The Architecture Question

- How do we build computer from contemporary silicon device technology that executes general-purpose programs quickly, efficiently, and at reasonable cost?
- i.e. How do we build the computer on your desk.

In the beginning...



The Difference Engine



ENIAC

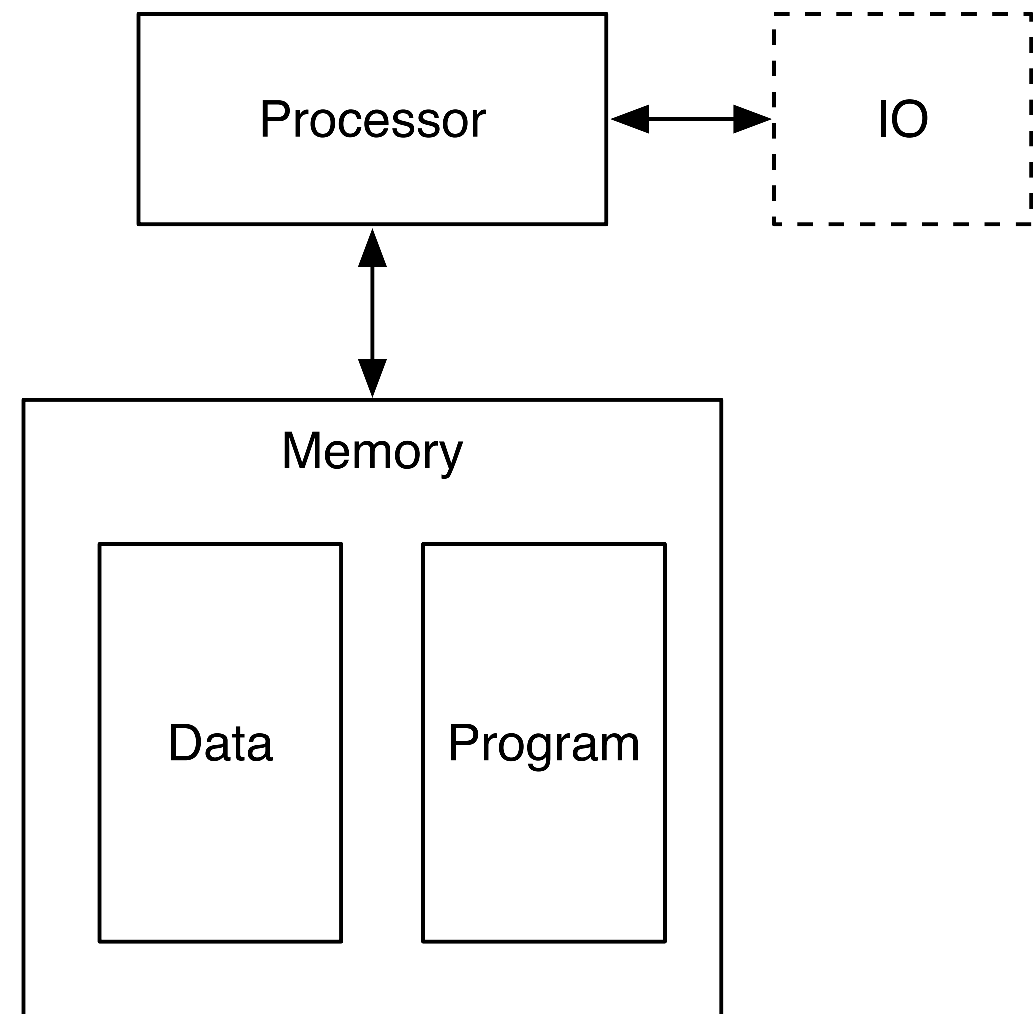
- Physical configuration specifies the computation

The Stored Program Computer

- The program is data
  - i.e., it is a sequence of *numbers* that machine interprets
- A very elegant idea
  - The same technologies can store and manipulate programs and data
  - Programs can manipulate programs.

The Stored Program Computer

- A very simple model
- Several questions
  - How are program represented?
  - How do we get algorithms out of our brains and into that representation?
  - How does the the computer interpret a program?

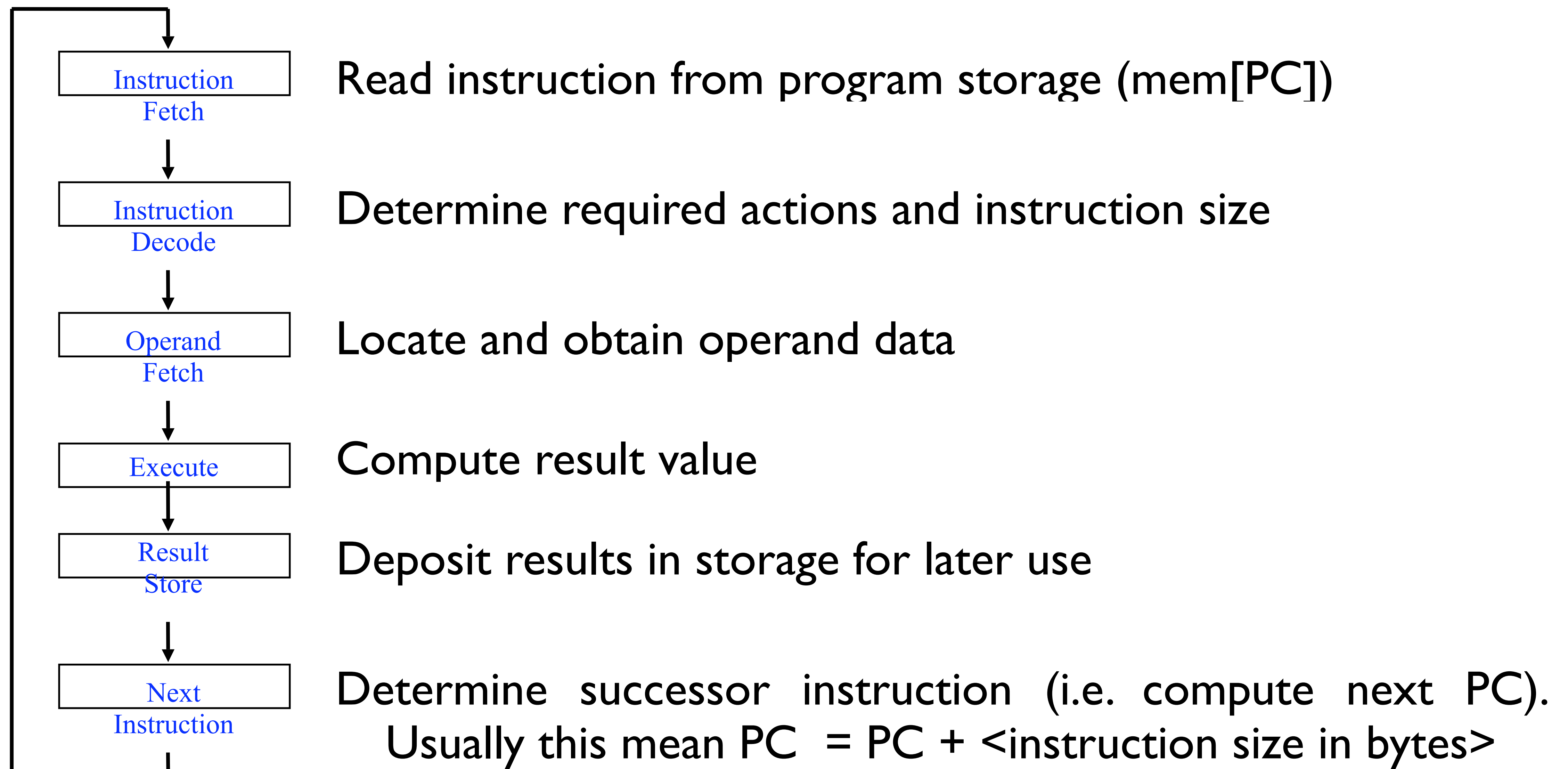


Representing Programs

- We need some basic building blocks -- call them "instructions"
- What does "execute a program" mean?
- What instructions do we need?
- What should instructions look like?
- Is it enough to just specify the instructions?
- How complex should an instruction be?

Program Execution

- This is the algorithm for a stored-program computer
  - The Program Counter (PC) is the key



Motivating Code segments

- a = b + c;
- a = b + c + d;
- a = b & c;
- a = b + 4;
- a = b - (c * (d/2) - 4);
- if (a) b = c;
- if (a == 4) b = c;
- while (a != 0) a--;
- a = 0xDEADBEEF;
- a = foo[4];
- foo[4] = a;
- a = foo.bar;
- a = a + b + c + d +... +z;
- a = foo(b); -- next class

What instructions do we need?

- Basic operations are a good choice.
  - Motivated by the programs people write.
  - Math: Add, subtract, multiply, bit-wise operations
  - Control: branches, jumps, and function calls.
  - Data access: Load and store.
- The exact set of operations depends on many, many things
  - Application domain, hardware trade-offs, performance, power, complexity requirements.
  - You will see these trade-offs first hand in the ISA project and in 141L.

What should instructions look like?

- They will be numbers -- i.e., strings of bits
- It is easiest if they are all the same size, say 32 bits
  - We can break up these bits into “fields” -- like members in a class or struct.
- This sets some limits
  - On the number of different instructions we can have
  - On the range of values any field of the instruction can specify

Is specifying the instructions sufficient?

- No! We also must what the instructions operate on.
- This is called the "Architectural State" of the machine.
  - Registers -- a few named data values that instructions can operate on
  - Memory -- a much larger array of bytes that is available for storing values.
  - How big is memory? 32 bits or 64 bits of addressing.
    - 64 is the standard today for desktops and larger.
    - 32 for phones and PDAs
    - Possibly fewer for embedded processors
- We also need to specify semantics of function calls
  - The "Stack Discipline," "Calling convention," or "Application binary interface (ABI)".

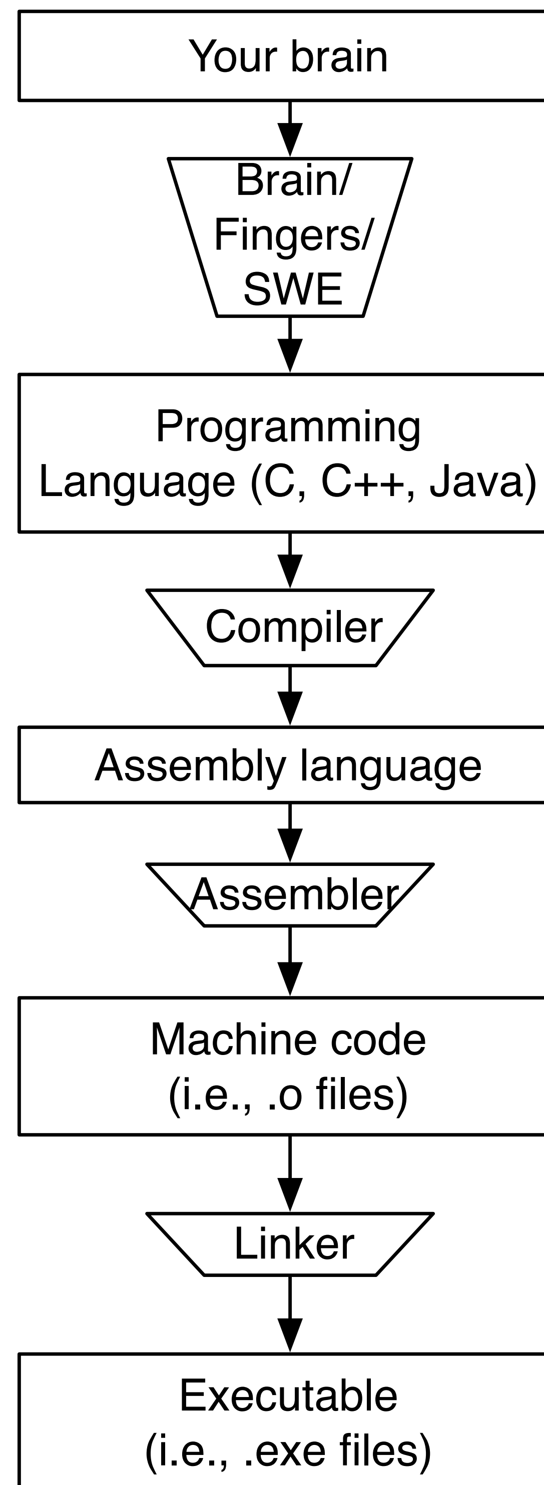
How complex should instructions be?

- More complexity
  - More different instruction types are required.
  - Increased design and verification costs
  - More complex hardware.
  - More difficult to use -- What's the right instruction in this context?
- Less complexity
  - Programs will require more instructions -- poor *code density*
  - Programs can be more difficult for humans to understand
  - In the limit, decremement-and-branch-if-negative is sufficient
    - Imagine trying to decipher programs written using just one instruction.
    - It takes many, many of these instructions to emulate simple operations.
- Today, what matters most is the compiler
  - *The Machine* must be able to understand program
  - *A program* must be able to decide which instructions to use

Big "A" Architecture

- The Architecture is a contract between the hardware and the software.
  - The hardware defines a set of operations, their semantics, and rules for their use.
  - The software agrees to follow these rules.
  - The hardware can implement those rules *IN ANY WAY IT CHOOSES!*
    - Directly in hardware
    - Via a software layer
    - Via a trained monkey with a pen and paper.
- This is a classic interface -- they are everywhere in computer science.
  - "Interface," "Separation of concerns," "API," "Standard,"
- For your project you are designing an Architecture -- not a processor.

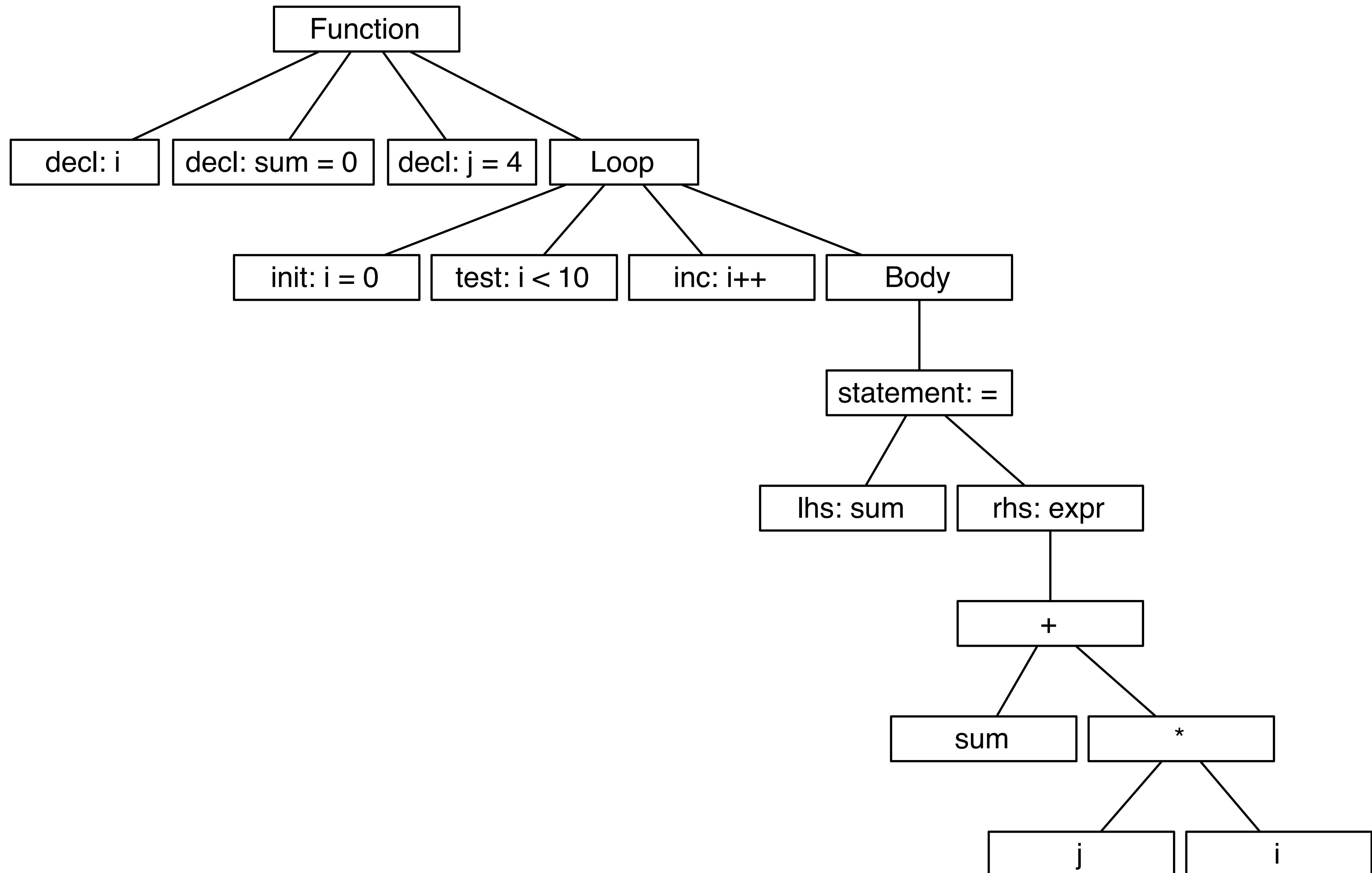
From Brain to Bits



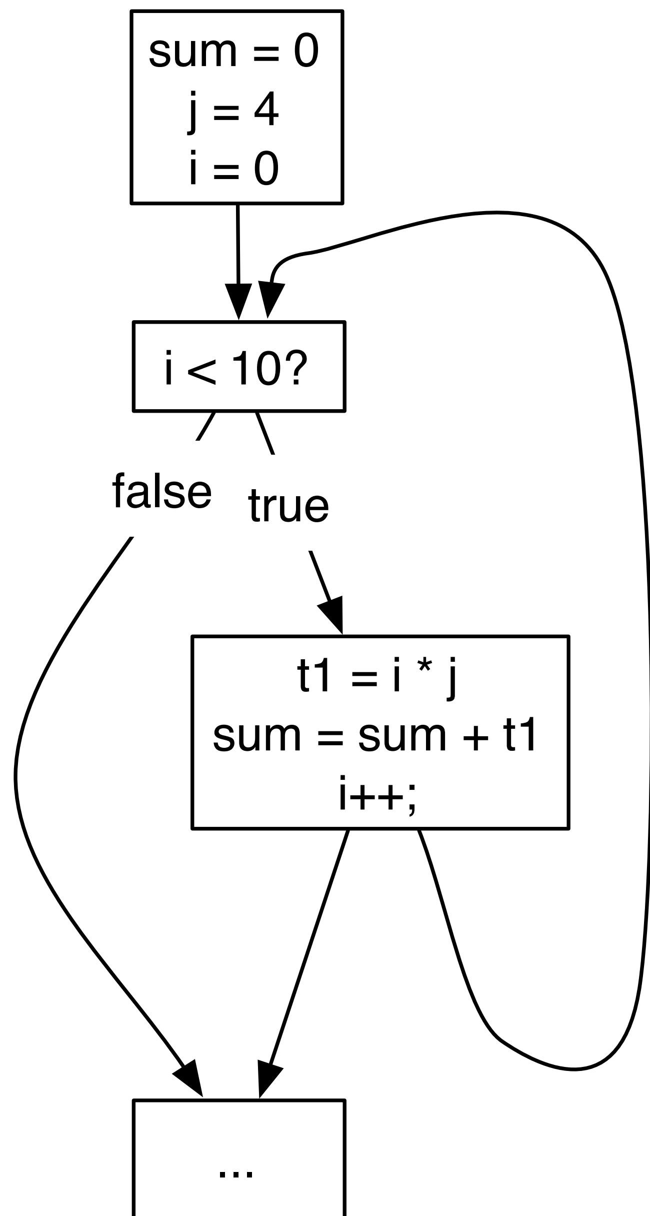
C Code

```
int i;
int sum = 0;
int j = 4;
for(i = 0; i < 10; i++) {
 sum = i * j + sum;
}
```

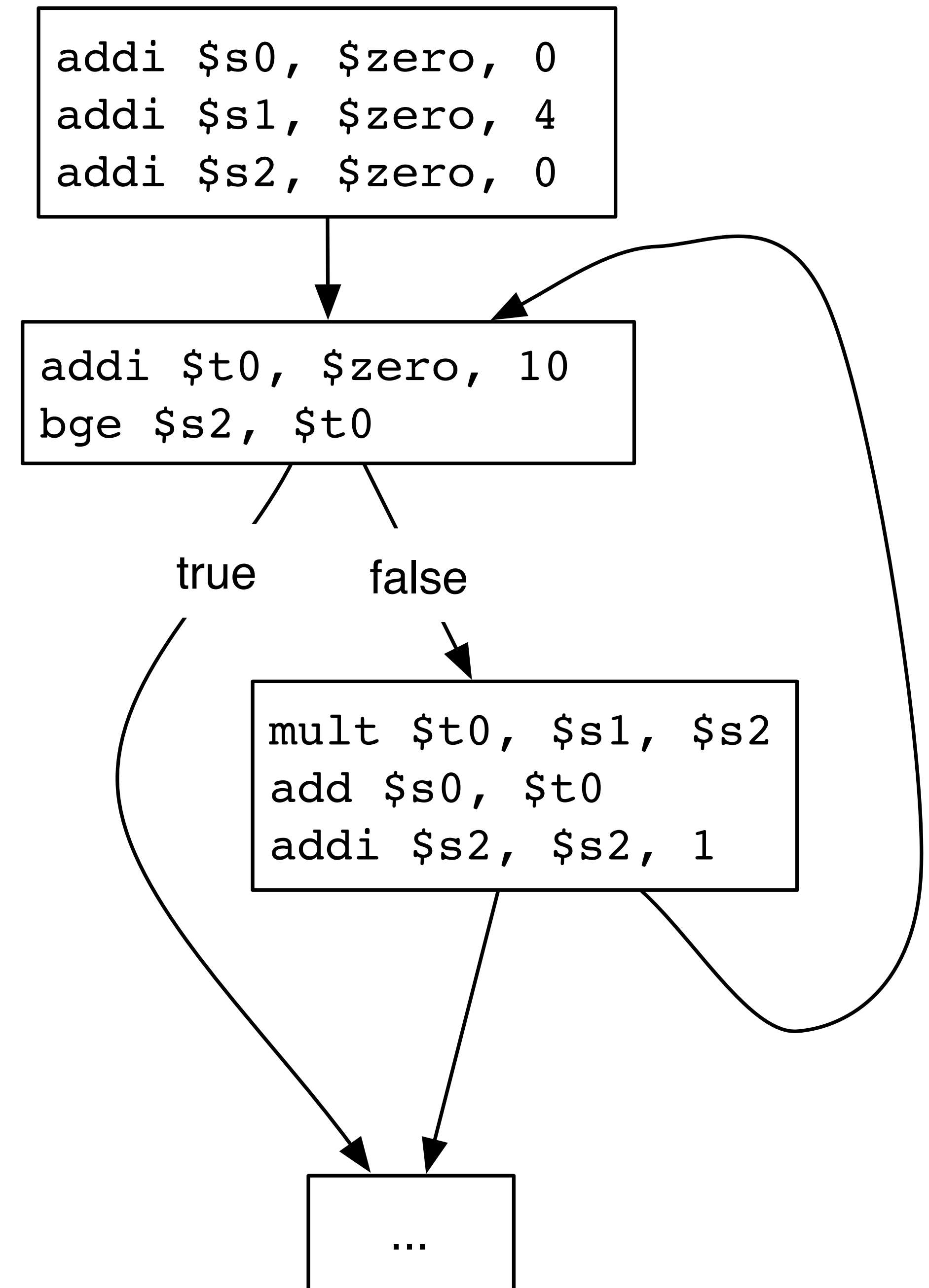
In the Compiler



In the Compiler

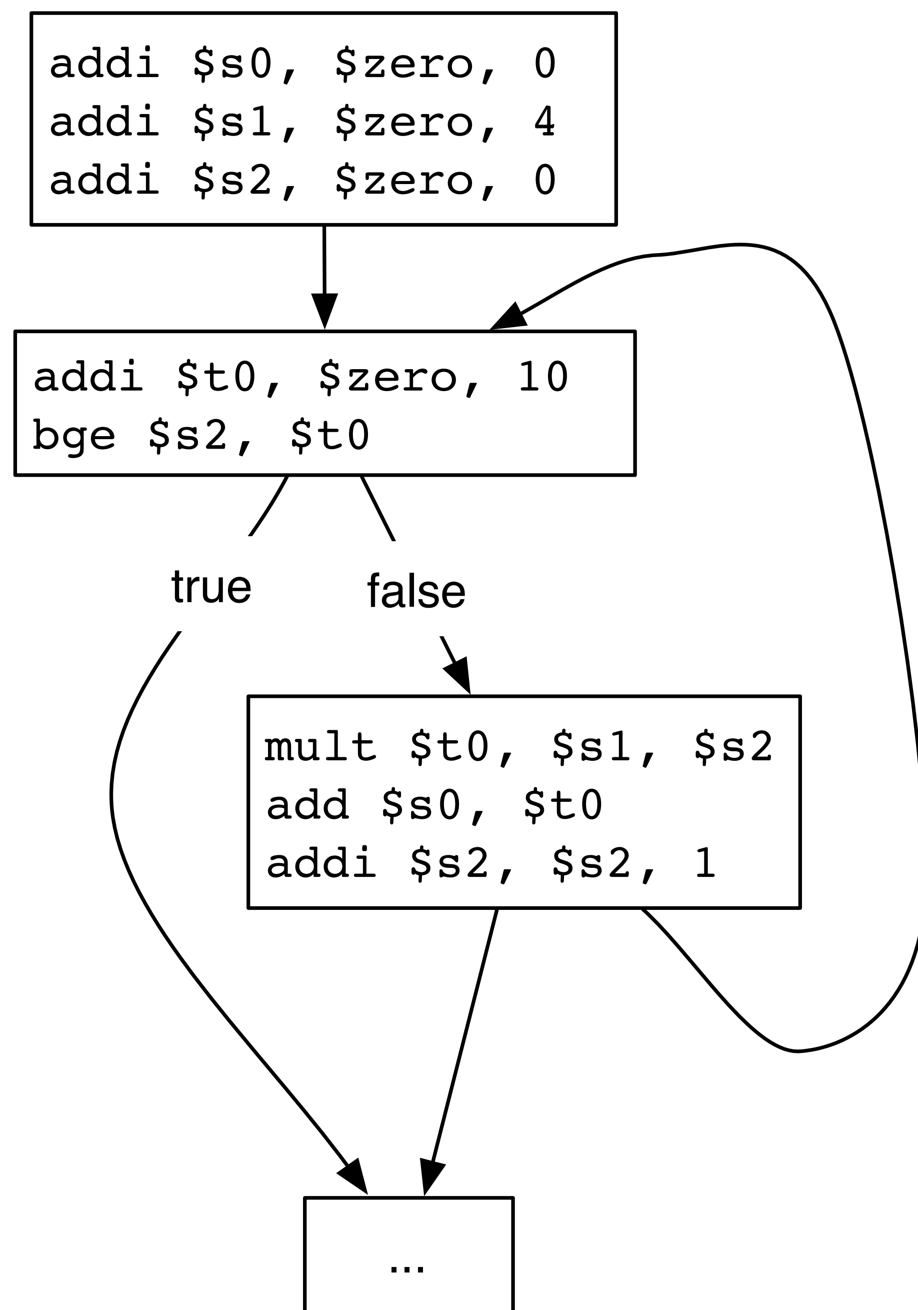


Control flow graph w/high-level instructions



Control flow graph w/real instructions

Out of the Compiler




```
addi $s0, $zero, 0
addi $s1, $zero, 4
addi $s2, $zero, 0

top:
addi $t0, $zero, 10
bge $s2, $t0, after

body:
mult $t0, $s1, $s2
add $s0, $t0
addi $s2, $s2, 1
br   top

after:
...
```

Assembly language

Labels in the Assembler

```
addi $s0, $zero, 0
addi $s1, $zero, 4
addi $s2, $zero, 0

top:
addi $t0, $zero, 10
bge $s2, $t0, after


mult $t0, $s1, $s2
add $s0, $t0
addi $s2, $s2, 1
br   top

after:
...
```

'after' is defined at 0x20
used at 0x10
The value of the immediate for the branch
is 0x20-0x10 = 0x10

'top' is defined at 0x0C
used at 0x1C
The value of the immediate for the branch
is 0x0C-0x1C = 0xFFFF0 (i.e., -0x10)

Labels in the Assembler

```
0x00  addi $s0, $zero, 0
0x04  addi $s1, $zero, 4
0x08  addi $s2, $zero, 0

      top:
0x0C  addi $t0, $zero, 10
0x10  bge $s2, $t0, after


      mult $t0, $s1, $s2
0x14  add $s0, $t0
0x18  addi $s2, $s2, 1
0x1C  br   top

0x20  after:
      ...
```

'after' is defined at 0x20
used at 0x10
The value of the immediate for the branch
is 0x20-0x10 = 0x10

'top' is defined at 0x0C
used at 0x1C
The value of the immediate for the branch
is 0x0C-0x1C = 0xFFFF0 (i.e., -0x10)

Assembly Language

- “Text section”
  - Hold assembly language instructions
  - In practice, there can be many of these.
- “Data section”
  - Contain definitions for static data.
  - It can contain labels as well.
- The addresses in the data section have no relation to the addresses in the data section.
- Pseudo instructions
  - Convenient shorthand for longer instruction sequences.

.data and pseudo instructions

```
void foo() {
  static int a = 0;
   a++;
   ...
}
```

```
.data
foo_a:
  .word 0

.text
foo:
  lda $t0, foo_a
  ld  $s0, 0($t0)
  addi $s0, $s0, 1
  st  $s0, 0($t0)
after:
...
```

.data and pseudo instructions

```
void foo() {
  static int a = 0;
  a++;
  ...
}
```

```
.data
foo_a:
  .word 0

.text
foo:
  lda $t0, foo_a
  ld  $s0, 0($t0)
  addi $s0, $s0, 1
  st  $s0, 0($t0)
after:
...
```

```
lda $t0, foo_a
```

becomes these instructions (this is not assembly language!)

```
andi $t0, $zero, ((foo_a & 0xff00) >> 16)
sll  $t0, $t0, 16
andi $t0, $t0, (foo_a & 0xff)
```

.data and pseudo instructions

```
void foo() {
   static int a = 0;
    a++;
    ...
}
```

```
.data
foo_a:
  .word 0

.text
foo:
  lda $t0, foo_a
  ld  $s0, 0($t0)
  addi $s0, $s0, 1
  st  $s0, 0($t0)
after:
...
```

```
lda $t0, foo_a
```

becomes these instructions (this is not assembly language!)

```
andi $t0, $zero, ((foo_a & 0xff00) >> 16)
sll  $t0, $t0, 16
andi $t0, $t0, (foo_a & 0xff)
```

The assembler computes and inserts these values.

.data and pseudo instructions

```
void foo() {
   static int a = 0;
    a++;
    ...
}
```

```
.data
foo_a:
  .word 0

.text
foo:
  lda $t0, foo_a
  ld  $s0, 0($t0)
  addi $s0, $s0, 1
  st  $s0, 0($t0)
after:
...
```

```
lda $t0, foo_a
```

becomes these instructions (this is not assembly language!)

```
andi $t0, $zero, ((foo_a & 0xff00) >> 16)
sll  $t0, $t0, 16
andi $t0, $t0, (foo_a & 0xff)
```

If `foo` is address 0x0, where is `after`?

The assembler computes and inserts these values.

.data and pseudo instructions

```
void foo() {
   static int a = 0;
    a++;
    ...
}
```

```
     .data
     foo_a:
       .word 0

     .text
     foo:
0x00   lda $t0, foo_a
0x0C   ld  $s0, 0($t0)
0x10   addi $s0, $s0, 1
0x14   st  $s0, 0($t0)
     after:
0x18...
```

`lda $t0, foo_a`

becomes these instructions (this is not assembly language!)

```
andi $t0, $zero, ((foo_a & 0xff00) >> 16)
sll  $t0, $t0, 16
andi $t0, $t0, (foo_a & 0xff)
```

If `foo` is address 0x0, where is `after`?

The assembler computes and inserts these values.

ISA Alternatives

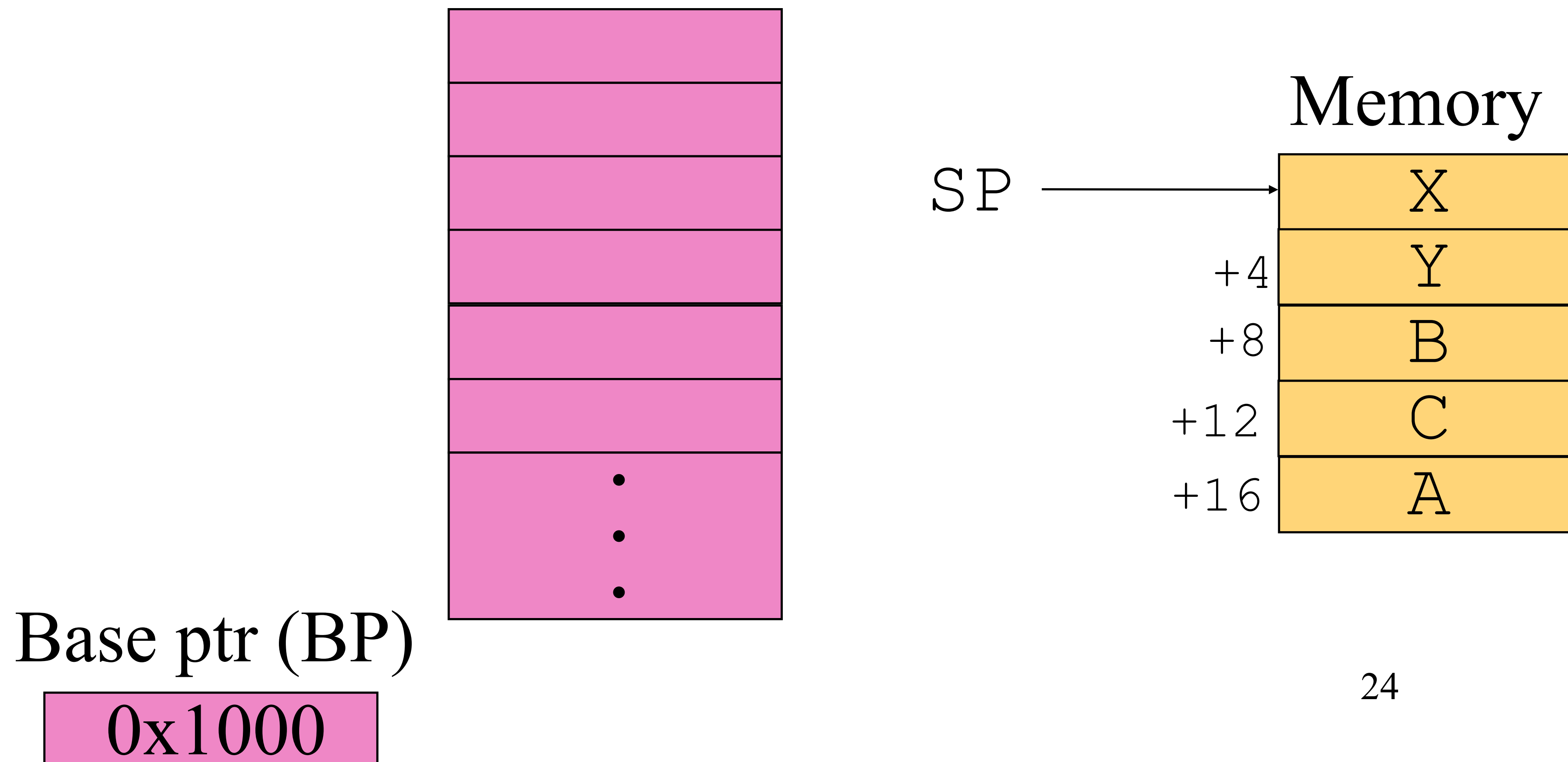
- MIPS is a 3-address, RISC ISA
  - add rs, rt, rd -- 3 operands
  - RISC -- reduced instruction set. Relatively small number of operation. Very regular encoding. RISC is the "right" way to build ISAs.
- 2-address
  - add r1, r2 --> r1 = r1 + r2
  - + few operands, so more bits for each.
  - - lots of extra copy instructions
- 1-address
  - Accumulator architectures
  - add r1 -> acc = acc + rl

Stack-based ISA

- A push-down stack holds arguments
- Some instruction manipulate the stack
  - push, pop, swap, etc.
- Most instructions operate on the contents of the stack
  - Zero-operand instructions
  - add --> t1 = pop; t2 = pop; push t1 + t2;
- Elegant in theory.
- Clumsy in hardware.
  - How big is the stack?
- Java byte code is a stack-based ISA
- So is the x86 floating point ISA

compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result



compute A = X * Y - B * C

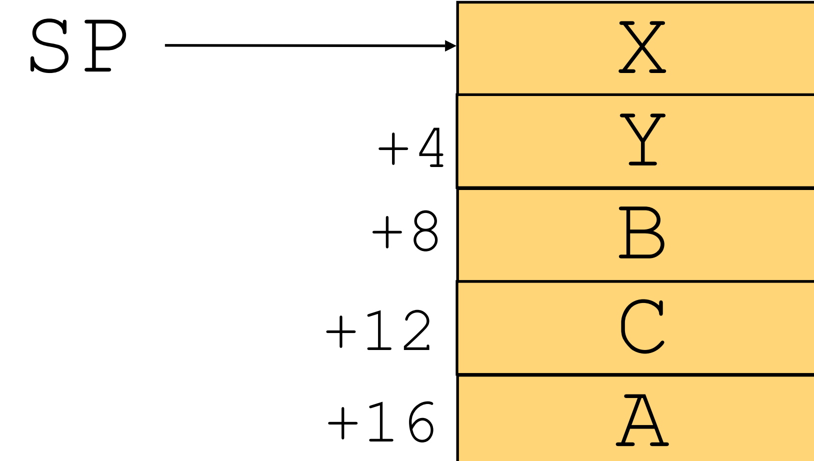
- Stack-based ISA
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result

PC

```
Push 8(BP)
Push 12(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop
```

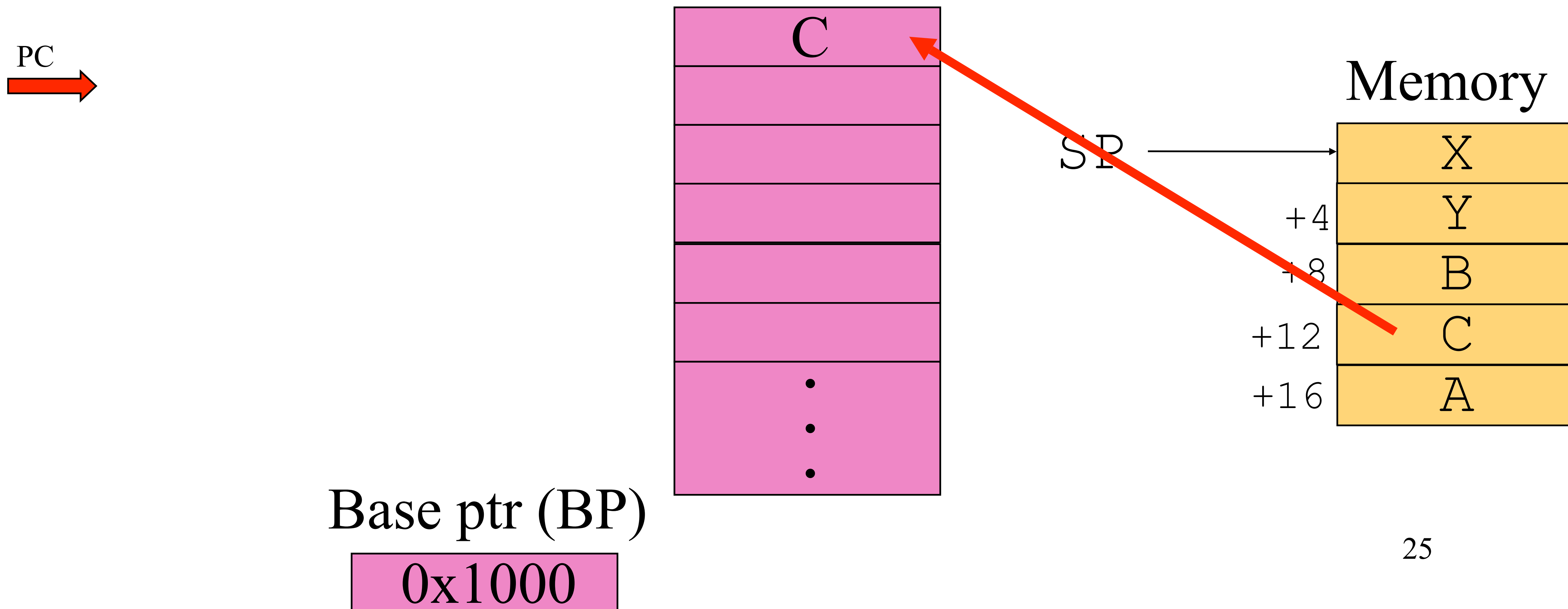
Base ptr (BP)

0x1000



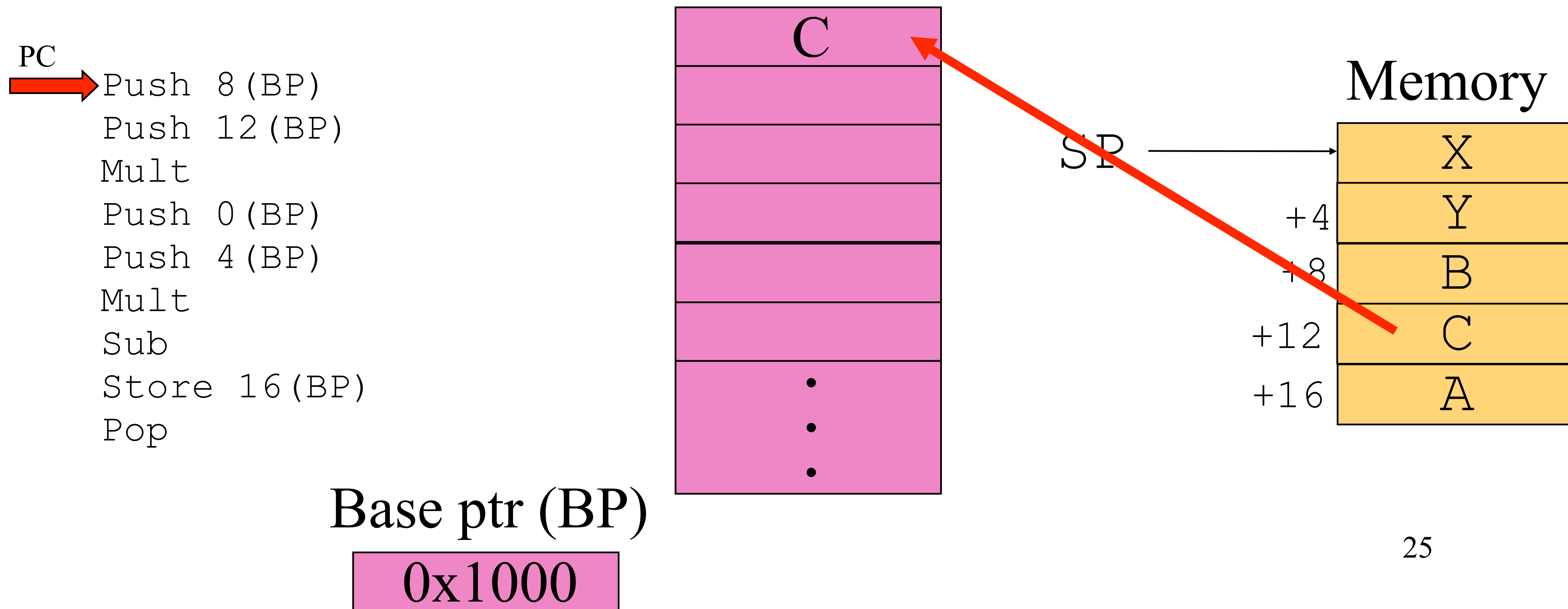
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result



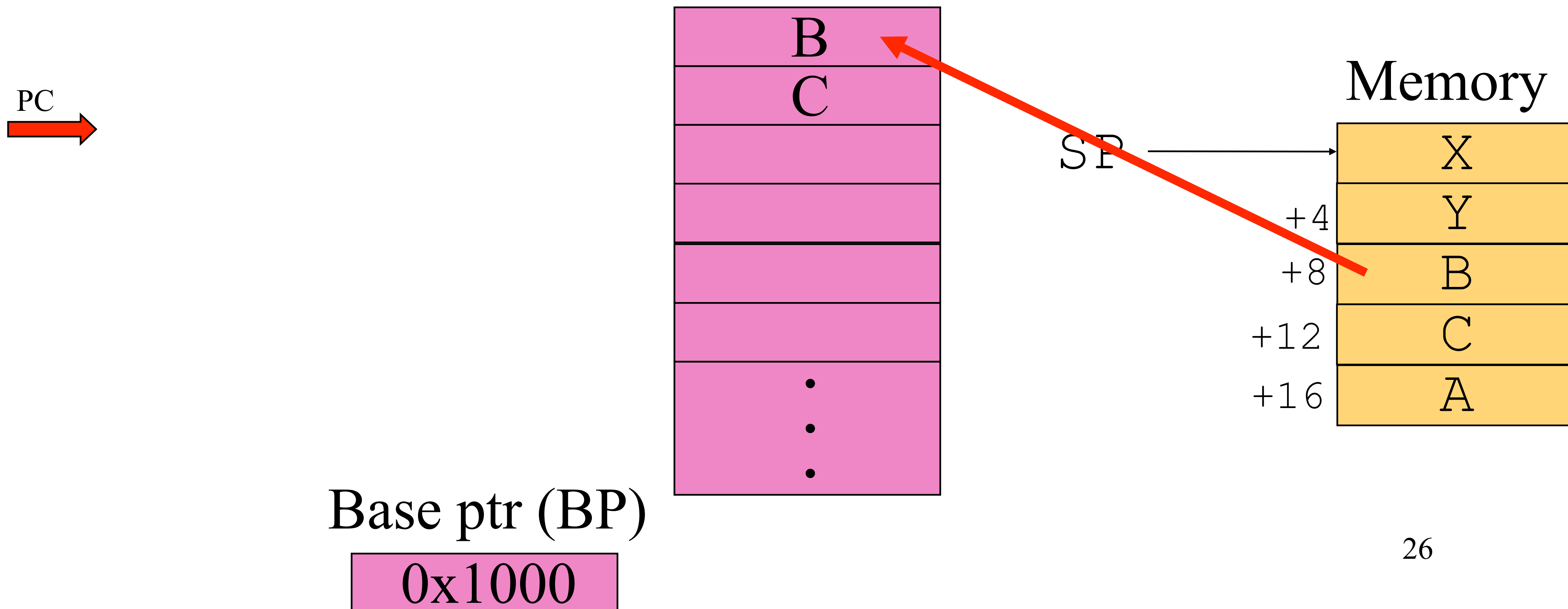
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result



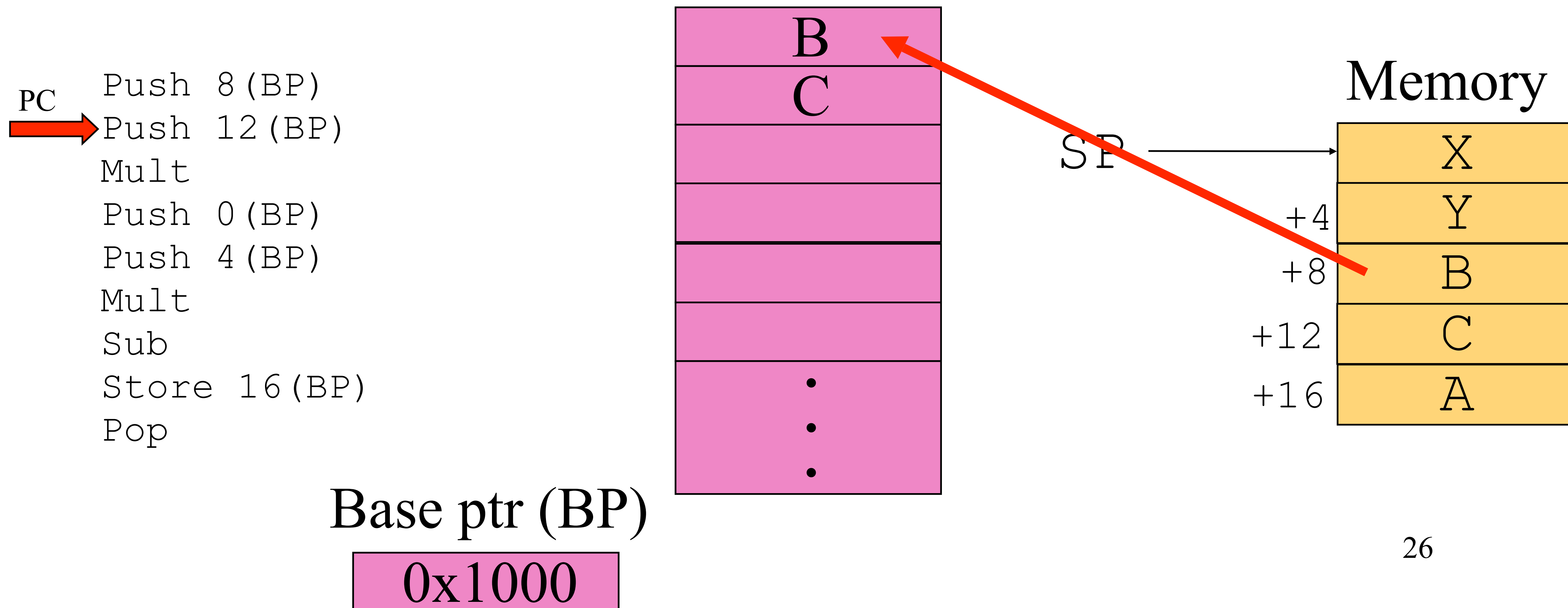
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result



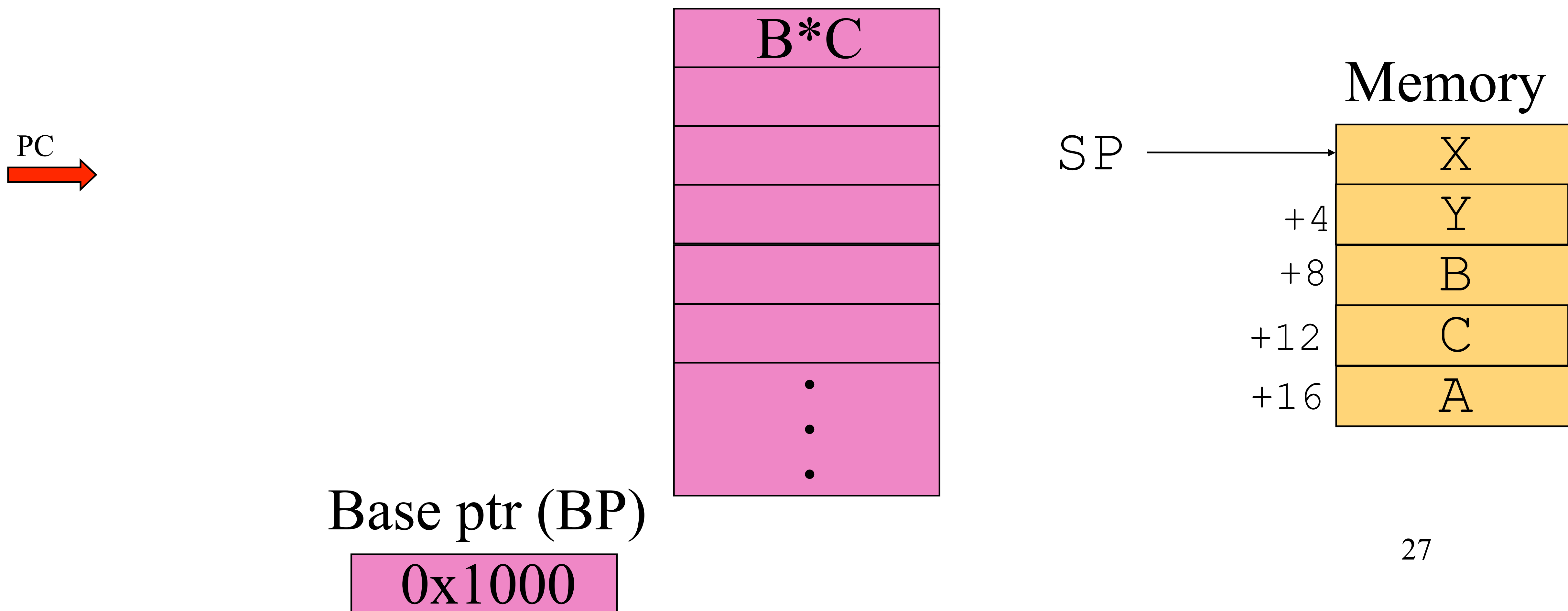
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result



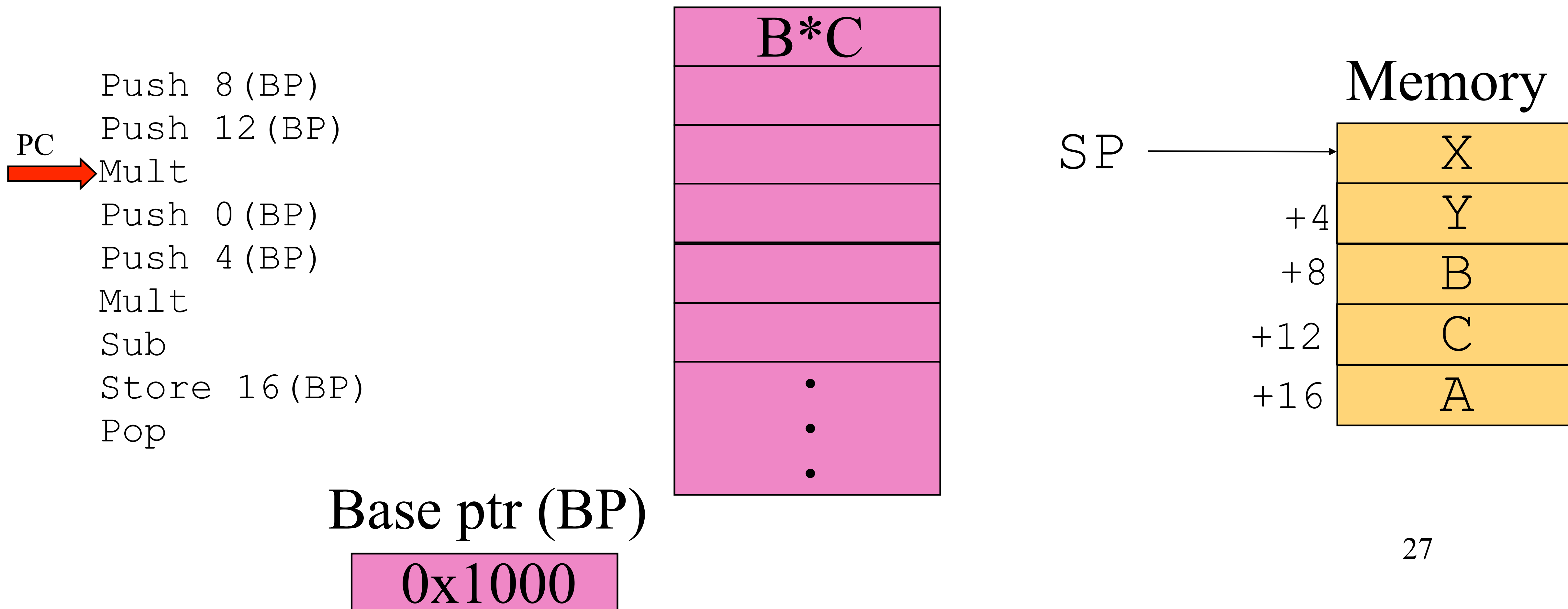
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result



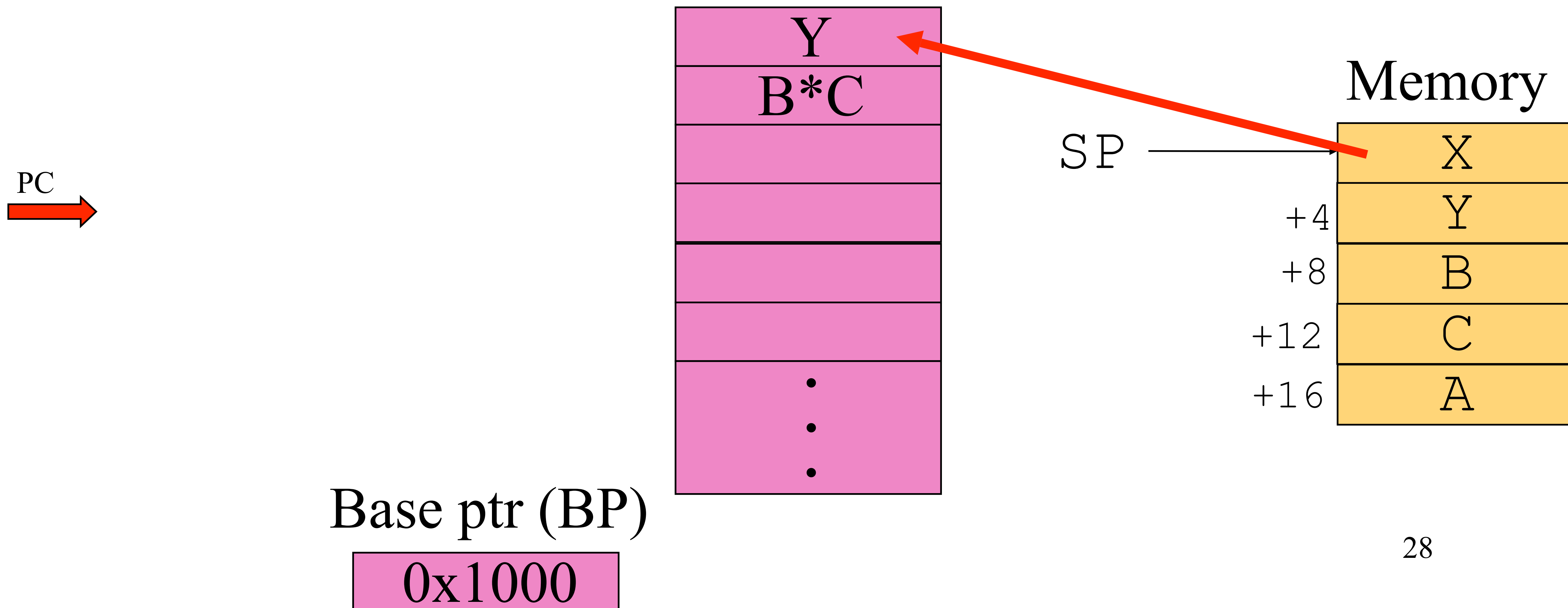
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result



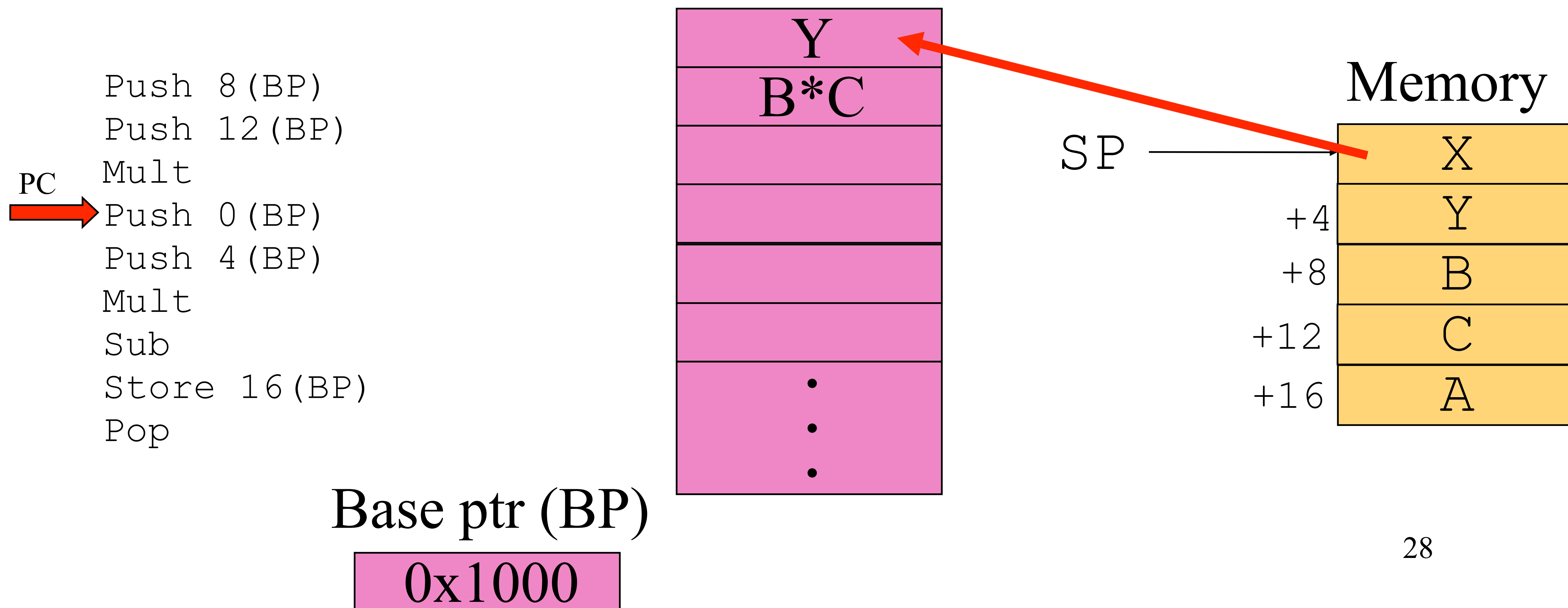
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result



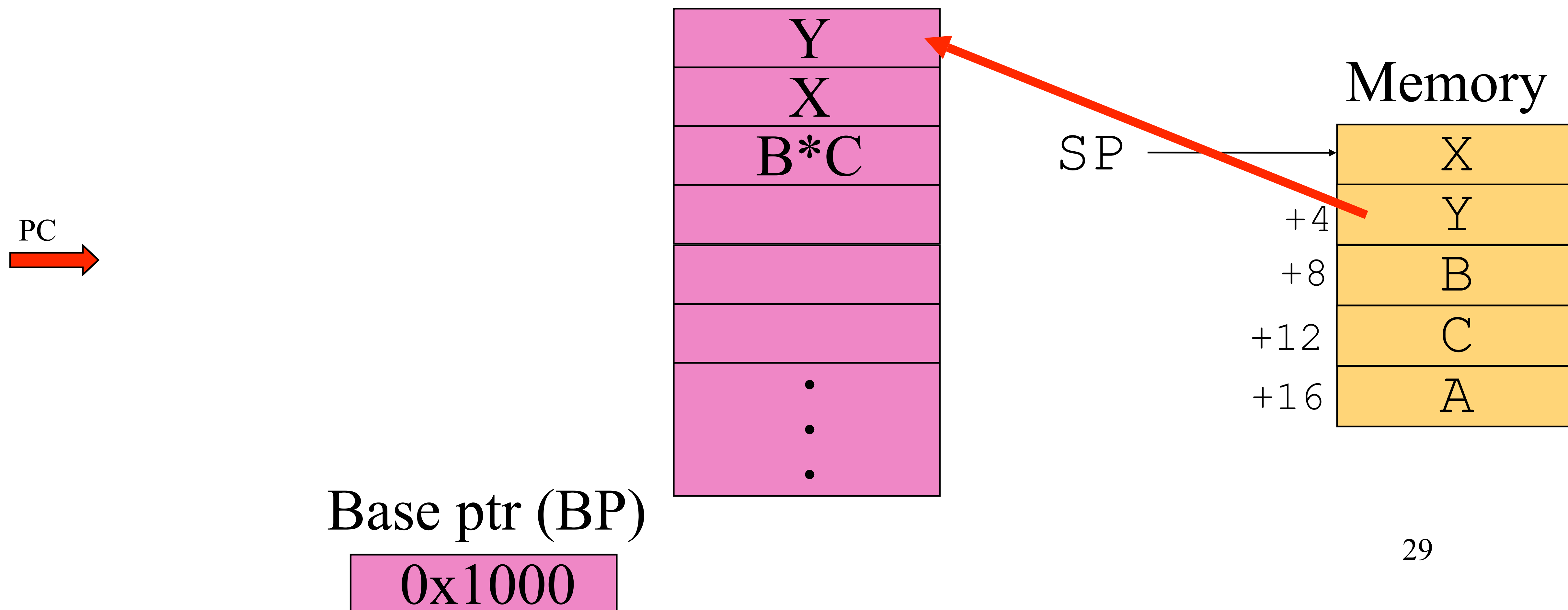
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result



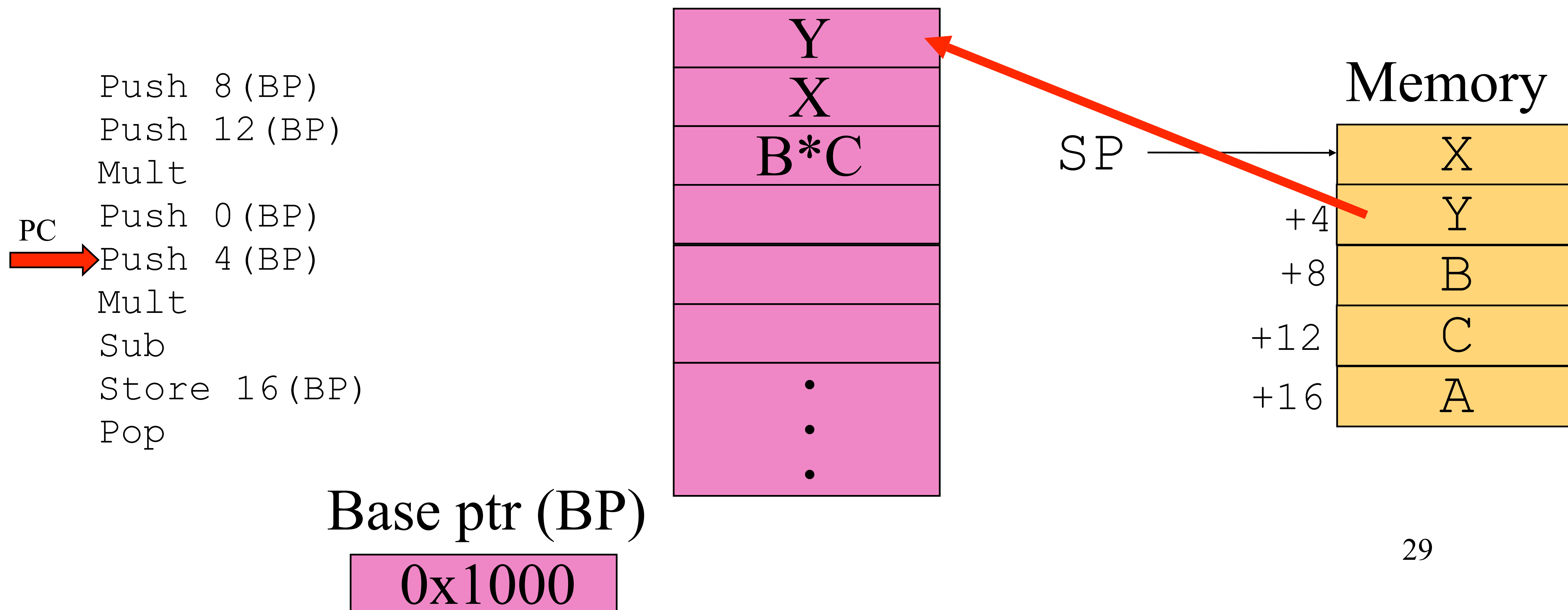
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result



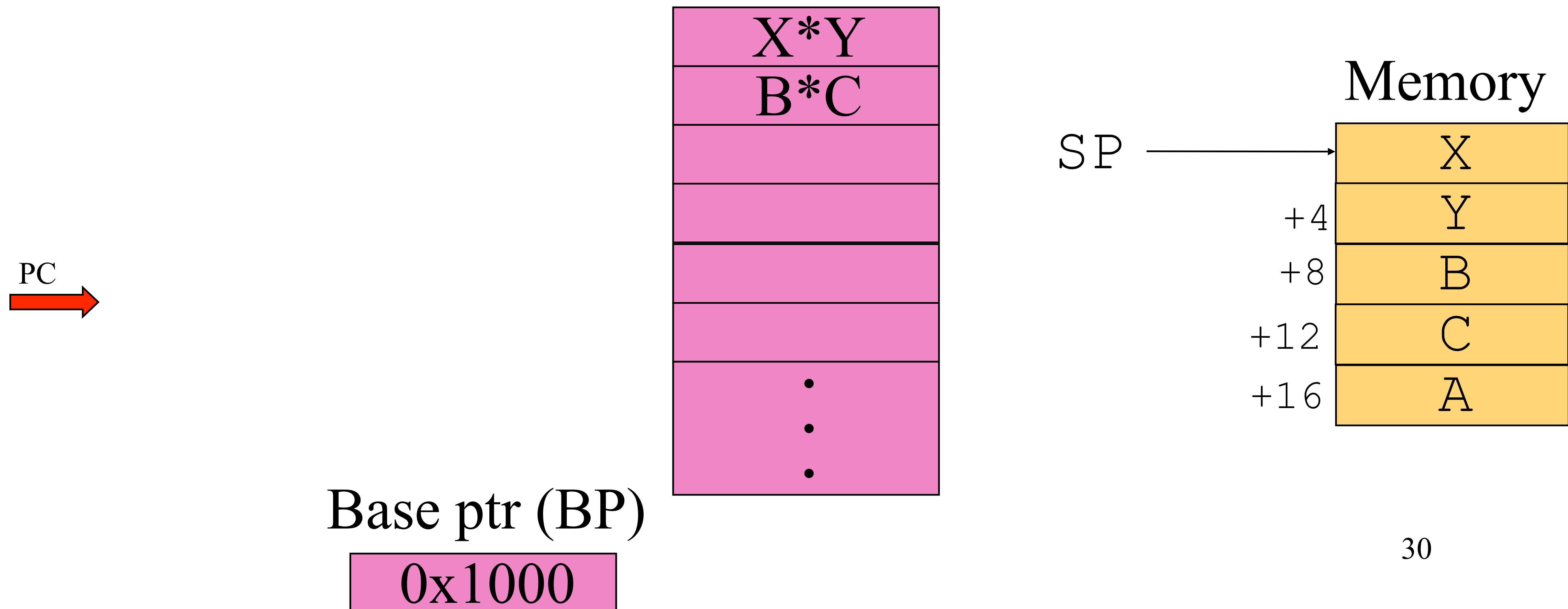
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result



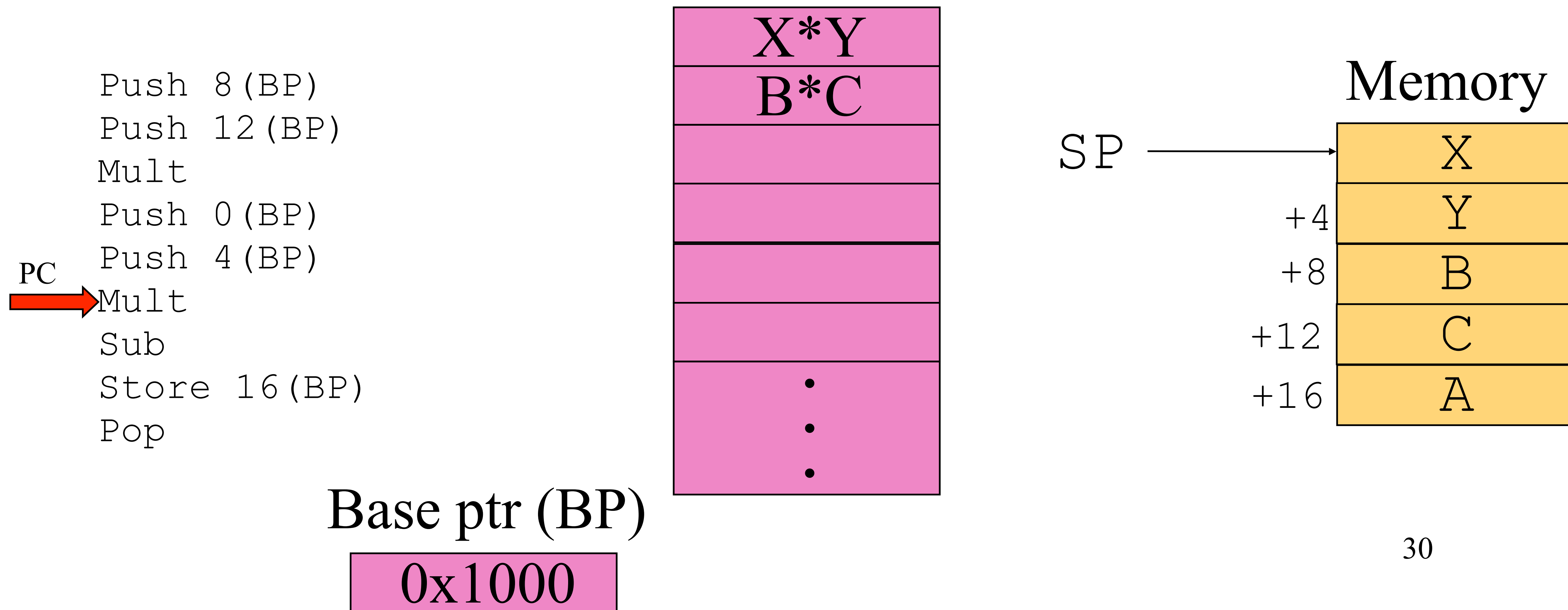
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result
  - Store -- Store the top of the stack



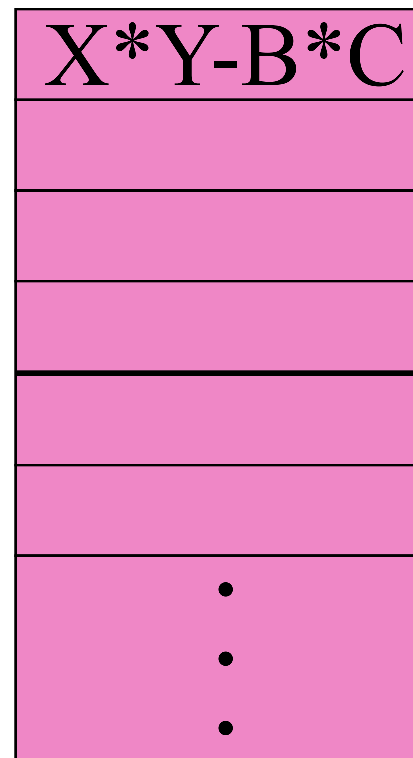
compute A = X * Y - B * C

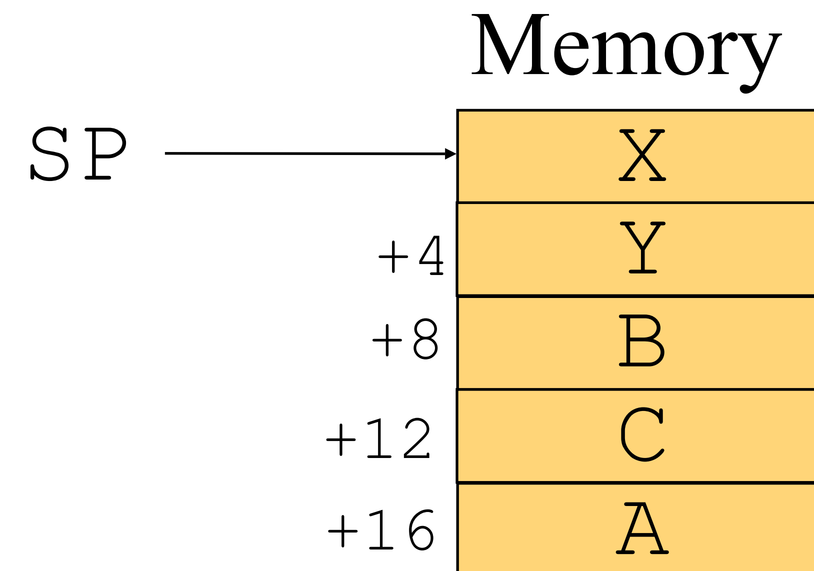
- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result
  - Store -- Store the top of the stack



compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,... -- Replace top two values with the result





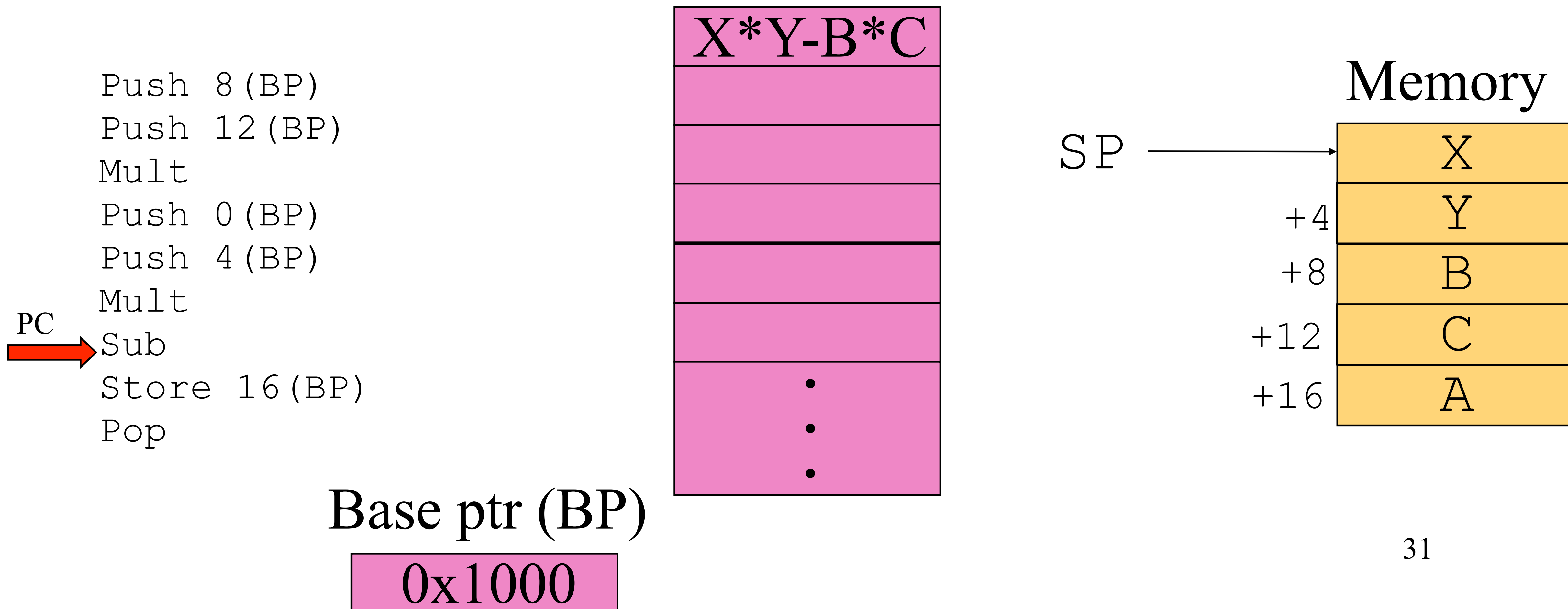
PC

Base ptr (BP)

0x1000

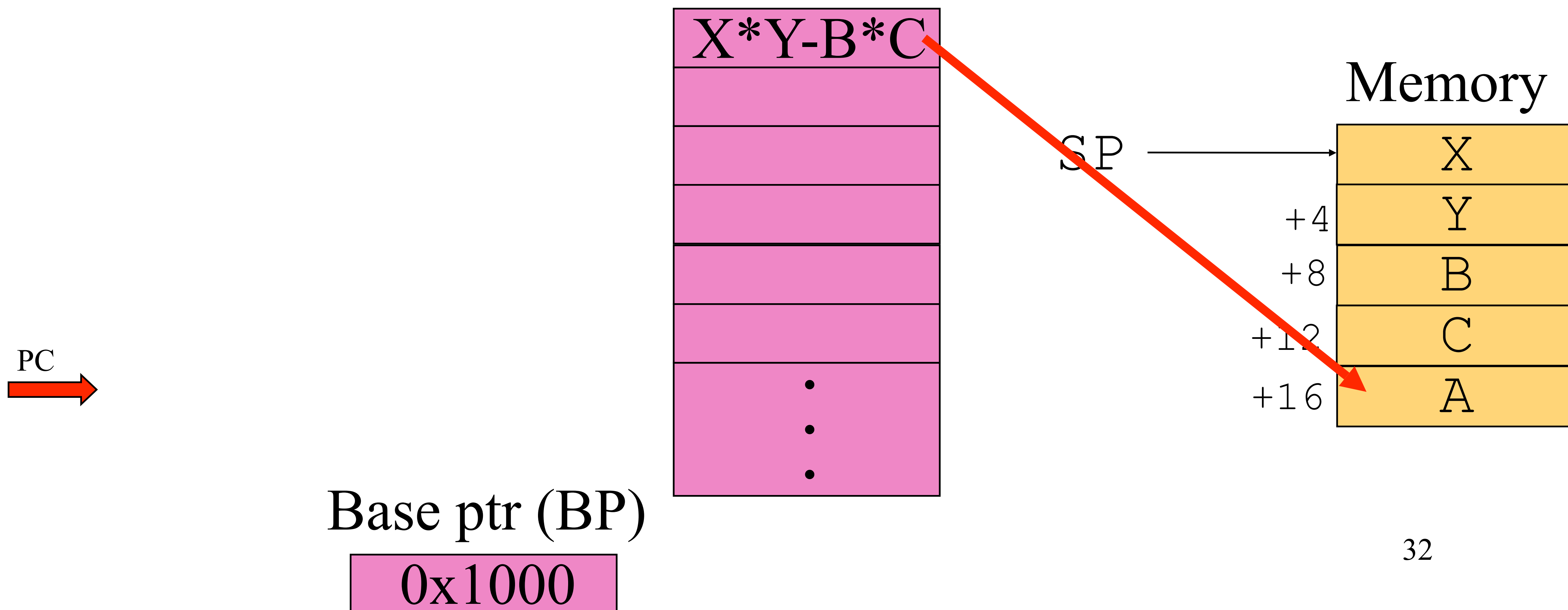
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result



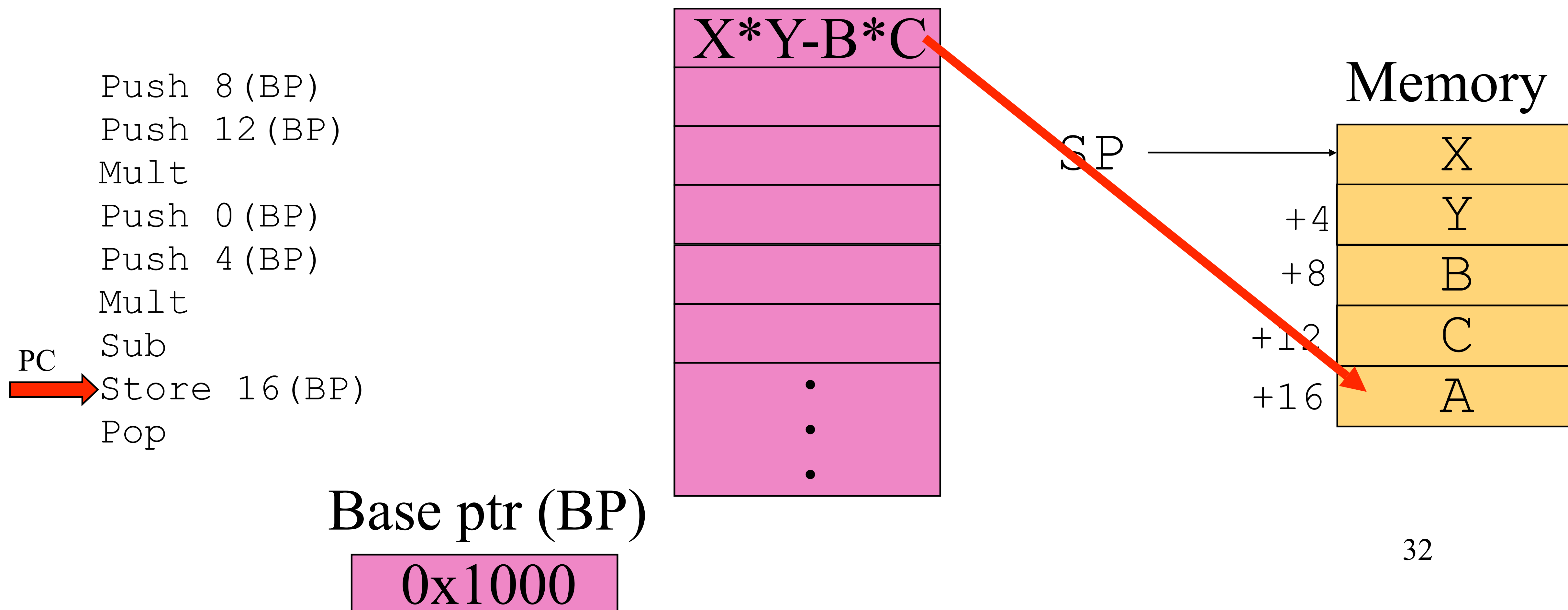
compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result
  - Store -- Store the top of the stack



compute A = X * Y - B * C

- Stack-based ISA
  - Processor state: PC, "operand stack", "Base ptr"
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result
  - Store -- Store the top of the stack



Supporting Function Calls

- Functions are an essential feature of modern languages
- What does a function need?
  - Arguments.
  - Storage for local variables.
  - To return control to the the caller.
  - To execute regardless of who called it.
  - To call other functions (that call other functions...that call other functions)
- There are not *instructions* for this
  - It is a contract about how the function behaves
  - In particular, how it treats the resources that are shared between functions -- the registers and memory

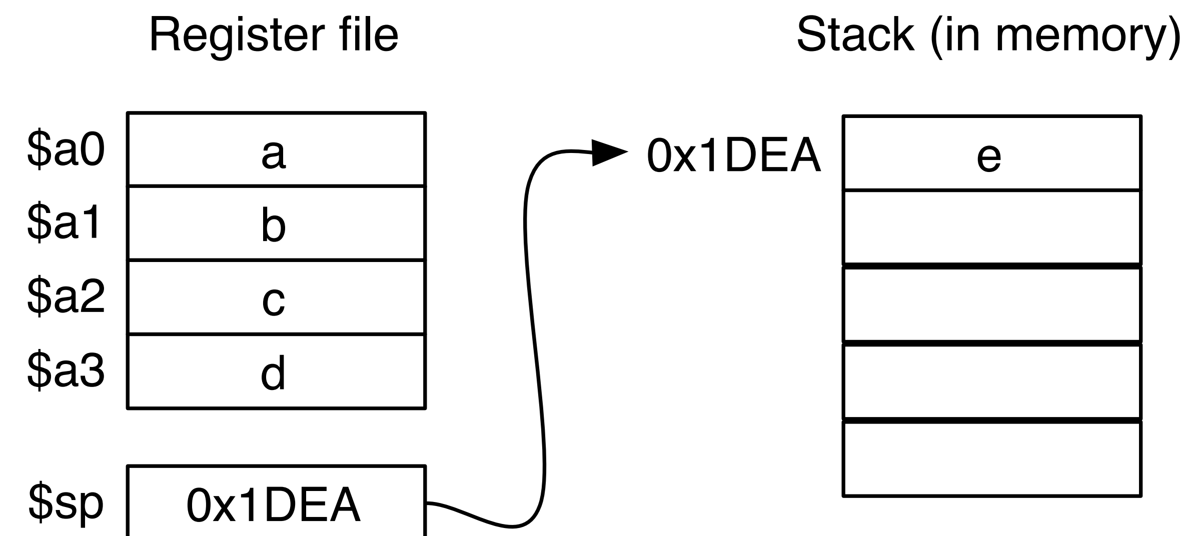
Register Discipline

- All registers are the same, but we assign them different uses.

Name	number	use	saved?
$zero	0	zero	n/a
$v0-$v1	2-3	return value	no
$a0-$a3	4-7	arguments	no
$t0-$t7	8-15	temporaries	no
$s0-$7	26-23	saved	yes
$t8-$t9	24-25	temporaries	no
$gp	26	global ptr	yes
$sp	29	stack ptr	yes
$fp	30	frame ptr	yes
$ra	31	return address	yes

Arguments

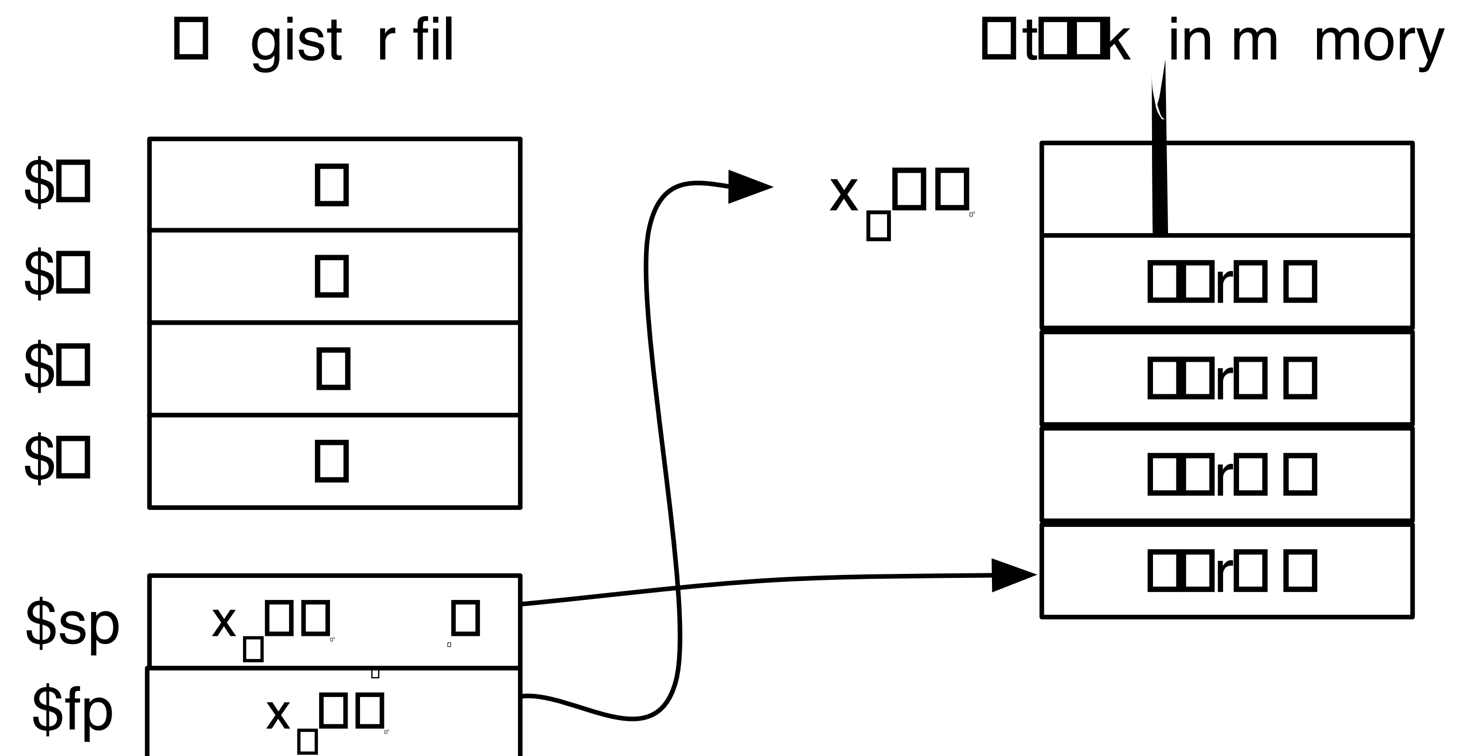
- How many arguments can function have?
  - unbounded.
  - But most functions have just a few.
- Make the common case fast
  - Put the first 4 argument in registers ($a0-$a3).
  - Put the rest on the "stack"




```
int Foo(int a, int b, int c, int d, int e) {
    ...
}
```

Storage for Local Variables

- Local variables go on the stack too.

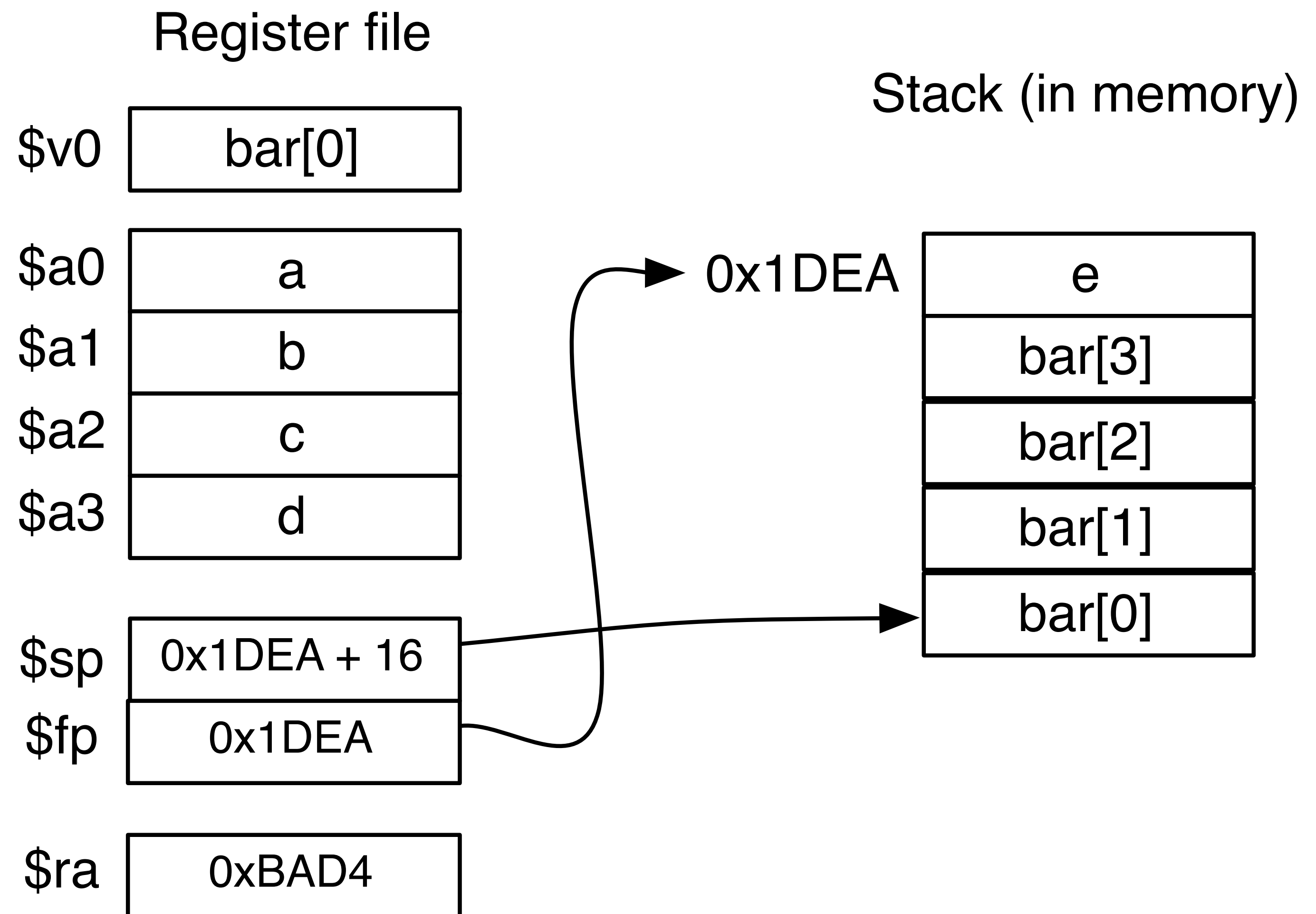



```
int Foo(int a, int b, int c, int d, int e) {
    int bar[4];
    ...
}
```

Returning Control

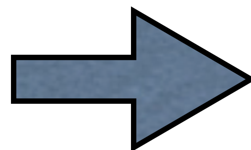
Caller

```
          ...
          move $a0, $t1
          move $a1, $s4
          move $a2, $s3
          move $a3, $s3
          sw   $t2, 0($sp)
          subi $sp, $sp, 4
0xBAD0:   jal  Foo
```



Callee

```
int Foo(int a, ...) {
    int bar[4];
    ...
     return bar[0];
}
```



```
...
subi $sp, $sp, 16 // Allocate bar
...
lw   $v0, 0($sp)
addi $sp, $sp, 16 // deallocate bar
jr   $ra          // return
```

Saving Registers

- Some registers are preserved across function calls
  - If a function needs a value after the call, it uses one of these
  - But it must also preserve the previous contents (so it can honor its obligation to its caller)
  - Push these registers onto the stack.
  - See figure 2.12 in the text.

Evaluating Computers: Bigger, better, faster, more?

What do you want in a computer?

What do you want in a computer?

- Low latency -- one unit of work in minimum time
  - 1/latency = responsiveness
- High throughput -- maximum work per time
  - High bandwidth (BW)
- Low cost
- Low power -- minimum jules per time
- Low energy -- minimum jules per work
- Reliability -- Mean time to failure (MTTF)
- Derived metrics
  - responsiveness/dollar
  - BW/$
  - BW/Watt
  - Work/Jule
  - Energy * latency -- Energy delay product
  - MTTF/$

Latency

- This is the simplest kind of performance
- How long does it take the computer to perform a task?
  - The task at hand depends on the situation.
- Usually measured in seconds
- Also measured in clock cycles
  - Caution: if you are comparing two different system, you must ensure that the cycle times are the same.

Measuring Latency

- Stop watch!
- System calls
  - `gettimeofday()`
  - `System.currentTimeMillis()`
- Command line
  - `time <command>`

Where latency matters

- Application responsiveness
  - Any time a person is waiting.
  - GUIs
  - Games
  - Internet services (from the users perspective)
- "Real-time" applications
  - Tight constraints enforced by the real world
  - Anti-lock braking systems
  - Manufacturing control
  - Multi-media applications
- The cost of poor latency
  - If you are selling computer time, latency is money.

Latency and Performance

- By definition:
- Performance = 1/Latency
- If Performance(X) > Performance(Y), X is faster.
- If Perf(X)/Perf(Y) = S, X is S times faster than Y.
- Equivalently: Latency(Y)/Latency(X) = S

- When we need to talk about specifically about other kinds of "performance" we must be more specific.

The Performance Equation

- We would like to model how architecture impacts performance (latency)
- This means we need to quantify performance in terms of architectural parameters.
  - Instructions -- this is the basic unit of work for a processor
  - Cycle time -- these two give us a notion of time.
  - Cycles

- The first fundamental theorem of computer architecture:

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

The Performance Equation

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

- The units work out! Remember your dimensional analysis!
- Cycles/Instruction == CPI
- Seconds/Cycle == 1/hz
- Example:
  - 1GHz clock
  - 1 billion instructions
  - CPI = 4
  - What is the latency?

Examples

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

- gcc runs in 100 sec on a 1 GHz machine
  - How many cycles does it take?

100G cycles

- gcc runs in 75 sec on a 600 MHz machine
  - How many cycles does it take?

45G cycles

How can this be?

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

- Different Instruction count?
  - Different ISAs ?
  - Different compilers ?
- Different CPI?
  - underlying machine implementation
  - Microarchitecture
- Different cycle time?
  - New process technology
  - Microarchitecture

Computing Average CPI

- Instruction execution time depends on instruction time (we'll get into why this is so later on)
  - Integer +, -, <<, |, & -- 1 cycle
  - Integer *, /, -- 5-10 cycles
  - Floating point +, - -- 3-4 cycles
  - Floating point *, /, sqrt() -- 10-30 cycles
  - Loads/stores -- variable
  - All theses values depend on the particular implementation, not the ISA
- Total CPI depends on the workload's Instruction mix -- how many of each type of instruction executes
  - What program is running?
  - How was it compiled?

The Compiler’s Role

- Compilers affect CPI…
  - Wise instruction selection
    - “Strength reduction”: x*2n -> x << n
    - Use registers to eliminate loads and stores
  - More compact code -> less waiting for instructions
- …and instruction count
  - Common sub-expression elimination
  - Use registers to eliminate loads and stores

Stupid Compiler

```
int i, sum = 0;
for(i=0;i<10;i++)
    sum += i;
```

Type	CPI	Static #	dyn #
mem	5	6	42
int	1	3	30
br	1	2	20
Total	2.8	11	92

(5*42 + 1*30 + 1*20)/92 = 2.8

```
sw   0($sp), $0 #sum = 0
sw   4($sp), $0 #i = 0
loop:
lw   $1, 4($sp)
sub  $3, $1, 10
beq  $3, $0, end
lw   $2, 0($sp)
add  $2, $2, $1
st   0($sp), $2
addi $1, $1, 1
st   4($sp), $1
b    loop
end:
```

Smart Compiler

```
int i, sum = 0;
for(i=0;i<10;i++)
    sum += i;
```

```
add  $1, $0, $0 # i
add  $2, $0, $0 # sum
loop:
sub  $3, $1, 10
beq  $3, $0, end
add  $2, $2, $1
addi $1, $1, 1
b    loop
end:
sw   0($sp), $2
```

Type	CPI	Static #	dyn #
mem	5	1	1
int	1	5	32
br	1	2	20
Total	1.01	8	53

$(5*1 + 1*32 + 1*20)/53 = 2.8$

Live demo

Program inputs affect CPI too!

```
int rand[1000] = {random 0s and 1s }
for(i=0;i<1000;i++)
    if(rand[i]) sum -= i;
    else sum *= i;

int ones[1000] = {1, 1, ...}
for(i=0;i<1000;i++)
    if(ones[i]) sum -= i;
    else sum *= i;
```

- Data-dependent computation
- Data-dependent micro-architectural behavior
  - Processors are faster when the computation is predictable (more later)

Live demo

Making Meaningful Comparisons

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

- Meaningful CPI exists only:
  - For a particular program with a particular compiler
  - ....with a particular input.
- You MUST consider all 3 to get accurate latency estimations or machine speed comparisons
  - Instruction Set
  - Compiler
  - Implementation of Instruction Set (386 vs Pentium)
  - Processor Freq (600 Mhz vs 1 GHz)
  - Same high level program with same input
- "wall clock" measurements are always comparable.
  - If the workloads (app + inputs) are the same

The Performance Equation

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

- Clock rate =
- Instruction count =
- Latency =
- Find the CPI!

Today

- DRAM
- Quiz 1 recap
- HW 1 recap
- Questions about ISAs
- More about the project?
- Amdahl's law

Key Points

- Amdahl's law and how to apply it in a variety of situations
- It's role in guiding optimization of a system
- It's role in determining the impact of localized changes on the entire system
-

Limits on Speedup: Amdahl's Law

- "The fundamental theorem of performance optimization"
- Coined by Gene Amdahl (one of the designers of the IBM 360)
- Optimizations do not (generally) uniformly affect the entire program
  - The more widely applicable a technique is, the more valuable it is
  - Conversely, limited applicability can (drastically) reduce the impact of an optimization.

Always heed Amdahl's Law!!!

It is central to many many optimization problems

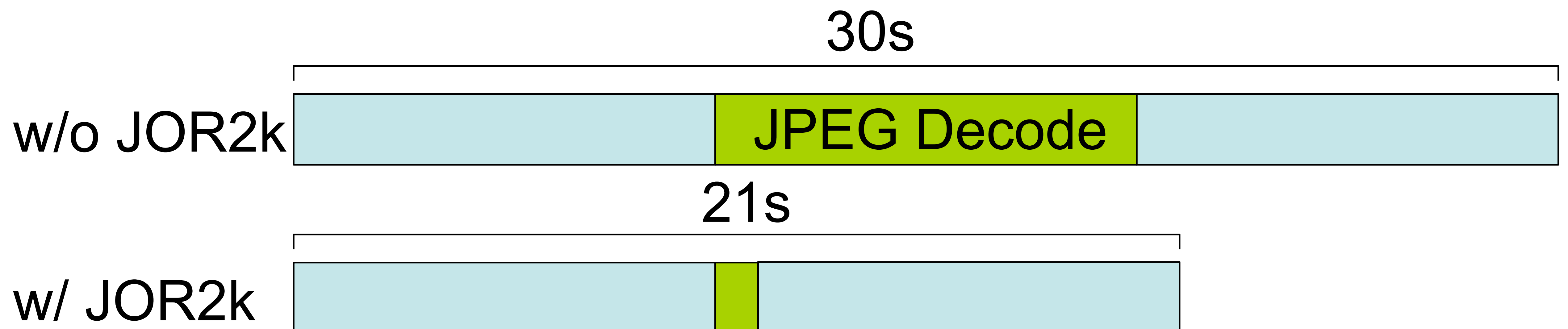
Amdahl's Law in Action

- SuperJPEG-O-Rama2000 ISA extensions **
  - –Speeds up JPEG decode by 10x!!!
  - –Act now! While Supplies Last!

** Increases processor cost by 45%

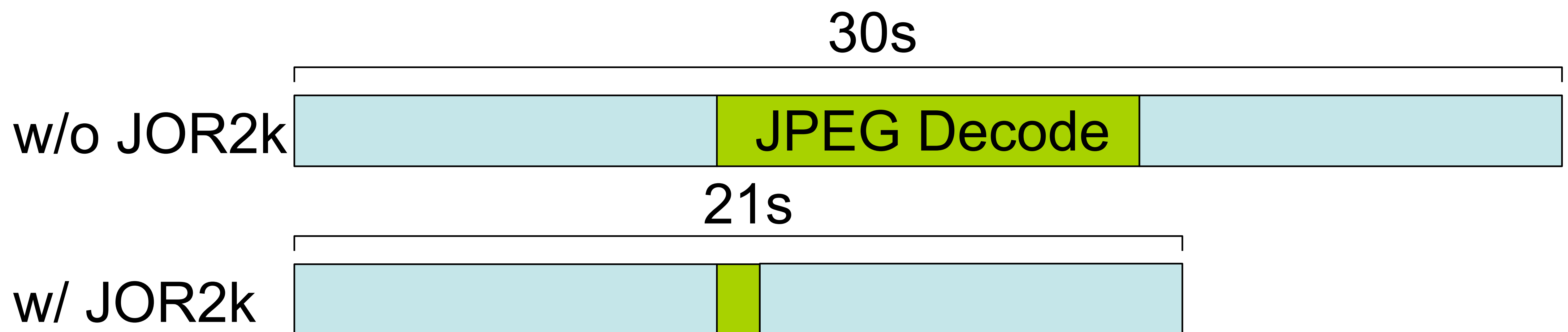
Amdahl’s Law in Action

- SuperJPEG-O-Rama2000 in the wild
- PictoBench spends 33% of it’s time doing JPEG decode
- How much does JOR2k help?



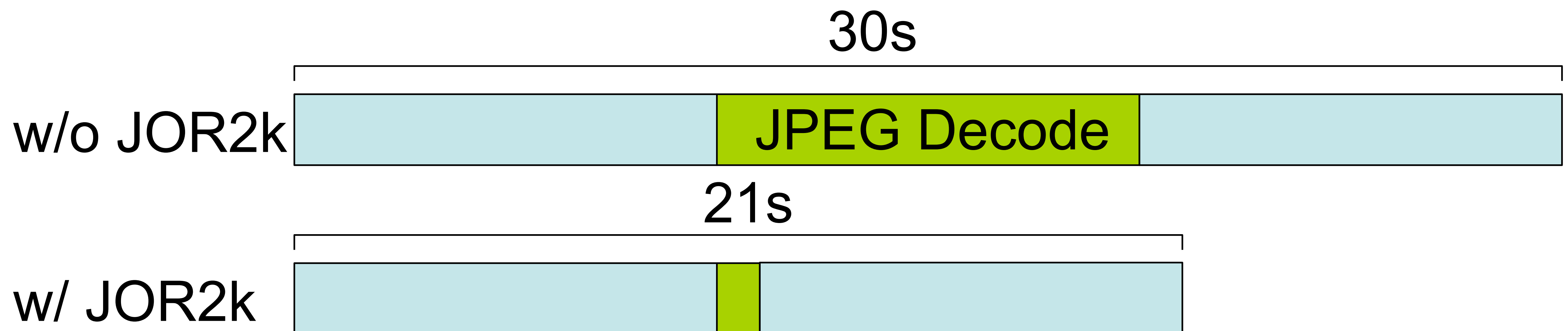
Amdahl's Law in Action

- SuperJPEG-O-Rama2000 in the wild
- PictoBench spends 33% of it's time doing JPEG decode
- How much does JOR2k help?



Amdahl's Law in Action

- SuperJPEG-O-Rama2000 in the wild
- PictoBench spends 33% of it's time doing JPEG decode
- How much does JOR2k help?

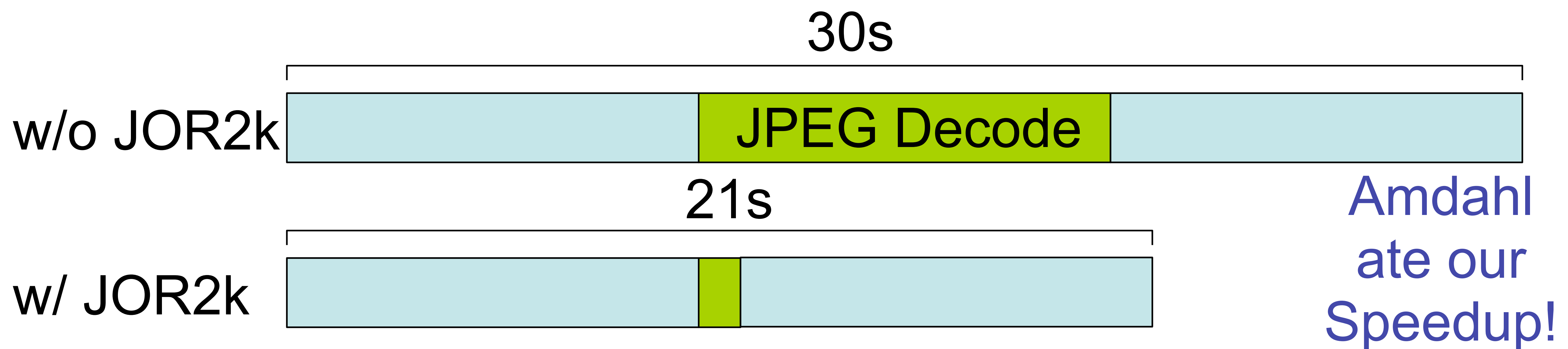


Performance: 30/21 = 1.4x Speedup != 10x

Is this worth the 45% increase in cost?

Amdahl's Law in Action

- SuperJPEG-O-Rama2000 in the wild
- PictoBench spends 33% of it's time doing JPEG decode
- How much does JOR2k help?



Amdahl's Law

- The second fundamental theorem of computer architecture.
- If we can speed up *X* of the program by *S* times
- Amdahl's Law gives the total speed up, S_{tot}

$$S_{tot} = \frac{1}{(x/S + (1-x))}.$$

Amdahl's Law

- The second fundamental theorem of computer architecture.
- If we can speed up *X* of the program by *S* times
- Amdahl's Law gives the total speed up, S_{tot}

$$S_{tot} = \frac{1}{(x/S + (1-x))}.$$

Sanity check:

$$x = 1 \Rightarrow S_{tot} = \frac{1}{(1/S + (1-1))} = \frac{1}{1/S} = S$$

Amdahl's Corollary #1

- Maximum possible speedup, S_{max}

$$S = infinity$$

$$S_{max} = \frac{1}{(1-x)}$$

Amdahl's Law Practice

- Protein String Matching Code
  - 200 hours to run on current machine, spends 20% of time doing integer instructions
  - How much faster must you make the integer unit to make the code run 10 hours faster?
  - How much faster must you make the integer unit to make the code run 50 hours faster?

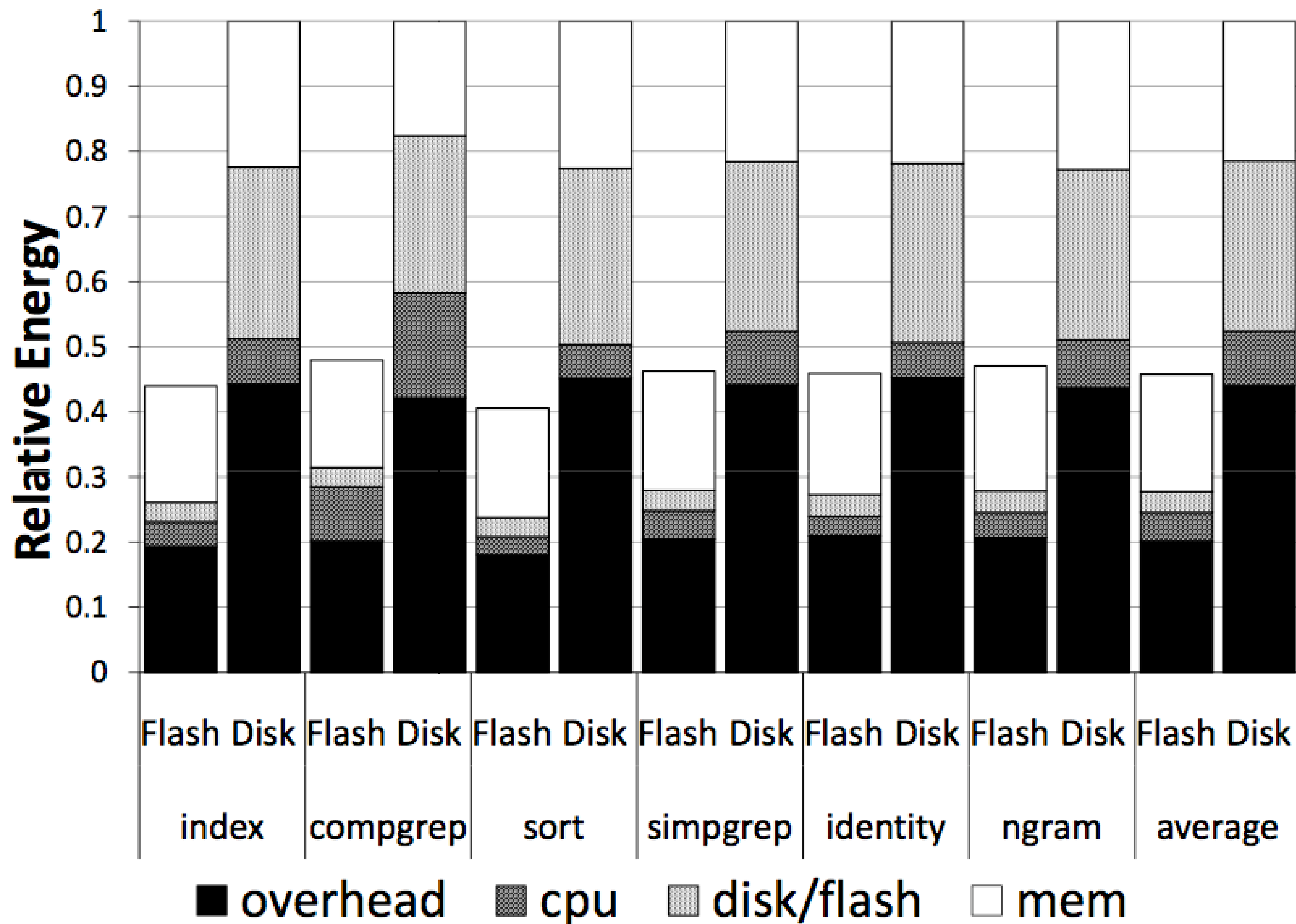
A) 1.1
B) 1.25
C) 1.75
D) 1.33

E) 10.0
F) 50.0
G) 1 million times
H) Other

Amdahl's Law Practice

- Protein String Matching Code
  - –4 days ET on current machine
    - 20% of time doing integer instructions
    - 35% percent of time doing I/O
  - –Which is the better economic tradeoff?
    - Compiler optimization that reduces number of integer instructions by 25% (assume each integer inst takes the same amount of time)
    - Hardware optimization that makes I/O run 20% faster?

Amdahl's Law Applies All Over

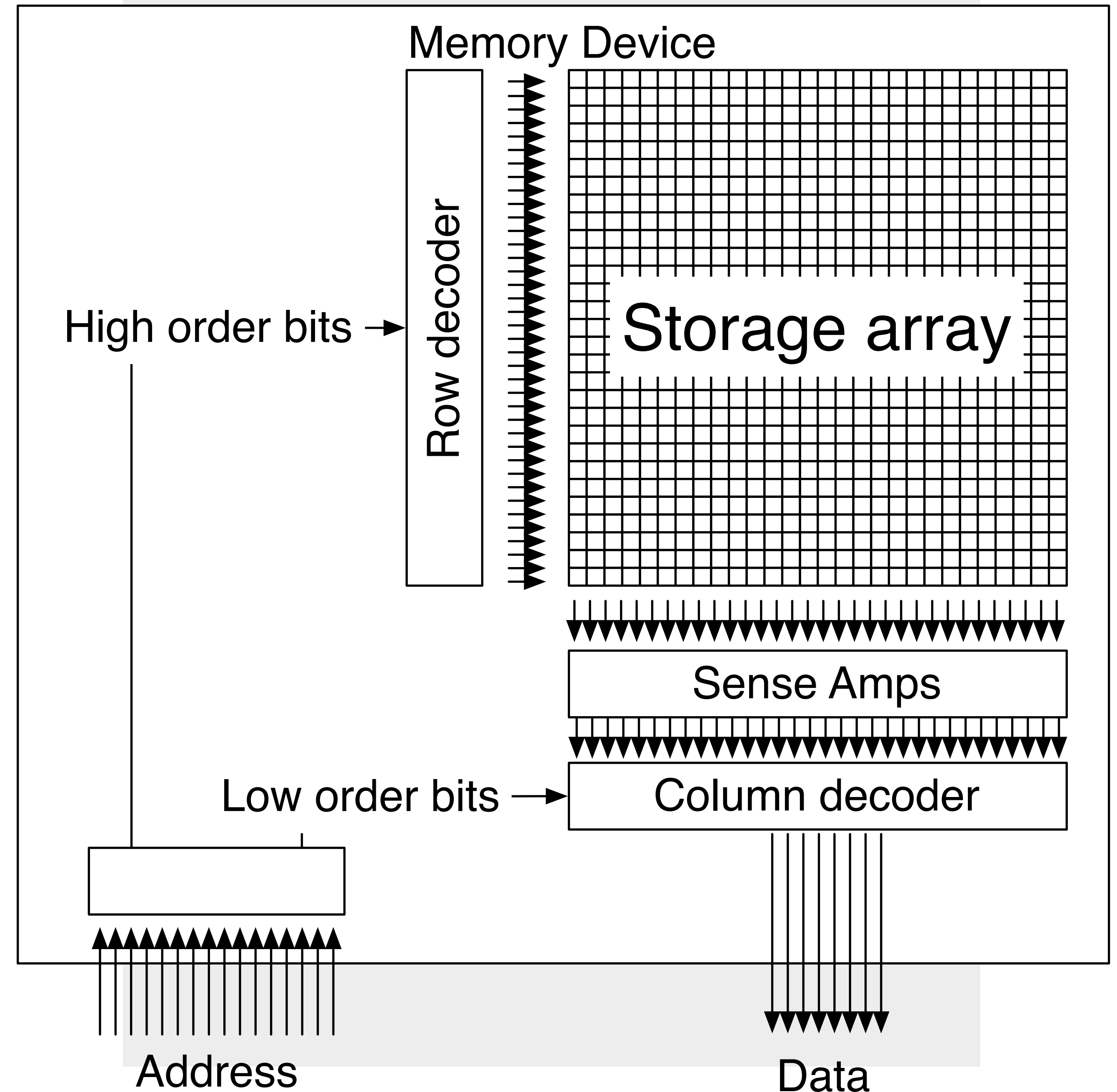


- SSDs use 10x less power than HDs
- But they only save you ~50% overall.

Amdahl's Law in Memory

- Storage array 90% of area
- Row decoder 4%
- Column decode 2%
- Sense amps 4%

- What's the benefit of reducing bit size by 10%?
- Reducing column decoder size by 90%?



Amdahl's Corollary #2

- Make the common case fast (i.e., *x* should be large)!
  - Common == "most time consuming" not necessarily "most frequent"
  - The uncommon case doesn't make much difference
  - Be sure of what the common case is
  - The common case changes.
- Repeat…
  - With optimization, the common becomes uncommon and vice versa.

Amdahl's Corollary #2: Example

Common case

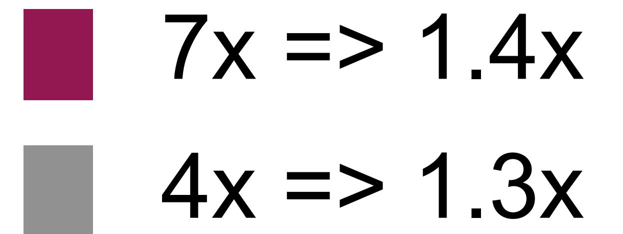
Amdahl's Corollary #2: Example

Common case

7x => 1.4x

Amdahl’s Corollary #2: Example

Common case



Amdahl's Corollary #2: Example

Common case

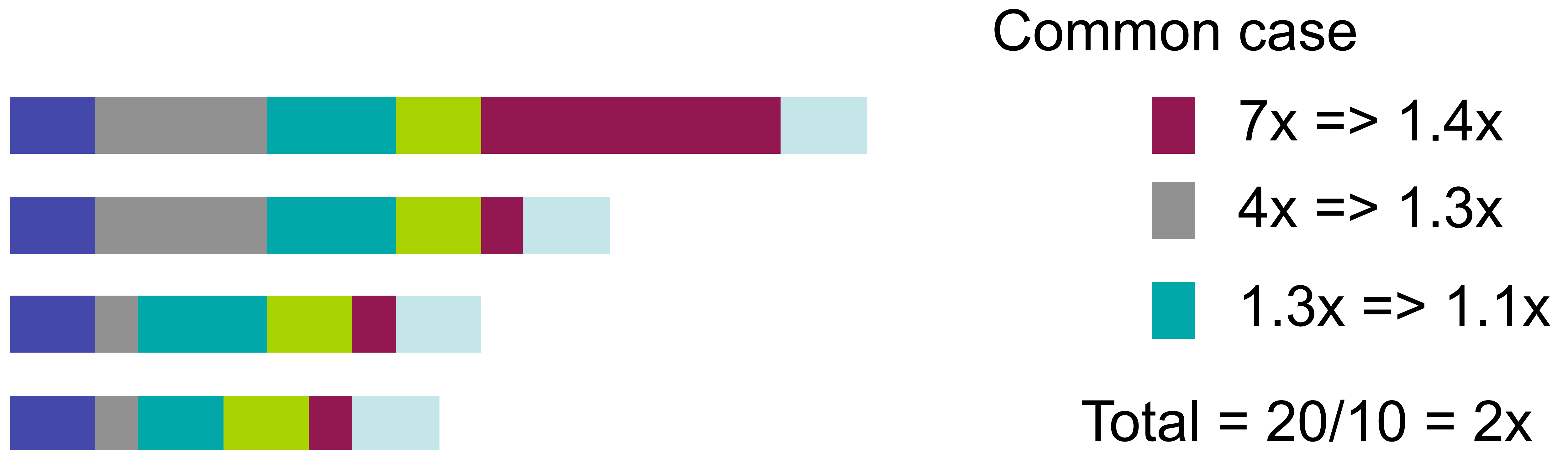
7x => 1.4x

4x => 1.3x

1.3x => 1.1x

Total = 20/10 = 2x

Amdahl's Corollary #2: Example



- In the end, there is no common case!
- Options:
  - Global optimizations (faster clock, better compiler)
  - Find something common to work on (i.e. memory latency)
  - War of attrition
  - Total redesign (You are probably well-prepared for this)

Amdahl's Corollary #3

- Benefits of parallel processing
- *p* processors
- *x*% is *p*-way parallizable
- maximum speedup, S_{par}

$$S_{par} = \frac{1}{(x/p + (1-x))}$$

Amdahl's Corollary #3

- Benefits of parallel processing
- p processors
- x% is p-way parallizable
- maximum speedup, S_{par}

$$S_{par} = \frac{1}{(x/p + (1-x))}$$

x is pretty small for desktop applications, even for p = 2

Amdahl's Corollary #3

- Benefits of parallel processing
- *p* processors
- *x*% is *p*-way parallizable
- maximum speedup, S_{par}

$$S_{par} = \frac{1}{(x/p + (1-x))}$$

x is pretty small for desktop applications, even for p = 2

Does Intel's 80-core processor make much sense?

Amdahl's Corollary #4

- Amdahl's law for latency (L)

$$L_{new} = L_{base} * 1/Speedup$$
$$L_{new} = L_{base} * (x/S + (1-x))$$
$$L_{new} = (L_{base} / S) * x + ET_{base} * (1-x)$$

- If you can speed up y% of the remaining (1-x), you can apply Amdahl's law recursively

$$L_{new} = (L_{base} / S_1) * x + (S_{base} * (1-x) / S_2 * y + L_{base} * (1-x) * (1-y))$$

- This is how we will analyze memory system performance

Amdahl's Non-Corollary

- Amdahl's law does not bound slowdown

$$L_{new} = (L_{base}/S)*x + L_{base}*(1-x)$$

- L_{new} is linear in 1/S
- Example: x = 0.01 of execution, $L_{base} = 1$
  - S = 0.001;
    - $E_{new} = 1000*L_{base}*0.01 + L_{base}*(0.99) \sim 10*L_{base}$
  - S = 0.00001;
    - $E_{new} = 100000*L_{base}*0.01 + L_{base}*(0.99) \sim 1000*L_{base}$
- Things can only get so fast, but they can get arbitrarily slow.
  - Do not hurt the non-common case too much!

Benchmarks: Standard Candles for Performance

- It's hard to convince manufacturers to run *your* program (unless you're a BIG customer)
- A benchmark is a set of programs that are representative of a class of problems.
- To increase predictability, collections of benchmark applications, called *benchmark suites*, are popular
  - "Easy" to set up
  - Portable
  - Well-understood
  - Stand-alone
  - Standardized conditions
  - These are all things that real software is not.

Classes of benchmarks

- Microbenchmark – measure one feature of system
  - e.g. memory accesses or communication speed
- Kernels – most compute-intensive part of applications
  - e.g. Linpack and NAS kernel b'marks (for supercomputers)
- Full application:
  - SpecInt / SpecFP (int and float) (for Unix workstations)
  - Other suites for databases, web servers, graphics,...

Bandwidth

- The amount of work (or data) per time
  - MB/s, GB/s -- network BW, disk BW, etc.
  - Frames per second -- Games, video transcoding
    - (why are games under both latency and BW?)
- Also called "throughput"

Measuring Bandwidth

- Measure how much work is done
- Measure latency
- Divide

Latency-BW Trade-offs

- Often, increasing latency for one task and increase BW for many tasks.
  - Think of waiting in line for one of 4 bank tellers
  - If the line is empty, your response time is minimized, but throughput is low because utilization is low.
  - If there is always a line, you wait longer (your latency goes up), but there is always work available for tellers.
- Much of computer performance is about scheduling work onto resources
  - Network links.
  - Memory ports.
  - Processors, functional units, etc.
  - IO channels.
  - Increasing contention for these resources generally increases throughput but hurts latency.

Stationwagon Digression

- IPv6 Internet 2: 272,400 terabit-meters per second
  - 585GB in 30 minutes over 30,000 Km
  - 9.08 Gb/s

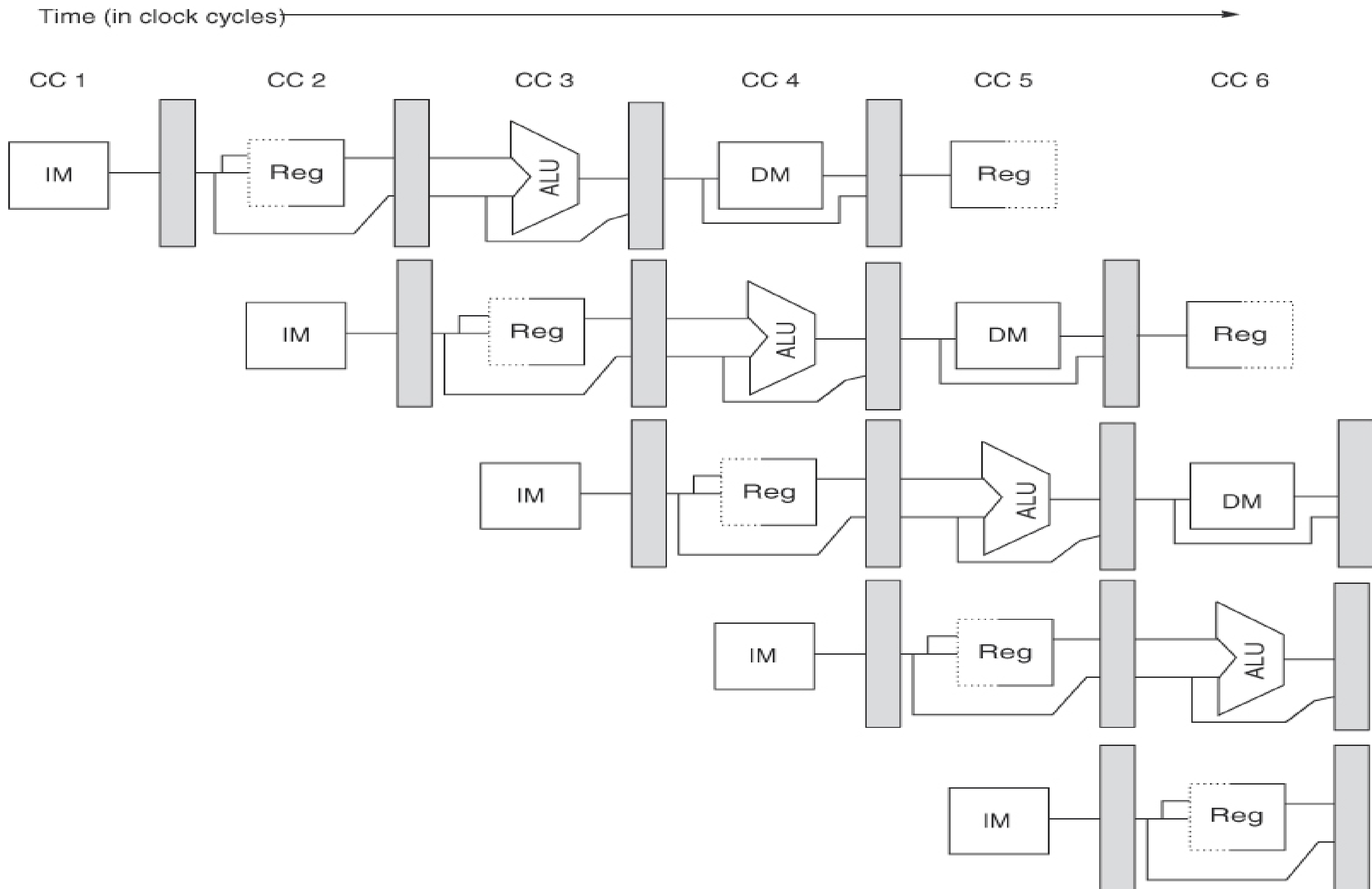
- Subaru outback wagon
  - Max load = 408Kg
  - 21Mpg
- MHX2 BT 300 Laptop drive
  - 300GB/Drive
  - 0.135Kg
- 906TB
- Legal speed: 75MPH (33.3 m/s)
- BW = 8.2 Gb/s
- Latency = 10 days
- 241,535 terabit-meters per second

Prius Digression

- IPv6 Internet 2: 272,400 terabit-meters per second
  - 585GB in 30 minutes over 30,000 Km
  - 9.08 Gb/s

- My Toyota Prius
  - Max load = 374Kg
  - 44Mpg (2x power efficiency)
- MHX2 BT 300
  - 300GB/Drive
  - 0.135Kg
- 831TB
- Legal speed: 75MPH (33.3 m/s)
- BW = 7.5 Gb/s
- Latency = 10 days
- 221,407 terabit-meters per second (13% performance hit)

A 5-Stage Pipeline



Source: H&P textbook

Pipeline Summary

	RR	ALU	DM	RW
ADD R1, R2, → R3	Rd R1,R2	R1+R2	--	Wr R3
BEQ R1, R2, 100	Rd R1, R2 Compare, Set PC	--	--	--
LD 8[R3] → R6	Rd R3	R3+8	Get data	Wr R6
ST 8[R3] ← R6	Rd R3,R6	R3+8	Wr data	--

Hazards

- Structural hazards: different instructions in different stages (or the same stage) conflicting for the same resource

- Data hazards: an instruction cannot continue because it needs a value that has not yet been generated by an earlier instruction

- Control hazard: fetch cannot continue because it does not know the outcome of an earlier branch – special case of a data hazard – separate category because they are treated in different ways

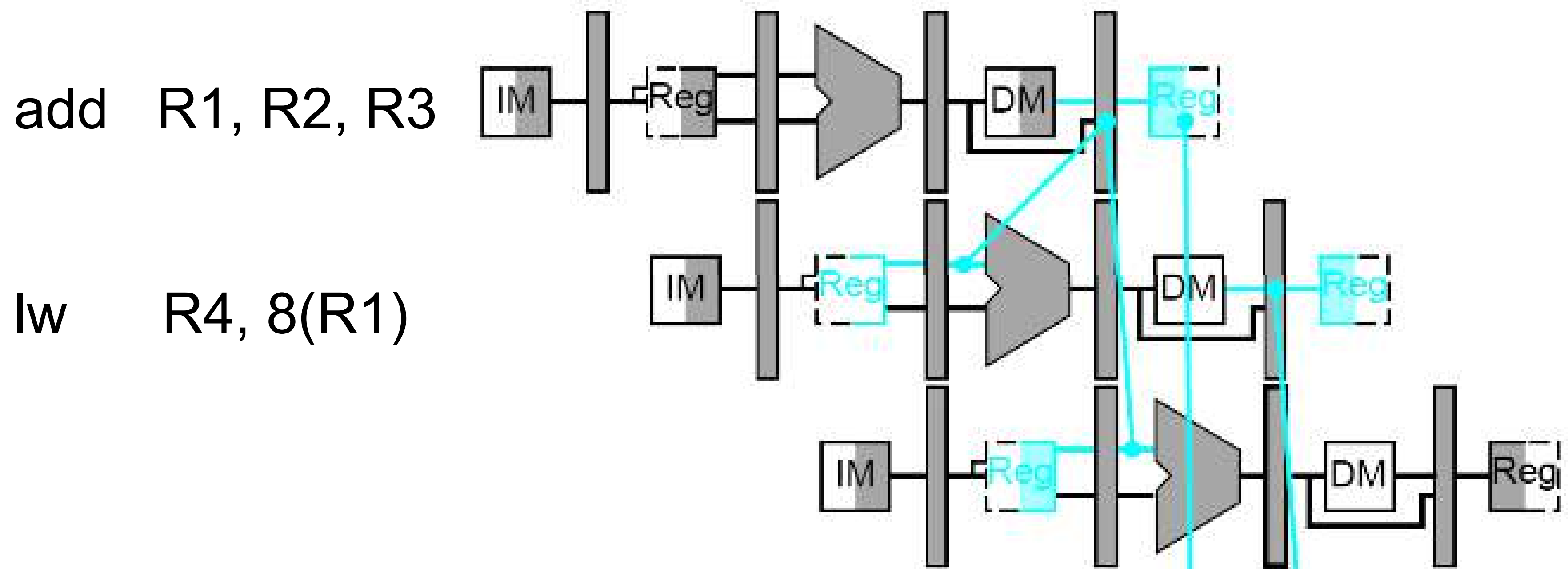
Data Hazards

- An instruction *produces* a value in a given pipeline stage
- A subsequent instruction *consumes* that value in a pipeline stage
- The consumer may have to be delayed so that the time of consumption is later than the time of production

Example 1

add R1, R2, R3

lw R4, 8(R1)

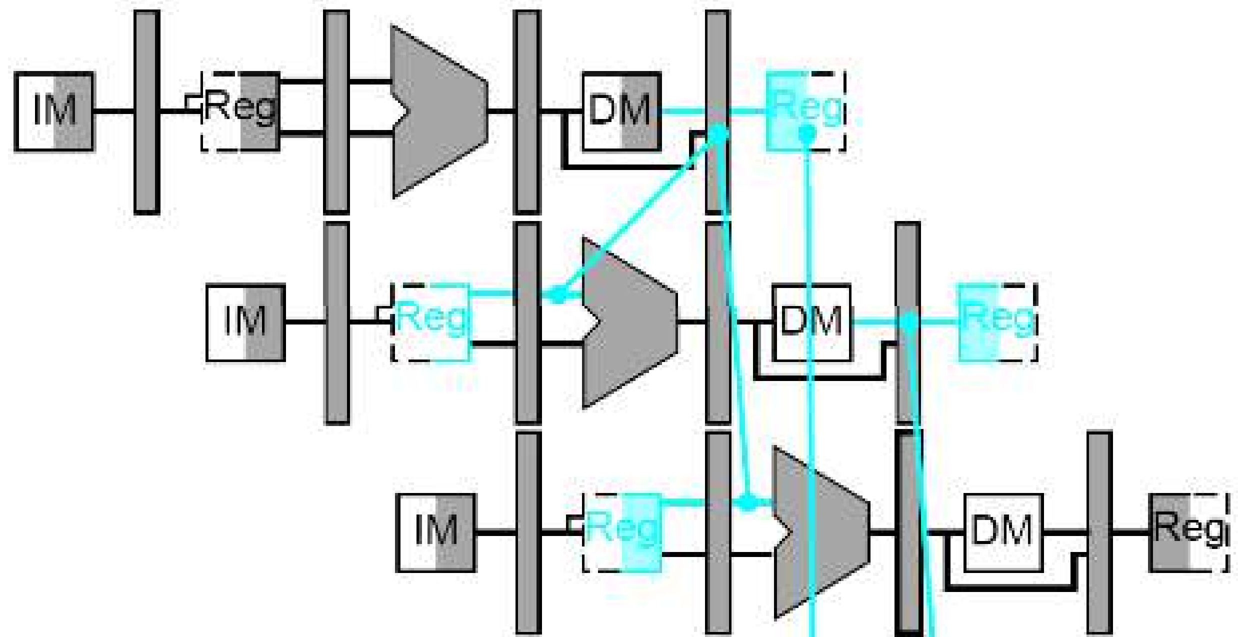


Source: H&P textbook

Example 2

lw R1, 8(R2)

lw R4, 8(R1)

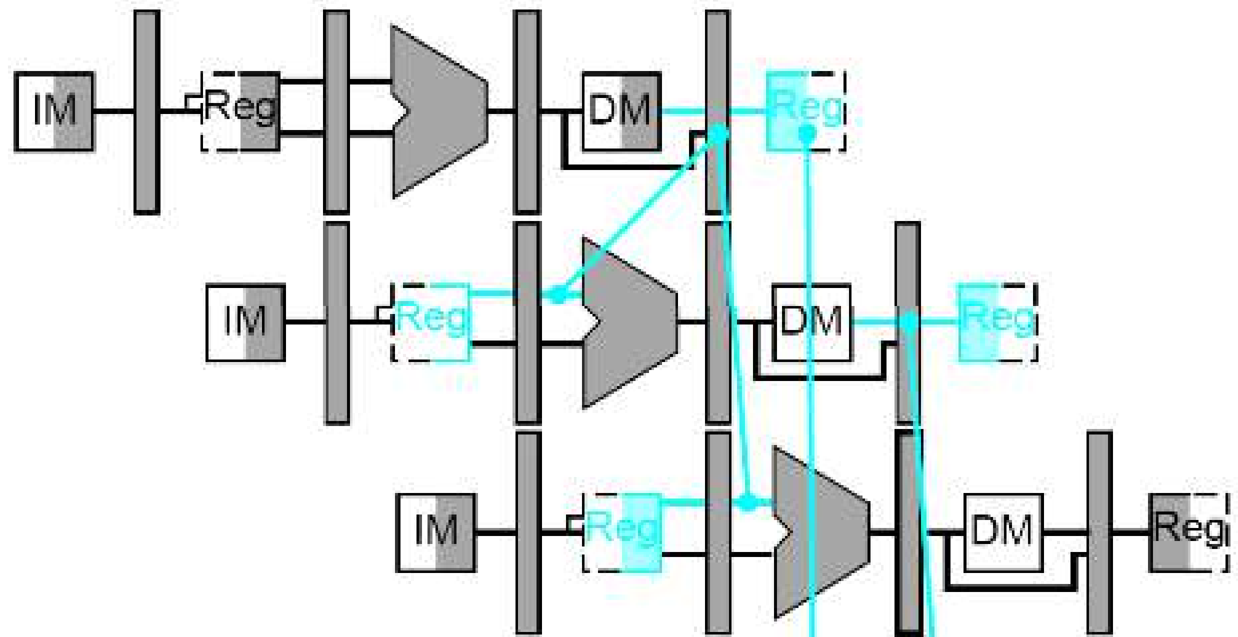


Source: H&P textbook

Example 3

lw R1, 8(R2)

sw R1, 8(R3)



Source: H&P textbook

Example 4

- Show the instruction occupying each stage in each cycle (no bypassing) if I1 is R1+R2→R3 and I2 is R3+R4→R5 and I3 is R7+R8→R9

CYC-1	CYC-2	CYC-3	CYC-4	CYC-5	CYC-6	CYC-7	CYC-8
IF	IF	IF	IF	IF	IF	IF	IF
D/R	D/R	D/R	D/R	D/R	D/R	D/R	D/R
ALU	ALU	ALU	ALU	ALU	ALU	ALU	ALU
DM	DM	DM	DM	DM	DM	DM	DM
RW	RW	RW	RW	RW	RW	RW	RW

Example 4

- Show the instruction occupying each stage in each cycle (no bypassing) if I1 is R1+R2→R3 and I2 is R3+R4→R5 and I3 is R7+R8→R9

	CYC-1	CYC-2	CYC-3	CYC-4	CYC-5	CYC-6	CYC-7	CYC-8
IF	I1	I2	I3	I3	I3	I4	I5	
D/R		I1	I2	I2	I2	I3	I4	
ALU			I1			I2	I3	
DM				I1			I2	I3
RW					I1			I2

Example 5

- Show the instruction occupying each stage in each cycle (with bypassing) if I1 is R1+R2→R3 and I2 is R3+R4→R5 and I3 is R3+R8→R9. Identify the input latch for each input operand.

CYC-1	CYC-2	CYC-3	CYC-4	CYC-5	CYC-6	CYC-7	CYC-8
IF	IF	IF	IF	IF	IF	IF	IF
D/R	D/R	D/R	D/R	D/R	D/R	D/R	D/R
ALU	ALU	ALU	ALU	ALU	ALU	ALU	ALU
DM	DM	DM	DM	DM	DM	DM	DM
RW	RW	RW	RW	RW	RW	RW	RW

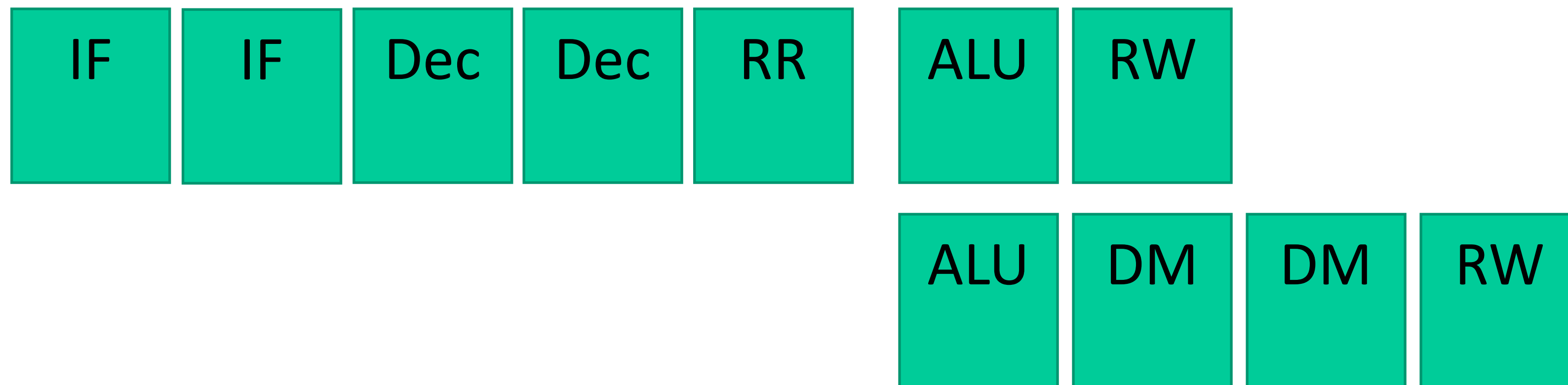
Example 5

- Show the instruction occupying each stage in each cycle (with bypassing) if I1 is R1+R2→R3 and I2 is R3+R4→R5 and I3 is R3+R8→R9. Identify the input latch for each input operand.

	CYC-1	CYC-2	CYC-3	CYC-4	CYC-5	CYC-6	CYC-7	CYC-8
IF	I1	I2	I3	I4	I5			
D/R		I1	I2	I3	I4			
ALU			L3 L3 I1	L4 L3 I2	L5 L3 I3			
DM				I1	I2	I3		
RW					I1	I2	I3	

Example 6

A 7 or 9 stage pipeline



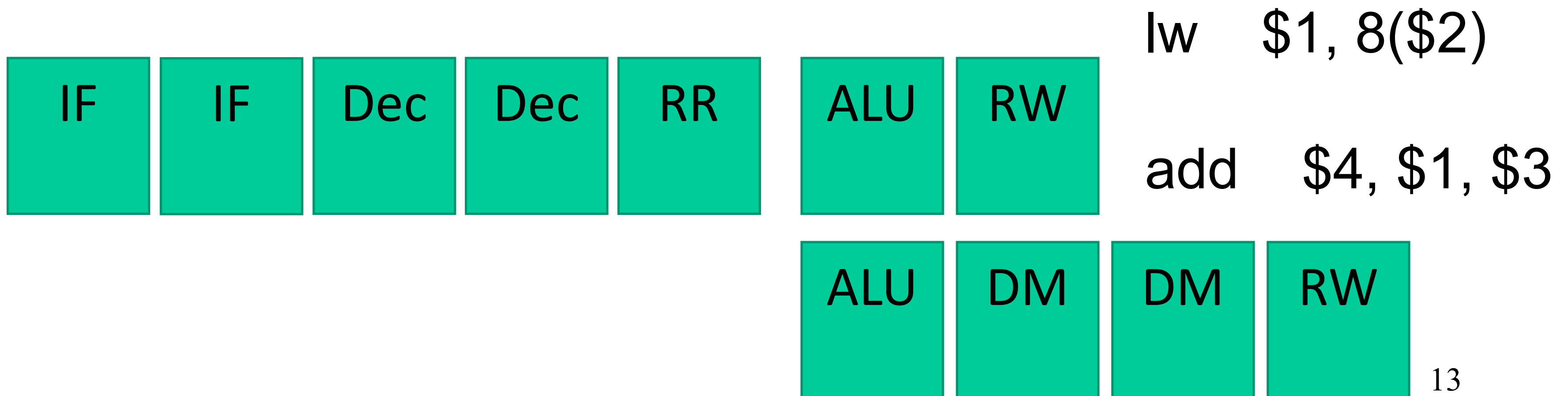
lw $1, 8($2)

add $4, $1, $3

Example 6

Without bypassing: 4 stalls
IF:IF:DE:DE:RR:AL:DM:DM:RW
IF: IF :DE:DE:DE:DE:DE :DE:RR:AL:RW

With bypassing: 2 stalls
IF:IF:DE:DE:RR:AL:DM:DM:RW
IF: IF :DE:DE:DE:DE:RR :AL:RW

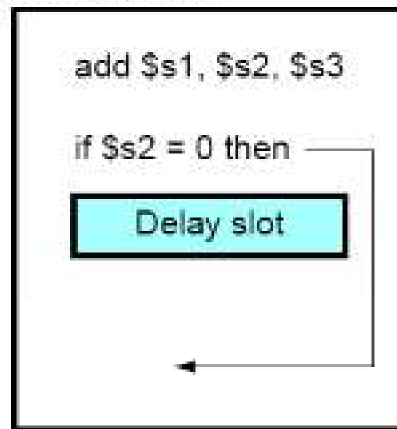


Control Hazards

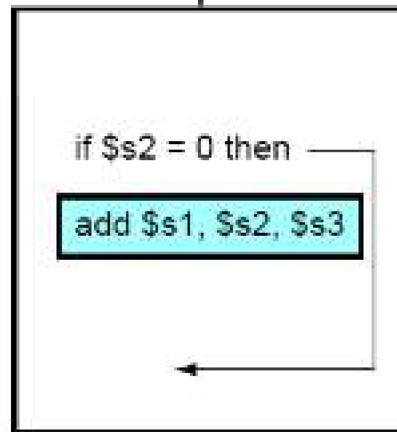
- Simple techniques to handle control hazard stalls:
  - for every branch, introduce a stall cycle (note: every 6th instruction is a branch!)
  - assume the branch is not taken and start fetching the next instruction – if the branch is taken, need hardware to cancel the effect of the wrong-path instruction
  - fetch the next instruction (branch delay slot) and execute it anyway – if the instruction turns out to be on the correct path, useful work was done – if the instruction turns out to be on the wrong path, hopefully program state is not lost
  - make a smarter guess and fetch instructions from the expected target

Branch Delay Slots

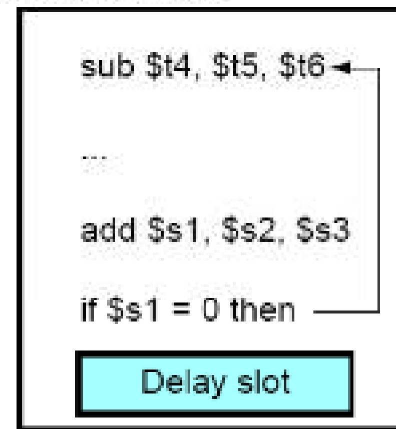
a. From before



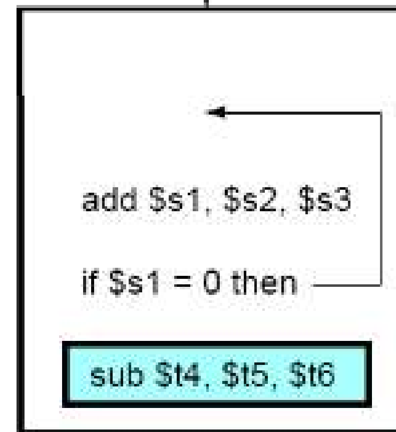
Becomes



b. From target



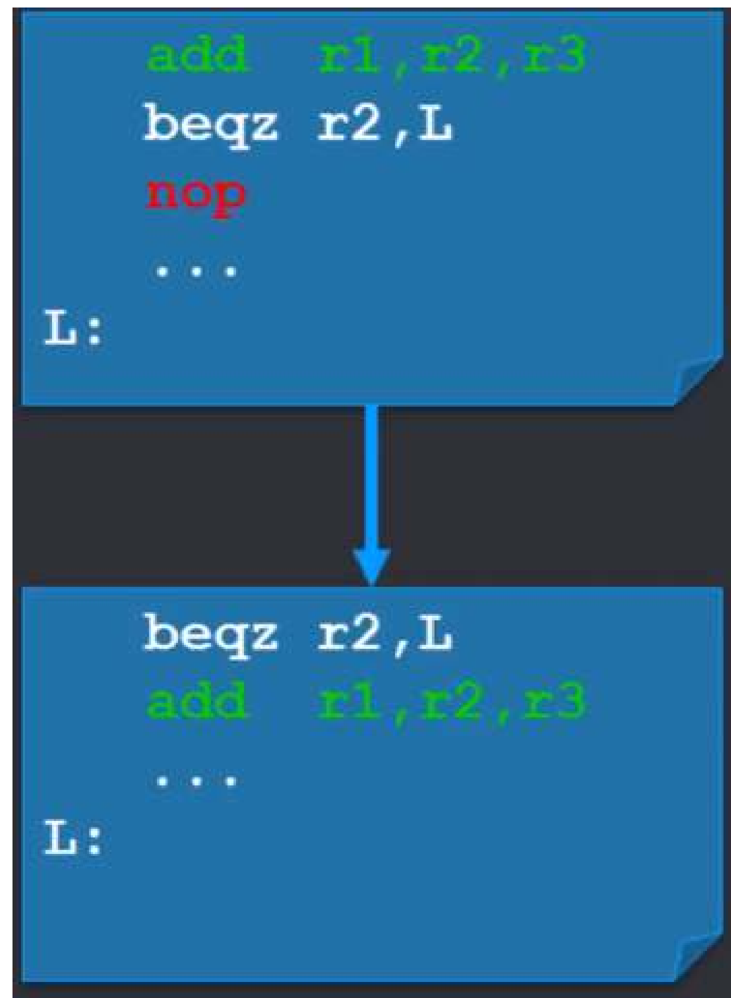
Becomes



Source: H&P textbook

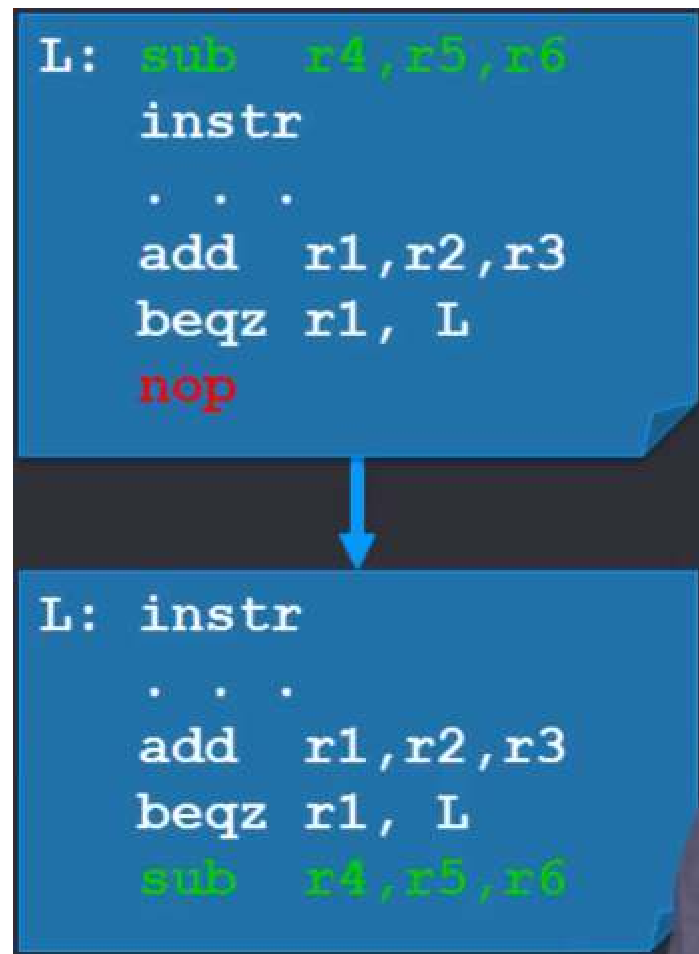
Scheduling branch delay slot: from before

- Always improves performance



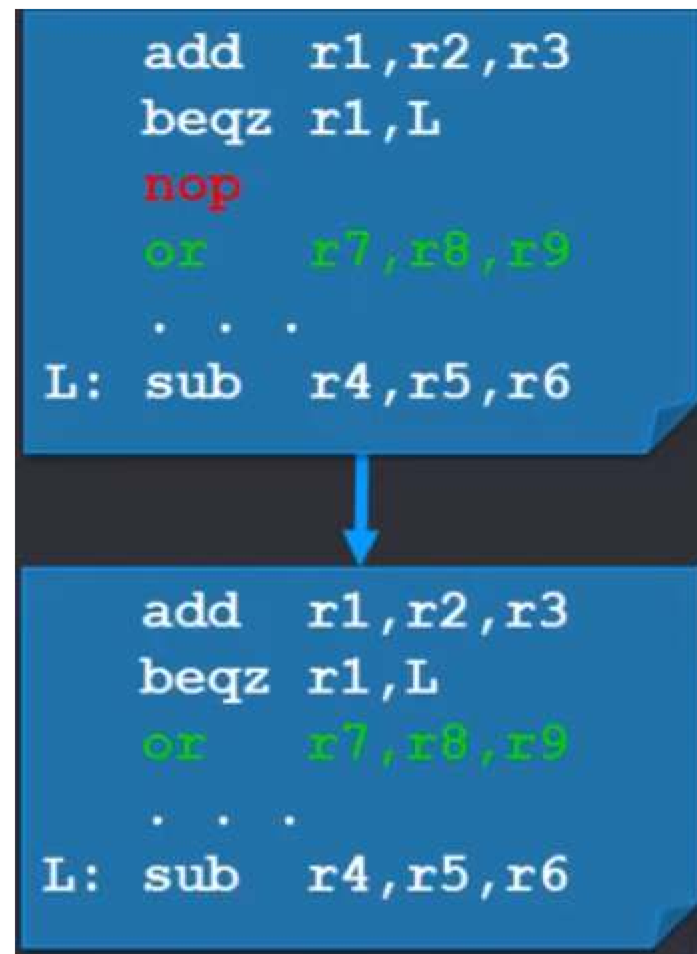
Scheduling branch delay slot: from target

- improves performance when branch is taken
- Must be ok to execute the inst if the branch is not taken



Scheduling branch delay slot: from the fall through

- improves performance when branch is not taken
- Must be ok to execute the inst if the branch is taken



Project:

Dheya Mustafa

HU Computer

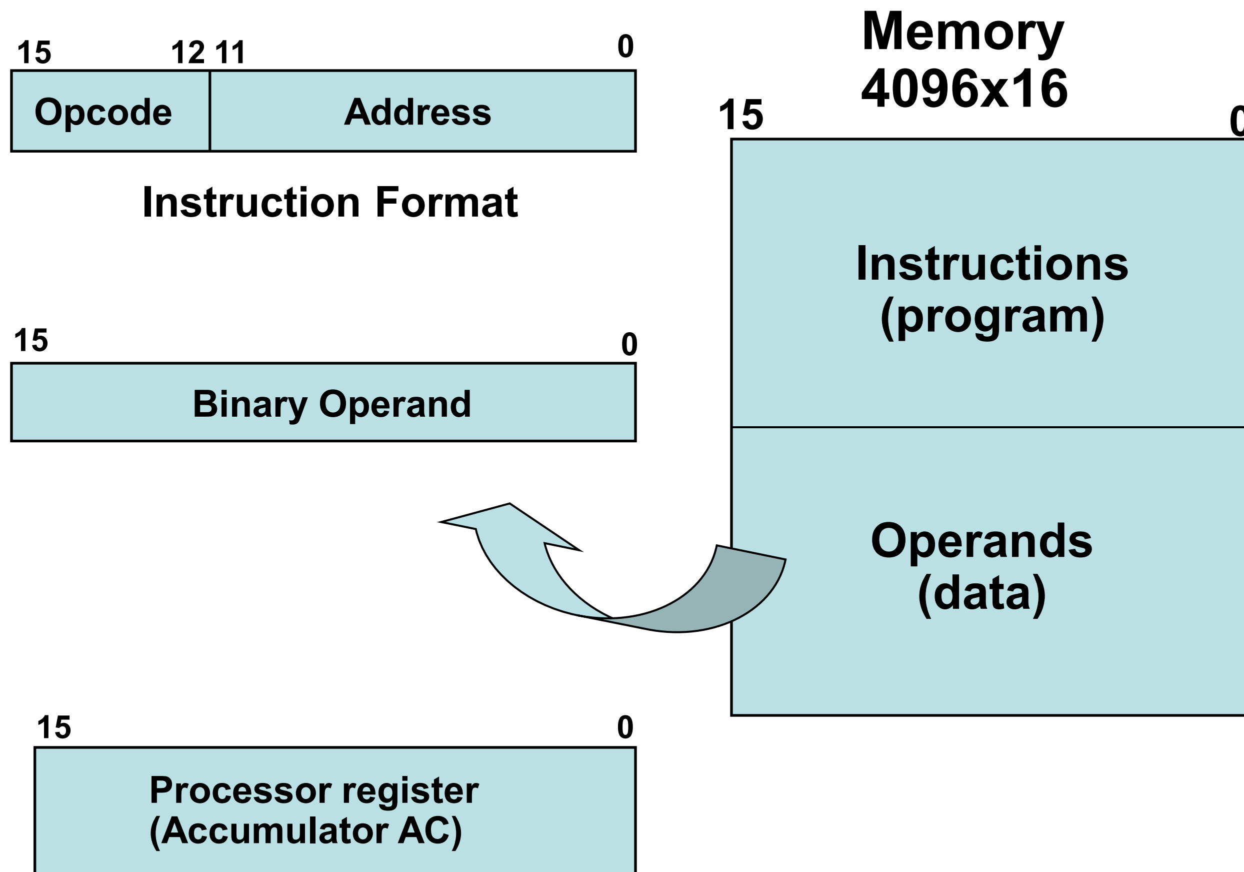
HU architecture

- Build your own computer architecture

Ask Questions:

- What is the memory model(size,address,data, Harvard or Princeton)
- What is the instruction size; fixed or variable
- What addressing modes we support, RISC,CISC
- What are the instructions formats, stack, accumelator two operand, one operand
- What instructions we support
- What registers we need

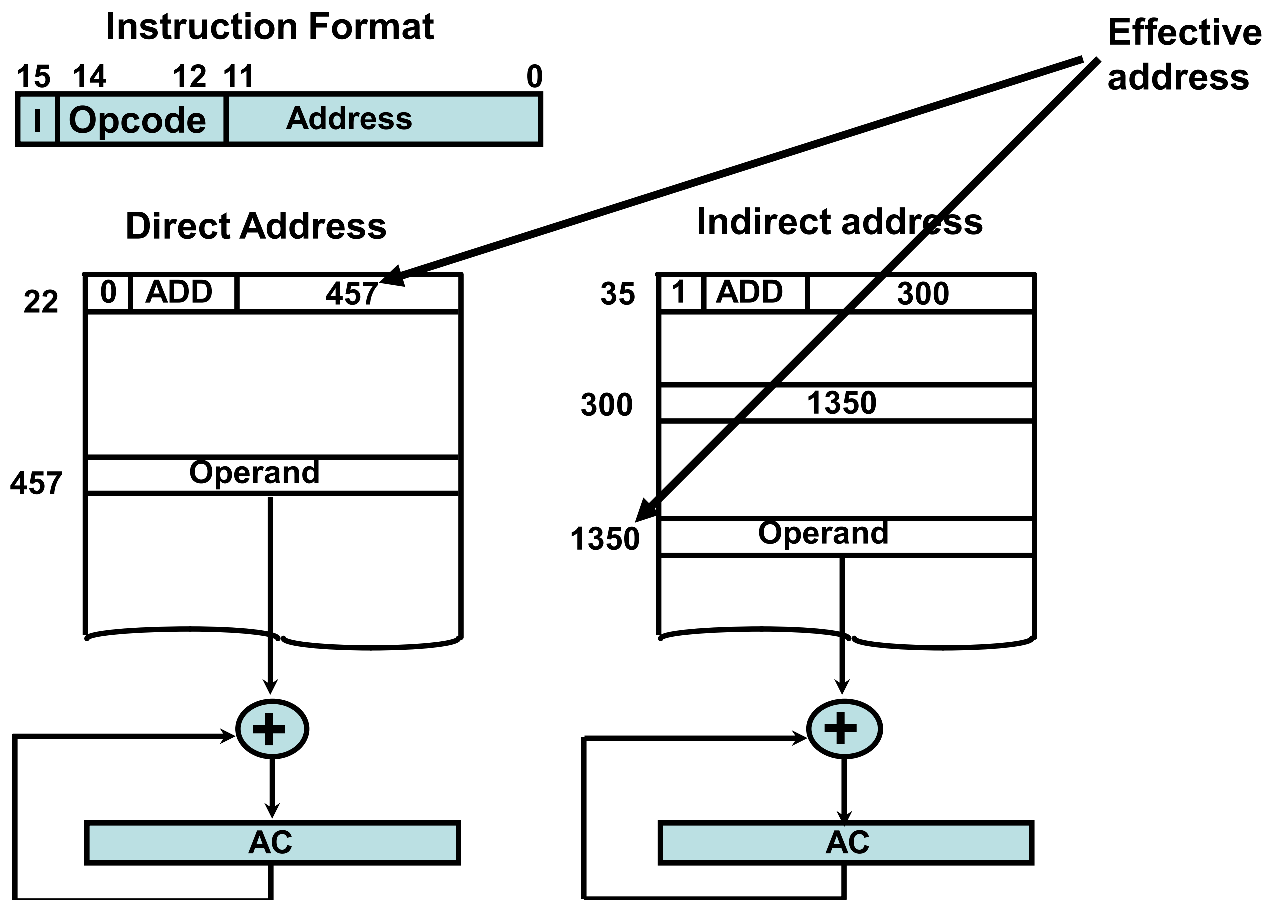
5-1 Instruction Codes Stored Program Organization cont.



5-1 Instruction Codes
Indirect Address

- There are three **Addressing Modes** used for address portion of the instruction code:
  - Immediate: the operand is given in the address portion (constant)
  - Direct: the address points to the operand stored in the memory
  - Indirect: the address points to the pointer (another address) stored in the memory that references the operand in memory
- One bit of the instruction code can be used to distinguish between direct & indirect addresses

5-1 Instruction Codes Indirect Address cont.



5-1 Instruction Codes Indirect Address cont.

- Effective address: the address of the operand in a computation-type instruction or the target address in a branch-type instruction
- The pointer can be placed in a processor register instead of memory as done in commercial computers

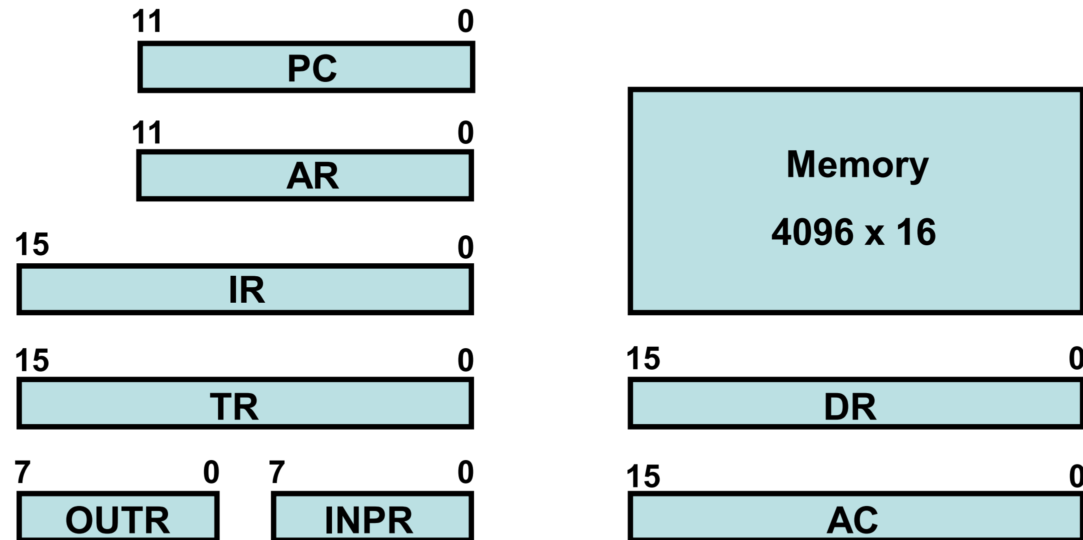
5-2 Computer Registers

- Computer instructions are normally stored in consecutive memory locations and executed sequentially one at a time
- The control reads an instruction from a specific address in memory and executes it, and so on
- This type of sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed

5-2 Computer Registers cont.

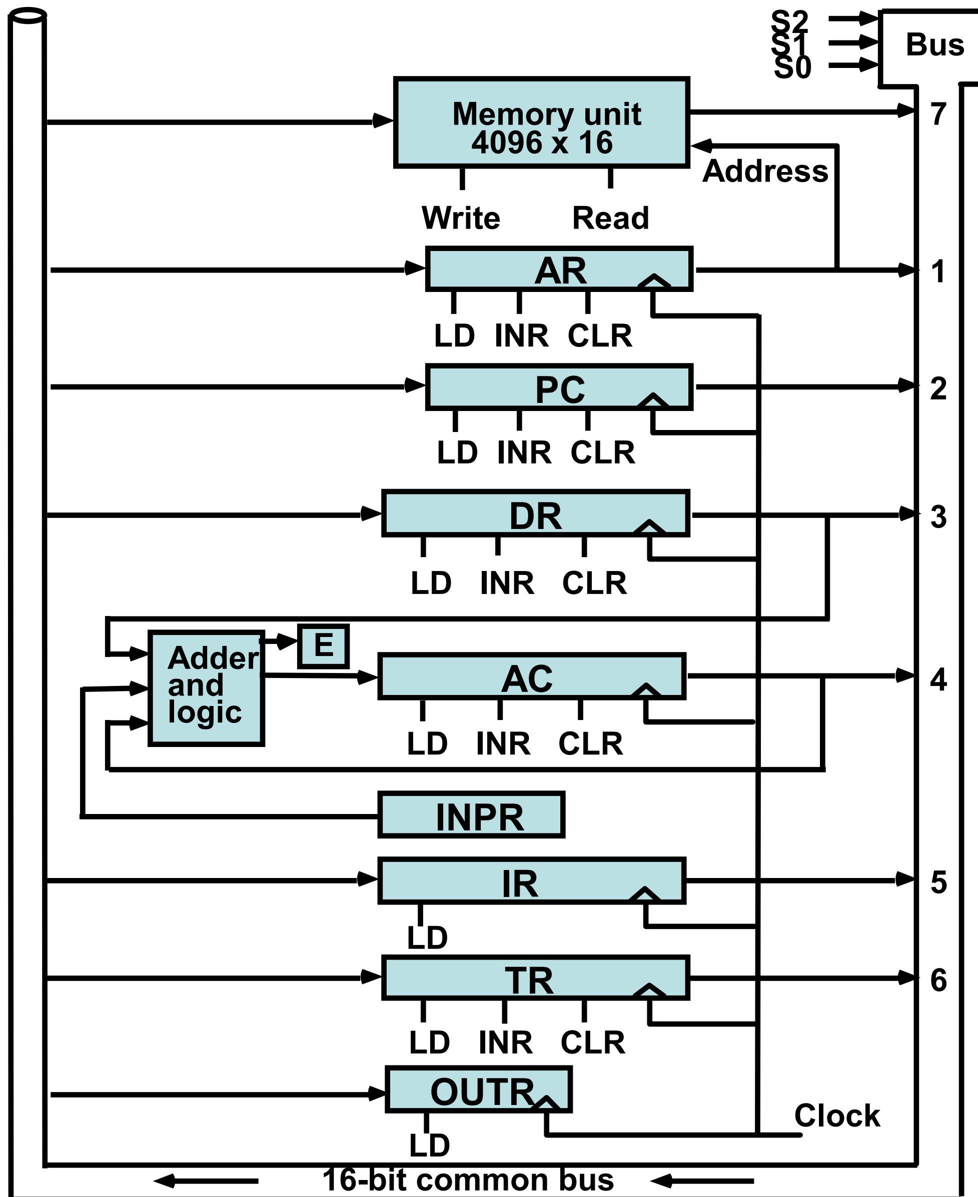
- It is also necessary to provide a register in the control unit for storing the instruction code after it is read from memory
- The computer needs processor registers for manipulating data and a register for holding a memory address

Registers in the Basic Computer



List of BC Registers

DR	16	Data Register	Holds memory operand
AR	12	Address Register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction Register	Holds instruction code
PC	12	Program Counter	Holds address of instruction
TR	16	Temporary Register	Holds temporary data
INPR	8	Input Register	Holds input character
OUTR	8	Output Register	Holds output character



Computer Registers Common Bus System

5-2 Computer Registers Common Bus System cont.

- $S_2S_1S_0$: Selects the register/memory that would use the bus
- LD (load): When enabled, the particular register receives the data from the bus during the next clock pulse transition
- E (extended AC bit): flip-flop holds the carry
- DR, AC, IR, and TR: have 16 bits each
- AR and PC: have 12 bits each since they hold a memory address

5-2 Computer Registers Common Bus System cont.

- When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to zeros
- When AR or PC receives information from the bus, only the 12 least significant bits are transferred into the register
- INPR and OUTR: communicate with the eight least significant bits in the bus

5-2 Computer Registers Common Bus System cont.

- INPR: Receives a character from the input device (keyboard,…etc) which is then transferred to AC
- OUTR: Receives a character from AC and delivers it to an output device (say a Monitor)
- Five registers have three control inputs: LD (load), INR (increment), and CLR (clear)
- Register ≡ binary counter with parallel load and synchronous clear

5-2 Computer Registers Memory Address

- The input data and output data of the memory are connected to the common bus
- But the memory address is connected to AR
- Therefore, AR must always be used to specify a memory address
- By using a single register for the address, we eliminate the need for an address bus that would have been needed otherwise

5-2 Computer Registers Memory Address cont.

- Register → Memory: Write operation
- Memory → Register: Read operation (note that AC cannot directly read from memory!!)
- Note that the content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle

5-2 Computer Registers Memory Address cont.

- The transition at the end of the cycle transfers the content of the bus into the destination register, and the output of the adder and logic circuit into the AC
- For example, the two microoperations

  $DR \leftarrow AC$ and $AC \leftarrow DR$ (Exchange)

  can be executed at the same time
- This is done by:

5-2 Computer Registers Memory Address cont.

- 1- place the contents of AC on the bus ($S_2S_1S_0$=100)
- 2- enabling the LD (load) input of DR
- 3- Transferring the contents of the DR through the adder and logic circuit into AC
- 4- enabling the LD (load) input of AC
- All during the same clock cycle
- The two transfers occur upon the arrival of the clock pulse transition at the end of the clock cycle

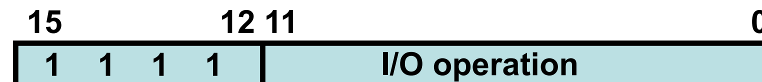
5-3 Computer Instructions

Basic Computer Instruction code format

Memory-Reference Instructions (OP-code = 000 ~ 110)

Register-Reference Instructions (OP-code = 111, I = 0)

Input-Output Instructions (OP-code =111, I = 1)



BASIC COMPUTER INSTRUCTIONS

Symbol	Hex Code I = 0	Hex Code I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

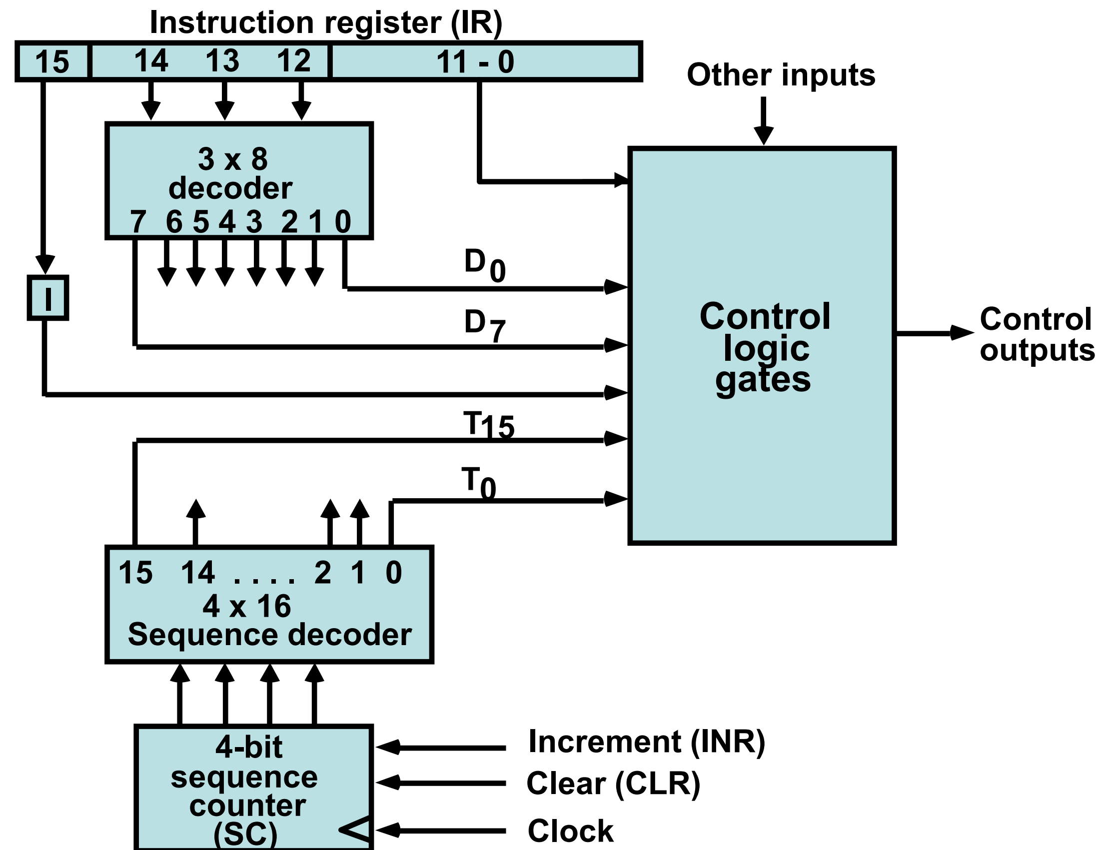
5-3 Computer Instructions
Instruction Set Completeness

- The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:
  - Arithmetic, logical, and shift instructions
  - Instructions for moving information to and from memory and processor registers
  - Program control instructions together with instructions that check status conditions
  - Input & output instructions

5-4 Timing & Control cont.

- In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits.
- In the microprogrammed organization, the control information is stored in a control memory (if the design is modified, the microprogram in control memory has to be updated)
- D_3T_4: $SC \leftarrow 0$

The Control Unit for the basic computer

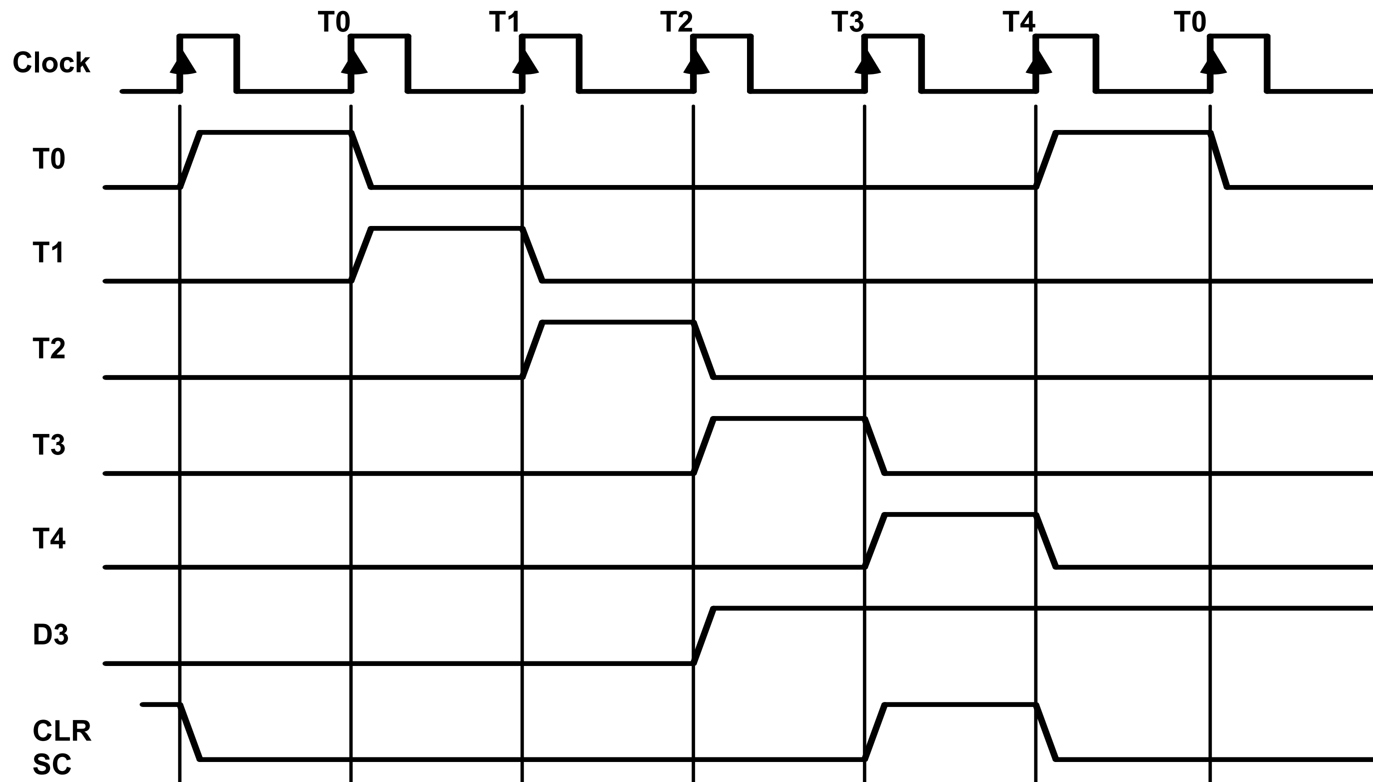


Hardwired Control Organization

- Generated by 4-bit sequence counter and 4x16 decoder
- The SC can be incremented or cleared.

- Example: T_0, T_1, T_2, T_3, T_4, T_0, T_1, . . .
  Assume: At time T_4, SC is cleared to 0 if decoder output D3 is active.

$$D_3T_4: SC \leftarrow 0$$



5-4 Timing & Control cont.

- A memory read or write cycle will be initiated with the rising edge of a timing signal
- Assume: memory cycle time **<** clock cycle time!
- So, a memory read or write cycle initiated by a timing signal will be completed by the time the next clock goes through its positive edge
- The clock transition will then be used to load the memory word into a register
- The memory cycle time is usually longer than the processor clock cycle → wait cycles

5-4 Timing & Control cont.

- T_0: AR←PC
  - Transfers the content of PC into AR if timing signal T_0 is active
  - T_0 is active during an entire clock cycle interval
  - During this time, the content of PC is placed onto the bus (with $S_2S_1S_0$=010) and the LD (load) input of AR is enabled
  - The actual transfer does not occur until the end of the clock cycle when the clock goes through a positive transition
  - This same positive clock transition increments the sequence counter SC from 0000 to 0001
  - The next clock cycle has T_1 active and T_0 inactive

5-5 Instruction Cycle

- A program is a sequence of instructions stored in memory
- The program is executed in the computer by going through a cycle for each instruction (in most cases)
- Each instruction in turn is subdivided into a sequence of sub-cycles or phases

5-5 Instruction Cycle cont.

- Instruction Cycle Phases:
  - 1- Fetch an instruction from memory
  - 2- Decode the instruction
  - 3- Read the effective address from memory if the instruction has an indirect address
  - 4- Execute the instruction
- This cycle repeats indefinitely unless a HALT instruction is encountered

5-5 Instruction Cycle Fetch and Decode

- Initially, the Program Counter (PC) is loaded with the address of the first instruction in the program
- The sequence counter SC is cleared to 0, providing a decoded timing signal T_0
- After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2, and so on

5-5 Instruction Cycle Fetch and Decode cont.

– T_0: AR←PC (this is essential!!)

The address of the instruction is moved to AR.

– T_1: IR←M[AR], PC←PC+1

The instruction is fetched from the memory to IR

,

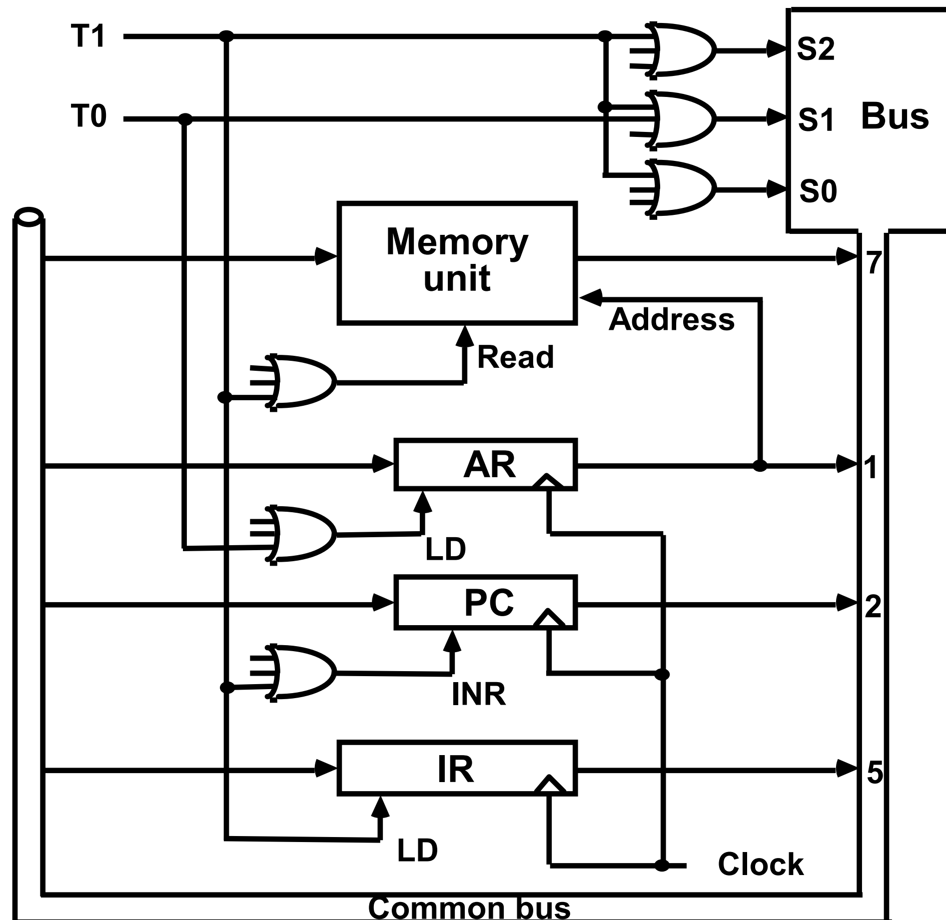
and the PC is incremented.

– T_2: D_0,…, D_7←Decode IR(12-14), AR←IR(0-11), I←IR(15)

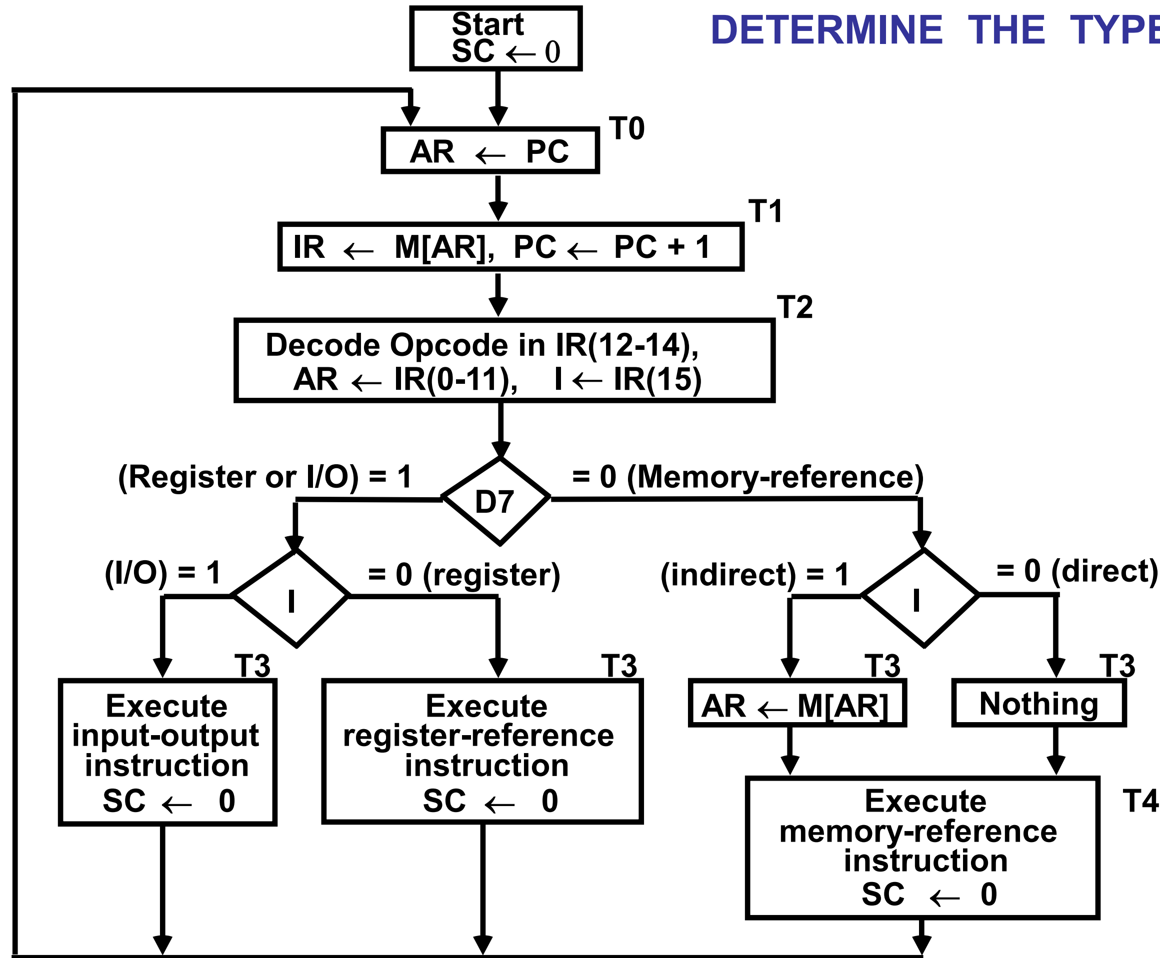
BC Instruction cycle: [Fetch Decode [Indirect] Execute]*

- **Fetch and Decode**

T0: AR ← PC (S0S1S2=010, T0=1)
T1: IR ← M [AR], PC ← PC + 1 (S0S1S2=111, T1=1)
T2: D0, . . . , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)



DETERMINE THE TYPE OF INSTRUCTION



D'_7IT_3: AR ← M[AR]

$D'_7I'T_3$: Nothing

$D_7I'T_3$: Execute a register-reference instr.

D_7IT_3: Execute an input-output instr.

REGISTER REFERENCE INSTRUCTIONS

Register Reference Instructions are identified when

- $D_7 = 1$, $I = 0$
- Register Ref. Instr. is specified in $B_0 \sim B_{11}$ of IR
- Execution starts with timing signal T_3

$r = D_7 I' T_3$ => Register Reference Instruction
$B_i = IR(i)$, $i=0,1,2,\ldots,11$, the ith bit of IR.

	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow AC'$
CME	rB_8:	$E \leftarrow E'$
CIR	rB_7:	$AC \leftarrow shr\ AC, AC(15) \leftarrow E, E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow shl\ AC, AC(0) \leftarrow E, E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	if $(AC(15) = 0)$ then $(PC \leftarrow PC+1)$
SNA	rB_3:	if $(AC(15) = 1)$ then $(PC \leftarrow PC+1)$
SZA	rB_2:	if $(AC = 0)$ then $(PC \leftarrow PC+1)$
SZE	rB_1:	if $(E = 0)$ then $(PC \leftarrow PC+1)$
HLT	rB_0:	$S \leftarrow 0$ (S is a start-stop flip-flop)

5.6 MEMORY REFERENCE INSTRUCTIONS

Symbol	Operation Decoder	Symbolic Description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR],\ E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC,\ PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, if $M[AR] + 1 = 0$ then $PC \leftarrow PC+1$

- The effective address of the instruction is in AR and was placed there during timing signal T_2 when I = 0, or during timing signal T3 when I = 1
- Memory cycle is assumed to be short enough to be completed in a CPU cycle
- The execution of MR Instruction starts with T_4

AND to AC

D_0T_4: $DR \leftarrow M[AR]$ — Read operand

D_0T_5: $AC \leftarrow AC \wedge DR,\ SC \leftarrow 0$ — AND with AC

ADD to AC

D_1T_4: $DR \leftarrow M[AR]$ — Read operand

D_1T_5: $AC \leftarrow AC + DR,\ E \leftarrow C_{out},\ SC \leftarrow 0$ — Add to AC and store carry in E

MEMORY REFERENCE INSTRUCTIONS cont.

LDA: Load to AC

D_2T_4: DR ← M[AR]

D_2T_5: AC ← DR, SC ← 0

STA: Store AC

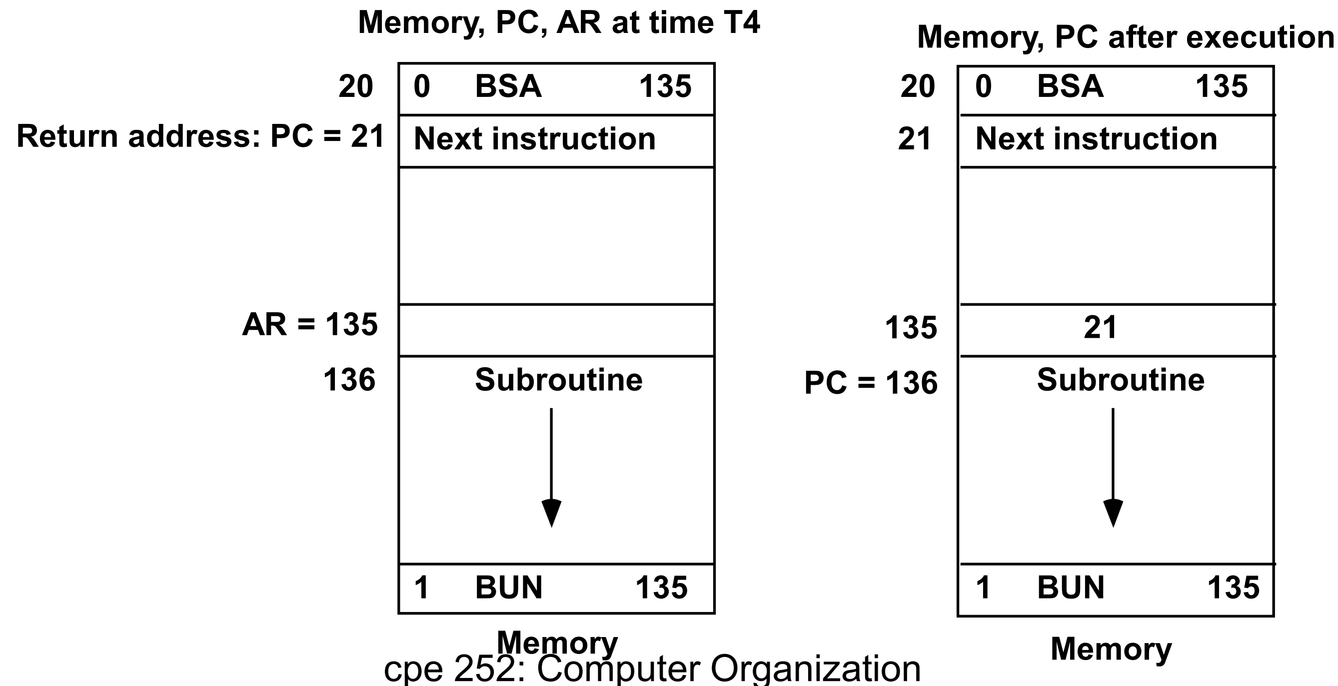
D_3T_4: M[AR] ← AC, SC ← 0

BUN: Branch Unconditionally

D_4T_4: PC ← AR, SC ← 0

BSA: Branch and Save Return Address

M[AR] ← PC, PC ← AR + 1



Memory Reference Instructions[cont.]

BSA: executed in a sequence of two micro-operations:

D_5T_4: M[AR] ← PC, AR ← AR + 1

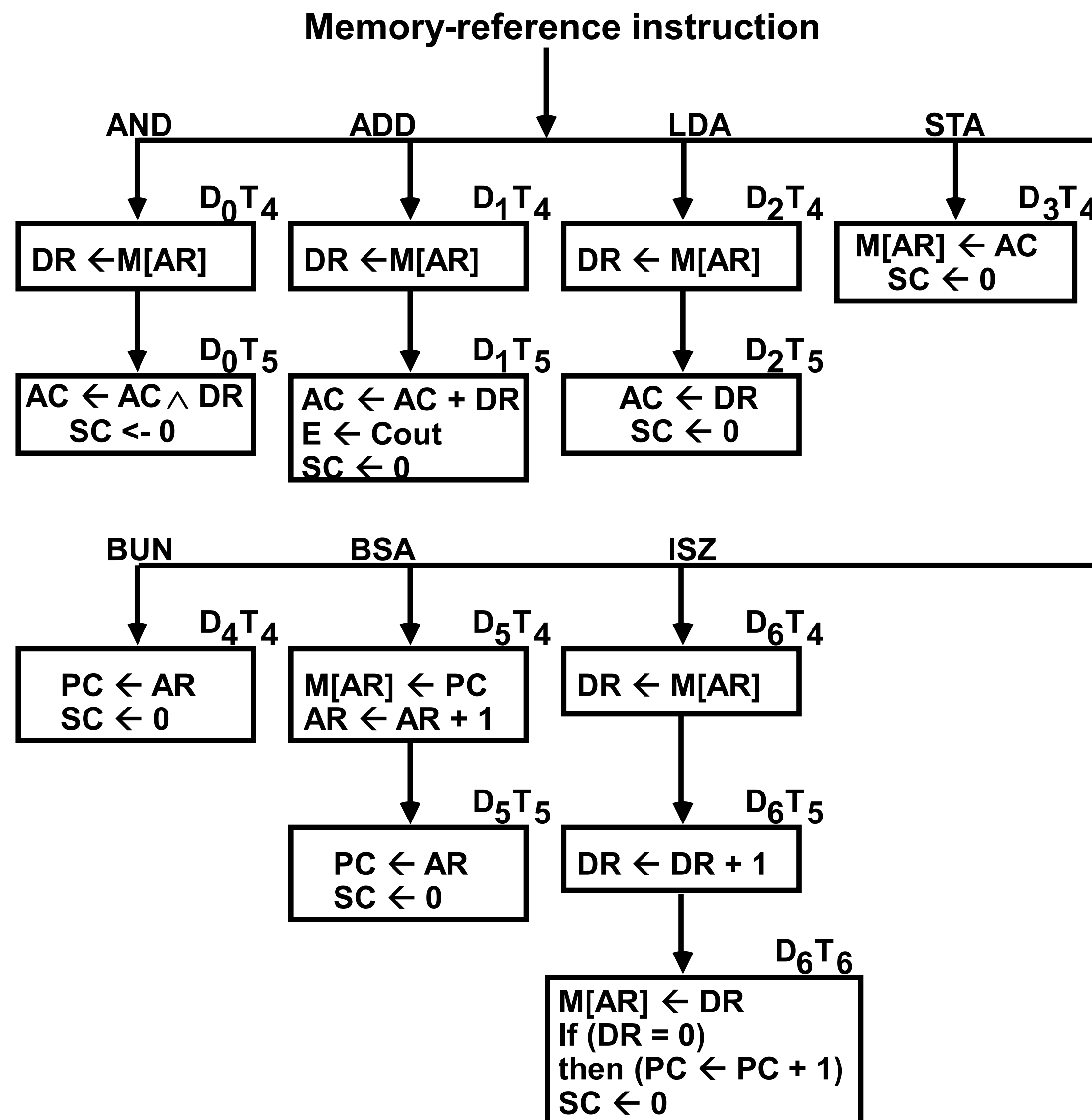
D_5T_5: PC ← AR, SC ← 0

ISZ: Increment and Skip-if-Zero

D_6T_4: DR ← M[AR]

D_6T_5: DR ← DR + 1

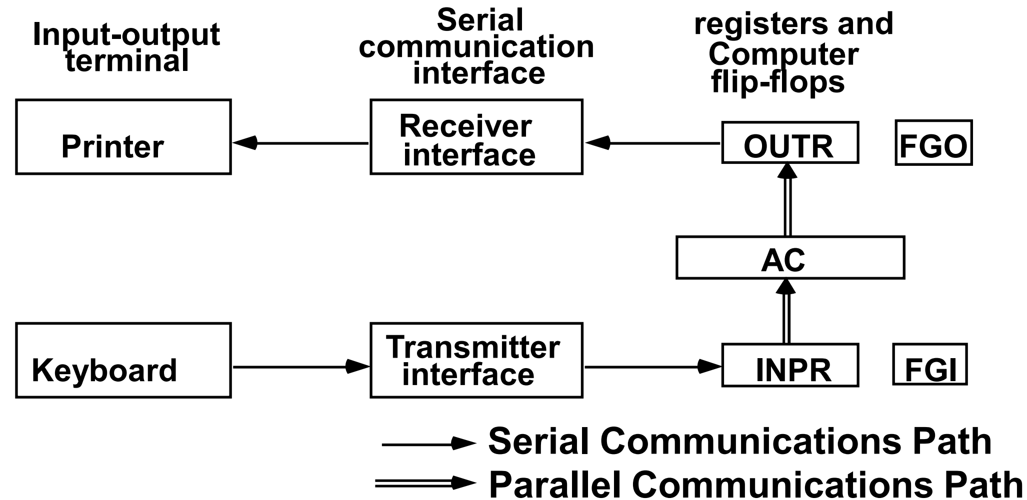
D_6T6: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0


Memory-reference instruction
AND
ADD
LDA
STA
D0T4
D1T4
D2T4
D3T4
DR ←M[AR]
DR ←M[AR]
DR ← M[AR]
M[AR] ← AC
SC ← 0
D0T5
D1T5
D2T5
AC ← AC ∧ DR
SC <- 0
AC ← AC + DR
E ← Cout
SC ← 0
AC ← DR
SC ← 0
BUN
BSA
ISZ
D4T4
D5T4
D6T4
PC ← AR
SC ← 0
M[AR] ← PC
AR ← AR + 1
DR ← M[AR]
D5T5
D6T5
PC ← AR
SC ← 0
DR ← DR + 1
D6T6
M[AR] ← DR
If (DR = 0)
then (PC ← PC + 1)
SC ← 0

5-7 Input-Output and Interrupt

- Instructions and data stored in memory must come from some input device
- Computational results must be transmitted to the user through some output device
- For the system to communicate with an input device, serial information is shifted into the input register INPR
- To output information, it is stored in the output register OUTR

5-7 Input-Output and Interrupt cont.



5-7 Input-Output and Interrupt[cont.]

- INPR and OUTR communicate with a communication interface serially and with the AC in parallel. They hold an 8-bit alphanumeric information
- I/O devices are slower than a computer system → we need to synchronize the timing rate difference between the input/output device and the computer.
- FGI: 1-bit input flag (Flip-Flop) aimed to control the input operation

5-7 Input-Output and Interrupt cont.

- FGI is set to 1 when a new information is available in the input device and is cleared to 0 when the information is accepted by the computer
- FGO: 1-bit output flag used as a control flip-flop to control the output operation
- If FGO is set to 1, then this means that the computer can send out the information from AC. If it is 0, then the output device is busy and the computer has to wait!

5-7 Input-Output and Interrupt cont.

- The process of input information transfer:
  - Initially, FGI is cleared to 0
  - An 8-bit alphanumeric code is shifted into INPR (Keyboard key strike) and the input flag FGI is set to 1
  - As long as the flag is set, the information in INPR cannot be changed by another data entry
  - The computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0

5-7 Input-Output and Interrupt^cont.

- Once the flag is cleared, new information can be shifted into INPR by the input device (striking another key)

- The process of outputting information:
  - Initially, the output flag FGO is set to 1
  - The computer checks the flag bit; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0
  - The output accepts the coded information (prints the corresponding character)

5-7 Input-Output and Interrupt[cont.]

- When the operation is completed, the output device sets FGO back to 1
- The computer does not load a new data information into OUTR when FGO is 0 because this condition indicates that the output device is busy to receive another information at the moment!!

Input-Output Instructions

- Needed for:
  - Transferring information to and from AC register
  - Checking the flag bits
  - Controlling the interrupt facility
- The control unit recognize it when D_7=1 and I = 1
- The remaining bits of the instruction specify the particular operation
- Executed with the clock transition associated with timing signal T_3
- Input-Output instructions are summarized next

Input-Output Instructions

$D_7IT_3 = p$
$IR(i) = B_i$, $i = 6, \ldots, 11$

INP	**pB_{11}: $AC(0\text{-}7) \leftarrow INPR$, $FGI \leftarrow 0$**	**Input char. to AC**
OUT	**pB_{10}: $OUTR \leftarrow AC(0\text{-}7)$, $FGO \leftarrow 0$**	**Output char. from AC**
SKI	**pB_9: if(FGI = 1) then ($PC \leftarrow PC + 1$)**	**Skip on input flag**
SKO	**pB_8: if(FGO = 1) then ($PC \leftarrow PC + 1$)**	**Skip on output flag**
ION	**pB_7: $IEN \leftarrow 1$**	**Interrupt enable on**
IOF	**pB_6: $IEN \leftarrow 0$**	**Interrupt enable off**

Program Interrupt

- The process of communication just described is referred to as ***Programmed Control Transfer***
- The computer keeps checking the flag bit, and when it finds it set, it initiates an information transform (this is sometimes called ***Polling***)
- This type of transfer is in-efficient due to the difference of information flow rate between the computer and the I/O device

Program Interrupt cont.

- The computer is wasting time while checking the flag instead of doing some other useful processing task
- An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer
- This type of transfer uses the interrupt facility

Program Interrupt[cont.]

- While the computer is running a program, it does not check the flags
- Instead:
  - When a flag is set, the computer is immediately interrupted from proceeding with the current program

Program Interrupt$^{cont.}$

- The computer stops what it is doing to take care of the input or output transfer
- Then, it returns to the current program to continue what it was doing before the interrupt

- The interrupt facility can be enabled or disabled via a flip-flop called IEN
- The interrupt enable flip-flop IEN can be set and cleared with two instructions (IOF, ION):
  - IOF: IEN $\leftarrow$ 0 (the computer cannot be interrupted)
  - ION: IEN $\leftarrow$ 1 (the computer can be interrupted)

Program Interrupt[cont.]

- Another flip-flop (called the interrupt flip-flop **R**) is used in the computer's interrupt facility to decide when to go through the interrupt cycle
- **FGI** and **FGO** are different here compared to the way they acted in an earlier discussion!!
- So, the computer is either in an **Instruction Cycle** or in an **Interrupt Cycle**

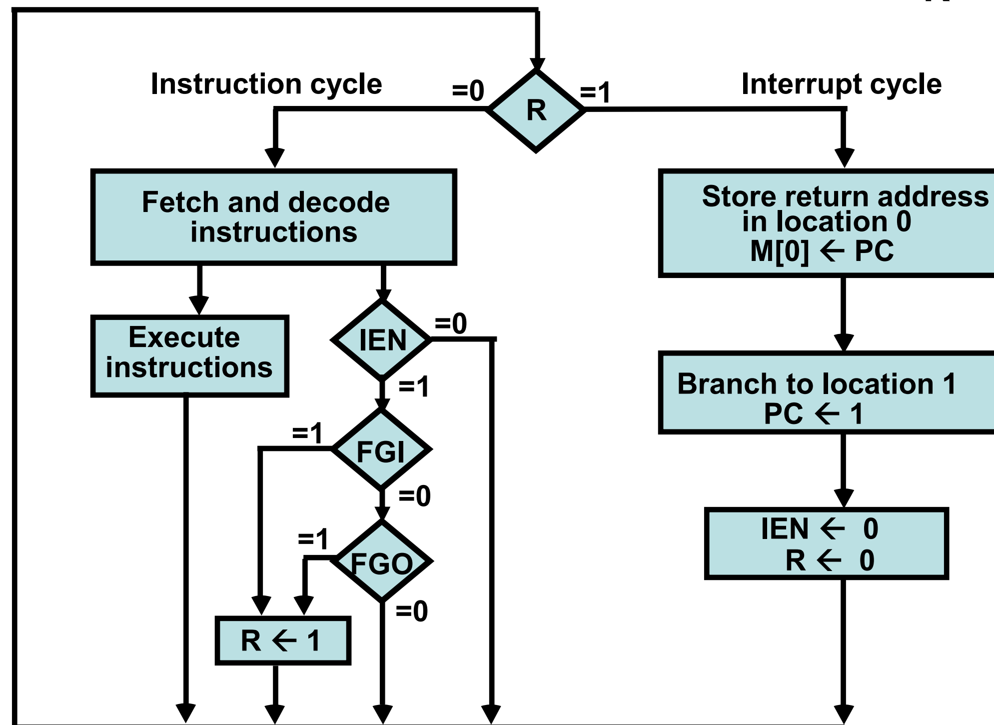
Program Interrupt[cont.]

- The interrupt cycle is a hardware implementation of a branch and save return address operation (BSA)
- The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted
- This location may be a processor register, a memory stack, or a specific memory location

Program Interrupt$^{cont.}$

- For our computer, we choose the memory location at address 0 as a place for storing the return address
- Control then inserts address 1 into PC: this means that the first instruction of the interrupt service routine should be stored in memory at address 1, or, the programmer must store a branch instruction that sends the control to an interrupt service routine!!

Program Interrupt$^{cont.}$



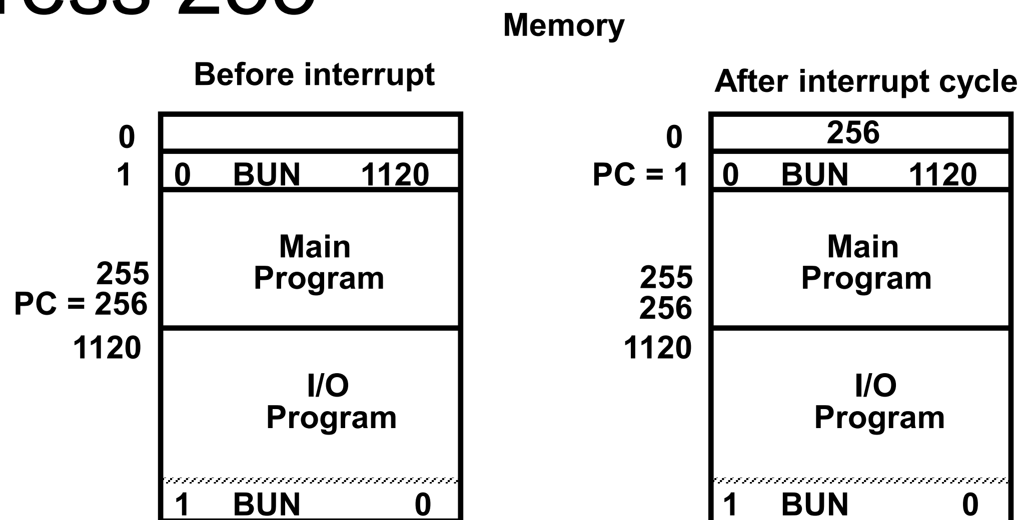
Flowchart for interrupt cycle

Program Interrupt[cont.]

- IEN, R ← 0: no more interruptions can occur until the interrupt request from the flag has been serviced
- The service routine must end with an instruction that re-enables the interrupt (IEN ← 1) and an instruction to return to the instruction at which the interrupt occurred
- The instruction that returns the control to the original program is "indirect BUN 0"

Program Interrupt$^{cont.}$

- Example: the computer is interrupted during execution of the instruction at address 255

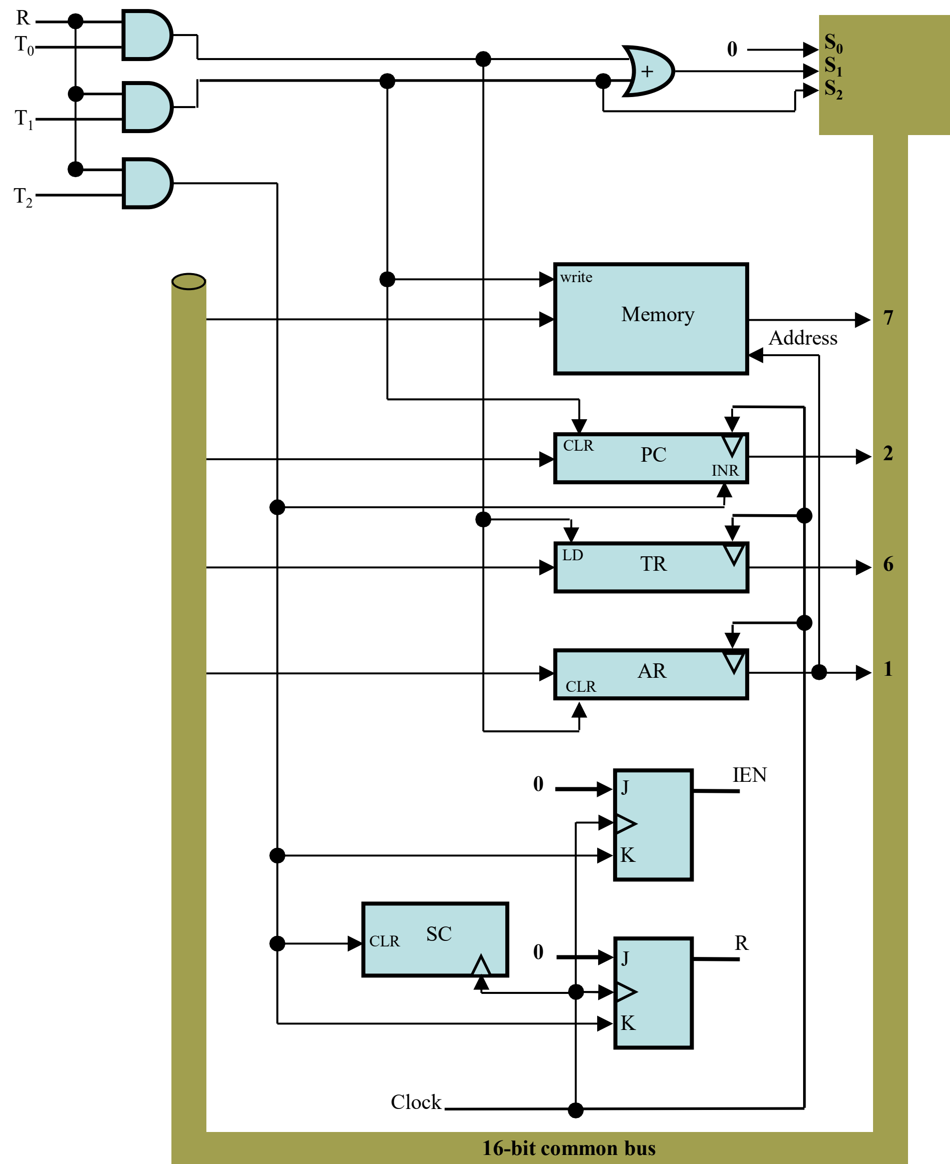


Interrupt Cycle

- The fetch and decode phases of the instruction cycle must be :

(Replace T0, T1, T2 → R'T0, R'T1, R'T2 (fetch and decode phases occur at the instruction cycle when R = 0)

- Interrupt Cycle:
  - **RT_0: AR ← 0, TR ← PC**
  - **RT_1: M[AR] ← TR, PC ← 0**
  - **RT_2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0**



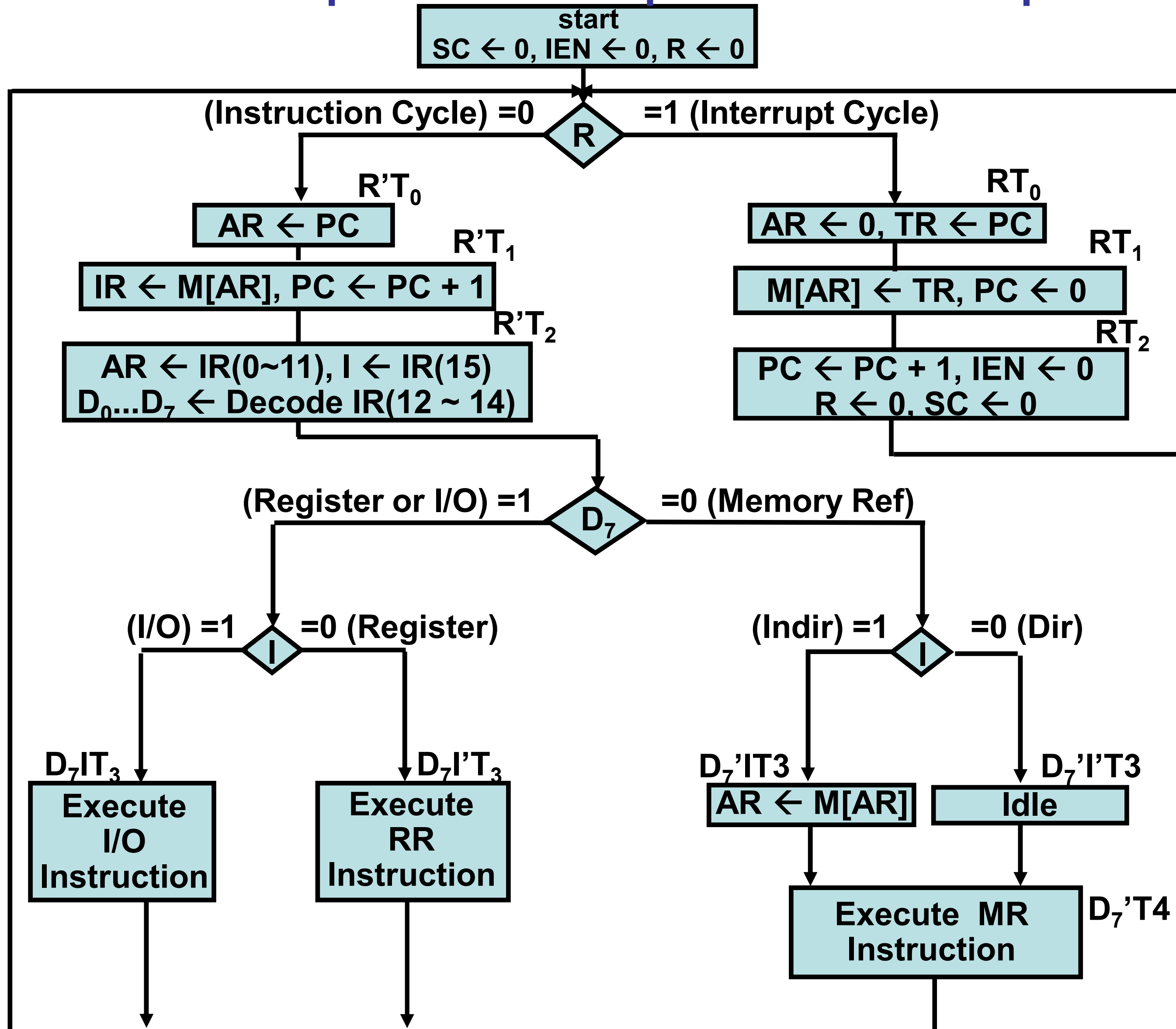
Register transfers for the Interrupt Cycle

Interrupt cont.

- **Further Questions:**
  - **How can the CPU recognize the device requesting an interrupt?**
  - **Since different devices are likely to require different interrupt service routines, how can the CPU obtain the starting address of the appropriate routine in each case?**
  - **Should any device be allowed to interrupt the CPU while another interrupt is being serviced?**
  - **How can the situation be handled when two or more interrupt requests occur simultaneously?**

5-8 Complete Computer Description

Fig 5-15



5-8 Complete Computer Description cont.

Fetch	R'T0:	AR ← PC
	R'T1:	IR ← M[AR], PC ← PC + 1
Decode	R'T2:	D0, ..., D7 ← Decode IR(12 ~ 14), AR ← IR(0 ~ 11), I ← IR(15)
Indirect	D7'IT3:	AR ← M[AR]
Interrupt:		
T0'T1'T2'(IEN)(FGI + FGO):		R ← 1
	RT0:	AR ← 0, TR ← PC
	RT1:	M[AR] ← TR, PC ← 0
	RT2:	PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0
Memory-Reference:		
AND	D0T4:	DR ← M[AR]
	D0T5:	AC ← AC . DR, SC ← 0
ADD	D1T4:	DR ← M[AR]
	D1T5:	AC ← AC + DR, E ← Cout, SC ← 0
LDA	D2T4:	DR ← M[AR]
	D2T5:	AC ← DR, SC ← 0
STA	D3T4:	M[AR] ← AC, SC ← 0
BUN	D4T4:	PC ← AR, SC ← 0
BSA	D5T4:	M[AR] ← PC, AR ← AR + 1
	D5T5:	PC ← AR, SC ← 0
ISZ	D6T4:	DR ← M[AR]
	D6T5:	DR ← DR + 1
	D6T6:	M[AR] ← DR, if(DR=0) then (PC ← PC + 1), SC ← 0

5-8 Complete Computer Description cont.

Register-Reference:

	D7I'T3 = r	(Common to all register-reference instructions)
	IR(i) = Bi	(i = 0,1,2, ..., 11)
	r:	SC ← 0
CLA	rB11:	AC ← 0
CLE	rB10:	E ← 0
CMA	rB9:	AC ← AC'
CME	rB8:	E ← E'
CIR	rB7:	AC ← shr AC, AC(15) ← E, E ← AC(0)
CIL	rB6:	AC ← shl AC, AC(0) ← E, E ← AC(15)
INC	rB5:	AC ← AC + 1
SPA	rB4:	If(AC(15) =0) then (PC ← PC + 1)
SNA	rB3:	If(AC(15) =1) then (PC ← PC + 1)
SZA	rB2:	If(AC = 0) then (PC ← PC + 1)
SZE	rB1:	If(E=0) then (PC ← PC + 1)
HLT	rB0:	S ← 0

Input-Output:

	D7IT3 = p	(Common to all input-output instructions)
	IR(i) = Bi	(i = 6,7,8,9,10,11)
	p:	SC ← 0
INP	pB11:	AC(0-7) ← INPR, FGI ← 0
OUT	pB10:	OUTR ← AC(0-7), FGO ← 0
SKI	pB9:	If(FGI=1) then (PC ← PC + 1)
SKO	pB8:	If(FGO=1) then (PC ← PC + 1)
ION	pB7:	IEN ← 1
IOF	pB6:	IEN ← 0

Table 5-6

5-9 Design of Basic Computer

1. **A memory unit: 4096 x 16.**
2. **Registers: AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC**
3. **Flip-Flops (Status): I, S, E, R, IEN, FGI, and FGO**
4. **Decoders:**
   1. **a 3x8 Opcode decoder**
   2. **a 4x16 timing decoder**
5. **Common bus: 16 bits**
6. **Control logic gates**
7. **Adder and Logic circuit: Connected to AC**

5-9 Design of Basic Computer[cont.]

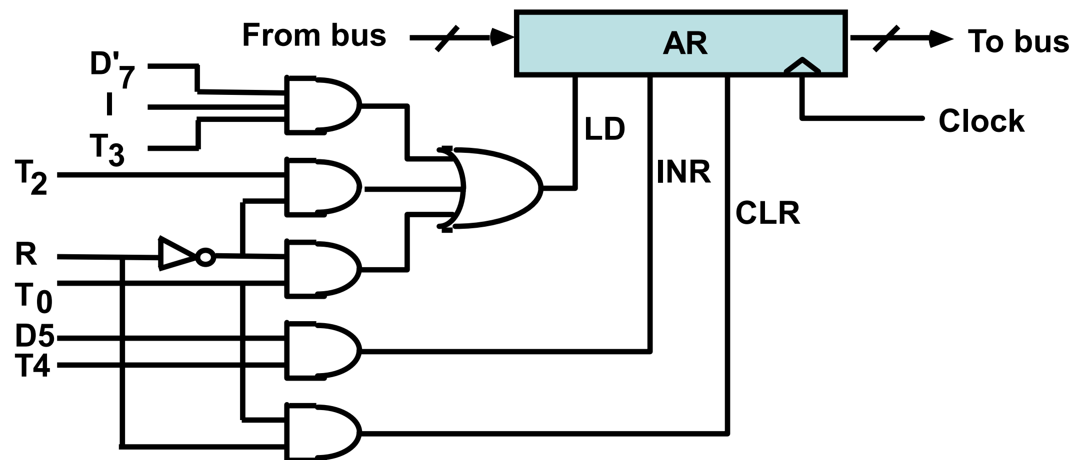
- The control logic gates are used to control:
  - Inputs of the nine registers
  - Read and Write inputs of memory
  - Set, Clear, or Complement inputs of the flip-flops
  - S2, S1, S0 that select a register for the bus
  - AC Adder and Logic circuit

5-9 Design of Basic Computer[cont.]

- Control of registers and memory
  - The control inputs of the registers are LD (load), INR (increment), and CLR (clear)
  - To control AR We scan table 5-6 to find out all the statements that change the content of AR:
    - **R'T0: AR ← PC LD(AR)**
    - **R'T2: AR ← IR(0-11) LD(AR)**
    - **D'7IT3: AR ← M[AR] LD(AR)**
    - **RT0: AR ← 0 CLR(AR)**
    - **D5T4: AR ← AR + 1 INR(AR)**

5-9 Design of Basic Computer$^{cont.}$

Control Gates associated with AR



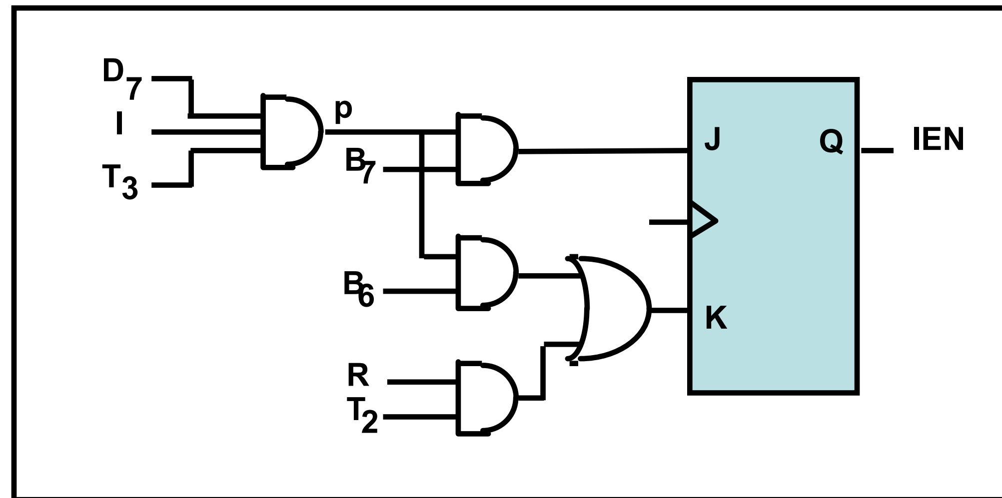
5-9 Design of Basic Computer[cont.]

- To control the Read input of the memory we scan the table again to get these:
  - D_0T_4: DR ← M[AR]
  - D_1T_4: DR ← M[AR]
  - D_2T_4: DR ← M[AR]
  - D_6T_4: DR ← M[AR]
  - $D_7'IT_3$: AR ← M[AR]
  - $R'T_1$: IR ← M[AR]
- → $Read = R'T_1 + D_7'IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$

5-9 Design of Basic Computer$^{cont.}$

- Control of Single Flip-flops (IEN for example)
  - **pB7: IEN ← 1 (I/O Instruction)**
  - **pB6: IEN ← 0 (I/O Instruction)**
  - **RT2: IEN ← 0 (Interrupt)**
    - **where p = D7IT3 (Input/Output Instruction)**
  - If we use a JK flip-flop for IEN, the control gate logic will be as shown in the following slide:

5-9 Design of Basic Computer$^{cont.}$

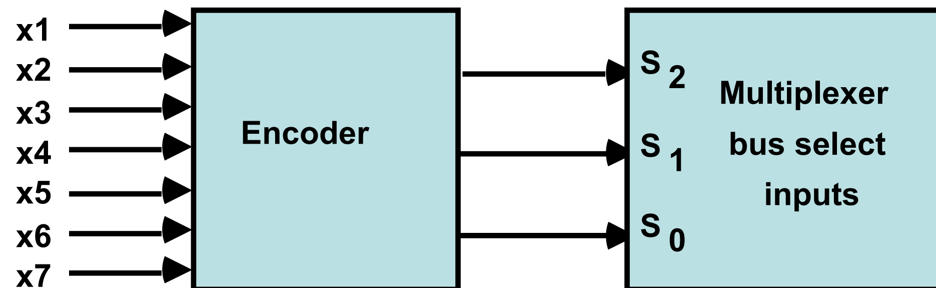


J	*K*	*Q(t+1)*
0	0	Q(t)
0	1	0
1	0	1
1	1	Q'(t)

JK FF Characteristic Table

5-9 Design of Basic Computer cont.

- Control of Common bus is accomplished by placing an encoder at the inputs of the bus selection logic and implementing the logic for each encoder input



5-9 Design of Basic Computer$^{cont.}$

- To select AR on the bus then x_1 must be 1. This is happen when:
  - **D_4T_4: PC ← AR**
  - **D_5T_5: PC ← AR**
- $\Rightarrow x_1 = D_4T_4 + D_5T_5$

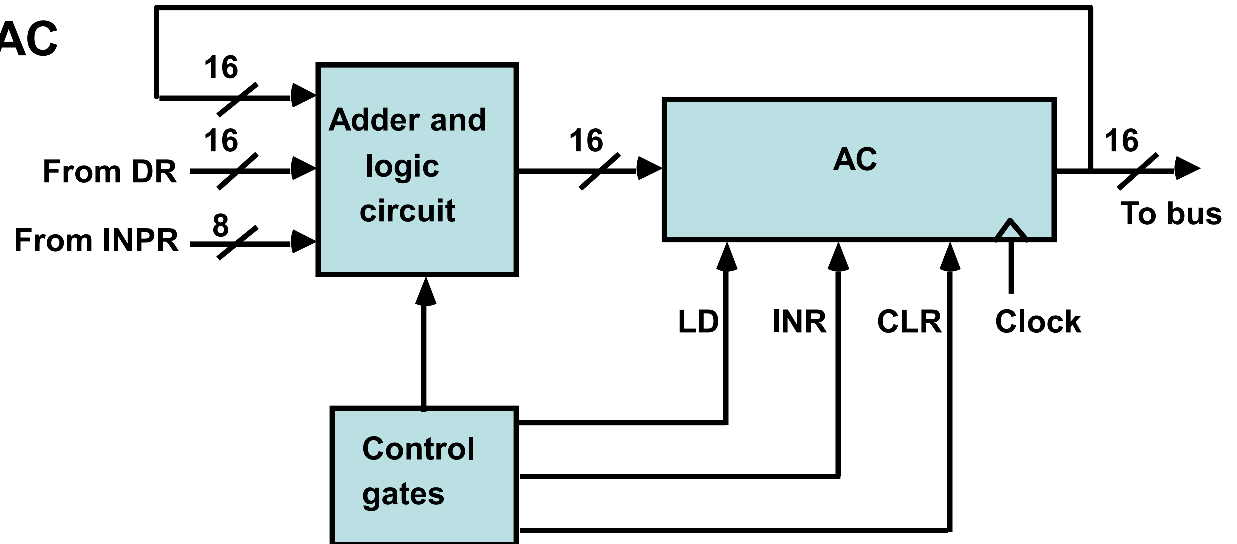
x1	x2	x3	x4	x5	x6	x7	S2	S1	S0	selected register
0	0	0	0	0	0	0	0	0	0	none
1	0	0	0	0	0	0	0	0	1	AR
0	1	0	0	0	0	0	0	1	0	PC
0	0	1	0	0	0	0	0	1	1	DR
0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	1	0	1	1	0	TR
0	0	0	0	0	0	1	1	1	1	Memory

5-9 Design of Basic Computer$^{cont.}$

- For x_7:
  - $X_7 = R'T_1 + D_7'IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$ where it is also applied to the read input

5-10 Design of Accumulator Logic

Circuits associated with AC

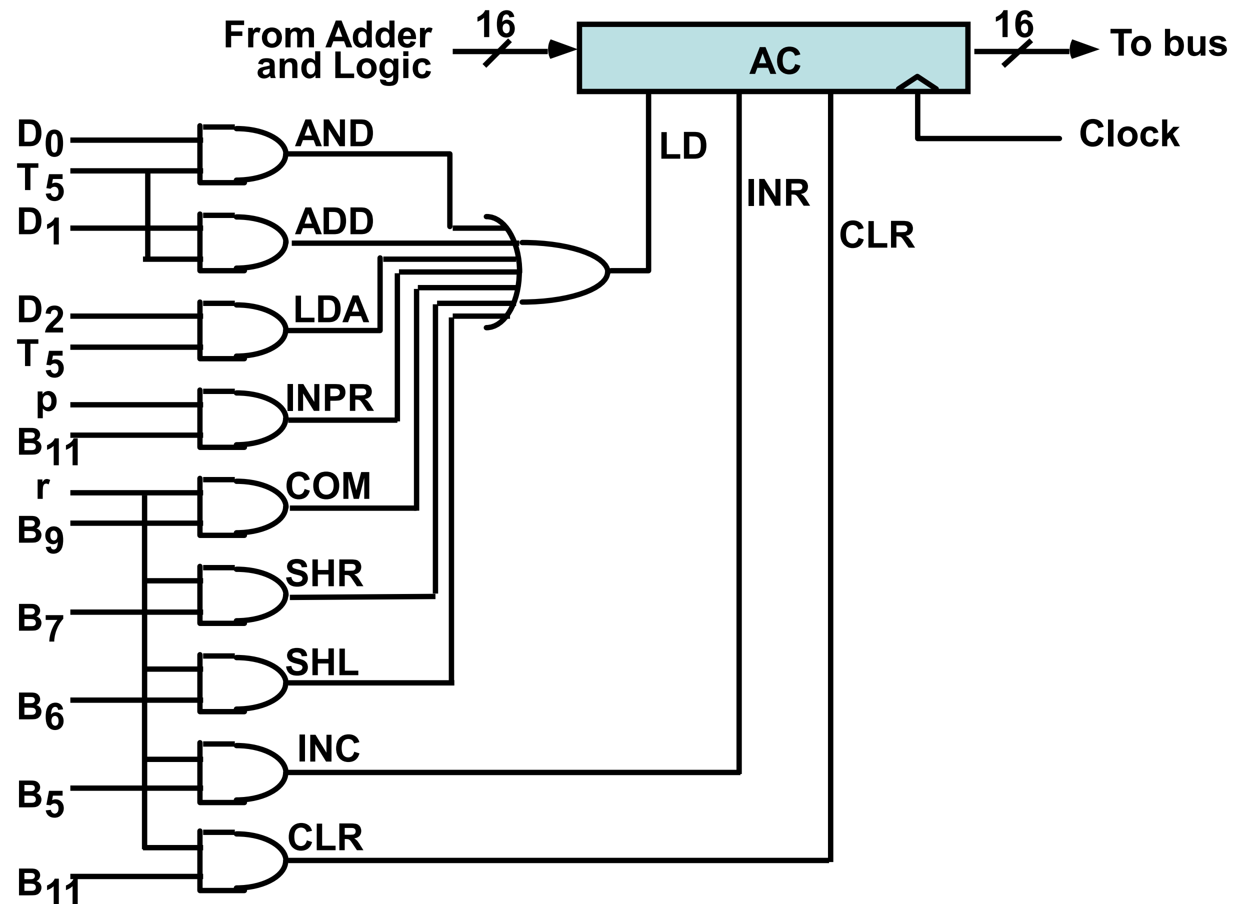


All the statements that change the content of AC

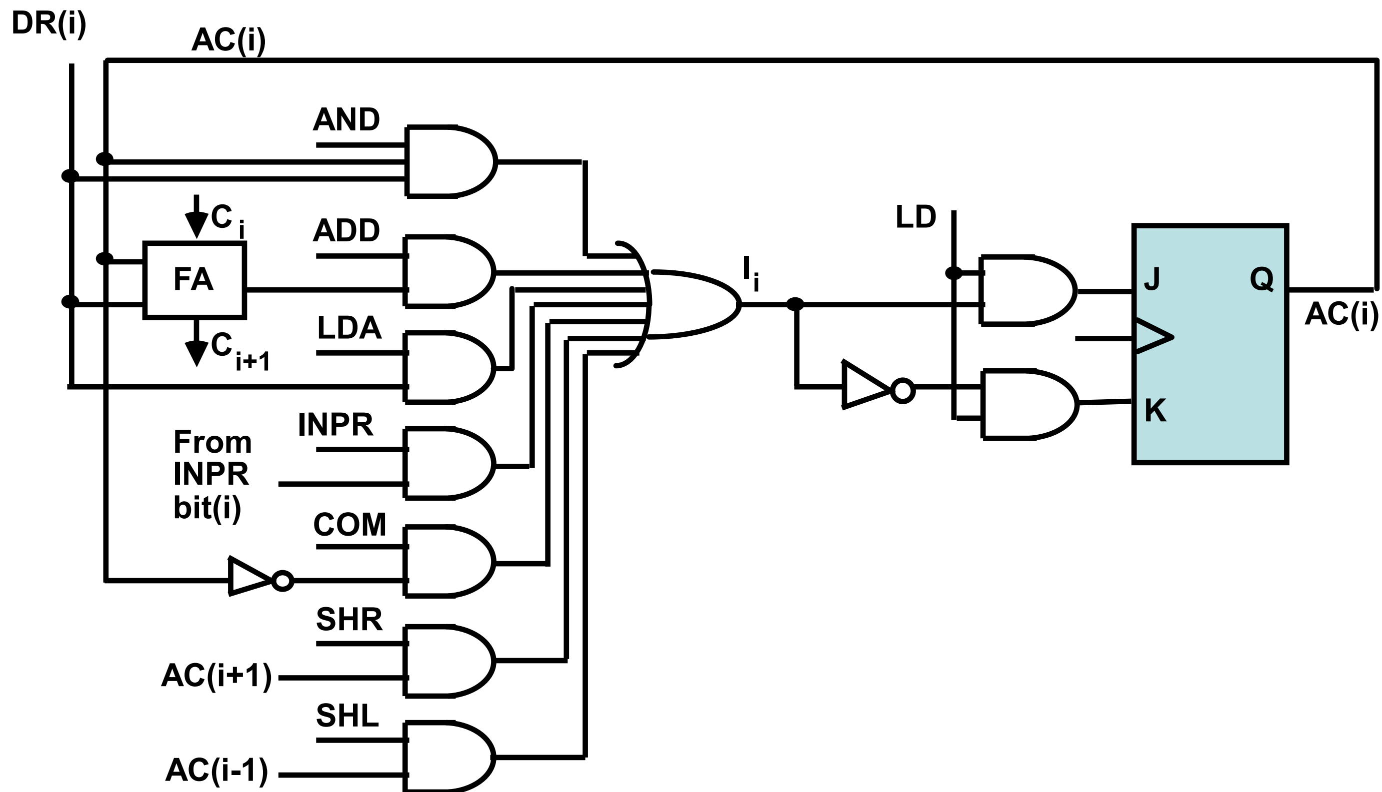
D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with DR
D_1T_5:	$AC \leftarrow AC + DR$	Add with DR
D_2T_5:	$AC \leftarrow DR$	Transfer from DR
pB_{11}:	$AC(0\text{-}7) \leftarrow INPR$	Transfer from INPR
rB_9:	$AC \leftarrow AC'$	Complement
rB_7 :	$AC \leftarrow shr\ AC,\ AC(15) \leftarrow E$	Shift right
rB_6 :	$AC \leftarrow shl\ AC,\ AC(0) \leftarrow E$	Shift left
rB_{11} :	$AC \leftarrow 0$	Clear
rB_5 :	$AC \leftarrow AC + 1$	Increment

5-10 Design of Accumulator Logic[cont.]

Gate structures for controlling the LD, INR, and CLR of AC



Adder and Logic Circuit



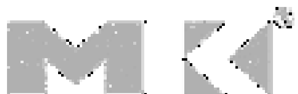
CPE 408340
Computer Organization

Chapter 5 : Large and Fast: Exploiting Memory Hierarchy The

Memory Technology

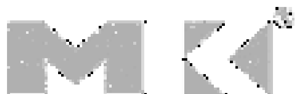
§5.1 Introduction

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $5000 per GB
- Dynamic RAM (DRAM)
  - 50ns – 70ns, $20 – $75 per GB
- Magnetic disk
  - 5ms – 20ms, $0.20 – $2 per GB
- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk



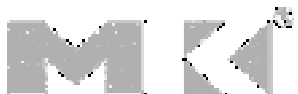
Principle of Locality

- Programs access a small proportion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

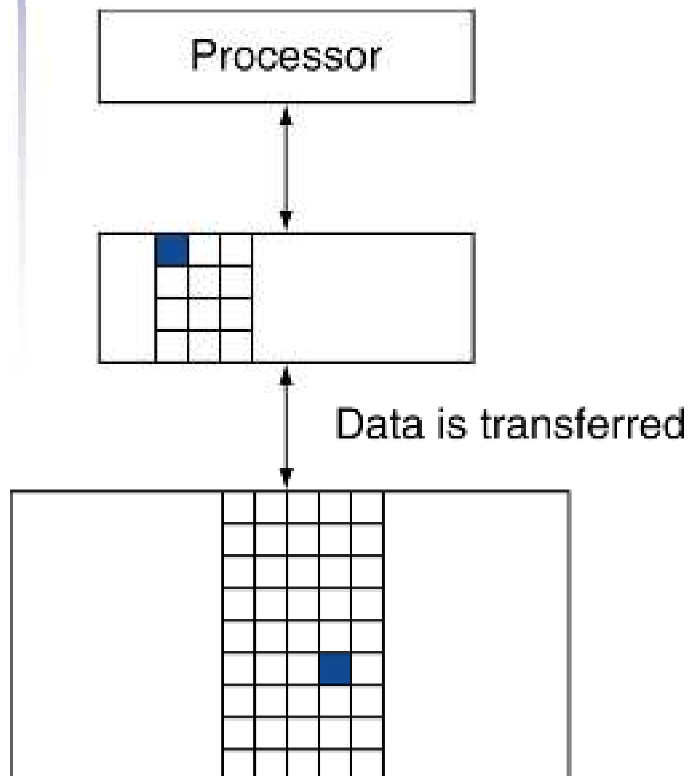


Taking Advantage of Locality

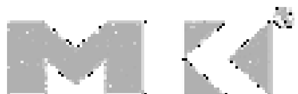
- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU



Memory Hierarchy Levels

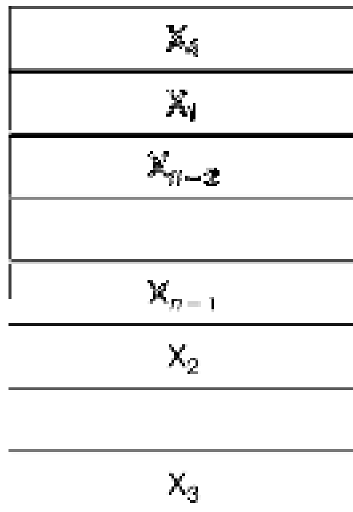


- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses
      = 1 – hit ratio
  - Then accessed data supplied from upper level

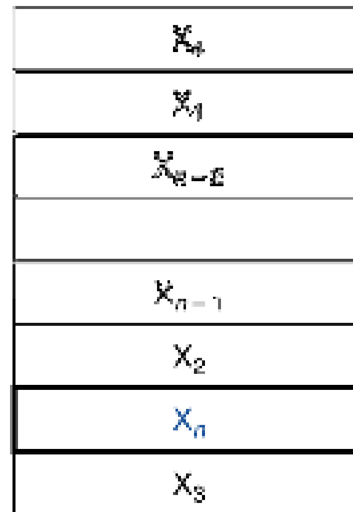


Cache Memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
- Given accesses $X_1, \ldots, X_{n-1}, X_n$

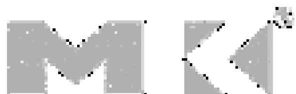


a. Before the reference to X_n



b. After the reference to X_n

- How do we know if the data is present?
- Where do we look?



Cache Design Rules

Address = [Block Address] [Block Offset]
Address = [Tag] [Index] [Word Offset] [Byte Offset]

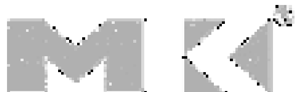
Block_bits = $\log_2$(Block_Size)
#Blocks in Cache = Cache_Size/Block_Size

#Sets in Cache = #Blocks / Set_Size
Set_Size = number of ways in the cache

- For direct cache : Set_Size=1 (#Sets = #Blocks)
- For fully associative : Set_Size= #Blocks (#Sets = 1)
- For k-way associative: Set_Size= k

Index_bits = $\log_2$ (#Sets)

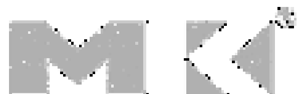
Tag_bits = Address_bits - (Block_bits + Index_bits)



Direct Cache Example

- A cache is direct-mapped and has 64 KB data. Each block contains 32 bytes. The address is 32 bits wide. What are the sizes of the tag, index, and block offset fields?

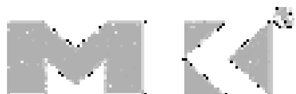
- # bits in block offset = 5 (since each block contains 2^5 bytes)
- # blocks in cache = 64×1024 / 32 = 2048 blocks
  - So # bits in index field = 11 (since there are 2^11 blocks)
- # bits in tag field = 32 - 5 - 11 = 16 (the rest!)



K-way Cache Example

- A cache is 4-way set-associative and has 64 KB data. Each block contains 32 bytes. The address is 32 bits wide. What are the sizes of the tag, index, and block offset fields?

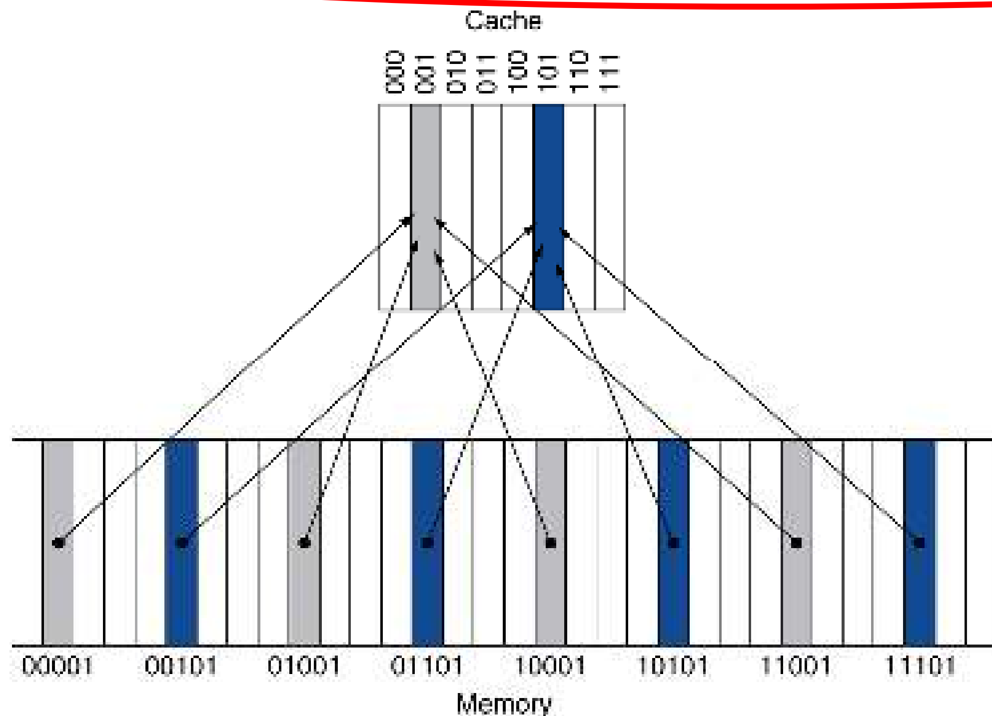
- # bits in block offset = 5 (since each block contains 2^5 bytes)
- # blocks in cache = 64×1024 / 32 = 2048 (2^11)
- # sets in cache = 2048 / 4 = 512 (2^9) sets (a set is 4 blocks kept in the cache for each index)
  - So # bits in index field = 9
- # bits in tag field = 32 – 5 - 9 = 18



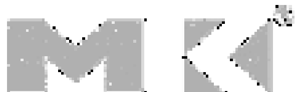
Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)

Index

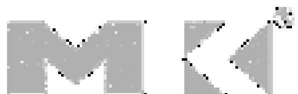


- #Blocks is a power of 2
- Use low-order address bits



Tags and Valid Bits

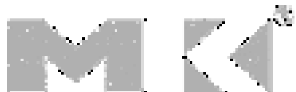
- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag
- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0



Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state, Mem=32 words (or blocks)

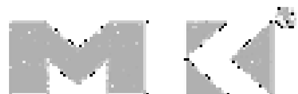
Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		



Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

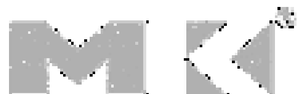
Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		



Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

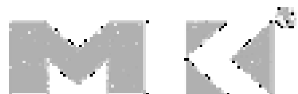
Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		



Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

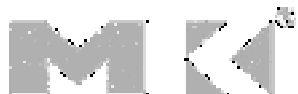
Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		



Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

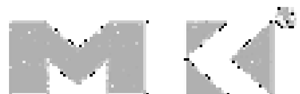


Cache Example

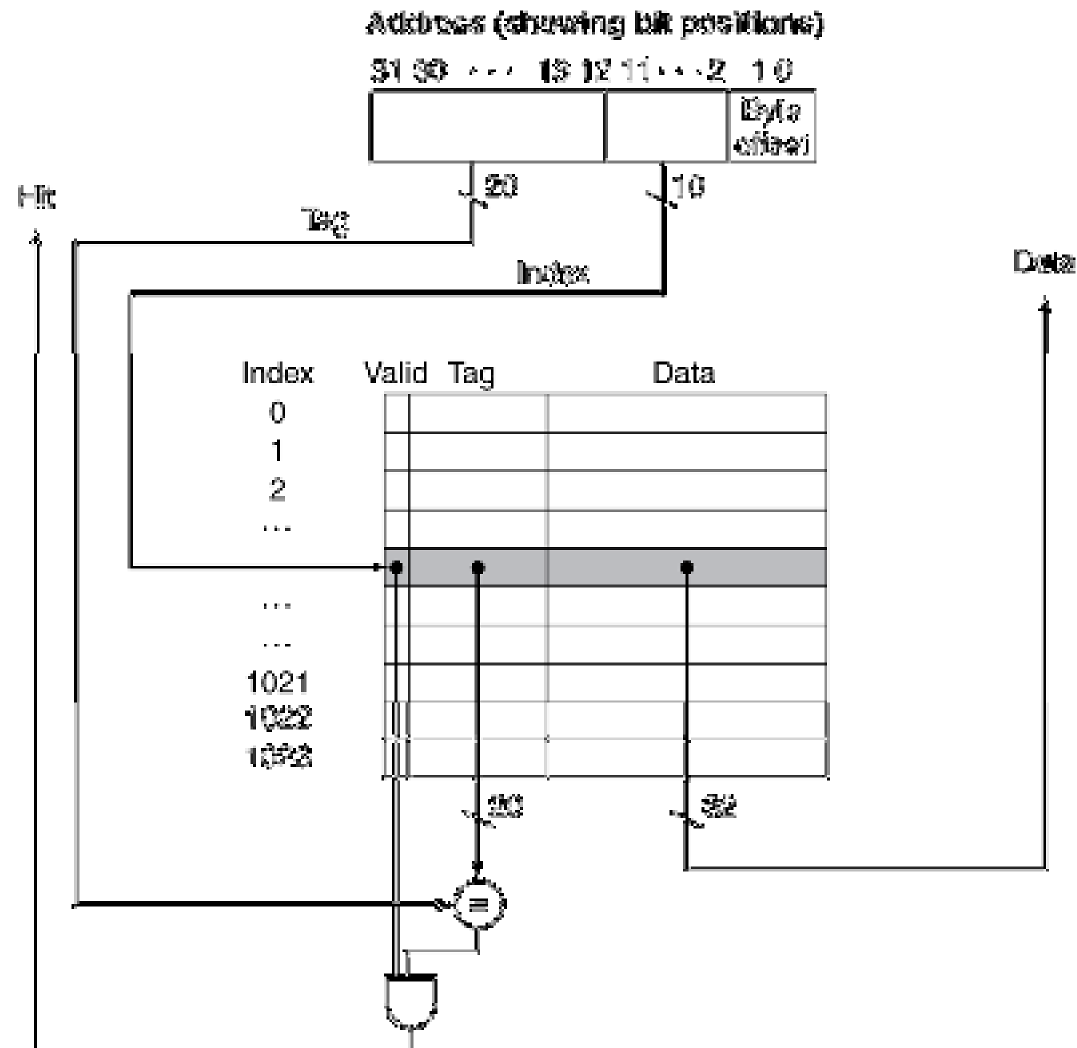
Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

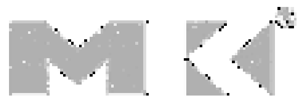
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Miss :Tag mismatch



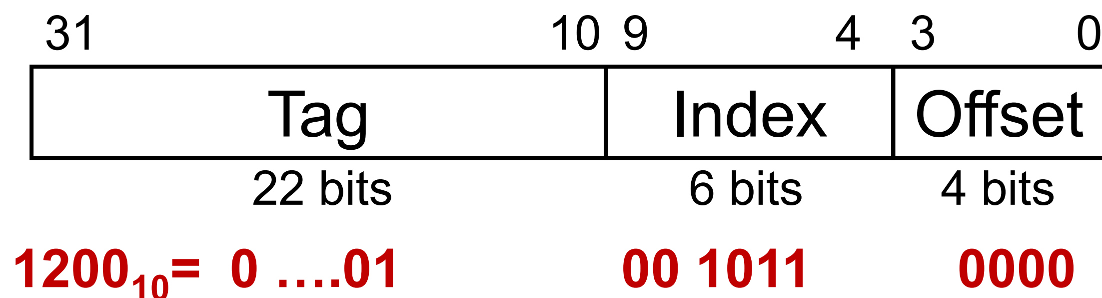
Address Subdivision

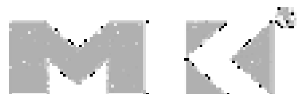




Example: Larger Block Size

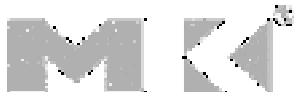
- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
- Block address = $\lfloor 1200/16 \rfloor = 75$
- Block number = 75 modulo 64 = 11



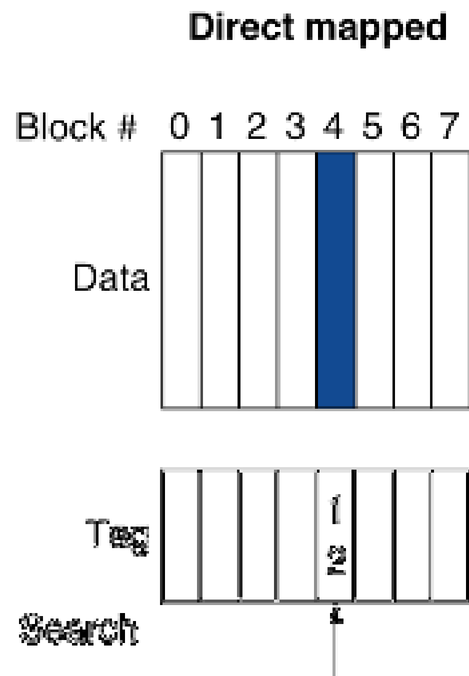


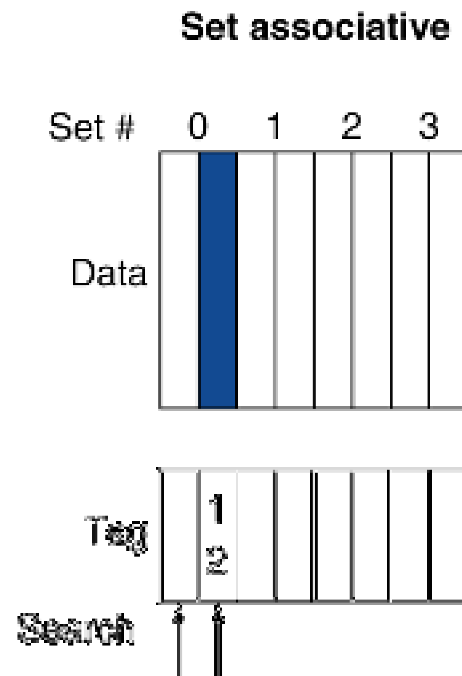
Associative Caches

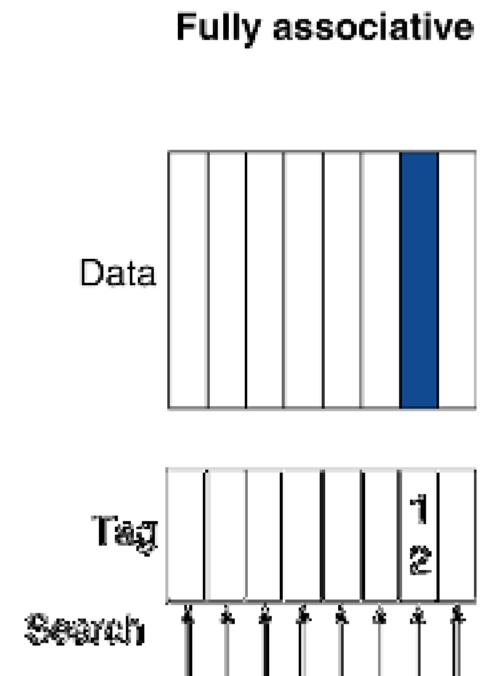
- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

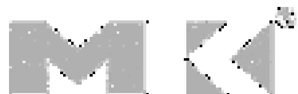


Associative Cache Example









Spectrum of Associativity

- For a cache with 8 entries

One-way set associative
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

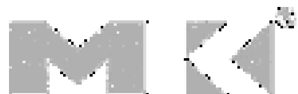
Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

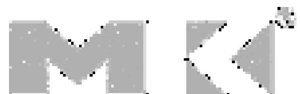


Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

In the next few slides we will measure:
1. Miss Rate
2. Miss Penalty

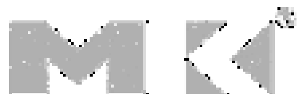
$$\text{Memory stall cycles}$$
$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$
$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$



Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	



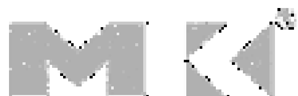
Associativity Example

- 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0	0	hit	Mem[0]	Mem[8]		
6	0	miss	Mem[0]	Mem[6]		
8	0	miss	Mem[8]	Mem[6]		

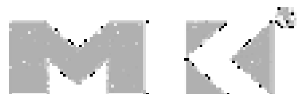
- Fully associative

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	Mem[8]	Mem[6]	



How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%



Set Associative Cache Organization

Address
31 30 ··· 12 11 10 9 8 ··· 3 2 1 0

22 8

Tag

Index

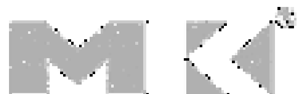
Index	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data
0												
1												
2												
253												
254												
255												

22 32

4-to-1 multiplexor

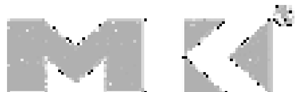
Hit

Data



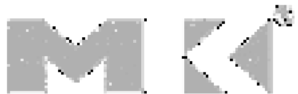
Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks $\Rightarrow$ fewer of them
    - More competition $\Rightarrow$ increased miss rate
  - Larger blocks $\Rightarrow$ pollution
- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help



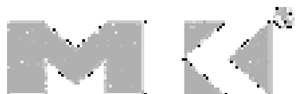
Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access



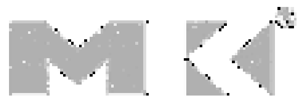
Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full



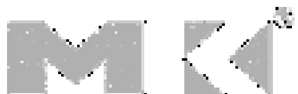
Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first



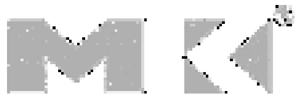
Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Write-allocate on miss: fetch the block
  - Write around (no write allocate): don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block



Example: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%



Example: Intrinsity FastMATH

Address (showing bit positions)

31 ... 14 13 ... 6 5 ... 2 1 0

18 8 4 Byte offset

Hit Tag Index Block offset Data

18 bits 512 bits

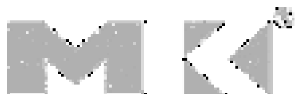
V Tag Data

256 entries

18 32 32 32

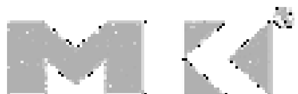
Mux

32



Main Memory Supporting Caches

- Use DRAMs for main memory
  - Fixed width (e.g., 1 word)
  - Connected by fixed-width clocked bus
    - Bus clock is typically slower than CPU clock
- Example cache block read
  - 1 bus cycle for address transfer
  - 15 bus cycles per DRAM access
  - 1 bus cycle per data transfer
- For 4-word block, 1-word-wide DRAM
  - Miss penalty = 1 + 4×15 + 4×1 = 65 bus cycles
  - Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle



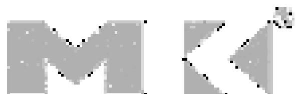
Increasing Memory Bandwidth

a. One-word-wide memory organization

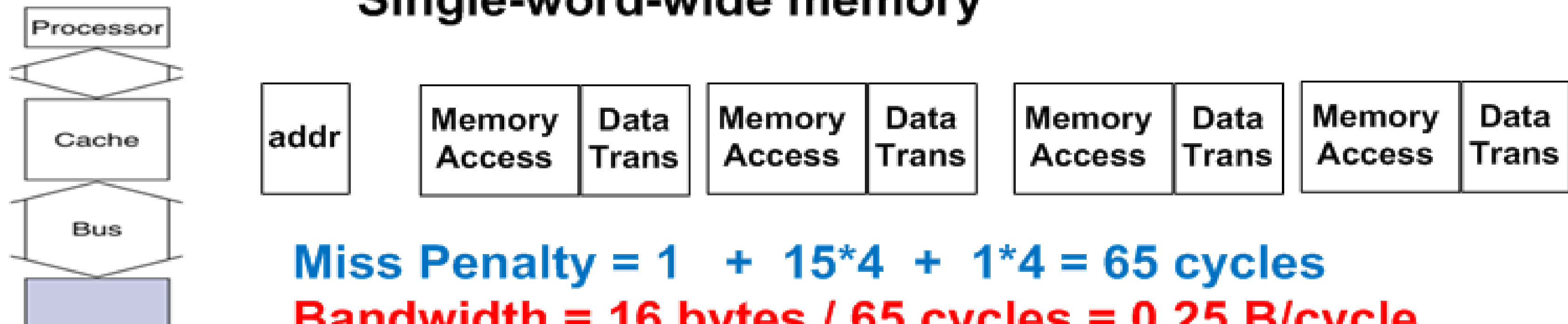
b. Wider memory organization

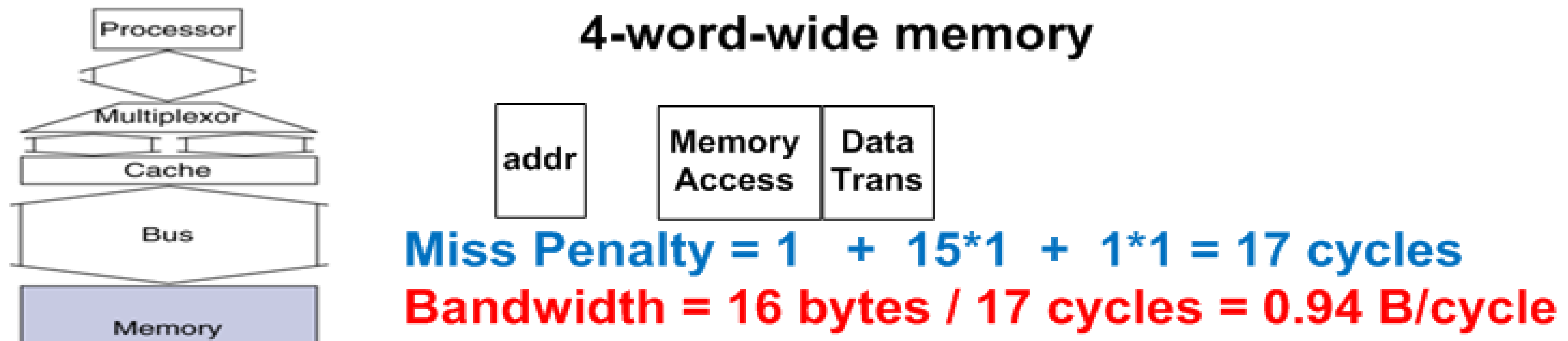
c. Interleaved memory organization

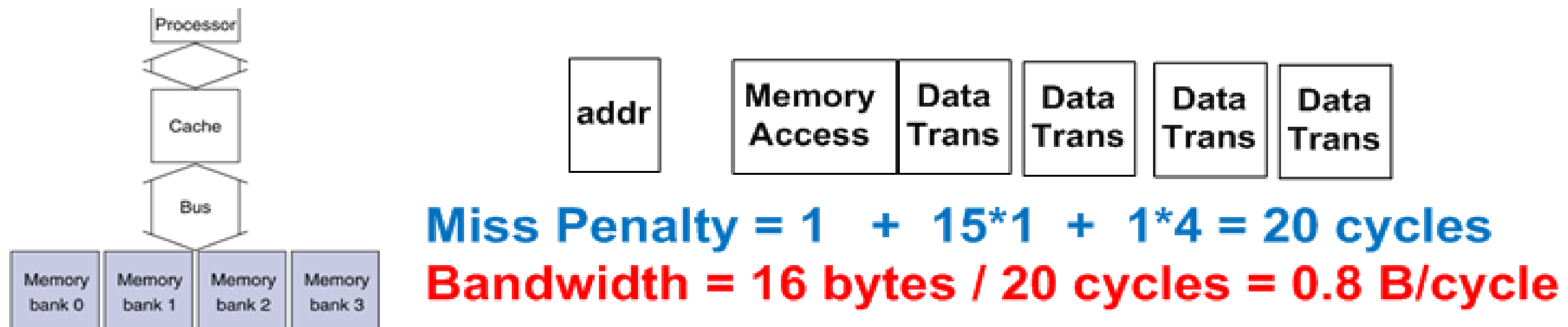
- 4-word wide memory
  - Miss penalty = 1 + 15 + 1 = 17 bus cycles
  - Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle
- 4-bank interleaved memory
  - Miss penalty = 1 + 15 + 4×1 = 20 bus cycles
  - Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle



Miss Penalty = Addr_Transfer + MemoryAccess + DataTrasfer

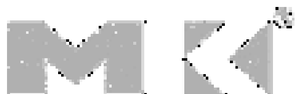






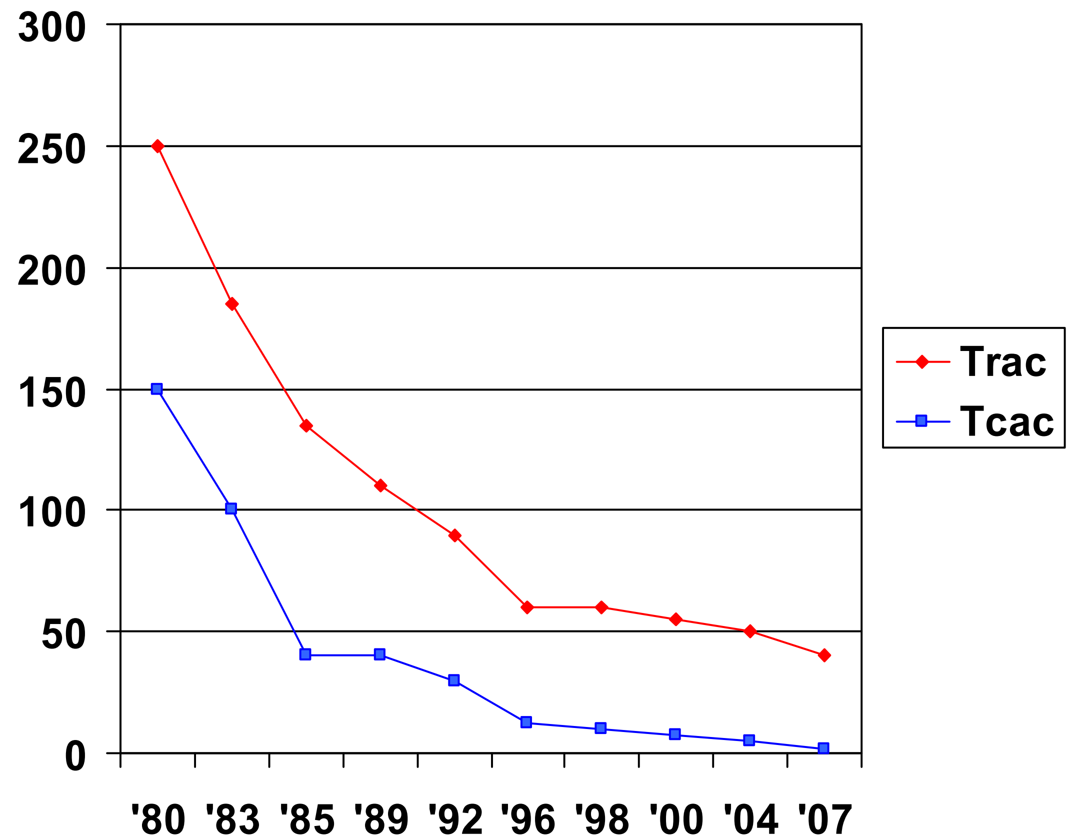
Advanced DRAM Organization

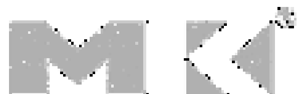
- Bits in a DRAM are organized as a rectangular array
  - DRAM accesses an entire row
  - Burst mode: supply successive words from a row with reduced latency
- Double data rate (DDR) DRAM
  - Transfer on rising and falling clock edges
- Quad data rate (QDR) DRAM
  - Separate DDR inputs and outputs



DRAM Generations

Year	Capacity	$/GB
1980	64Kbit	$1500000
1983	256Kbit	$500000
1985	1Mbit	$200000
1989	4Mbit	$50000
1992	16Mbit	$15000
1996	64Mbit	$10000
1998	128Mbit	$4000
2000	256Mbit	$1000
2004	512Mbit	$250
2007	1Gbit	$50



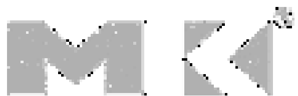


Measuring Cache Performance

§5.3 Measuring and Improving Cache Performance

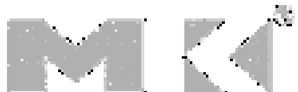
- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

$$\begin{aligned}&\text{Memory stall cycles}\\&=\frac{\text{Memory accesses}}{\text{Program}}\times\text{Miss rate}\times\text{Miss penalty}\\&=\frac{\text{Instructions}}{\text{Program}}\times\frac{\text{Misses}}{\text{Instruction}}\times\text{Miss penalty}\end{aligned}$$



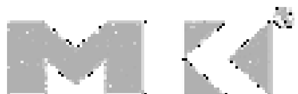
Cache Performance Example

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: $0.02 \times 100 = 2$
  - D-cache: $0.36 \times 0.04 \times 100 = 1.44$
- Actual CPI = 2 + 2 + 1.44 = 5.44
  - Ideal CPU is 5.44/2 =2.72 times faster



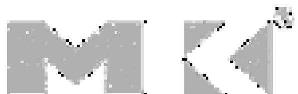
Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction



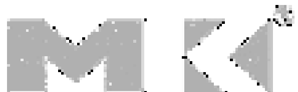
Performance Summary

- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can't neglect cache behavior when evaluating system performance



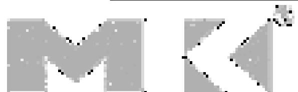
Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity



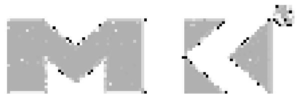
Cache Misses

Cache Misses	The Cause	Dependency
Capacity misses	Occur due to the finite size of the cache.	Cache size
Conflict misses	Occur because the cache had evicted an entry earlier.	Associatively
Compulsory misses (Cold misses)	Caused by the first reference to a location in memory.	Block size



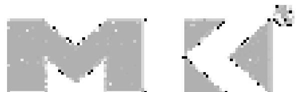
Cache Design Trade-offs

Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.



Memory Protection

- Different tasks can share parts of their virtual address spaces
  - But need to protect against errant access
  - Requires OS assistance
- Hardware support for OS protection
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., syscall in MIPS)

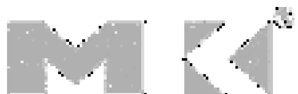


The Memory Hierarchy

§5.5 A Common Framework for Memory Hierarchies

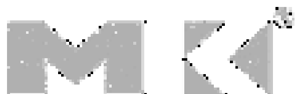
The BIG Picture

- Common principles apply at all levels of the memory hierarchy
  - Based on notions of caching
- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy



Block Placement

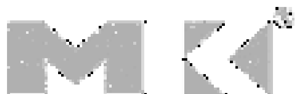
- Determined by associativity
  - Direct mapped (1-way associative)
    - One choice for placement
  - n-way set associative
    - n choices within a set
  - Fully associative
    - Any location
- Higher associativity reduces miss rate
  - Increases complexity, cost, and access time



Finding a Block

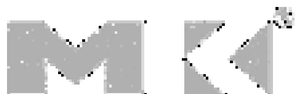
Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

- Hardware caches
  - Reduce comparisons to reduce cost
- Virtual memory
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate



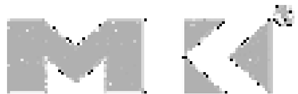
Replacement

- Choice of entry to replace on a miss
  - Least recently used (LRU)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU, easier to implement
- Virtual memory
  - LRU approximation with hardware support



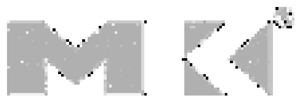
Write Policy

- Write-through
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer
- Write-back
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state
- Virtual memory
  - Only write-back is feasible, given disk write latency



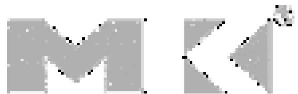
Sources of Misses

- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size



Cache Design Trade-offs

Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

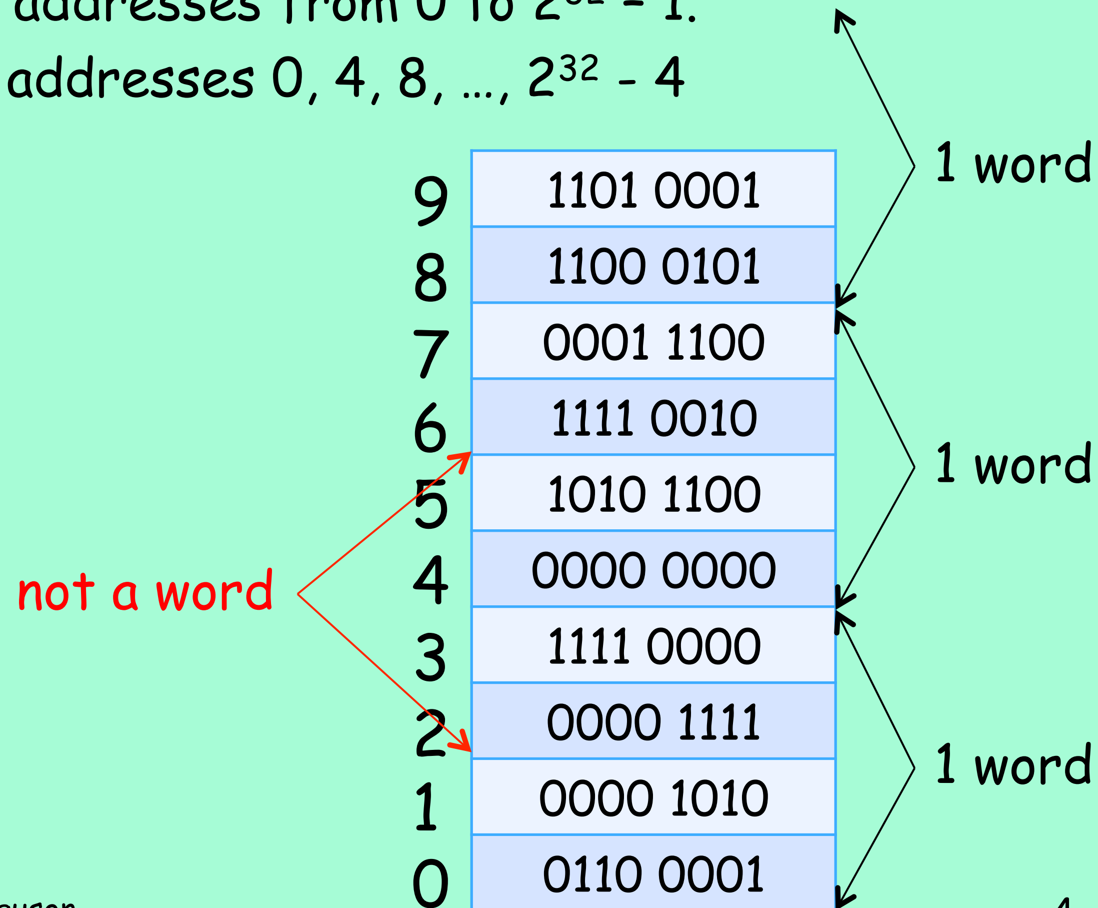


MIPS Addressing Modes and Memory Architecture

(Second Edition:Section 3.8
Fourth Edition: Section 2.10)
from Dr. Andrea Di Blas' notes

Memory Organization and Addressing

- Memory may be viewed as a single-dimensional array of individually addressable bytes. 32-bit words are aligned to 4 byte boundaries.
  - 2^{32} bytes, with addresses from 0 to $2^{32} - 1$.
  - 2^{30} words with addresses 0, 4, 8, …, 2^{32} - 4



Byte ordering within words

- Little Endian: word address is LSB
- Big Endian: word address is MSB

Ex: 0000 0001 0010 0011 0100 0101 0110 0111

Little Endian

Address	Byte
x..x1101	
x..x1100	
x..x0111	00000001
x..x0110	00100011
x..x0101	01000101
x..x0100	01100111
x..x0011	
x..x0010	

Big Endian

Address	Byte
x..x1101	
x..x1100	
x..x0111	01100111
x..x0110	01000101
x..x0101	00100011
x..x0100	00000001
x..x0011	
x..x0010	

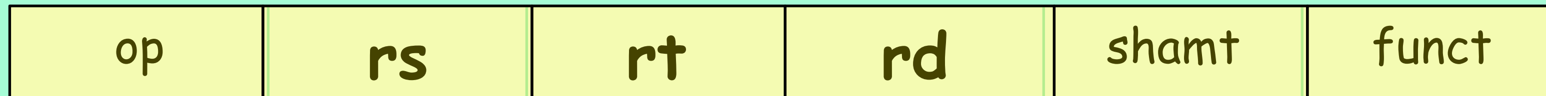
MIPS addressing modes

Addressing modes are the ways of specifying an operand or a memory address.

- Register addressing
- Immediate addressing
- Base addressing
- PC-relative addressing
- Indirect addressing
- Direct addressing (almost)

Register addressing

- Operands are in a register.
- Example: add $3,$4,$5
- Takes n bits to address 2^n registers



Register addressing

op	**rs**	**rt**	**rd**	shamt	funct

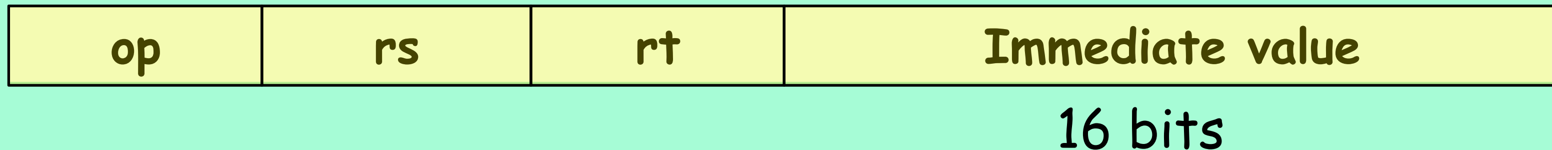
registers

ALU

memory

Immediate Addressing

- The operand is embedded inside the encoded instruction.



16 bit two's-complement number:

$-2^{15} - 1 = -32{,}769 < \text{value} < +2^{15} = +32{,}768$

Immediate addressing

op	rs	rt	Immediate value

16 bits

Sign-extended

----------------xxxxxxxxxxxxxxxx

registers

ALU

memory

Example is addi or similar

Base (or Base-offset or displacement) Addressing

- The address of the operand is the sum of the immediate and the value in a register (rs).
- 16-bit immediate is a two's complement number
- Ex: lw $15,16($12)

op	rs	rt	Immediate value

Base addressing

op	rs	rt	Immediate value

16 bits

Sign-extended xxxxxxxxxxxxxxxx

registers

ALU

memory

```
lw $8,128($5)
```

Effective address

PC-relative addressing: the value in the immediate field is interpreted as an offset of the next instruction (PC+4 of current instruction)

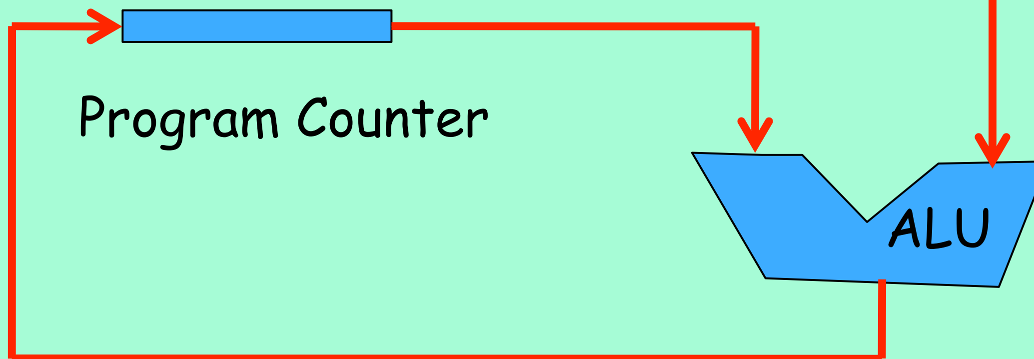
Example: `beq $0,$3,Label`

op	rs	rt	Immediate value

PC-relative addressing

16 bits

Shifted by 2 and Sign-extended




```
beq $0,$5,Label
```

Detail of MIPS PC-Relative

address	instruction
40000008	addi $5, $5, 1
4000000C	beq $0, $5, label
40000010	addi $5, $5, 1
40000014	addi $5, $5, 1
40000018	label addi $5, $5, 1
4000001C	addi $5, $5, 1
40000020	etc…

Binary code to beq $0,$5, label is 0x10050002, which means 2 instructions from the next instruction.

PC =	0x4000000C
PC+4=	0x40000010
Add 4*2 =	0x00000008
Eff. Add. =	0x40000018

op	rs	rt	Immediate value
00010	00000	00101	0000000000000010

Register Direct Addressing: the value the (memory) effective address is in a register. Also called "Indirect Addressing".

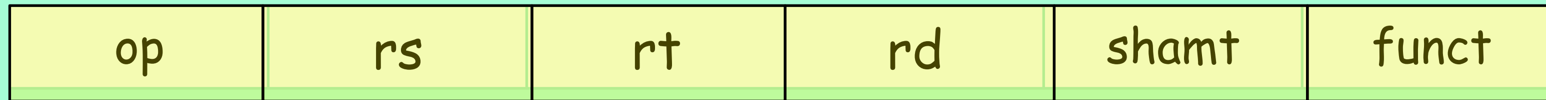
Special case of base addressing where offset is 0.

Used with the jump register instructions (jr, jalr).

Example: `jr $31`

op	rs	rt	rd	shamt	funct
000000	rs	00000	00000	00000	001000

Register Direct



registers

program counter

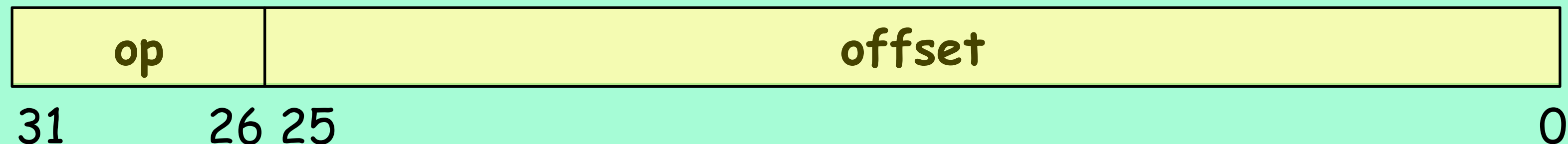
memory

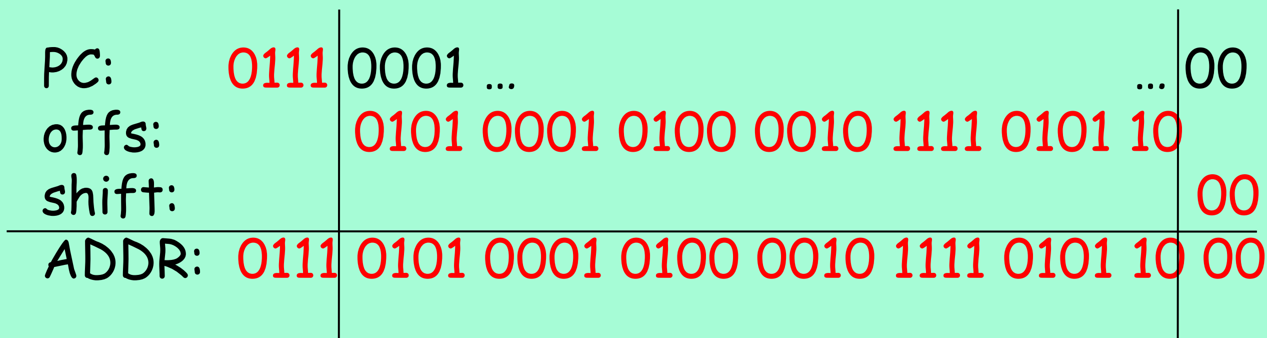
jr $5

Direct Addressing: the address is "the immediate". 32-bit address cannot be embedded in a 32-bit instruction.

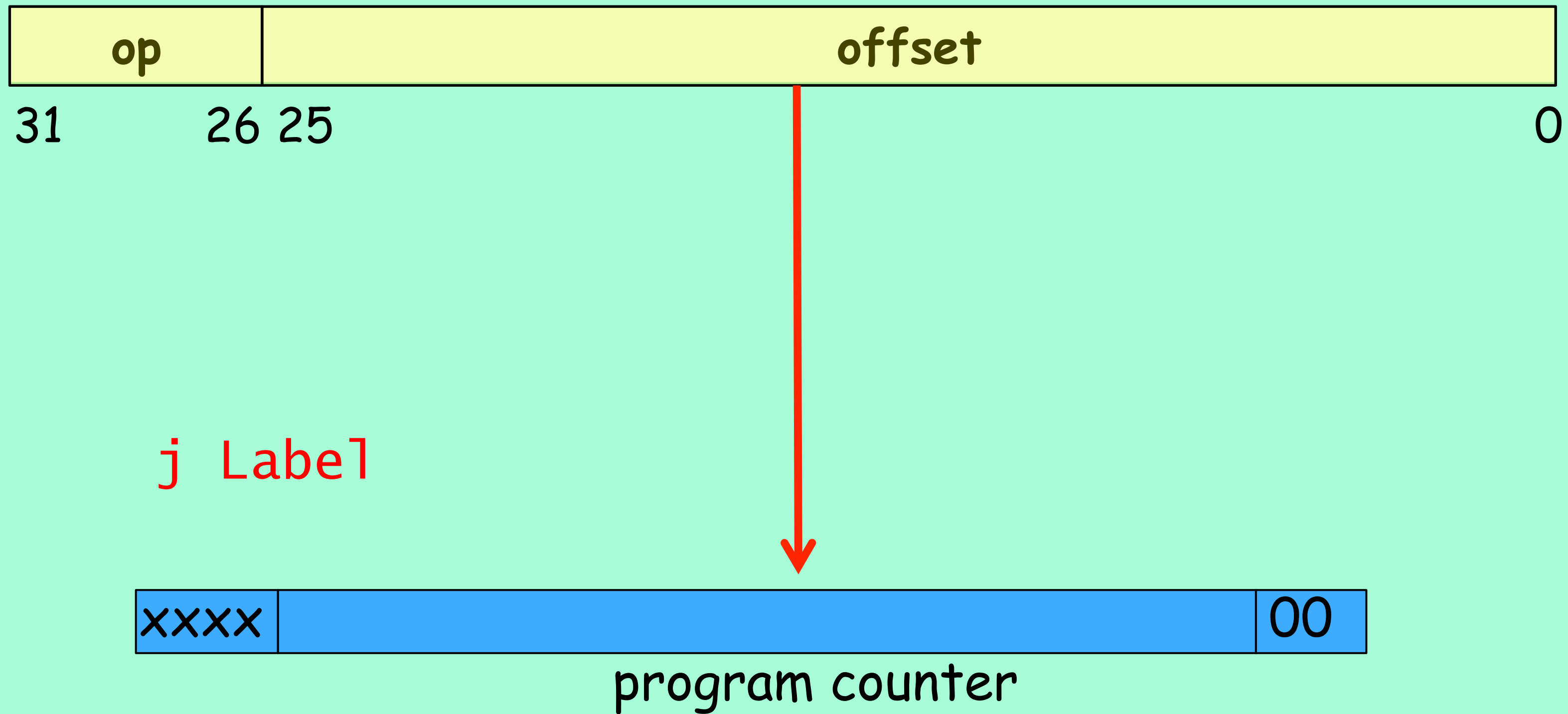
Pseudodirect addressing: 26 bits of the address is embedded as the immediate, and is used as the instruction offset within the current 256MB (64MWord) region defined by the MS 4 bits of the PC.

Example: `j Label`





Pseudodirect addressing



Caution: Addressing mode is not Instruction type

- Addressing mode is how an address (memory or register) is determined.
- Instruction type is how the instruction is put together.
- Example: addi, beq, and lw are all I-types instructions. But
  - addi uses immediate addressing mode (and register)
  - beq uses pc-relative addressing (and register)
  - lw uses base addressing (and register)

MIPS Addressing Modes

1. REGISTER: a source or destination *operand* is specified as content of one of the registers $0–$31 .
2. IMMEDIATE: a numeric value embedded in the instruction is the actual operand. .
3. PC-RELATIVE: a data or instruction memory location is specified as an offset relative to the incremented PC. .
4. BASE: a data or instruction memory location is specified as a signed offset from a register.
5. REGISTER-DIRECT: the value the effective address is in a register.
6. PSEUDODIRECT: the memory address is (mostly) embedded in the instruction.

PowerPC and x86 addressing modes and instructions

- **PowerPC:** 2nd edition: pp. 175-177, 4th edition: Appendix E.
- **80x86:** 2nd edtion: pp. 177-185, 4th edition: Section 2.17.

Additional PowerPC addressing modes - 1

Indexed Addressing: The address is the sum of two registers. (note indexed addressing is different here than usually used)

```
MIPS code: add $10, $20, $13   ;$20 is base,$13 is index
           lw $5, 0($10)

PowerPC:   lw $5, $20+$13      ; $5 ← ($20+$13)
```

Saves instruction for incrementing array index.

No extra hardware.

PowerPC: Indexed Addressing

op	rs	rt	rd		

registers

ALU

memory

Additional PowerPC addressing mode - 2

Update Addressing: base addressing with automatic base register increment.

```
MIPS code:  lw $10, 4($13)       ; $10 ← Mem[$10+4]
            addi $13, $13, 4     ; $13 ← $13+4

PowerPC:    lwu $10, 4($13)      ; $10 ← Mem[$10+4]
                                 ; and $13 ← $13+4
```

Requires that two registers be written at the same time → more hardware.

PowerPC: Update Addressing

op	rs	rt	Immediate value

16 bits

memory

registers

ALU

eff. add.

(for **base** and index addressing)

Additional non-RISC addressing mode

Memory Indirect Addressing: read effective address from memory. (Usually PC-relative addressing is used to get the effective address from memory).

```
RISC code:   lw $10, 0($13)
             lw $5, 0($10)

CISC:        ldi $5, Label        ; $5 ← Mem[Label]
```

Requires two sequential data memory accesses.

CISC: Memory Indirect Addressing

op	rs	rt	Immediate value

(or from PC)

ALU

memory

registers

eff. add.

CSAIL

Early Developments: From *Difference Engine* to *IBM 701*

Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.

Based on the material prepared by Arvind and Krste Asanovic

Charles Babbage 1791-1871

Lucasian Professor of Mathematics, Cambridge University, 1827-1839

Charles Babbage

Image removed due to copyright restrictions.
To view image, visit
http://www.rtpnet.org/robroy/Babbage/hawks.html

Charles Babbage

- *Difference Engine* 1823
- *Analytic Engine* 1833
  - The forerunner of modern digital computer!

Application
- Mathematical Tables – Astronomy
- Nautical Tables – Navy

Background
- Any continuous function can be approximated by a polynomial --- *Weierstrass*

Technology
- mechanical - gears, Jacquard's loom, simple calculators

CSAIL

Difference Engine
A machine to compute mathematical tables

Weierstrass:
- Any continuous function can be approximated by a polynomial
- Any Polynomial can be computed from *difference* tables

An example

$$f(n) = n^2+n+41$$
$$d1(n) = f(n) - f(n-1) = 2n$$
$$d2(n) = d1(n) - d1(n-1) = 2$$

$$f(n) = f(n-1) + d1(n) = f(n-1) + (d1(n-1) + 2)$$

n	0	1	2	3	4 ...
d2(n)			2	2	2
d1(n)		2	4	6	8
f(n)	41	43	47	53	61

all you need is an adder!

CSAIL

Difference Engine

1823
- Babbage's paper is published

1834
- The paper is read by Scheutz & his son in Sweden

1842
- Babbage gives up the idea of building it; he is onto Analytic Engine!

1855
- Scheutz displays his machine at the Paris World Fare
- Can compute any 6th degree polynomial
- *Speed:* 33 to 44 32-digit numbers per minute!

Now the machine is at the Smithsonian

Analytic Engine

1833: Babbage's paper was published

- *conceived during a hiatus in the development of the difference engine*

Inspiration: *Jacquard Looms*

- looms were controlled by punched cards
  - The set of cards with fixed punched holes dictated the pattern of weave ⇒ *program*
  - The same set of cards could be used with different colored threads ⇒ *numbers*

1871: Babbage dies

- The machine remains unrealized.

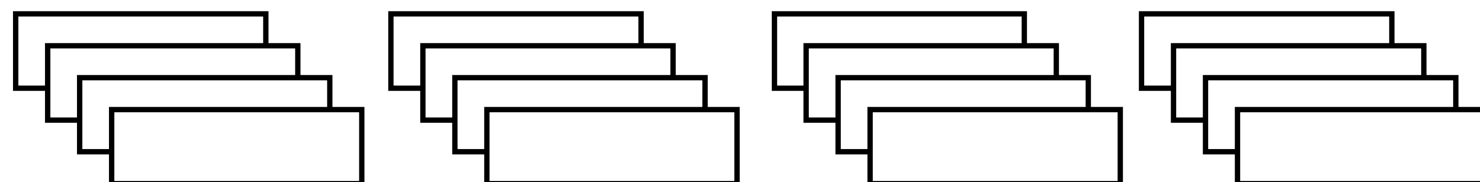
It is not clear if the analytic engine could be built even today using only mechanical technology

CSAIL

Analytic Engine
The first conception of a general purpose computer

1. The *store* in which all variables to be operated upon, as well as all those quantities which have arisen from the results of the operations are placed.
2. The *mill* into which the quantities about to be operated upon are always brought.

The *program*



An operation in the *mill* required feeding two punched cards and producing a new punched card for the *store.*

An operation to alter the sequence was also provided!

CSAIL

The first programmer
Ada Byron *aka* “Lady Lovelace” 1815-52

Ada Byron a.k.a "Lady Lovelace"

Image removed due to copyright restrictions. To view image, visit http://www.sdsc.edu/ScienceWomen/lovelace.html

Ada’s tutor was Babbage himself!

CSAIL

Babbage's Influence

- Babbage's ideas had great influence later primarily because of
  - *Luigi Menabrea,* who published notes of Babbage's lectures in Italy
  - *Lady Lovelace,* who translated Menabrea's notes in English and thoroughly expanded them.

    "... Analytic Engine weaves *algebraic patterns*...."

- In the early twentieth century - the focus shifted to analog computers but
  - *Harvard Mark I built in 1944 is very close in spirit to the Analytic Engine.*

CSAIL

Harvard Mark I

- Built in 1944 in IBM Endicott laboratories
  - Howard Aiken – Professor of Physics at Harvard
  - Essentially mechanical but had some electro-magnetically controlled relays and gears
  - Weighed *5 tons* and had *750,000* components
  - A synchronizing clock that beat every *0.015* seconds

Performance:

0.3 seconds for addition
6 seconds for multiplication
1 minute for a sine calculation

Broke down once a week!

CSAIL

Linear Equation Solver
John Atanasoff, Iowa State University

1930's:
- Atanasoff built the Linear Equation Solver.
- It had 300 tubes!

Application:
- Linear and Integral differential equations

Background:
- Vannevar Bush's Differential Analyzer
  --- *an analog computer*

Technology:
- Tubes and Electromechanical relays

Atanasoff decided that the correct mode of computation was by electronic digital means.

CSAIL

Electronic Numerical Integrator and Computer (ENIAC)

- Inspired by Atanasoff and Berry, Eckert and Mauchly designed and built ENIAC (1943-45) at the University of Pennsylvania
- The first, completely electronic, operational, general-purpose analytical calculator!
  - 30 tons, 72 square meters, 200KW
- Performance
  - Read in 120 cards per minute
  - Addition took 200 μs, Division 6 ms
  - 1000 times faster than Mark I
- Not very reliable!

Application: Ballistic calculations

angle = f (location, tail wind, cross wind,
air density, temperature, weight of shell,
propellant charge, ...)

Electronic Discrete Variable Automatic Computer (EDVAC)

- ENIAC's programming system was external
  - Sequences of instructions were executed independently of the results of the calculation
  - Human intervention required to take instructions "out of order"
- Eckert, Mauchly, John von Neumann and others designed EDVAC (1944) to solve this problem
  - Solution was the *stored program computer*

    $\Rightarrow$ *"program can be manipulated as data"*
- *First Draft of a report on EDVAC* was published in 1945, but just had von Neumann's signature!
  - In 1973 the court of Minneapolis attributed the honor of *inventing the computer* to John Atanasoff

CSAIL

Stored Program Computer

Program = A sequence of instructions

How to control instruction sequencing?

manual control	calculators
automatic control	
external (paper tape)	Harvard Mark I , 1944 Zuse's Z1, WW2
internal	
plug board	ENIAC 1946
read-only memory	ENIAC 1948
read-write memory	EDVAC 1947 *(concept)*

– The same storage can be used to store program and data

EDSAC 1950 Maurice Wilkes

CSAIL

Technology Issues

ENIAC ⇒ EDVAC

18,000 tubes
20 10-digit numbers

4,000 tubes
2000 word storage
mercury delay lines

ENIAC had many asynchronous parallel units but only one was active at a time

BINAC : Two processors that checked each other for reliability.

Didn't work well because processors never agreed

CSAIL

The Spread of Ideas

ENIAC & EDVAC had immediate impact

brilliant engineering: Eckert & Mauchley
lucid paper: Burks, Goldstein & von Neumann

IAS	Princeton	46-52	Bigelow
EDSAC	Cambridge	46-50	Wilkes
MANIAC	Los Alamos	49-52	Metropolis
JOHNIAC	Rand	50-53	
ILLIAC	Illinois	49-52	
	Argonne	49-53	
SWAC	UCLA-NBS		

UNIVAC - the first commercial computer, 1951

Alan Turing's direct influence on these developments is still being debated by historians.

Dominant Problem: *Reliability*

Mean time between failures (MTBF)

MIT's Whirlwind with an MTBF of 20 min. was perhaps the most reliable machine !

Reasons for unreliability:

1. Vacuum Tubes

2. Storage medium
   acoustic delay lines
   mercury delay lines
   Williams tubes
   Selections

CORE J. Forrester 1954

Commercial Activity: 1948-52

IBM's SSEC

Selective Sequence Electronic Calculator

- 150 word store.
- Instructions, constraints, and tables of data were read from paper tapes.
- 66 Tape reading stations!
- Tapes could be glued together to form a loop!
- Data could be output in one phase of computation and read in the next phase of computation.

And then there was IBM 701

IBM 701 -- 30 machines were sold in 1953-54

IBM 650 -- a cheaper, drum based machine,
more than 120 were sold in 1954
and there were orders for 750 more!

Users stopped building their own machines.

Why was IBM late getting into computer technology?

IBM was making too much money!
Even without computers, IBM revenues were doubling every 4 to 5 years in 40's and 50's.

Software Developments

up to 1955 Libraries of numerical routines

- Floating point operations
- Transcendental functions
- Matrix manipulation, equation solvers, . . .

1955-60 *High level Languages* - Fortran 1956
Operating Systems -

- Assemblers, Loaders, Linkers, Compilers
- Accounting programs to keep track of usage and charges

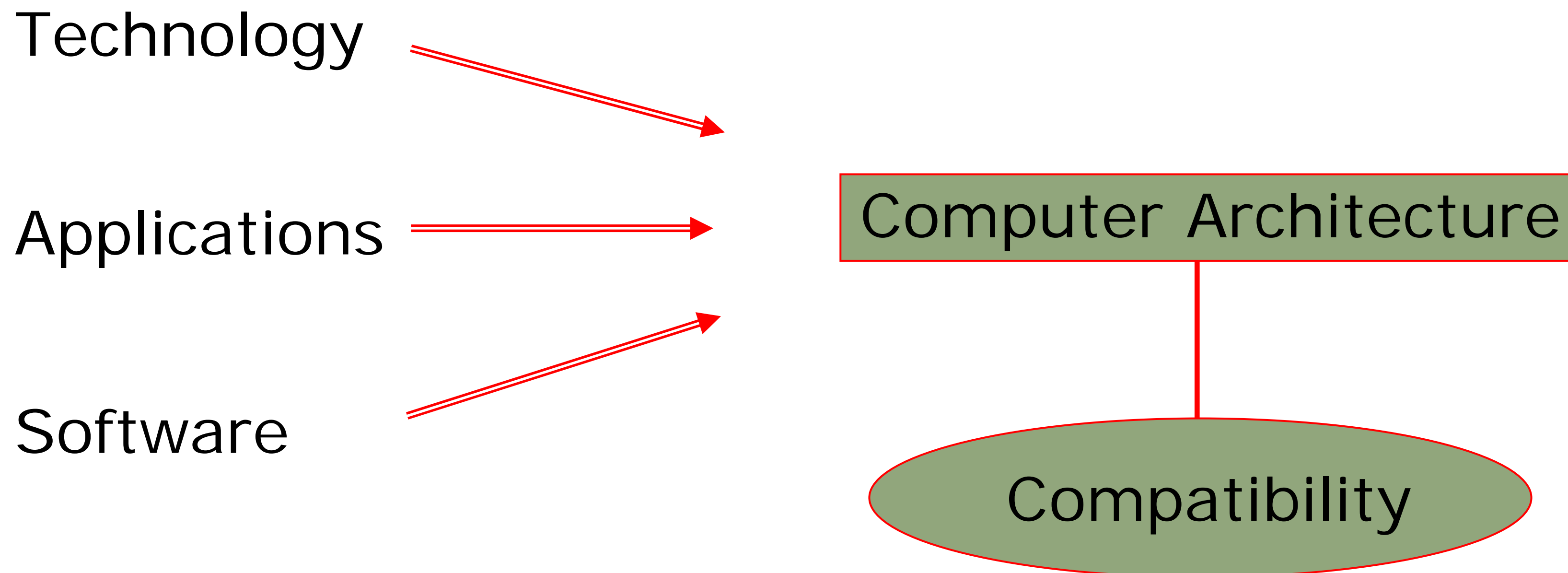
Machines required *experienced operators*

⇒ Most users could not be expected to understand these programs, much less write them

⇒ Machines had to be sold with a lot of resident software

CSAIL

Factors that Influence Computer Architecture



Software played almost no role in defining an architecture before mid fifties.

special-purpose *versus* general-purpose machines

Microprocessors Economics *since 1990's*

- Huge teams design state-of-the-art microprocessors

  PentiumPro ~ 500 engineers
  Itanium ~ 1000 engineers

- Huge investments in fabrication lines and technology

  ⇒ to improve clock-speeds and yields
  ⇒ to build new peripheral chips (memory controllers, ...)

- Economics

  ⇒ price drops to one tenth in 2-3 years
  ⇒ need to sell 2 to 4 million units to breakeven

The cost of launching a new ISA is prohibitive and the advantage is dubious!

CSAIL

Compatibility

Essential for *portability* and *competition*

Its importance increases with the market size

but it is also the most *regressive* force

What does compatibility mean?

Instruction Set Architecture (ISA) compatibility

The same assembly program can run on an upward compatible model

then IBM 360/370 ... *now* Intel x86 (IA32), IA64

System and application software developers expect more than ISA compatibility *(API's)*

applications
operating system
proc + mem + I/O

Java?

Wintel

CSAIL

Perpetual tension

Language/ Compiler/ System software designer		*Architect/Hardware designer*
Need mechanisms to support important abstractions		Decompose each mechanism into essential micro-mechanisms and determine its feasibility and cost effectiveness
Determine compilation strategy; new language abstractions		Propose mechanisms and features for performance

Architects main concerns are performance (both absolute and MIPs/$), and power (both absolute and MIPs/watt) in supporting a broad class of software systems.

Influence of Technology and Software on Instruction Sets: Up to the dawn of IBM 360

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Based on the material prepared by
Arvind and Krste Asanovic

Importance of Technology

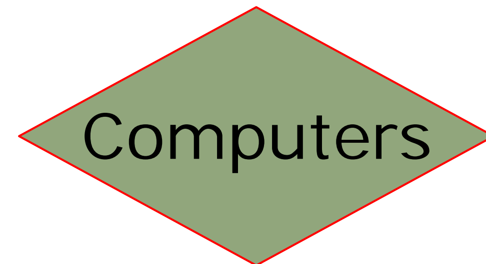
New technologies not only provide greater speed, size and reliability at lower cost, but more importantly these dictate the kinds of structures that can be considered and thus come to shape our whole view of what a computer is.

Bell & Newell

Technology is the dominant factor in computer design

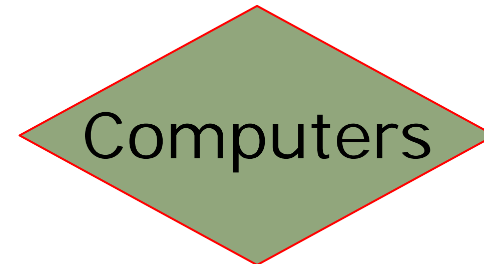
Technology

Transistors
Integrated circuits
VLSI (initially)
Laser disk, CD's



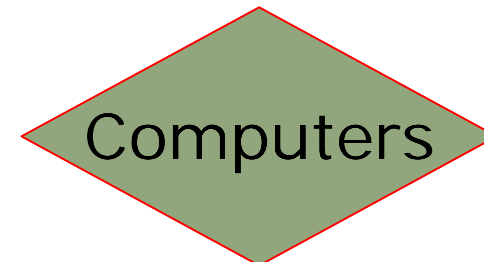
Technology

Core memories
Magnetic tapes
Disks



Technology

ROMs, RAMs
VLSI
Packaging
Low Power

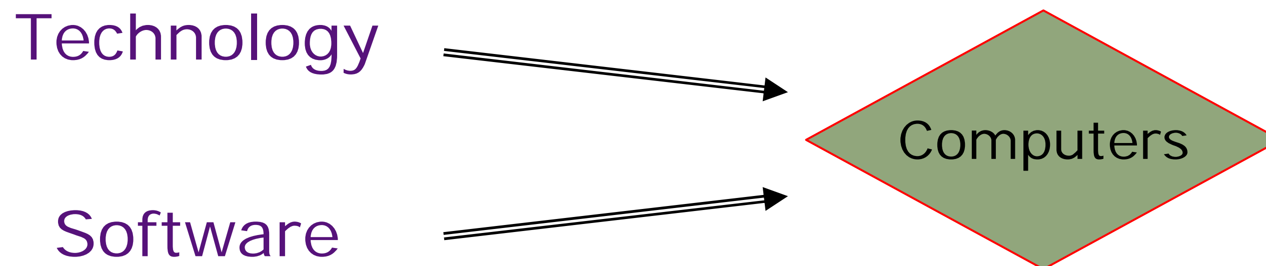


But Software…

As people write programs and use computers, our understanding of *programming* and *program behavior* improves.

This has profound though slower impact on computer architecture

Modern architects cannot avoid paying attention to software and compilation issues.



Computers in mid 50's

- Hardware was expensive
- Stores were small (1000 words)
  - ⇒ No resident system-software!
- Memory access time was 10 to 50 times slower than the processor cycle
  - ⇒ Instruction execution time was totally dominated by the *memory reference time*.
- The *ability to design complex control circuits* to execute an instruction was the central design concern as opposed to *the speed* of decoding or an ALU operation
- Programmer's view of the machine was inseparable from the actual hardware implementation

CSAIL

Programmer's view of the machine IBM 650

A drum machine with 44 instructions

Instruction: 60 1234 1009

- "Load the contents of location 1234 into the *distribution*; put it also into the *upper accumulator*; set *lower accumulator* to zero; and then go to location 1009 for the next instruction."

Good programmers optimized the placement of instructions on the drum to reduce latency!

CSAIL

The Earliest Instruction Sets

Single Accumulator - A carry-over from the calculators.

LOAD	x	$AC \leftarrow M[x]$
STORE	x	$M[x] \leftarrow (AC)$
ADD	x	$AC \leftarrow (AC) + M[x]$
SUB	x	
MUL	x	Involved a quotient register
DIV	x	
SHIFT LEFT		$AC \leftarrow 2 \times (AC)$
SHIFT RIGHT		
JUMP	x	$PC \leftarrow x$
JGE	x	if $(AC) \geq 0$ then $PC \leftarrow x$
LOAD ADR	x	$AC \leftarrow$ Extract address field(M[x])
STORE ADR	x	

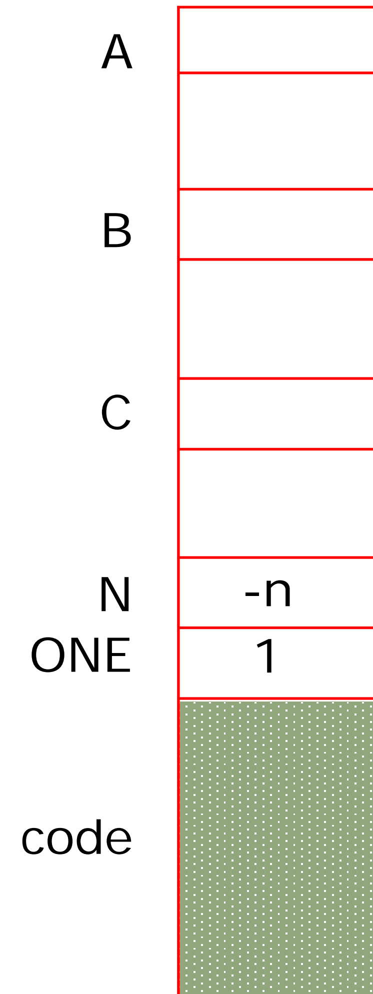
Typically less than 2 dozen instructions!

CSAIL

Programming: Single Accumulator Machine

$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$

```
LOOP    LOAD     N
        JGE      DONE
        ADD      ONE
        STORE    N
F1      LOAD     A
F2      ADD      B
F3      STORE    C
        JUMP     LOOP
DONE    HLT
```



How to modify the addresses A, B and C ?

Self-Modifying Code

$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$

```
LOOP    LOAD        N
        JGE         DONE
        ADD         ONE
        STORE       N
F1      LOAD        A
F2      ADD         B
F3      STORE       C
        LOAD ADR    F1
        ADD         ONE
        STORE ADR   F1
        LOAD ADR    F2
        ADD         ONE
        STORE ADR   F2
        LOAD ADR    F3
        ADD         ONE
        STORE ADR   F3
        JUMP        LOOP
DONE    HLT
```

modify the program for the next iteration

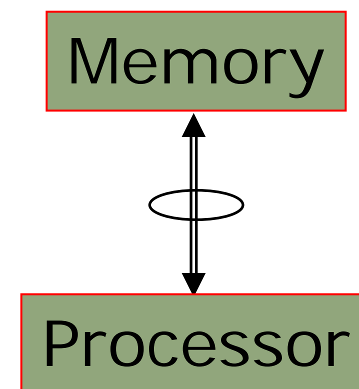
Each iteration involves

	total	*book-keeping*
instruction fetches	*17*	*14*
operand fetches	*10*	*8*
stores	*5*	*4*

CSAIL

Processor-Memory Bottleneck: Early Solutions

- Fast local storage in the processor
  - 8-16 registers as opposed to one accumulator
- Indexing capability
  - to reduce book keeping instructions
- Complex instructions
  - to reduce instruction fetches
- Compact instructions
  - implicit address bits for operands, to reduce instruction fetches



Processor State

The information held in the processor at the end of an instruction to provide the processing context for the next instruction.

Program Counter, Accumulator, . . .

Programmer visible state of the processor (and memory) plays a central role in computer organization for both hardware and software:

- *Software* must make efficient use of it
- If the processing of an instruction can be interrupted then the *hardware* must save and restore the state in a transparent manner

Programmer's machine model is a contract between the hardware and software

Index Registers

Tom Kilburn, Manchester University, mid 50's

One or more specialized registers to simplify address calculation

Modify existing instructions

LOAD	x, IX	AC ← M[x + (IX)]
ADD	x, IX	AC ← (AC) + M[x + (IX)]
...		

Add new instructions to manipulate *index registers*

JZi	x, IX	if (IX)=0 *then* PC ← x
		else IX ← (IX) + 1
LOADi	x, IX	IX ← M[x] (truncated to fit IX)
...		

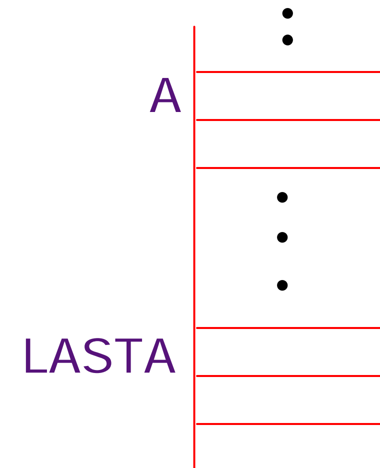
Index registers have accumulator-like characteristics

CSAIL

Using Index Registers

$C_i \leftarrow A_i + B_i, \quad 1 \le i \le n$

```
        LOADi   -n, IX
LOOP    JZi     DONE, IX
        LOAD    LASTA, IX
        ADD     LASTB, IX
        STORE   LASTC, IX
        JUMP    LOOP
DONE    HALT
```



- *Program does not modify itself*
- *Efficiency has improved dramatically (ops / iter)*

	with index regs	without index regs
instruction fetch	5(2)	17 (14)
operand fetch	2	10 (8)
store	1	5 (4)

- *Costs:* Instructions are 1 to 2 bits longer
  Index registers with ALU-like circuitry
  Complex control

CSAIL

Indexing *vs.* Index Registers

Suppose instead of registers, memory locations are used to implement index registers.

LOAD x, IX

Arithmetic operations on index registers can be performed by bringing the contents to the accumulator

Most bookkeeping instructions will be avoided but each instruction will implicitly cause several fetches and stores

$\Rightarrow$ *complex control circuitry*

CSAIL

Operations on Index Registers

To increment index register by k

$AC \leftarrow (IX)$ *new instruction*

$AC \leftarrow (AC) + k$

$IX \leftarrow (AC)$ *new instruction*

also the AC must be saved and restored.

It may be better to increment IX directly

INCi k, IX $IX \leftarrow (IX) + k$

More instructions to manipulate index register

STOREi x, IX $M[x] \leftarrow (IX)$ (extended to fit a word)

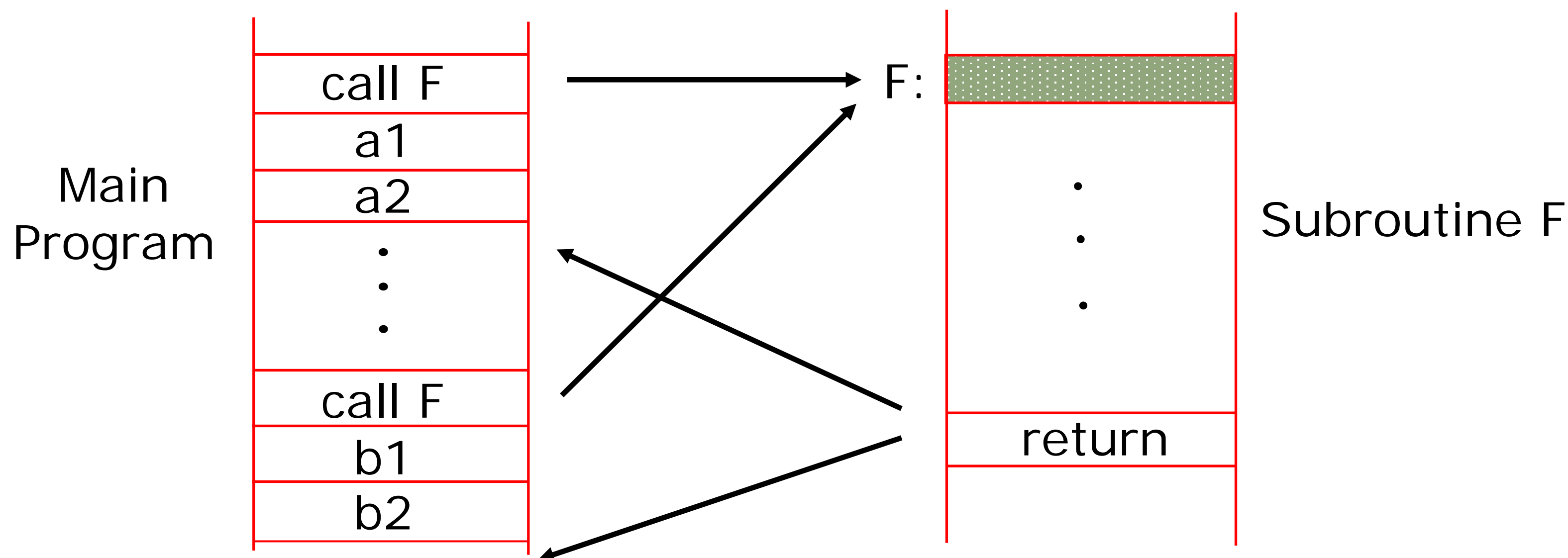
...

IX begins to look like an accumulator

$\Rightarrow$ several index registers
several accumulators

$\Rightarrow$ *General Purpose Registers*

Support for Subroutine Calls



A special *subroutine jump instruction*

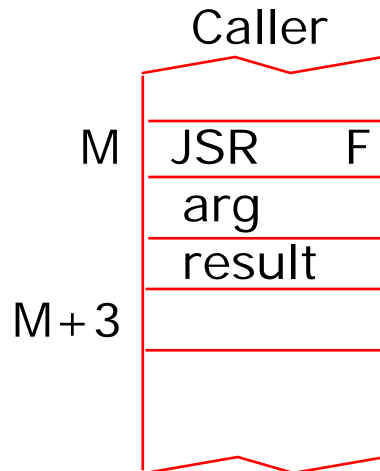
M: JSR F

$F \leftarrow M + 1$ and jump to $F+1$

Indirect Addressing and Subroutine Calls

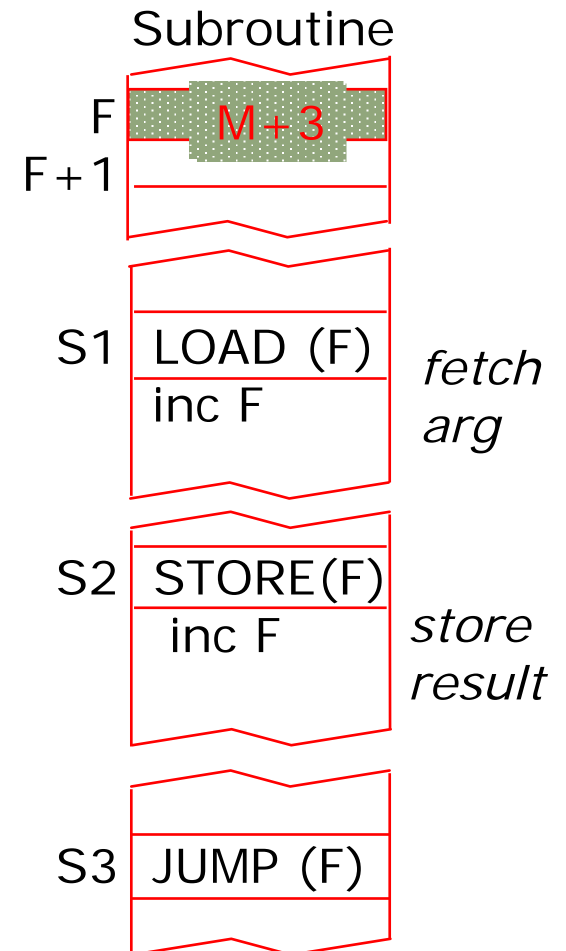
Indirect addressing

LOAD (x) means AC ← M[M[x]]
...



Events:

Execute M
Execute S1
Execute S2
Execute S3



Indirect addressing almost eliminates the need to write self-modifying code (location F still needs to be modified)

⇒ *Problems with recursive procedure calls*

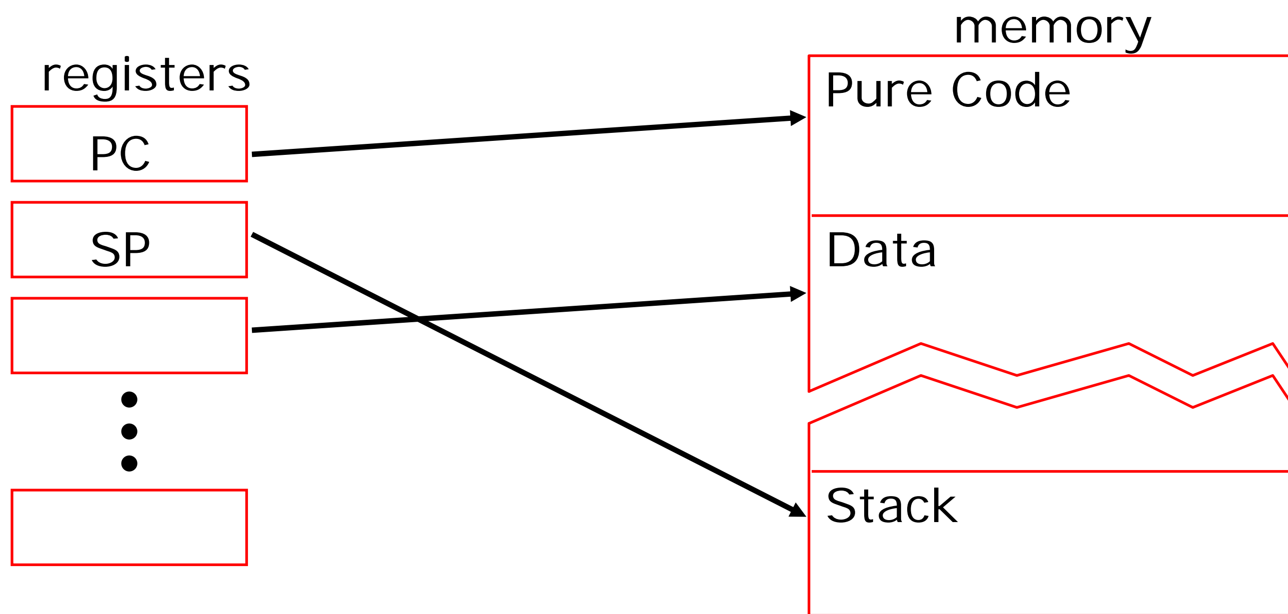
CSAIL

Recursive Procedure Calls and Reentrant Codes

Indirect Addressing through a register

LOAD R_1, (R_2)

Load register R_1 with the contents of the word whose address is contained in register R_2



CSAIL

Evolution of Addressing Modes

1. Single accumulator, absolute address

   LOAD x

2. Single accumulator, index registers

   LOAD x, IX

3. Indirection

   LOAD (x)

4. Multiple accumulators, index registers, indirection

   LOAD R, IX, x

   or LOAD R, IX, (x) the meaning?

   $R \leftarrow M[M[x] + (IX)]$

   or $R \leftarrow M[M[x + (IX)]]$

5. Indirect through registers

   LOAD R_I, (R_J)

6. The works

   LOAD R_I, R_J, (R_K) R_J = index, R_K = base addr

Variety of Instruction Formats

- *Two address formats:* the destination is same as one of the operand sources

(Reg × Reg) to Reg $R_I \leftarrow (R_I) + (R_J)$
(Reg × Mem) to Reg $R_I \leftarrow (R_I) + M[x]$

  - x can be specified directly or via a register
  - effective address calculation for x could include indexing, indirection, ...

- *Three address formats:* One destination and up to two operand sources per instruction

(Reg x Reg) to Reg $R_I \leftarrow (R_J) + (R_K)$
(Reg x Mem) to Reg $R_I \leftarrow (R_J) + M[x]$

CSAIL

More Instruction Formats

- *Zero address formats:* operands on a stack

add	M[sp-1] ← M[sp] + M[sp-1]
load	M[sp] ← M[M[sp]]

  – Stack can be in registers or in memory (usually top of stack cached in registers)

- *One address formats:* Accumulator machines
  – Accumulator is always other implicit operand

Many different formats are possible!

CSAIL

Data Formats and Memory Addresses

Data formats:

Bytes, Half words, words and double words

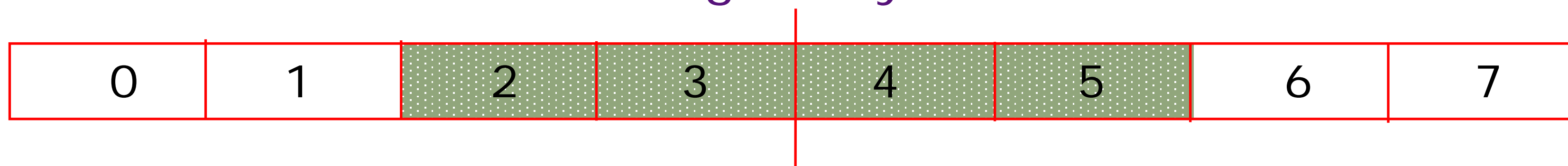
Some issues

- *Byte addressing*

Big Endian	0	1	2	3
vs. Little Endian	3	2	1	0

- *Word alignment*

Suppose the memory is organized in 32-bit words.
Can a word address begin only at 0, 4, 8, ... ?



Some Problems

- Should all addressing modes be provided for every operand?

  ⇒ *regular vs. irregular instruction formats*

- Separate instructions to manipulate Accumulators, Index registers, Base registers

  ⇒ *large number of instructions*

- Instructions contained implicit memory references -- several contained more than one

  ⇒ *very complex control*

CSAIL

Compatibility Problem at IBM

By early 60's, *IBM had 4 incompatible lines of computers!*

701	→	7094
650	→	7074
702	→	7080
1401	→	7010

Each system had its own

- Instruction set
- I/O system and Secondary Storage:
  magnetic tapes, drums and disks
- assemblers, compilers, libraries,...
- market niche
  business, scientific, real time, ...

⇒ *IBM 360*

CSAIL

IBM 360 : Design Premises

Amdahl, Blaauw and Brooks, 1964

- The design must lend itself to *growth and successor machines*
- General method for connecting I/O devices
- Total performance - answers per month rather than bits per microsecond $\Rightarrow$ *programming aids*
- Machine must be capable of *supervising itself* without manual intervention
- Built-in *hardware fault checking* and locating aids to reduce down time
- Simple to assemble systems with redundant I/O devices, memories etc. for *fault tolerance*
- Some problems required floating point words larger than 36 bits

IBM 360: *A General-Purpose Register (GPR) Machine*

- Processor State
  - 16 General-Purpose 32-bit Registers
    - *may be used as index and base register*
    - *Register 0 has some special properties*
  - 4 Floating Point 64-bit Registers
  - A Program Status Word (PSW)
    - *PC, Condition codes, Control flags*
- A 32-bit machine with 24-bit addresses
  - No instruction contains a 24-bit address !
- Data Formats
  - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words

IBM 360: Implementation

	Model 30	. . .	*Model 70*
Storage	8K - 64 KB		256K - 512 KB
Datapath	8-bit		64-bit
Circuit Delay	30 nsec/level		5 nsec/level
Local Store	Main Store		Transistor Registers
Control Store	Read only 1μsec		Conventional circuits

IBM 360 instruction set architecture completely hid the underlying technological differences between various models.

With minor modifications it survives till today

CSAIL

IBM S/390 z900 Microprocessor

- 64-bit virtual addressing
  - first 64-bit S/390 design (original S/360 was 24-bit, and S/370 was 31-bit extension)
- 1.1 GHz clock rate (announced ISSCC 2001)
  - 0.18μm CMOS, 7 layers copper wiring
  - 770MHz systems shipped in 2000
- Single-issue 7-stage CISC pipeline
- Redundant datapaths
  - every instruction performed in two parallel datapaths and results compared
- 256KB L1 I-cache, 256KB L1 D-cache on-chip
- 20 CPUs + 32MB L2 cache per Multi-Chip Module
- Water cooled to 10°C junction temp

CSAIL

What makes a good instruction set?

One that provides a simple software interface yet allows simple, fast, efficient hardware implementations

... but across 25+ year time frame

Example of difficulties:

- Current machines have register files with more storage than entire main memory of early machines!
- On-chip test circuitry in current machines has hundreds of times more transistors than entire early computers!

CSAIL

Full Employment for Architects

- Good news: "Ideal" instruction set changes continually
  - Technology allows larger CPUs over time
  - Technology constraints change (e.g., now it is power)
  - Compiler technology improves (e.g., register allocation)
  - Programming styles change (assembly, HLL, object-oriented, ...)
  - Applications change (e.g., multimedia, ...)
- Bad news: Software compatibility imposes huge damping coefficient on instruction set innovation
  - Software investment dwarfs hardware investment
  - Innovate at microarchitecture level, below the ISA level (this is what most computer architects do)
- New instruction set can only be justified by new large market and technological advantage
  - Network processors
  - Multimedia processors
  - DSP's

CSAIL

Instruction Set Evolution in the Sixties: *GPR, Stack, and Load-Store Architectures*

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Based on the material prepared by Arvind and Krste Asanovic

The Sixties

- Hardware costs started dropping
  - memories beyond 32K words seemed likely
  - separate I/O processors
  - large register files
- Systems software development became essential
  - Operating Systems
  - I/O facilities
- Separation of Programming Model from implementation become essential
  - family of computers

CSAIL

Issues for Architects in the Sixties

- Stable base for software development
- Support for operating systems
  - processes, multiple users, I/O
- Implementation of high-level languages
  - recursion, ...
- Impact of large memories on instruction size
- How to organize the *processor state* from the programming point of view
- Architectures for which fast implementations could be developed

Three Different Directions in the Sixties

- A machine with only a high-level language interface
  - Burrough's 5000, a stack machine
- A family of computers based on a common ISA
  - IBM 360, a General Register Machine
- A pipelined machine with a fast clock (Supercomputer)
  - CDC 6600, a Load/Store architecture

CSAIL

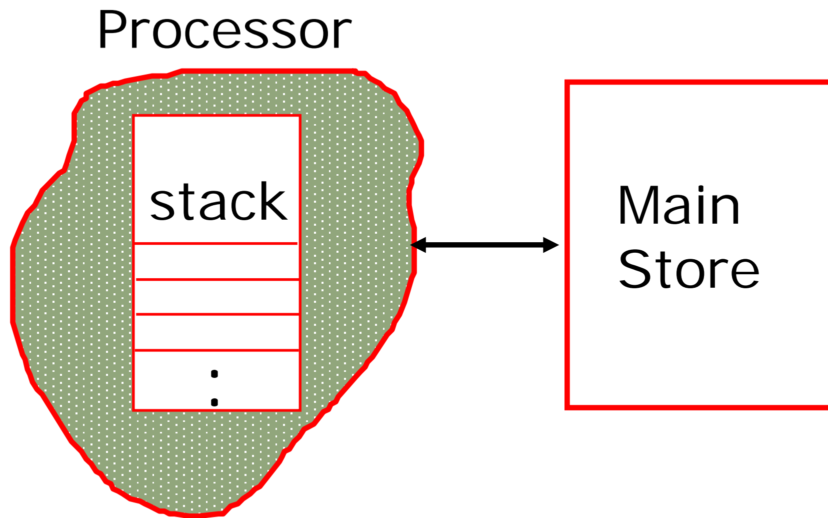
The Burrough's B5000:
An ALGOL Machine, Robert Barton, 1960

- Machine implementation can be completely hidden if the programmer is provided only a high-level language interface.

- *Stack machine* organization because stacks are convenient for:
    1. expression evaluation;
    2. subroutine calls, recursion, nested interrupts;
    3. accessing variables in block-structured languages.

- B6700, a later model, had many more innovative features
    – tagged data
    – virtual memory
    – multiple processors and memories

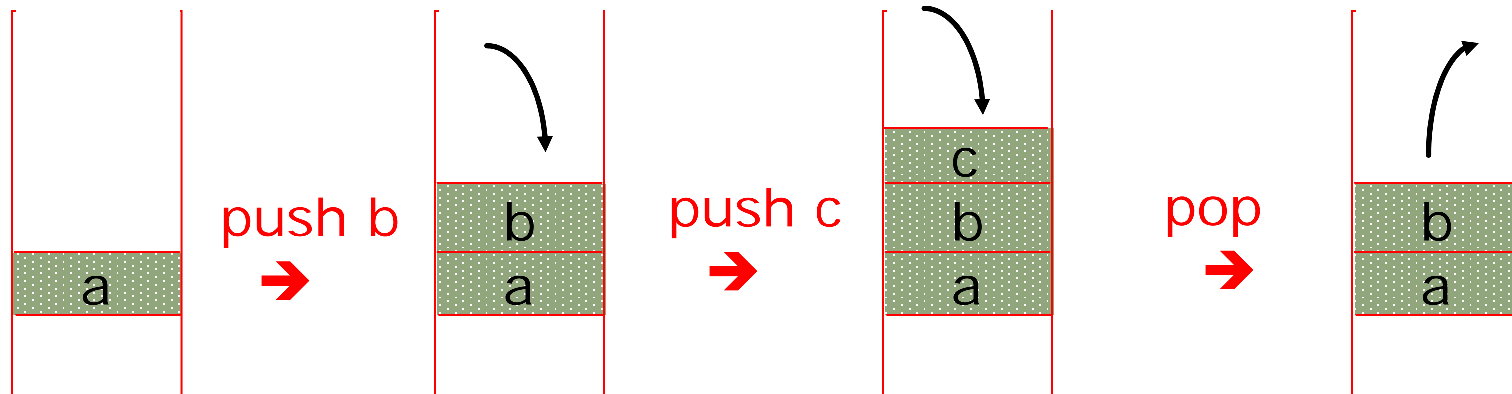
A Stack Machine



A Stack machine has a stack as a part of the processor state

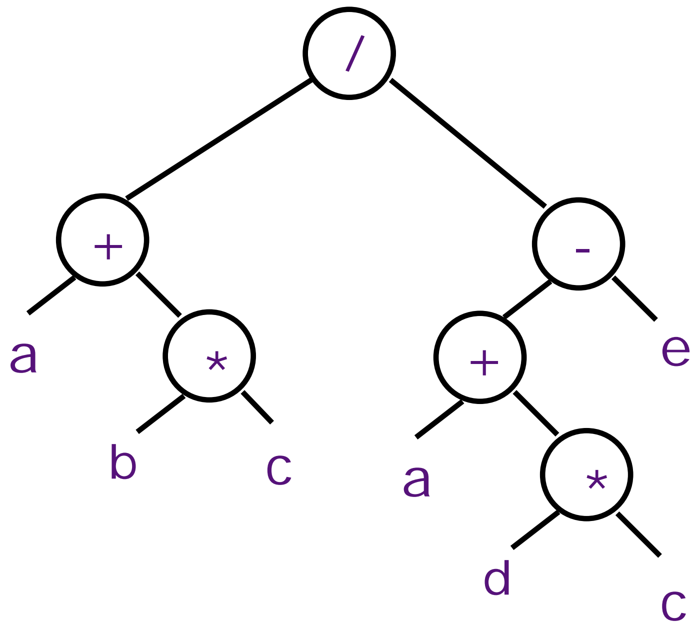
typical operations:
push, pop, +, *, ...

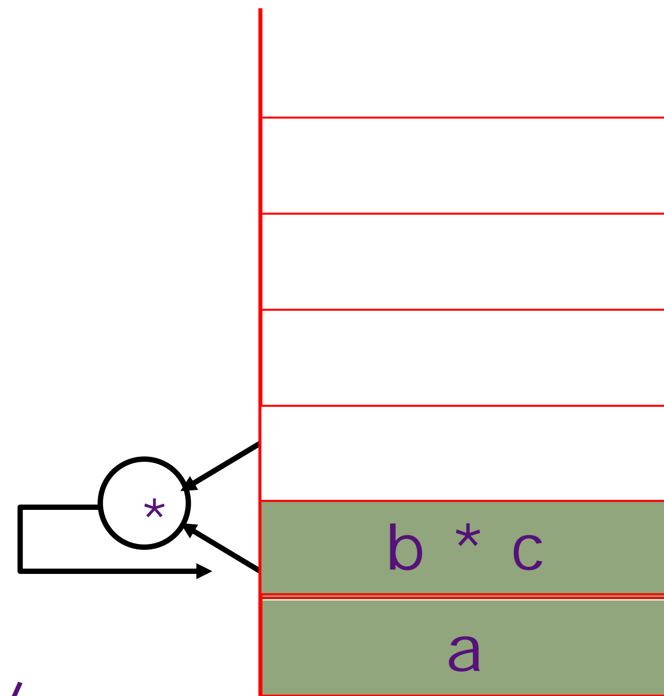
Instructions like + implicitly specify the top 2 elements of the stack as operands.



Evaluation of Expressions

(a + b * c) / (a + d * c - e)

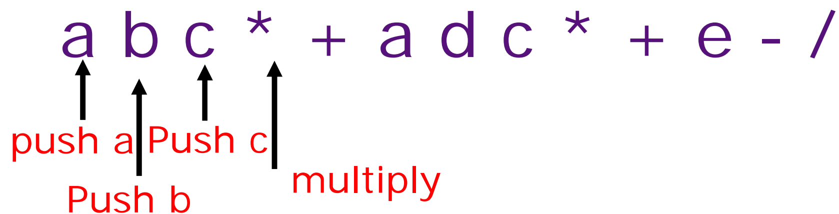




Evaluation Stack

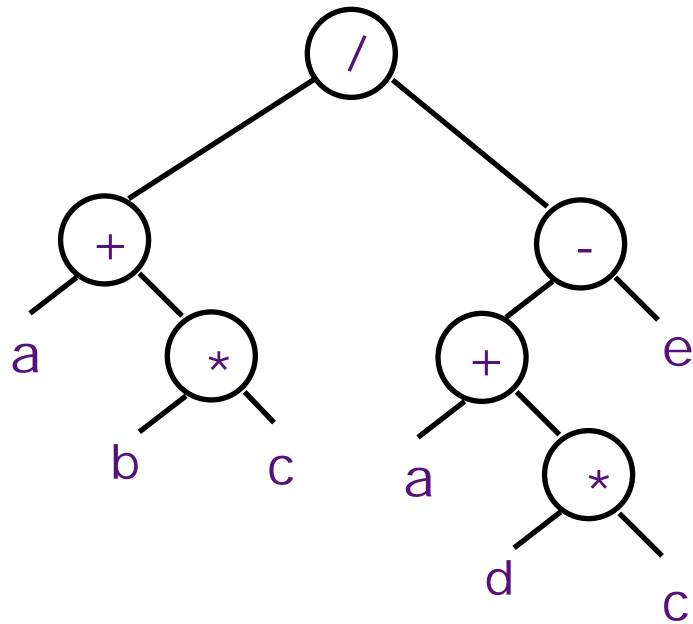
Reverse Polish

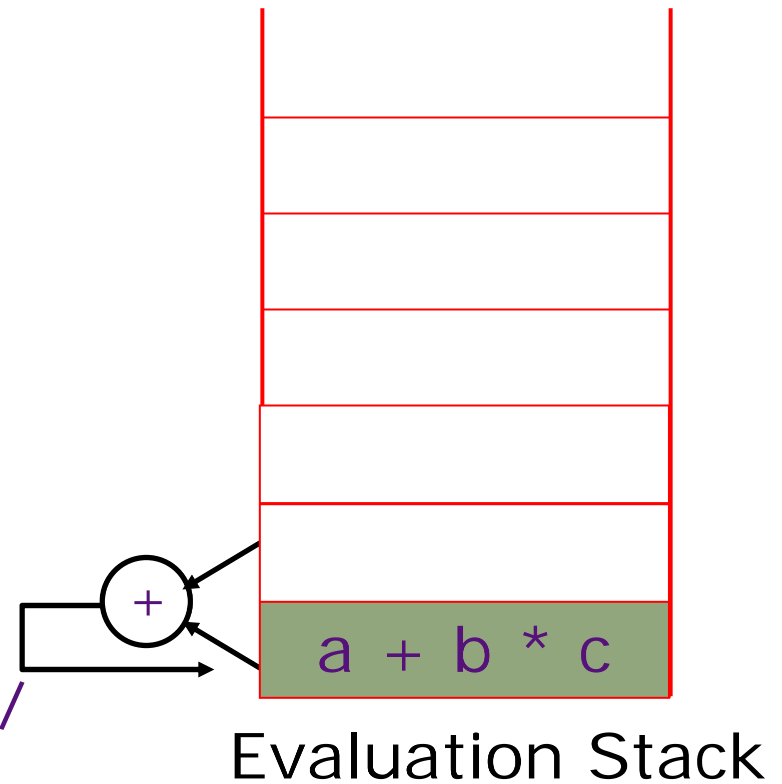
a b c * + a d c * + e - /



Evaluation of Expressions

(a + b * c) / (a + d * c - e)





Reverse Polish

a b c * + a d c * + e - /

add

Hardware organization of the stack

- Stack is part of the processor state
  - ⇒ *stack must be bounded and small*
    ≈ number of Registers,
    not the size of main memory

- Conceptually stack is unbounded
  - ⇒ *a part of the stack is included in the processor state; the rest is kept in the main memory*

CSAIL

Stack Size and Memory References

a b c * + a d c * + e - /

program	*stack (size = 2)*	*memory refs*
push a	R0	a
push b	R0 R1	b
push c	R0 R1 R2	c, ss(a)
*	R0 R1	sf(a)
+	R0	
push a	R0 R1	a
push d	R0 R1 R2	d, ss(a+b*c)
push c	R0 R1 R2 R3	c, ss(a)
*	R0 R1 R2	sf(a)
+	R0 R1	sf(a+b*c)
push e	R0 R1 R2	e,ss(a+b*c)
-	R0 R1	sf(a+b*c)
/	R0	

4 stores, 4 fetches (implicit)

CSAIL

Stack Operations and Implicit Memory References

- Suppose the top 2 elements of the stack are kept in registers and the rest is kept in the memory.

Each *push* operation $\Rightarrow$ 1 memory reference
pop operation $\Rightarrow$ 1 memory reference

No Good!

- Better performance can be got if the top N elements are kept in registers and memory references are made only when register stack overflows or underflows.

Issue - when to Load/Unload registers ?

CSAIL

Stack Size and Expression Evaluation

a b c * + a d c * + e - /

program	*stack (size = 2)*
push a	R0
push b	R0 R1
push c	R0 R1 R2
*	R0 R1
+	R0
push a	R0 R1
push d	R0 R1 R2
push c	R0 R1 R2 R3
*	R0 R1 R2
+	R0 R1
push e	R0 R1 R2
-	R0 R1
/	R0

a and c are "loaded" twice

. ⇒

not the best use of registers!

CSAIL

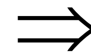
Register Usage in a GPR Machine

(a + b * c) / (a + d * c - e)

	Load	R0	a
	Load	R1	c
	Load	R2	b
Reuse R2	Mul	R2	R1
	Add	R2	R0
	Load	R3	d
Reuse R3	Mul	R3	R1
	Add	R3	R0
Reuse R0	Load	R0	e
	Sub	R3	R0
	Div	R2	R3

More control over register usage since registers can be named explicitly

Load	Ri m
Load	Ri (Rj)
Load	Ri (Rj) (Rk)

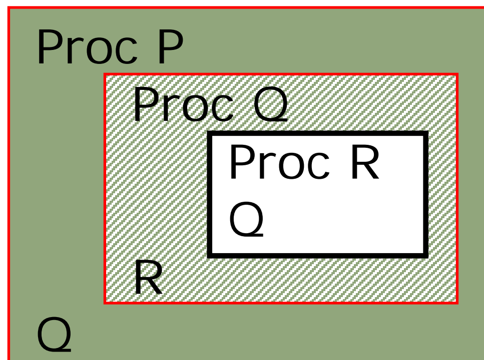


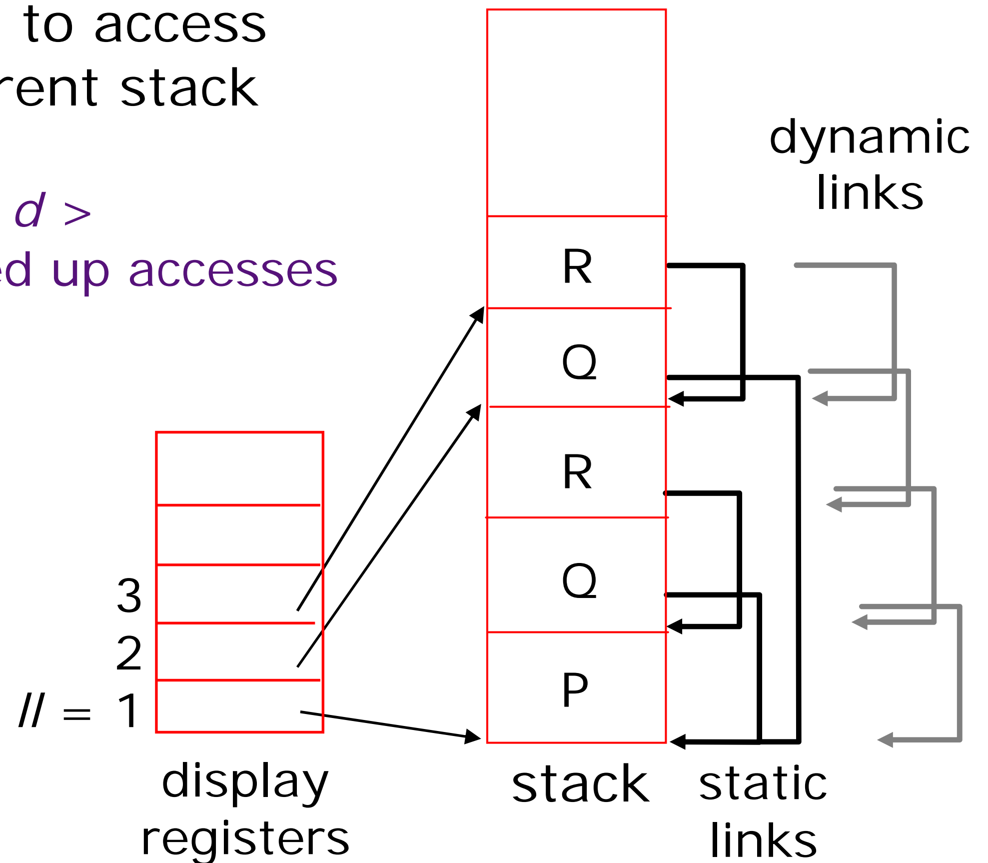
- *eliminates unnecessary Loads and Stores*
- *fewer Registers*

but instructions may be longer!

Procedure Calls

- Storage for procedure calls also follows a stack discipline
- However, there is a need to access variables beyond the current stack frame
  - lexical addressing < *ll* , *d* >
  - display registers to speed up accesses to stack frames

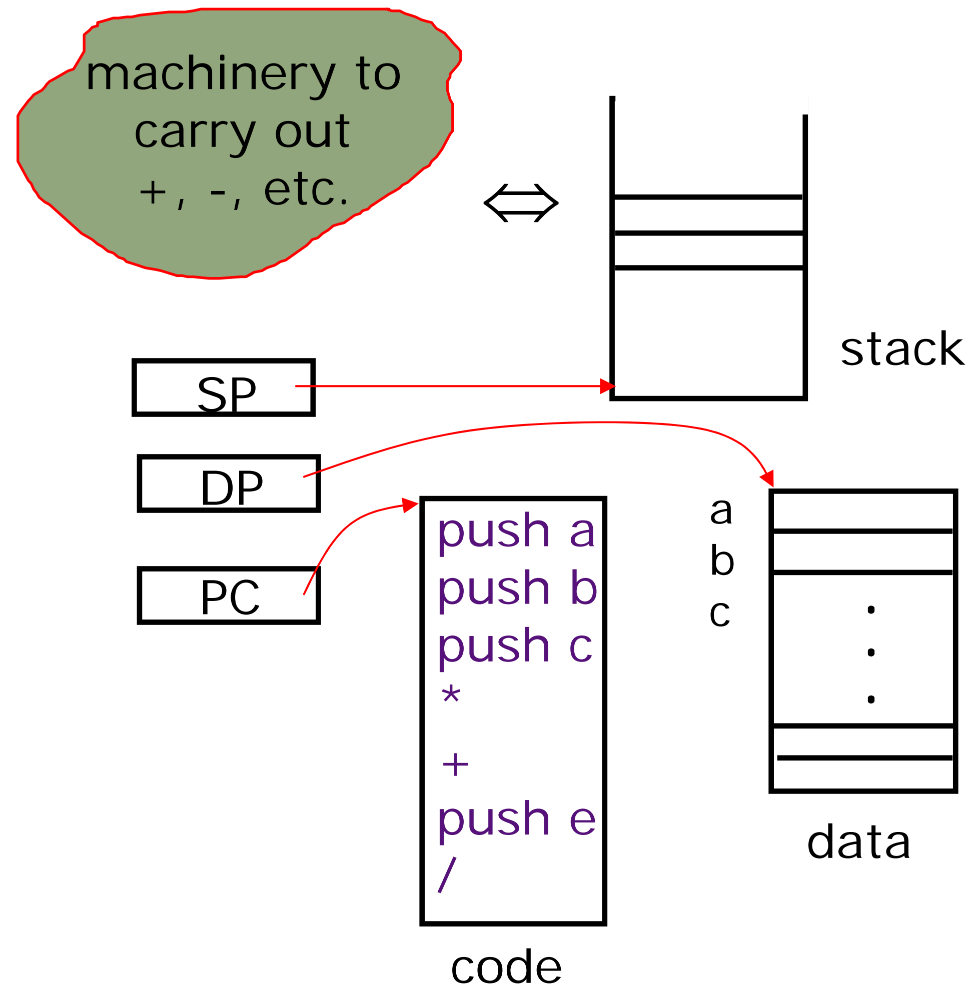




automatic loading of display registers?

Stack Machines: Essential features

- In addition to push, pop, + etc., the instruction set must provide the capability to
  - *refer to any element in the data area*
  - *jump to any instruction in the code area*
  - *move any element in the stack frame to the top*



Stack versus GPR Organization
Amdahl, Blaauw and Brooks, 1964

1. The performance advantage of push down stack organization is derived from the presence of fast registers and not the way they are used.
2. "Surfacing" of data in stack which are "profitable" is approximately 50% because of constants and common subexpressions.
3. Advantage of instruction density because of implicit addresses is equaled if short addresses to specify registers are allowed.
4. Management of finite depth stack causes complexity.
5. Recursive subroutine advantage can be realized only with the help of an independent stack for addressing.
6. Fitting variable length fields into fixed width word is awkward.

CSAIL

Stack Machines (Mostly) Died by 1980

1. Stack programs are not smaller if short (Register) addresses are permitted.

2. Modern compilers can manage fast register space better than the stack discipline.

3. Lexical addressing is a useful abstract model for compilers but hardware support for it (i.e. display) is not necessary.

GPR's and caches are better than stack and displays

Early language-directed architectures often did not take into account the role of compilers!

B5000, B6700, HP 3000, ICL 2900, Symbolics 3600

CSAIL

Stacks post-1980

- Inmos Transputers (1985-2000)
  - Designed to support many parallel processes in Occam language
  - Fixed-height stack design simplified implementation
  - Stack trashed on context swap (fast context switches)
  - Inmos T800 was world's fastest microprocessor in late 80's
- Forth machines
  - Direct support for Forth execution in small embedded real-time environments
  - Several manufacturers (Rockwell, Patriot Scientific)
- Java Virtual Machine
  - Designed for software emulation not direct hardware execution
  - Sun PicoJava implementation + others
- Intel x87 floating-point unit
  - Severely broken stack model for FP arithmetic
  - Deprecated in Pentium-4 replaced with SSE2 FP registers

A five-minute break to stretch your legs

IBM 360: A General-Purpose Register (GPR) Machine

- Processor State
  - 16 General-Purpose 32-bit Registers
    - *may be used as index and base register*
    - *Register 0 has some special properties*
  - 4 Floating Point 64-bit Registers
  - A Program Status Word (PSW)
    - *PC, Condition codes, Control flags*
- A 32-bit machine with 24-bit addresses
  - No instruction contains a 24-bit address !
- Data Formats
  - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words

IBM 360: Precise Interrupts

- IBM 360 ISA (Instruction Set Architecture) preserves sequential execution model
- Programmers view of machine was that each instruction either completed or signaled a fault before next instruction began execution
- Exception/interrupt behavior constant across family of implementations

CSAIL

IBM 360: Original family

	Model 30	. . .	*Model 70*
Storage	8K - 64 KB		256K - 512 KB
Datapath	8-bit		64-bit
Circuit Delay	30 nsec/level		5 nsec/level
Local Store	Main Store		Transistor Registers
Control Store	Read only 1μsec		Conventional circuits

IBM 360 instruction set architecture completely hid the underlying technological differences between various models.

With minor modifications it survives till today

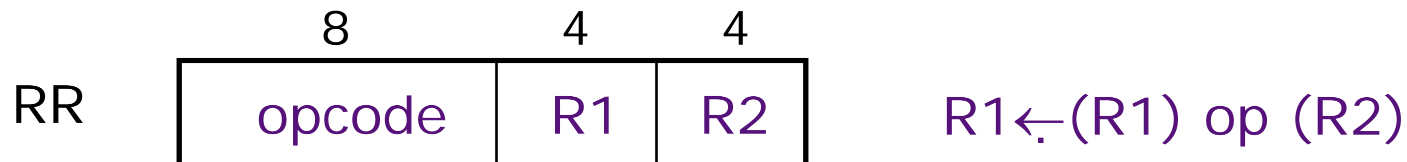
CSAIL

IBM S/390 z900 Microprocessor

- 64-bit virtual addressing
  - first 64-bit S/390 design (original S/360 was 24-bit, and S/370 was 31-bit extension)
- 1.1 GHz clock rate (announced ISSCC 2001)
  - 0.18μm CMOS, 7 layers copper wiring
  - 770MHz systems shipped in 2000
- Single-issue 7-stage CISC pipeline
- Redundant datapaths
  - every instruction performed in two parallel datapaths and results compared
- 256KB L1 I-cache, 256KB L1 D-cache on-chip
- 20 CPUs + 32MB L2 cache per Multi-Chip Module
- Water cooled to 10°C junction temp

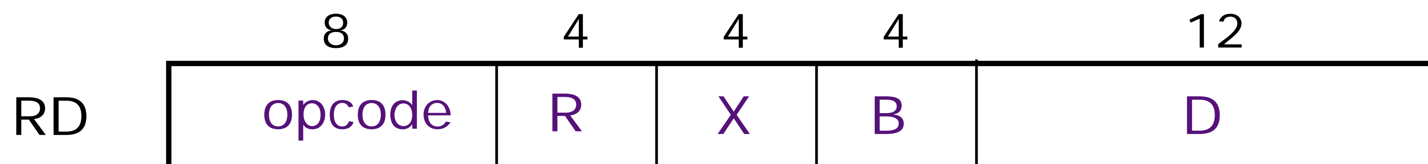
IBM 360: Some Addressing Modes

RR



R1←(R1) op (R2)

RD



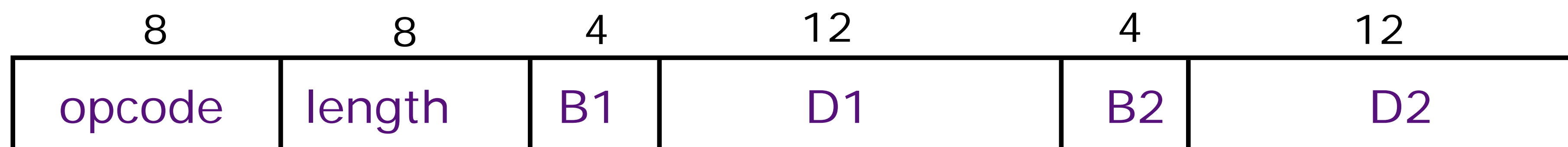
R ← (R) op M[(X) + (B) + D]

a 24-bit address is formed by adding the 12-bit displacement (D) to a base register (B) and an Index register (X), if desired

The most common formats for arithmetic & logic instructions, as well as Load and Store instructions

CSAIL

IBM 360: Character String Operations



SS format: store to store instructions

M[(B1) + D1] ← M[(B1) + D1] op M[(B2) + D2]

iterate "length" times

Most operations on decimal and character strings use this format

MVC move characters
MP multiply two packed decimal strings
CLC compare two character strings
...

Multiple memory operations per instruction complicates exception & interrupt handling

CSAIL

IBM 360: Branches & Condition Codes

- Arithmetic and logic instructions set *condition codes*
  - equal to zero
  - greater than zero
  - overflow
  - carry...
- I/O instructions also set condition codes
  - channel busy
- Conditional branch instructions are based on testing condition code registers (CC's)
  - RX and RR formats
    - BC_ branch conditionally
    - BAL_ branch and link, i.e., R15 ← (PC)+1 *for subroutine calls*

      ⇒ CC's must be part of the PSW

CSAIL

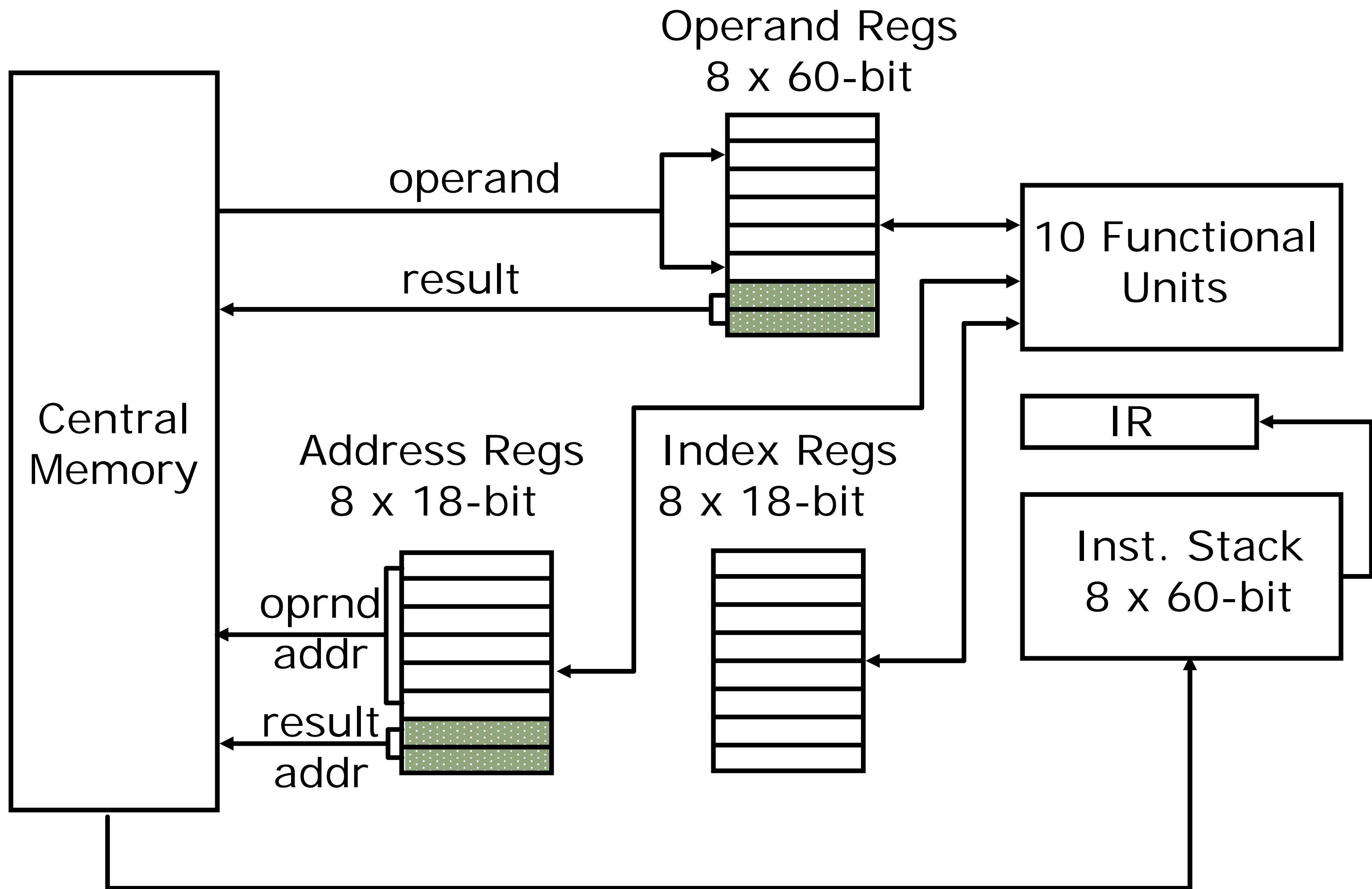
CDC 6600 *Seymour Cray, 1964*

- A fast pipelined machine with 60-bit words
- Ten functional units
  - Floating Point: adder, multiplier, divider
  - Integer: adder, multiplier

  ...
- Hardwired control (no microcoding)
- Dynamic scheduling of instructions using a scoreboard
- Ten Peripheral Processors for Input/Output
  - a fast time-shared 12-bit integer ALU
- Very fast clock
- Novel freon-based technology for cooling

CSAIL

CDC 6600: Datapath

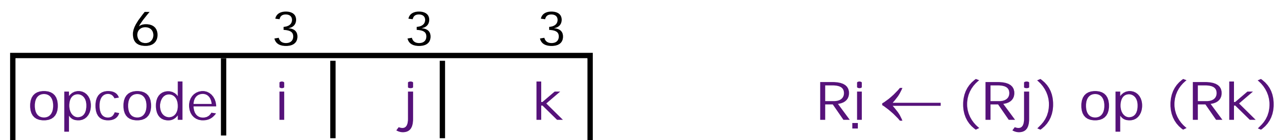


CDC 6600: A Load/Store Architecture

- Separate instructions to manipulate three types of reg.

  8 60-bit data registers (X)
  8 18-bit address registers (A)
  8 18-bit index registers (B)

- All arithmetic and logic instructions are reg-to-reg



Ri ← (Rj) op (Rk)

- Only Load and Store instructions refer to memory!

Ri ← M[(Rj) + disp]

Touching address registers 1 to 5 initiates a load
6 to 7 initiates a store
- *very useful for vector operations*

CDC6600: Vector Addition

```
        B0 ← - n
loop:   JZE  B0, exit
        A0 ← B0 + a0        load X0
        A1 ← B0 + b0        load X1
        X6 ← X0 + X1
        A6 ← B0 + c0        store X6
        B0 ← B0 + 1
        jump loop
```

Ai = address register
Bi = index register
Xi = data register

CSAIL

What makes a good instruction set?

One that provides a simple software interface yet allows simple, fast, efficient hardware implementations

... but across 25+ year time frame

Example of difficulties:

- Current machines have register files with more storage than entire main memory of early machines!
- On-chip test circuitry in current machines has hundreds of times more transistors than entire early computers!

CSAIL

Full Employment for Architects

- Good news: "Ideal" instruction set changes continually
  - Technology allows larger CPUs over time
  - Technology constraints change (e.g., now it is power)
  - Compiler technology improves (e.g., register allocation)
  - Programming styles change (assembly, HLL, object-oriented, …)
  - Applications change (e.g., multimedia, ….)
- Bad news: Software compatibility imposes huge damping coefficient on instruction set innovation
  - Software investment dwarfs hardware investment
  - Innovate at microarchitecture level, below the ISA level (this is what most computer architects do)
- New instruction set can only be justified by new large market and technological advantage
  - Network processors
  - Multimedia processors
  - DSP's

Microprogramming

Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.

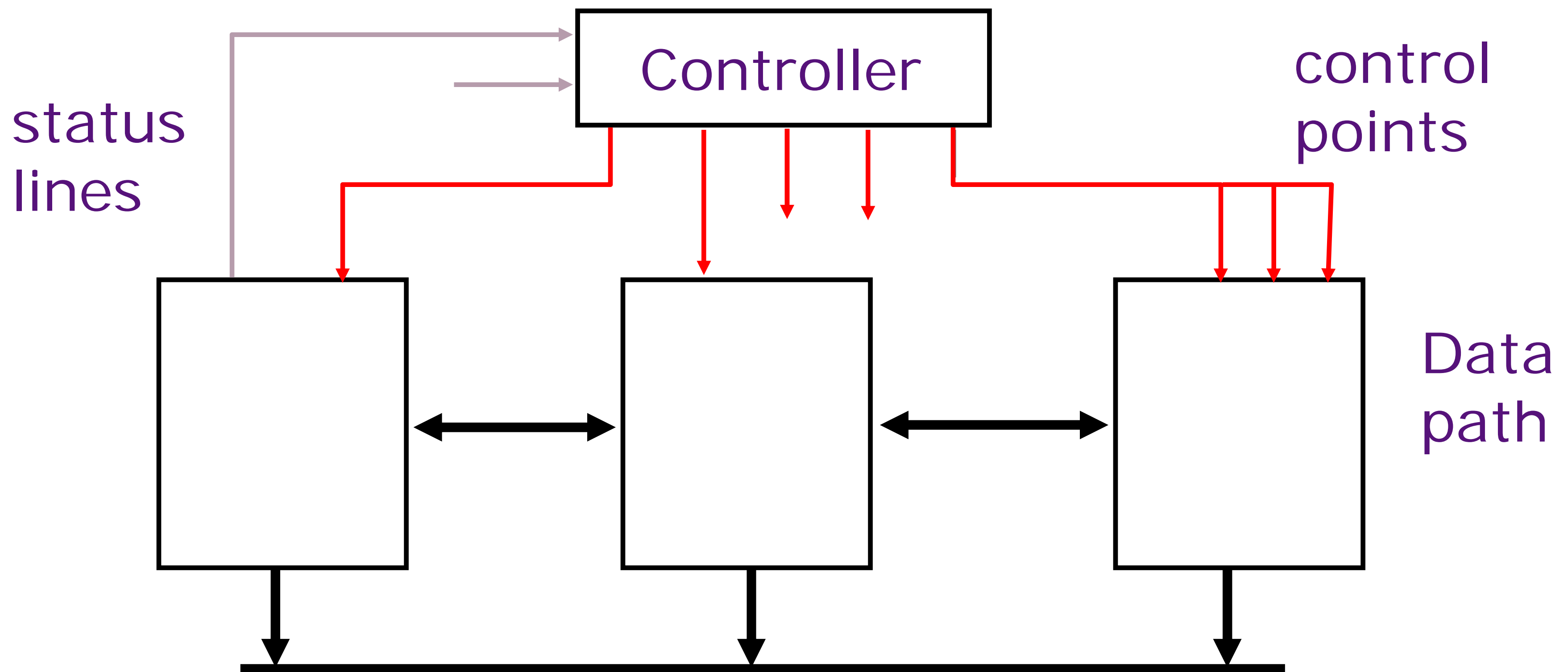
Based on the material prepared by
Arvind and Krste Asanovic

ISA to Microarchitecture Mapping

- An ISA often designed for a particular microarchitectural style, e.g.,
  - CISC ⇒ microcoded
  - RISC ⇒ hardwired, pipelined
  - VLIW ⇒ fixed latency in-order pipelines
  - JVM ⇒ software interpretation
- But an ISA can be implemented in any microarchitectural style
  - Pentium-4: hardwired pipelined CISC (x86) machine (with some microcode support)
  - This lecture: a microcoded RISC (MIPS) machine
  - Intel will probably eventually have a dynamically scheduled out-of-order VLIW (IA-64) processor
  - PicoJava: A hardware JVM processor

CSAIL

Microarchitecture: *Implementation of an ISA*

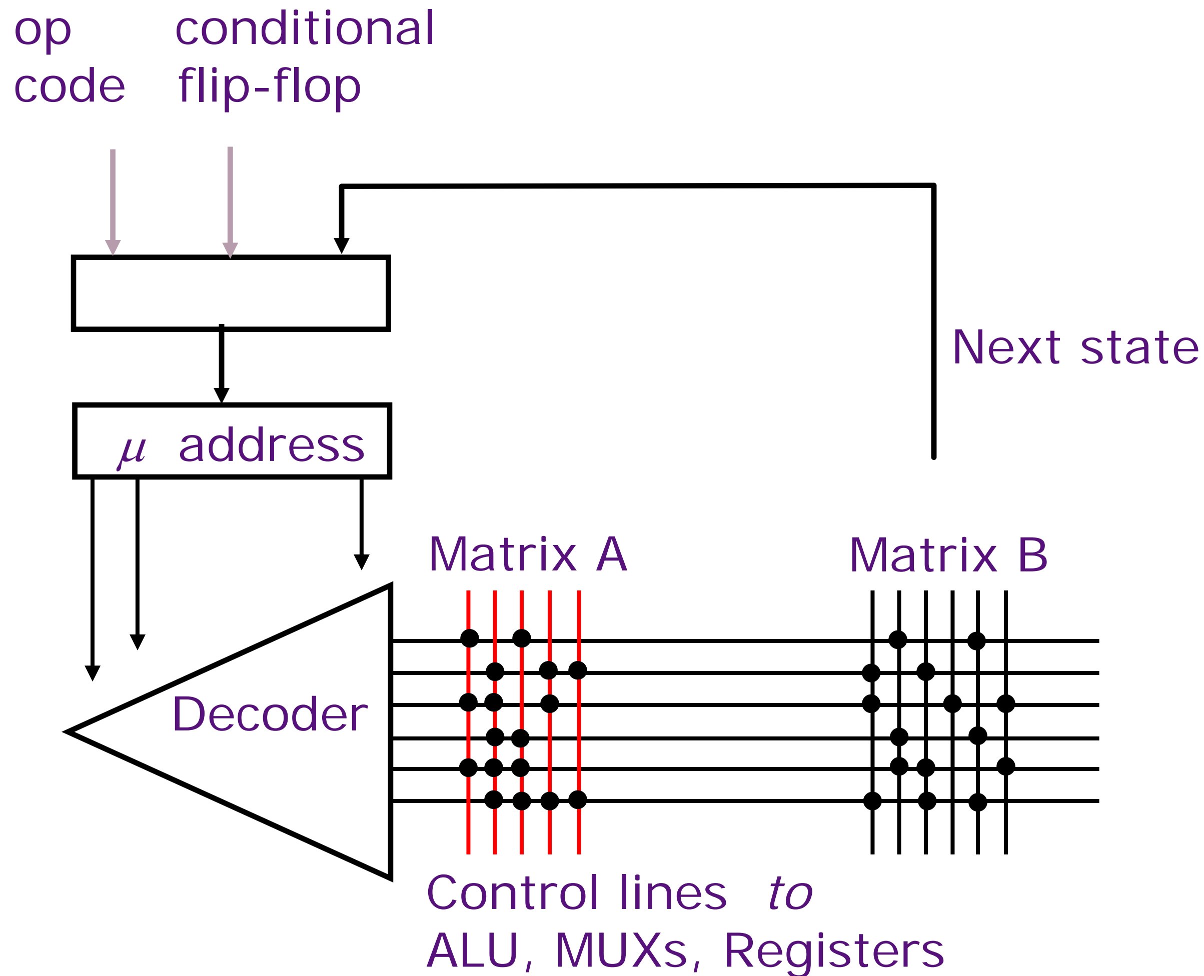


Structure: How components are connected.
Static

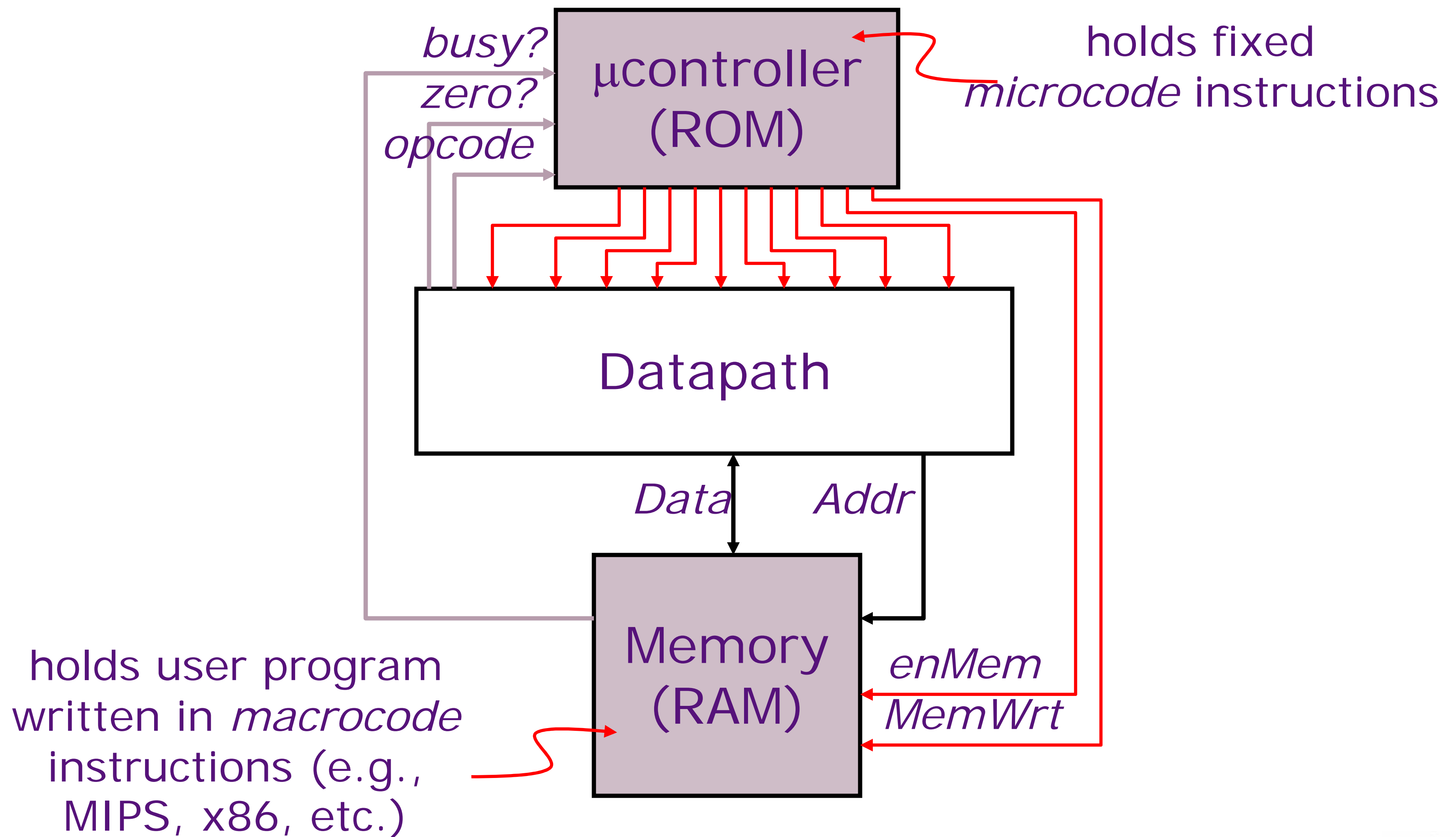
Behavior: How data moves between components
Dynamic

Microcontrol Unit *Maurice Wilkes, 1954*

Embed the control logic state table in a memory array



Microcoded Microarchitecture



The MIPS32 ISA

- Processor State
  - 32 32-bit GPRs, R0 always contains a 0
  - 16 double-precision/32 single-precision FPRs
  - FP status register, used for FP compares & exceptions
  - PC, the program counter
  - some other special registers

See H&P p129-137 & Appendix C (online) for full description

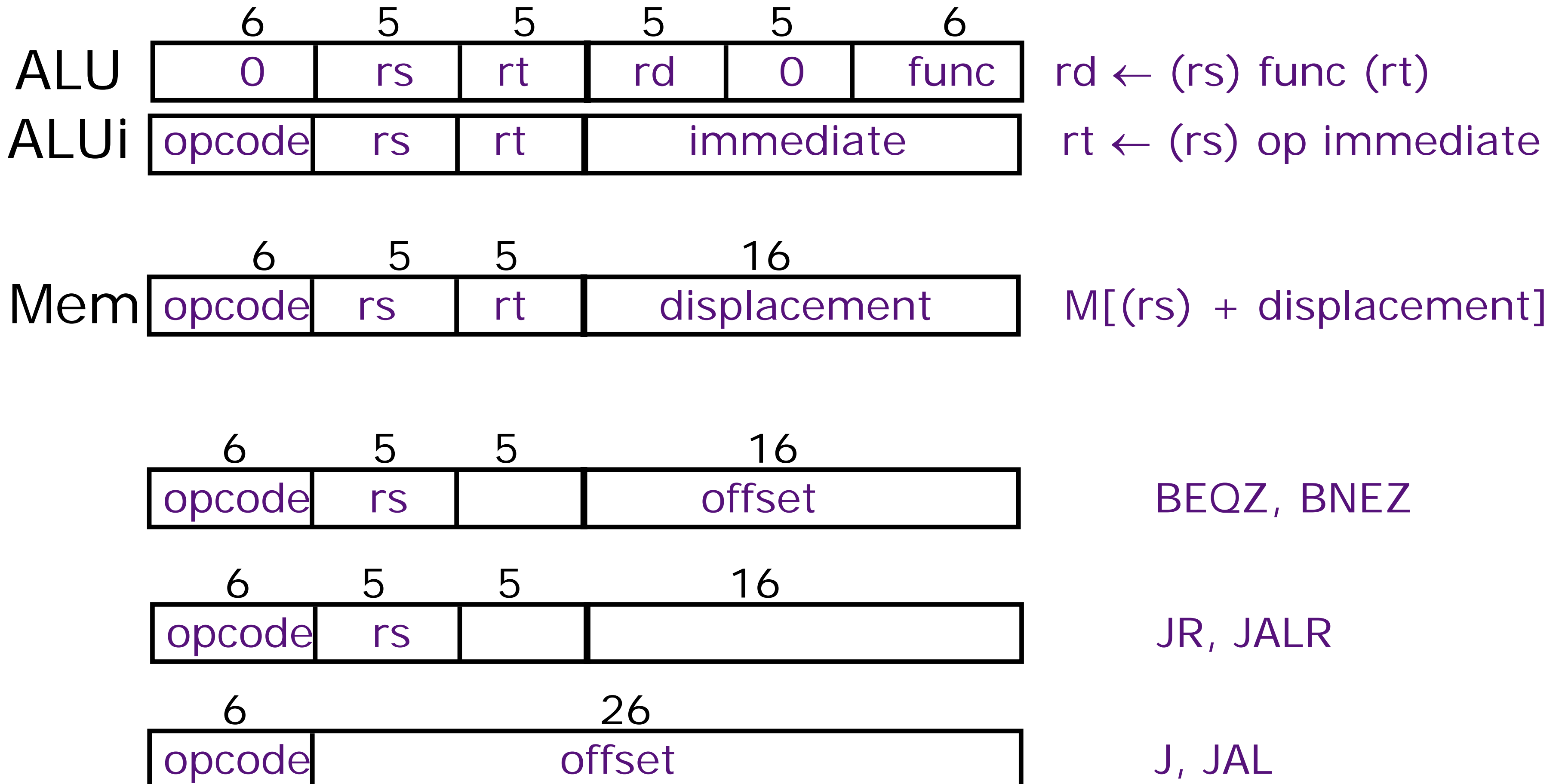
- Data types
  - 8-bit byte, 16-bit half word
  - 32-bit word for integers
  - 32-bit word for single precision floating point
  - 64-bit word for double precision floating point

- Load/Store style instruction set
  - data addressing modes- immediate & indexed
  - branch addressing modes- PC relative & register indirect
  - Byte addressable memory- big-endian mode

All instructions are 32 bits

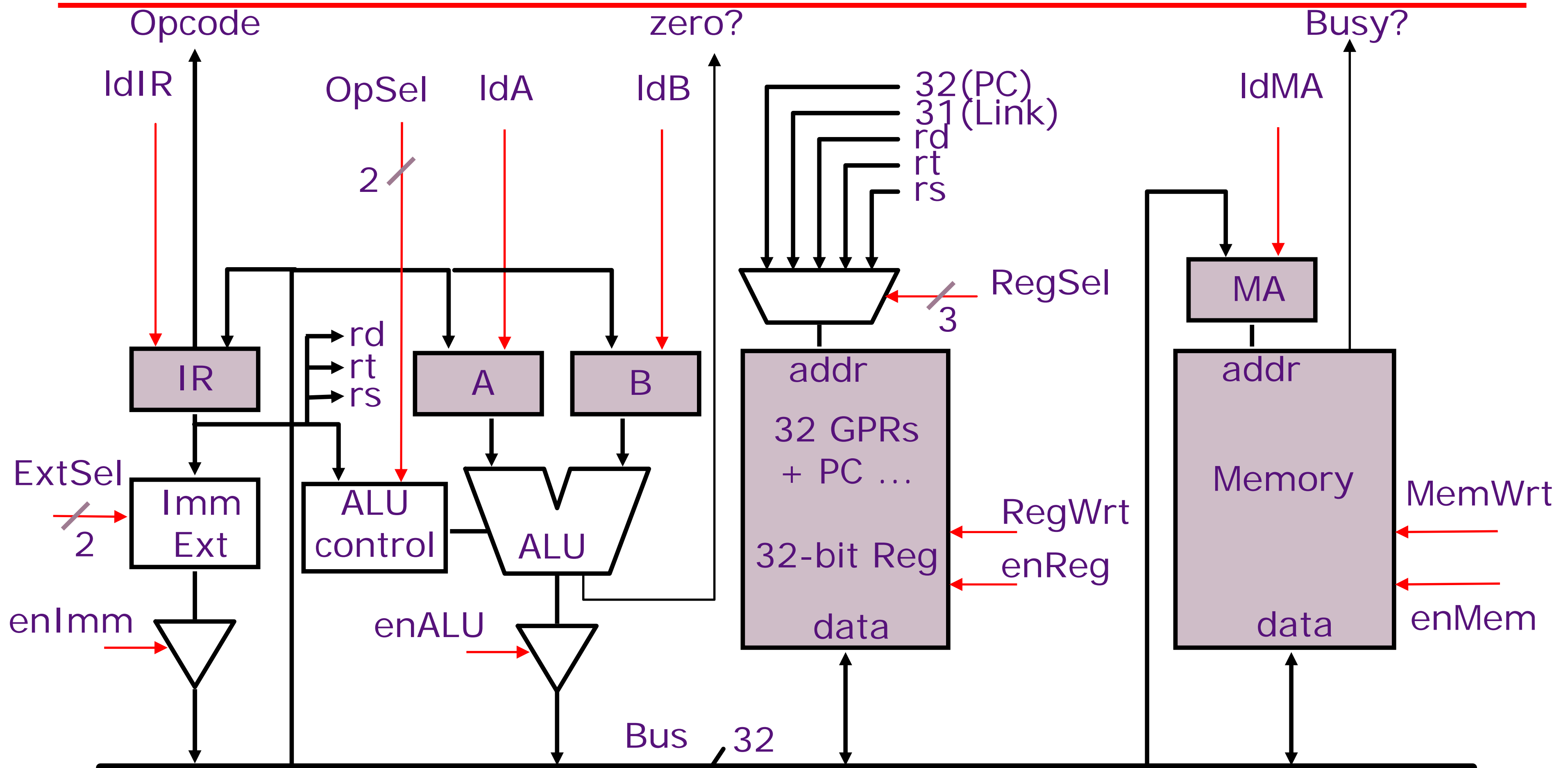
CSAIL

MIPS Instruction Formats



CSAIL

A Bus-based Datapath for MIPS

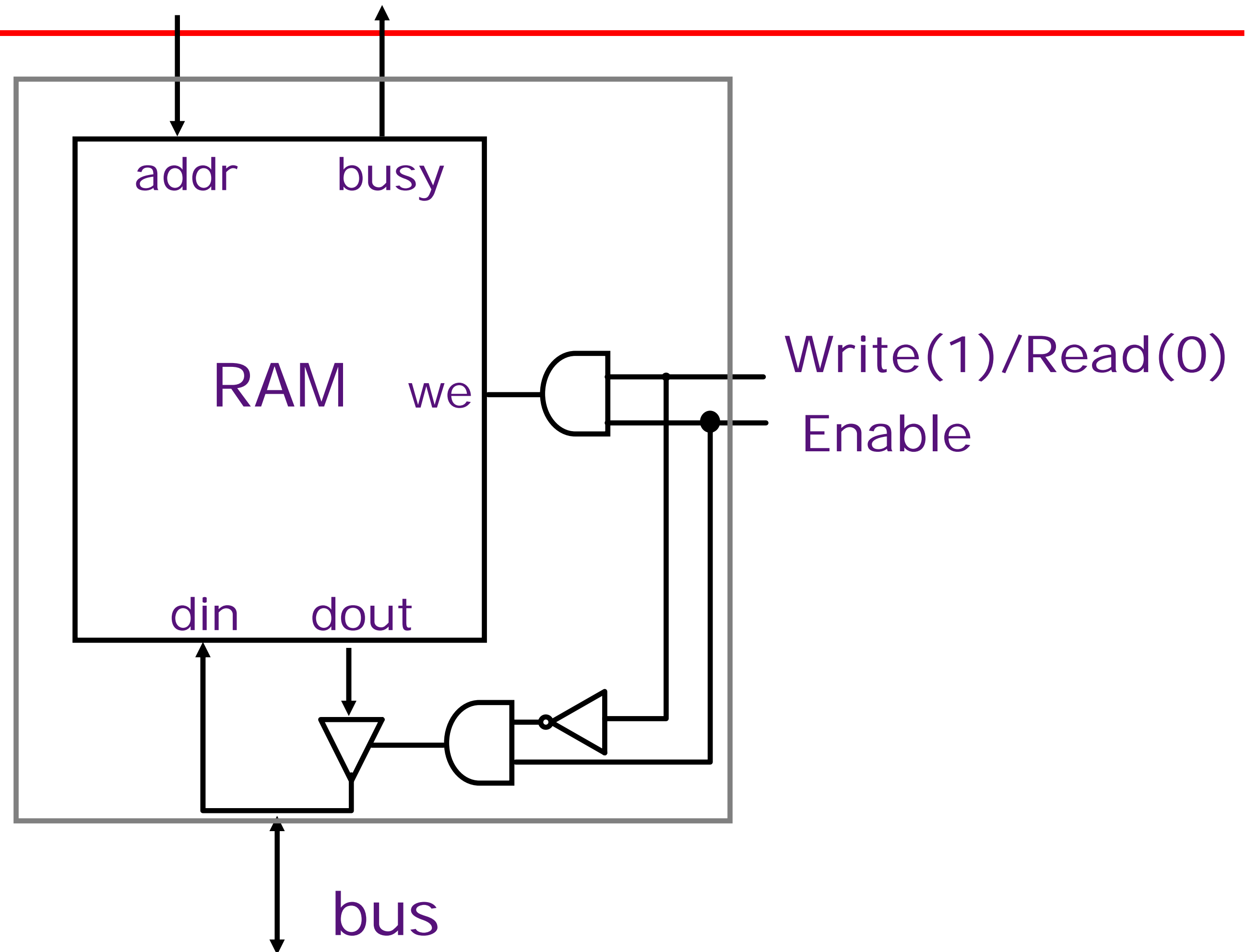


Microinstruction: register to register transfer (17 control signals)

MA ← PC *means* RegSel = PC; enReg=yes; ldMA= yes

B ← Reg[rt] *means* RegSel = rt; enReg=yes; ldB = yes

Memory Module



Assumption: Memory operates asynchronously and is slow as compared to Reg-to-Reg transfers

CSAIL

Instruction Execution

Execution of a MIPS instruction involves

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back to register file (optional)

+ the computation of the *next instruction* address

CSAIL

Microprogram Fragments

instr fetch:
MA ← PC
A ← PC
IR ← Memory
PC ← A + 4
dispatch on OPcode

can be treated as a macro

ALU:
A ← Reg[rs]
B ← Reg[rt]
Reg[rd] ← func(A,B)
do instruction fetch

ALUi:
A ← Reg[rs]
B ← Imm *sign extension ...*
Reg[rt] ← Opcode(A,B)
do instruction fetch

CSAIL

Microprogram Fragments *(cont.)*

LW: $A \leftarrow Reg[rs]$
$B \leftarrow Imm$
$MA \leftarrow A + B$
$Reg[rt] \leftarrow Memory$
do instruction fetch

J: $A \leftarrow PC$
$B \leftarrow IR$
$PC \leftarrow JumpTarg(A,B)$
do instruction fetch

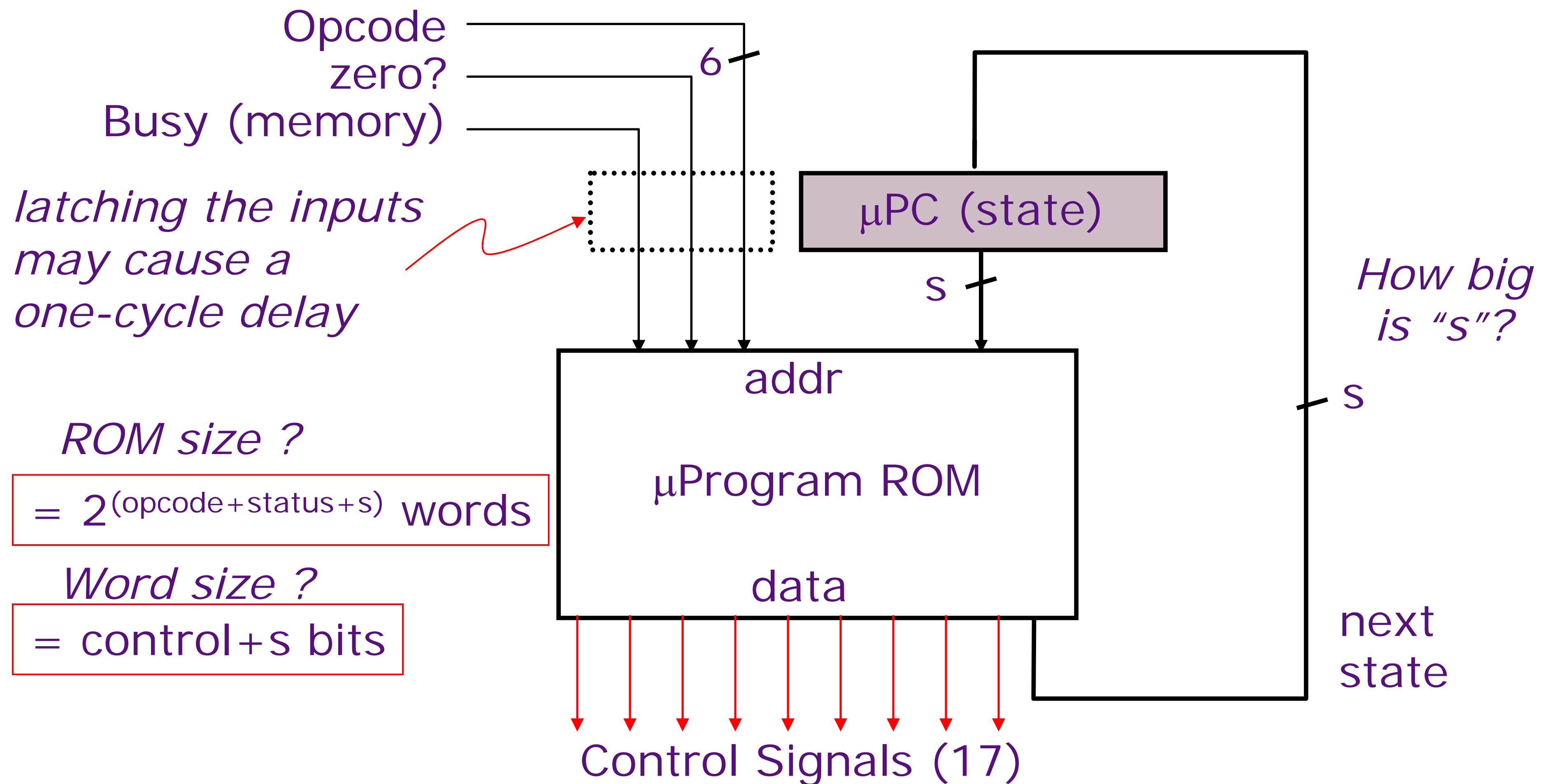
JumpTarg(A,B) = {A[31:28],B[25:0],00}

beqz: $A \leftarrow Reg[rs]$
If zero?(A) *then go to* bz-taken
do instruction fetch

bz-taken: $A \leftarrow PC$
$B \leftarrow Imm << 2$
$PC \leftarrow A + B$
do instruction fetch

CSAIL

MIPS Microcontroller: *first attempt*



Microprogram in the ROM *worksheet*

State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	MA ← PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	*	*	*	A ← PC	$fetch_3$
~~$fetch_3$~~	~~*~~	~~*~~	~~*~~	~~PC ← A + 4~~	~~?~~
$fetch_3$	ALU	*	*	PC ← A + 4	ALU_0
ALU_0	*	*	*	A ← Reg[rs]	ALU_1
ALU_1	*	*	*	B ← Reg[rt]	ALU_2
ALU_2	*	*	*	Reg[rd] ← func(A,B)	$fetch_0$

CSAIL

Microprogram in the ROM

State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	MA ← PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	*	*	*	A ← PC	$fetch_3$
$fetch_3$	ALU	*	*	PC ← A + 4	ALU_0
$fetch_3$	ALUi	*	*	PC ← A + 4	$ALUi_0$
$fetch_3$	LW	*	*	PC ← A + 4	LW_0
$fetch_3$	SW	*	*	PC ← A + 4	SW_0
$fetch_3$	J	*	*	PC ← A + 4	J_0
$fetch_3$	JAL	*	*	PC ← A + 4	JAL_0
$fetch_3$	JR	*	*	PC ← A + 4	JR_0
$fetch_3$	JALR	*	*	PC ← A + 4	$JALR_0$
$fetch_3$	beqz	*	*	PC ← A + 4	$beqz_0$
...					
ALU_0	*	*	*	A ← Reg[rs]	ALU_1
ALU_1	*	*	*	B ← Reg[rt]	ALU_2
ALU_2	*	*	*	Reg[rd] ← func(A,B)	$fetch_0$

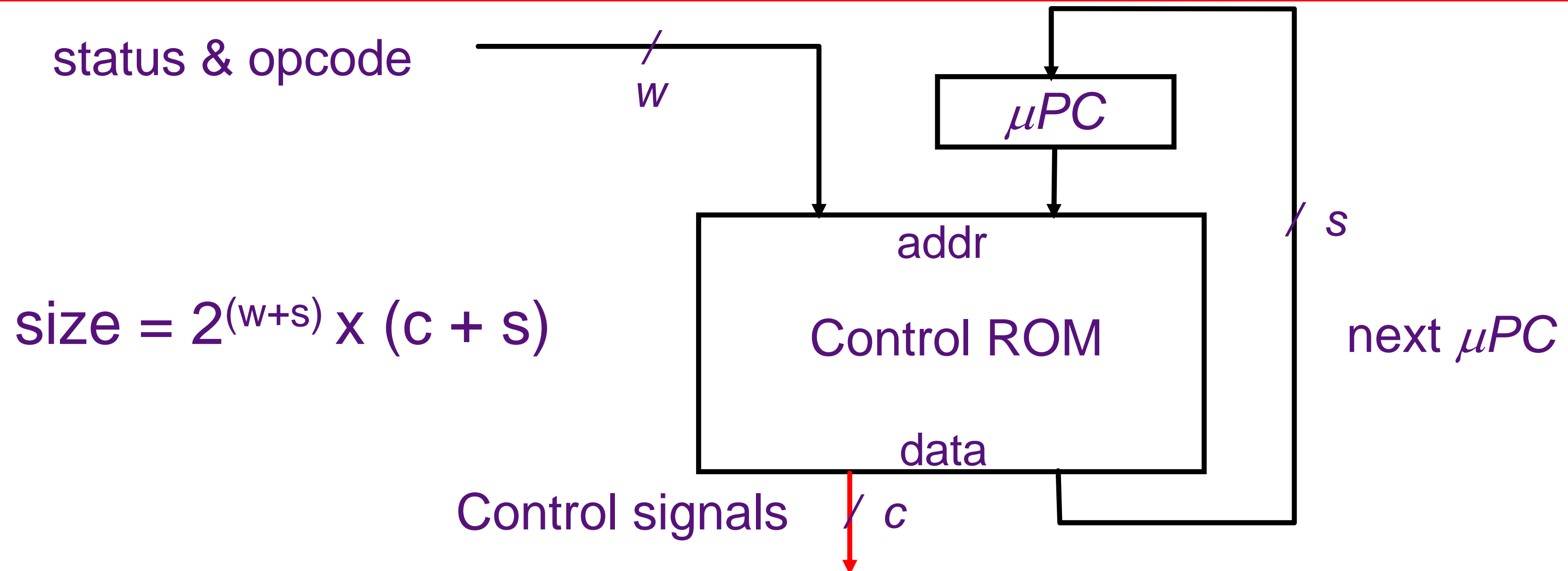
CSAIL

Microprogram in the ROM *Cont.*

State	Op	zero?	busy	Control points	next-state
$ALUi_0$	*	*	*	$A \leftarrow Reg[rs]$	$ALUi_1$
$ALUi_1$	sExt	*	*	$B \leftarrow sExt_{16}(Imm)$	$ALUi_2$
$ALUi_1$	uExt	*	*	$B \leftarrow uExt_{16}(Imm)$	$ALUi_2$
$ALUi_2$	*	*	*	$Reg[rd] \leftarrow Op(A,B)$	$fetch_0$
...					
J_0	*	*	*	$A \leftarrow PC$	J_1
J_1	*	*	*	$B \leftarrow IR$	J_2
J_2	*	*	*	$PC \leftarrow JumpTarg(A,B)$	$fetch_0$
...					
$beqz_0$	*	*	*	$A \leftarrow Reg[rs]$	$beqz_1$
$beqz_1$	*	yes	*	$A \leftarrow PC$	$beqz_2$
$beqz_1$	*	no	*	....	$fetch_0$
$beqz_2$	*	*	*	$B \leftarrow sExt_{16}(Imm)$	$beqz_3$
$beqz_3$	*	*	*	$PC \leftarrow A+B$	$fetch_0$
...					

JumpTarg(A,B) = {A[31:28],B[25:0],00}

Size of Control Store



size = $2^{(w+s)}$ x (c + s)

MIPS: w = 6+2 c = 17 s = ?

no. of steps per opcode = 4 to 6 + fetch-sequence

no. of states ≈ (4 steps per op-group) x op-groups + common sequences

= 4 x 8 + 10 states = 42 states ⇒ s = 6

Control ROM = $2^{(8+6)}$ x 23 bits ≈ 48 Kbytes

CSAIL

Reducing Control Store Size

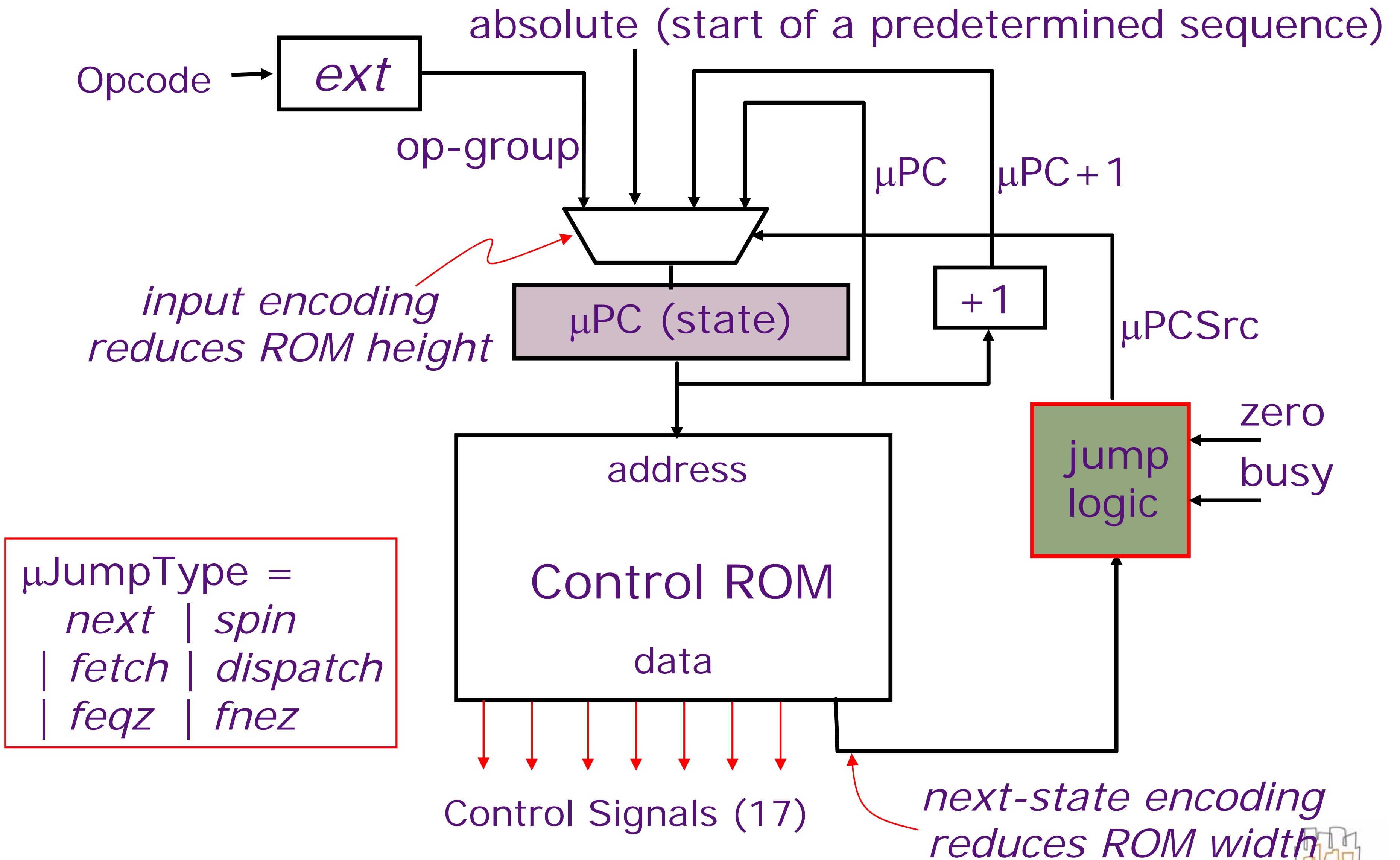
Control store has to be *fast* ⇒ *expensive*

- Reduce the ROM height (= address bits)
  - *reduce inputs by extra external logic*
    each input bit doubles the size of the control store
  - *reduce states by grouping opcodes*
    find common sequences of actions
  - *condense input status bits*
    combine all exceptions into one, i.e., exception/no-exception

- Reduce the ROM width
  - *restrict the next-state encoding*
    Next, Dispatch on opcode, Wait for memory, ...
  - *encode control signals (vertical microcode)*

CSAIL

MIPS Controller V2



Jump Logic

μPCSrc = *Case* μJumpTypes

next ⇒ μPC+1

spin ⇒ if (busy) then μPC else μPC+1

fetch ⇒ absolute

dispatch ⇒ op-group

feqz ⇒ if (zero) then absolute else μPC+1

fnez ⇒ if (zero) then μPC+1 else absolute

CSAIL

Instruction Fetch & ALU: *MIPS-Controller-2*

State	Control points	next-state
$fetch_0$	MA ← PC	next
$fetch_1$	IR ← Memory	spin
$fetch_2$	A ← PC	next
$fetch_3$	PC ← A + 4	dispatch
...		
ALU_0	A ← Reg[rs]	next
ALU_1	B ← Reg[rt]	next
ALU_2	Reg[rd]←func(A,B)	fetch
$ALUi_0$	A ← Reg[rs]	next
$ALUi_1$	B ← $sExt_{16}$(Imm)	next
$ALUi_2$	Reg[rd]← Op(A,B)	fetch

CSAIL

Load & Store: *MIPS-Controller-2*

State	Control points	next-state
LW_0	$A \leftarrow Reg[rs]$	next
LW_1	$B \leftarrow sExt_{16}(Imm)$	next
LW_2	$MA \leftarrow A+B$	next
LW_3	$Reg[rt] \leftarrow Memory$	spin
LW_4		fetch
SW_0	$A \leftarrow Reg[rs]$	next
SW_1	$B \leftarrow sExt_{16}(Imm)$	next
SW_2	$MA \leftarrow A+B$	next
SW_3	$Memory \leftarrow Reg[rt]$	spin
SW_4		fetch

Branches: *MIPS-Controller-2*

State	Control points	next-state
$BEQZ_0$	$A \leftarrow Reg[rs]$	next
$BEQZ_1$		fnez
$BEQZ_2$	$A \leftarrow PC$	next
$BEQZ_3$	$B \leftarrow sExt_{16}(Imm<<2)$	next
$BEQZ_4$	$PC \leftarrow A+B$	fetch
$BNEZ_0$	$A \leftarrow Reg[rs]$	next
$BNEZ_1$		feqz
$BNEZ_2$	$A \leftarrow PC$	next
$BNEZ_3$	$B \leftarrow sExt_{16}(Imm<<2)$	next
$BNEZ_4$	$PC \leftarrow A+B$	fetch

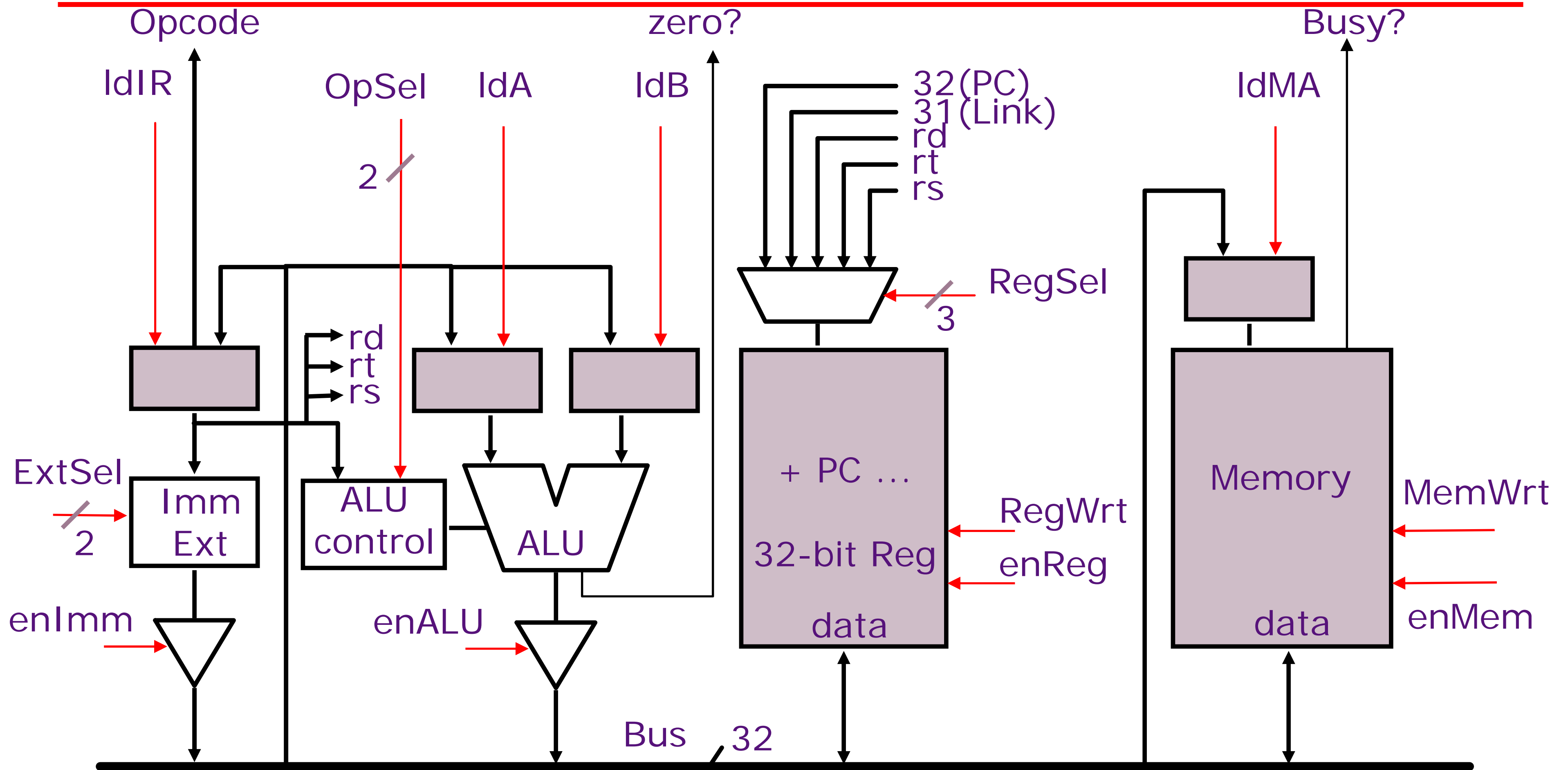
Jumps: *MIPS-Controller-2*

State	Control points	next-state
J_0	A ← PC	next
J_1	B ← IR	next
J_2	PC ← JumpTarg(A,B)	fetch
JR_0	A ← Reg[rs]	next
JR_1	PC ← A	fetch
JAL_0	A ← PC	next
JAL_1	Reg[31] ← A	next
JAL_2	B ← IR	next
JAL_3	PC ← JumpTarg(A,B)	fetch
$JALR_0$	A ← PC	next
$JALR_1$	B ← Reg[rs]	next
$JALR_2$	Reg[31] ← A	next
$JALR_3$	PC ← B	fetch

CSAIL

Five-minute break to stretch your legs

Implementing Complex Instructions



rd ← M[(rs)] op (rt) *Reg-Memory-src ALU op*

M[(rd)] ← (rs) op (rt) *Reg-Memory-dst ALU op*

M[(rd)] ← M[(rs)] op M[(rt)] *Mem-Mem ALU op*

Mem-Mem ALU Instructions:
MIPS-Controller-2

Mem-Mem ALU op M[(rd)] ← M[(rs)] op M[(rt)]

$ALUMM_0$	MA ← Reg[rs]	next
$ALUMM_1$	A ← Memory	spin
$ALUMM_2$	MA ← Reg[rt]	next
$ALUMM_3$	B ← Memory	spin
$ALUMM_4$	MA ←Reg[rd]	next
$ALUMM_5$	Memory ← func(A,B)	spin
$ALUMM_6$		fetch

Complex instructions usually do not require datapath modifications in a microprogrammed implementation
-- only extra space for the control program

Implementing these instructions using a hardwired controller is difficult without datapath modifications

CSAIL

Performance Issues

Microprogrammed control

$\Rightarrow$ multiple cycles per instruction

Cycle time ?

$t_C > \max(t_{reg\text{-}reg}, t_{ALU}, t_{\mu ROM}, t_{RAM})$

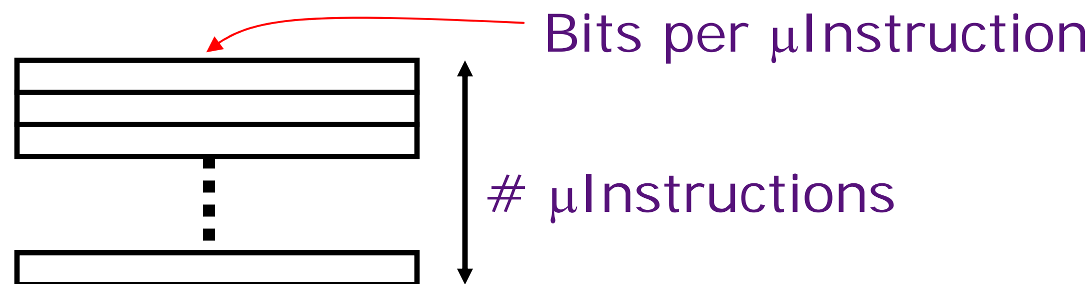
Given complex control, t_{ALU} & t_{RAM} can be broken into multiple cycles. However, $t_{\mu ROM}$ cannot be broken down. Hence

$t_C > \max(t_{reg\text{-}reg}, t_{\mu ROM})$

Suppose $10 * t_{\mu ROM} < t_{RAM}$

Good performance, relative to the single-cycle hardwired implementation, can be achieved even with a CPI of 10

Horizontal vs Vertical µCode

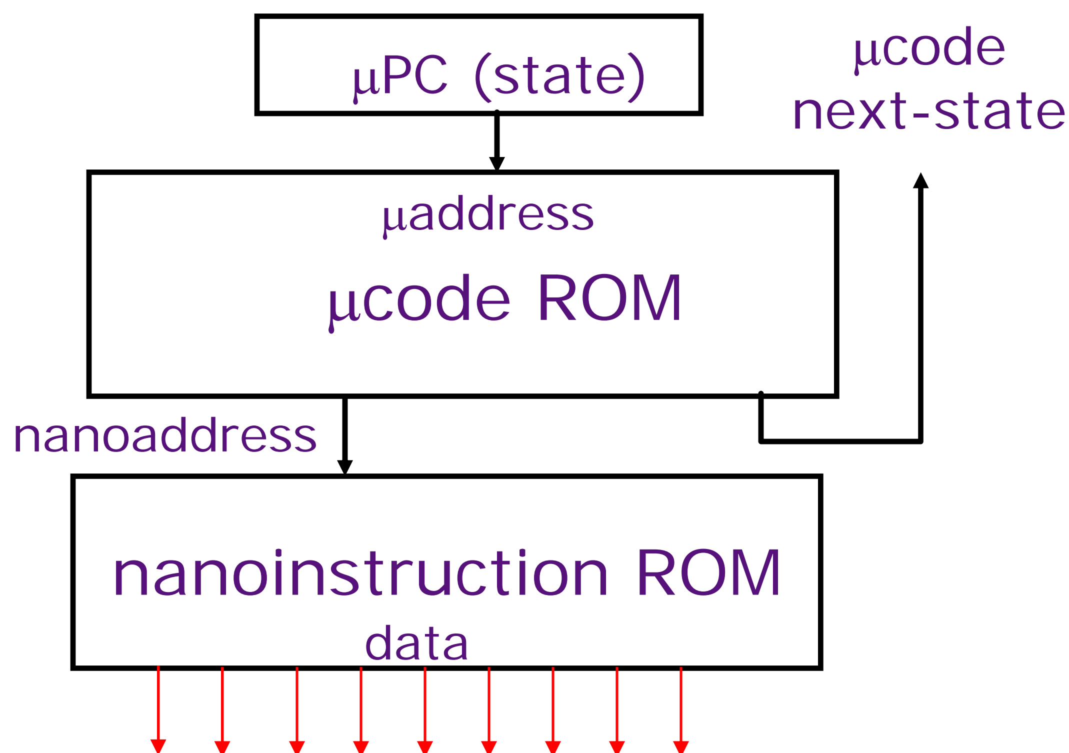


- Horizontal µcode has wider µinstructions
  - Multiple parallel operations per µinstruction
  - Fewer steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical µcode has narrower µinstructions
  - Typically a single datapath operation per µinstruction
    - separate µinstruction for branches
  - More steps to per macroinstruction
  - More compact ⇒ less bits
- Nanocoding
  - Tries to combine best of horizontal and vertical µcode

Nanocoding

Exploits recurring control signal patterns in µcode, e.g.,

ALU_0 A ← Reg[rs]
...
$ALUi_0$ A ← Reg[rs]
...



- MC68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  – Nanoinstructions were 68 bits wide, decoded to give 196 control signals

Some more history …

- IBM 360
- Microcoding through the seventies
- Microcoding now

CSAIL

Microprogramming in IBM 360

	M30	M40	M50	M65
Datapath width (bits)	8	16	32	64
μinst width (bits)	50	52	85	87
μcode size (K minsts)	4	4	2.75	2.75
μstore technology	CCROS	TCROS	BCROS	BCROS
μstore cycle (ns)	750	625	500	200
memory cycle (ns)	1500	2500	2000	750
Rental fee ($K/month)	4	7	15	35

Only the fastest models (75 and 95) were hardwired

Microcode Emulation

- IBM initially miscalculated the importance of software compatibility with earlier models when introducing the 360 series
- Honeywell stole some IBM 1401 customers by offering translation software ("Liberator") for Honeywell H200 series machine
- IBM retaliated with optional additional microcode for 360 series that could emulate IBM 1401 ISA, later extended for IBM 7000 series
  - one popular program on 1401 was a 650 simulator, so some customers ran many 650 programs on emulated 1401s
    - *(650 simulated on 1401 emulated on 360)*

CSAIL

Microprogramming thrived in the Seventies

- Significantly faster ROMs than DRAMs were available
- For complex instruction sets, datapath and controller were *cheaper and simpler*
- *New instructions* , e.g., floating point, could be supported without datapath modifications
- *Fixing bugs* in the controller was easier
- ISA compatibility across various models could be achieved easily and cheaply

Except for the cheapest and fastest machines, all computers were microprogrammed

CSAIL

Writable Control Store (WCS)

- Implement control store with SRAM not ROM
  - MOS SRAM memories now almost as fast as control store (core memories/DRAMs were 2-10x slower)
  - Bug-free microprograms difficult to write
- User-WCS provided as option on several minicomputers
  - Allowed users to change microcode for each process
- User-WCS *failed*
  - Little or no programming tools support
  - Difficult to fit software into small space
  - Microcode control tailored to original ISA, less useful for others
  - Large WCS part of processor state - expensive context switches
  - Protection difficult if user can change microcode
  - Virtual memory required *restartable* microcode

CSAIL

Microprogramming: *late seventies*

- With the advent of VLSI technology assumptions about ROM & RAM speed became invalid
- Micromachines became more complicated
  - Micromachines were pipelined to overcome slower ROM
  - Complex instruction sets led to the need for subroutine and call stacks in μcode
  - Need for fixing bugs in control programs was in conflict with read-only nature of μROM
    $\Rightarrow$ *WCS (B1700, QMachine, Intel432, ...)*
- Introduction of caches and buffers, especially for instructions, made multiple-cycle execution of reg-reg instructions unattractive

CSAIL

Modern Usage

- *Microprogramming is far from extinct*

- Played a crucial role in micros of the Eighties

  Motorola 68K series
  Intel 386 and 486

- Microcode pays an assisting role in most modern CISC micros *(AMD Athlon, Intel Pentium-4 ...)*
  - Most instructions are executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke the microcode engine

- *Patchable* microcode common for post-fabrication bug fixes, e.g. Intel Pentiums load μcode patches at bootup

CSAIL

Thank you !

Single-Cycle Processors: Datapath & Control

Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.

Based on the material prepared by Arvind and Krste Asanovic

Instruction Set Architecture (ISA) versus Implementation

- ISA is the hardware/software interface
  - Defines set of programmer visible state
  - Defines instruction format (bit encoding) and instruction semantics
  - Examples: *MIPS, x86, IBM 360, JVM*
- Many possible implementations of one ISA
  - 360 implementations: model 30 (c. 1964), z900 (c. 2001)
  - x86 implementations: *8086 (c. 1978), 80186, 286, 386, 486, Pentium, Pentium Pro, Pentium-4 (c. 2000), AMD Athlon, Transmeta Crusoe, SoftPC*
  - MIPS implementations: *R2000, R4000, R10000, ...*
  - JVM: *HotSpot, PicoJava, ARM Jazelle, ...*

Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

– Instructions per program depends on source code, compiler technology, and ISA

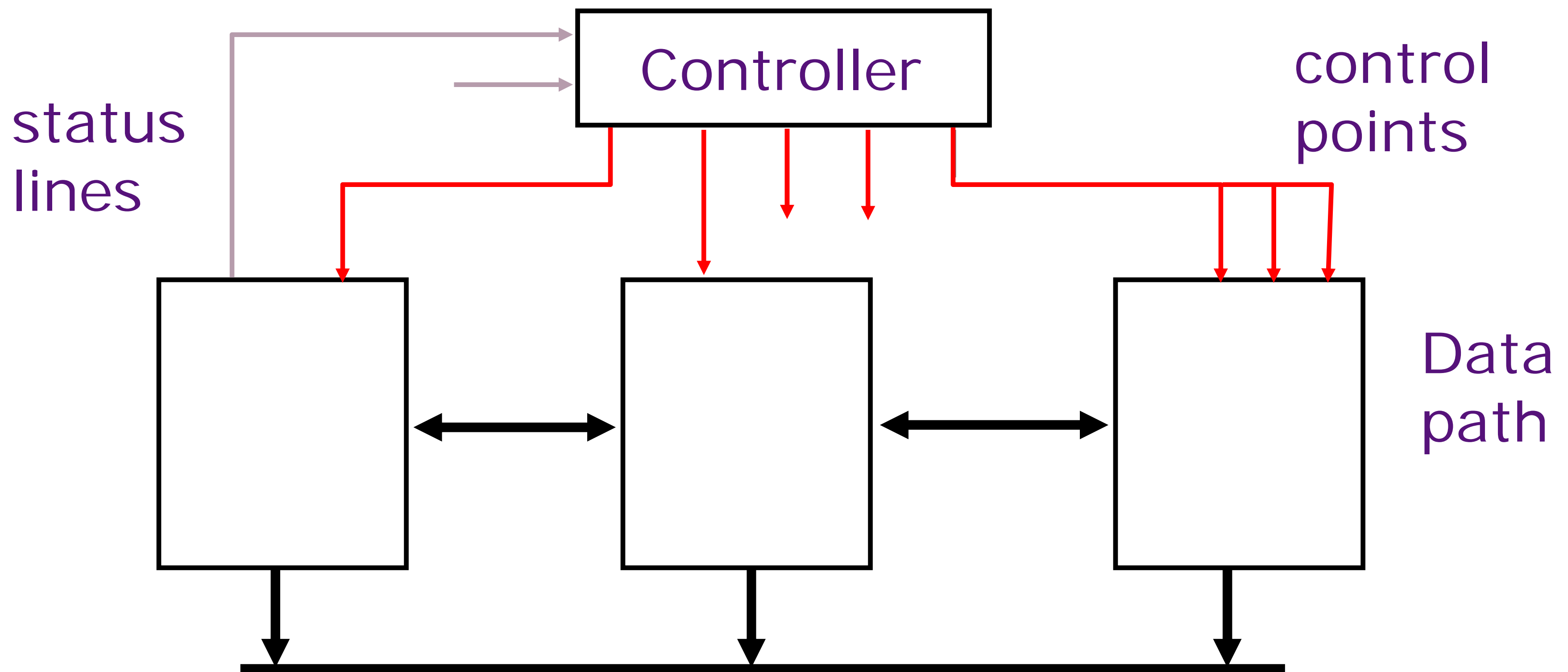
– Cycles per instructions (CPI) depends upon the ISA and the microarchitecture

– Time per cycle depends upon the microarchitecture and the base technology

this lecture

Microarchitecture	CPI	cycle time
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

Microarchitecture: *Implementation of an ISA*

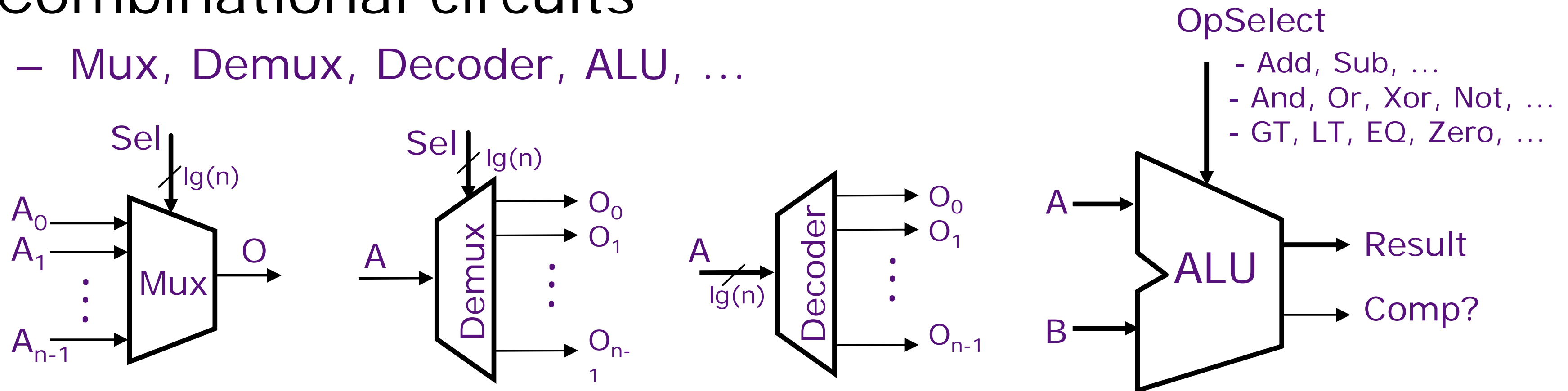


Structure: How components are connected.
Static

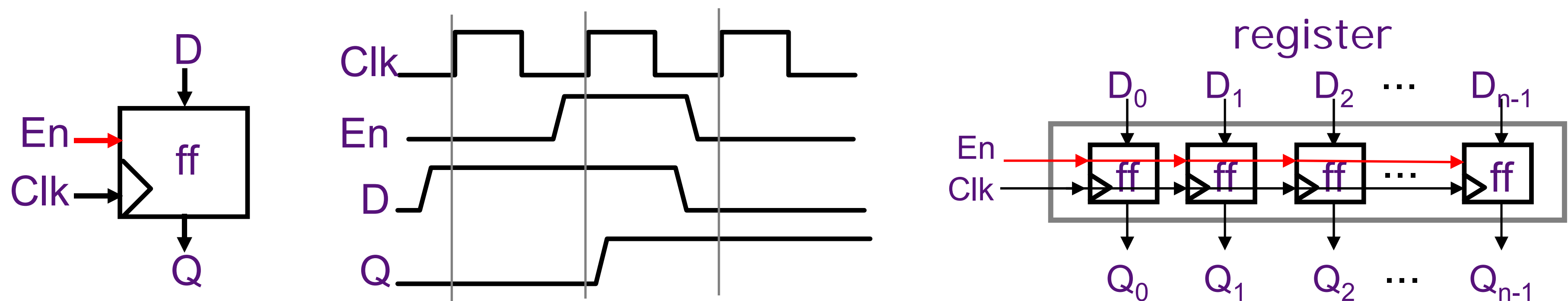
Behavior: How data moves between components
Dynamic

Hardware Elements

- Combinational circuits
  - Mux, Demux, Decoder, ALU, ...



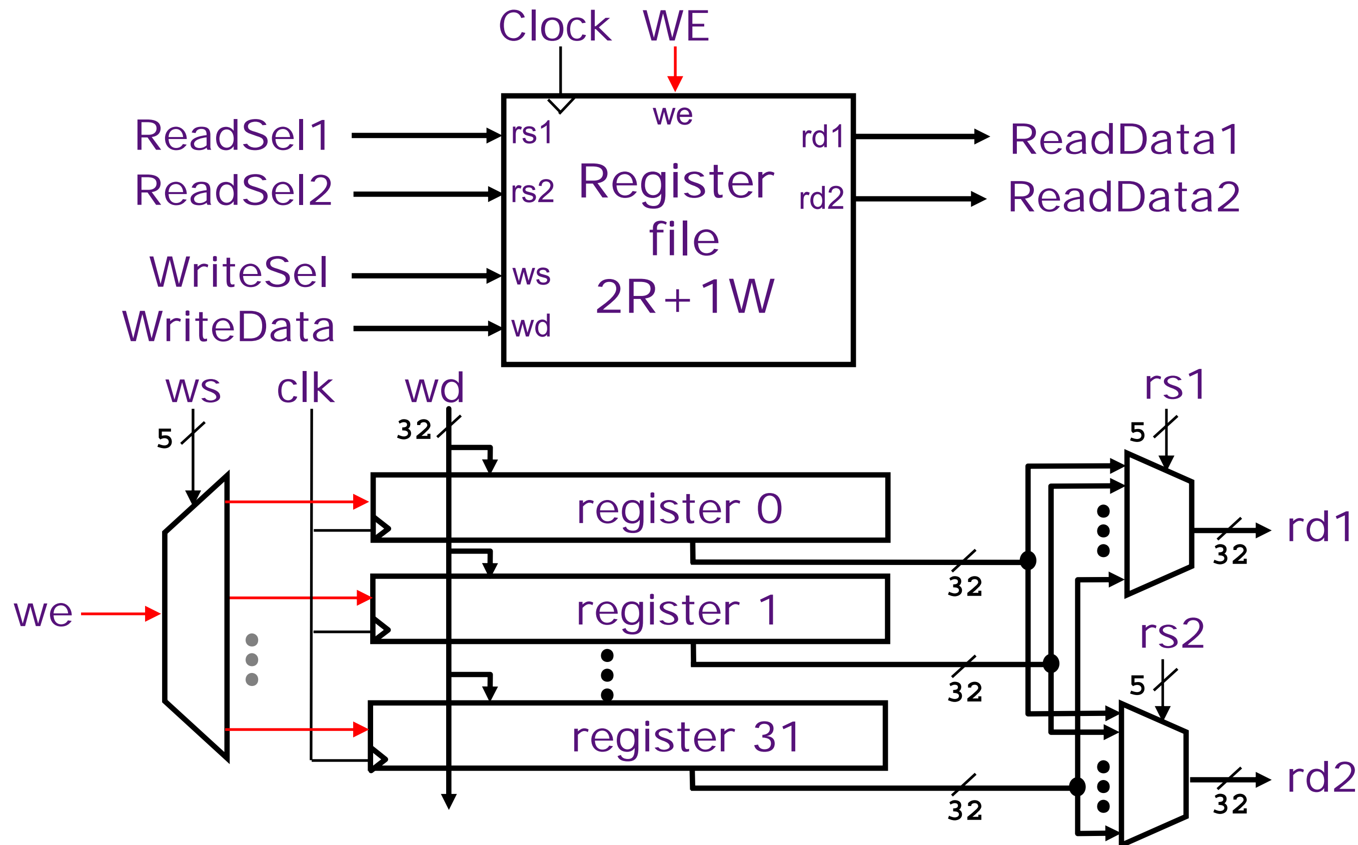
- Synchronous state elements
  - Flipflop, Register, Register file, SRAM, DRAM



Edge-triggered: Data is sampled at the rising edge

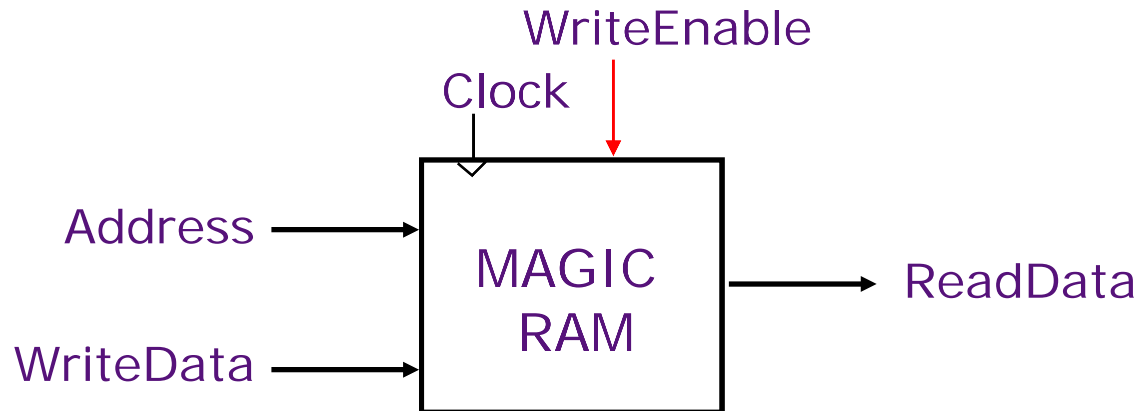
CSAIL

Register Files



- No timing issues in reading a selected register
- Register files with a large number of ports are difficult to design
  - *Intel's Itanium, GPR File has 128 registers with 8 read ports and 4 write ports!!!*

A Simple Memory Model



Reads and writes are always completed in one cycle

- a Read can be done any time (i.e. combinational)
- a Write is performed at the rising clock edge
  if it is enabled

  $\Rightarrow$ *the write address and data must be stable at the clock edge*

Later in the course we will present a more realistic model of memory

Implementing MIPS:

Single-cycle per instruction datapath & control logic

The MIPS ISA

Processor State

32 32-bit GPRs, R0 always contains a 0
32 single precision FPRs, may also be viewed as
16 double precision FPRs
FP status register, used for FP compares & exceptions
PC, the program counter
some other special registers

Data types

8-bit byte, 16-bit half word
32-bit word for integers
32-bit word for single precision floating point
64-bit word for double precision floating point

Load/Store style instruction set

data addressing modes- immediate & indexed
branch addressing modes- PC relative & register indirect
Byte addressable memory- big endian mode

All instructions are 32 bits

CSAIL

Instruction Execution

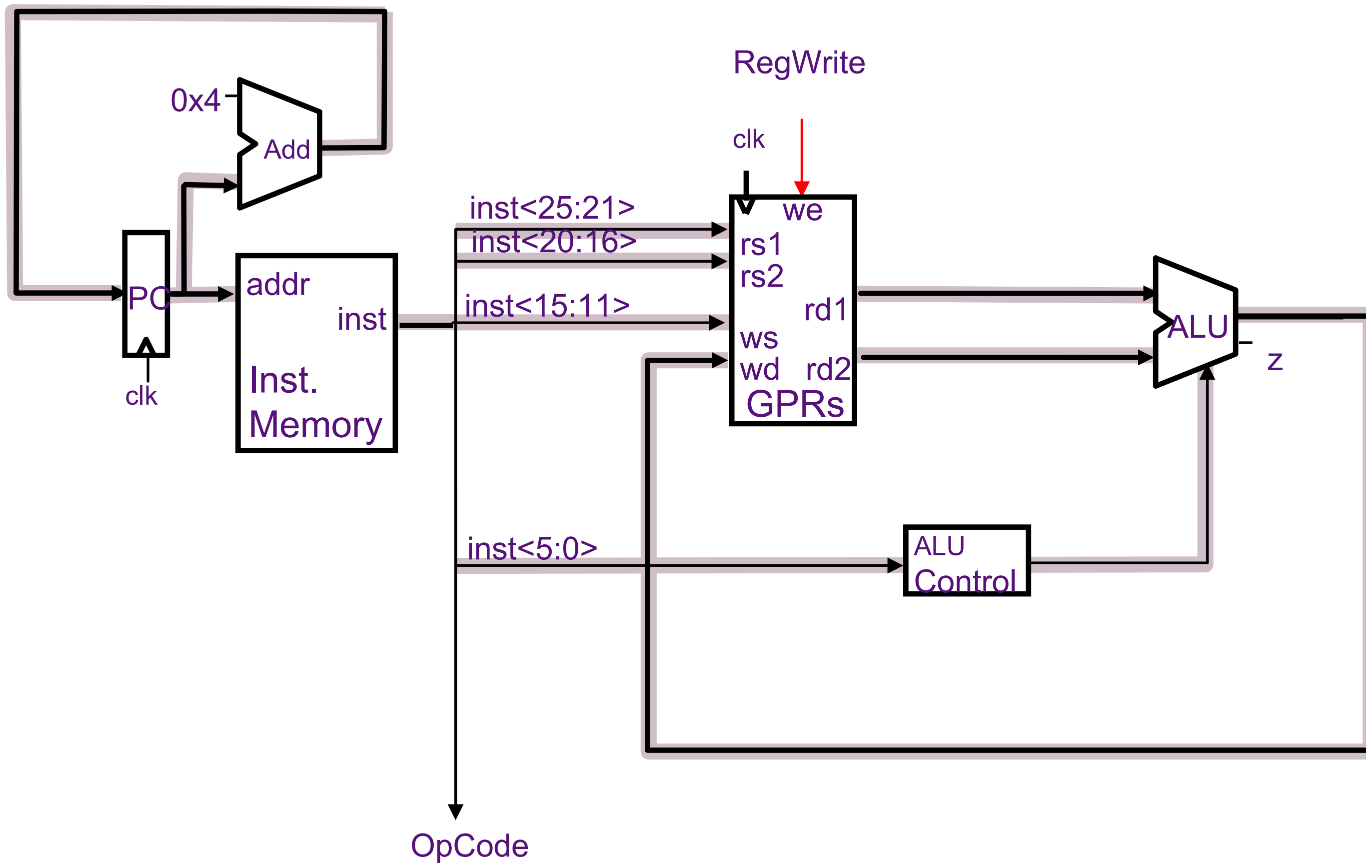
Execution of an instruction involves

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back

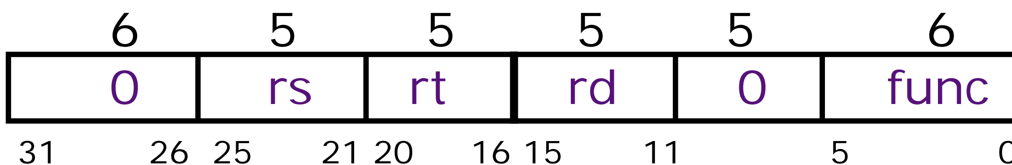
and the computation of the address of the *next instruction*

CSAIL

Datapath: Reg-Reg ALU Instructions

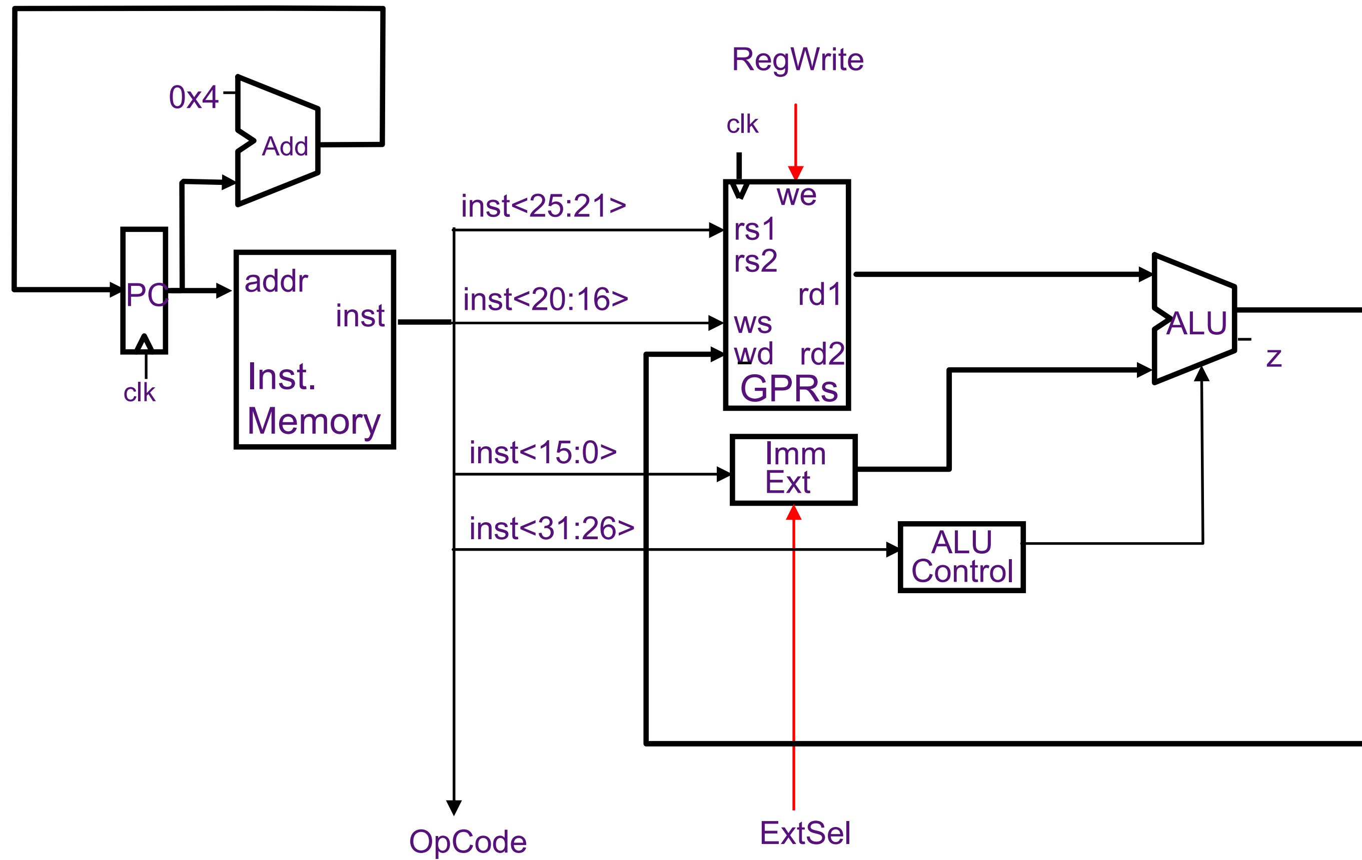


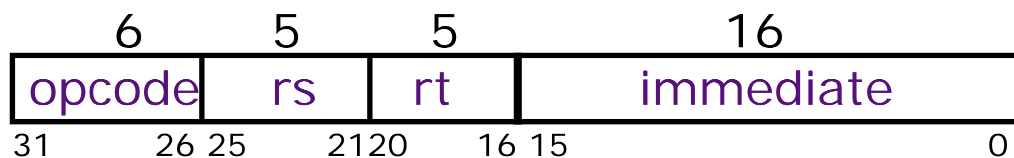
RegWrite Timing?



rd ← (rs) func (rt)

Datapath: Reg-Imm ALU Instructions





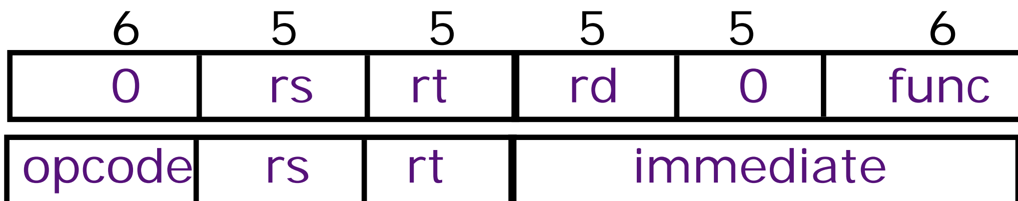
rt ← (rs) op immediate

CSAIL

Conflicts in Merging Datapath

Introduce muxes

0x4
Add
PC
clk
addr
inst
Inst.
Memory
RegWrite
clk
we
rs1
rs2
rd1
ws
wd
rd2
GPRs
ALU
z
inst<25:21>
inst<20:16>
inst<15:11>
inst<15:0>
Imm
Ext
inst<31:26>
inst<5:0>
ALU
Control
OpCode
ExtSel



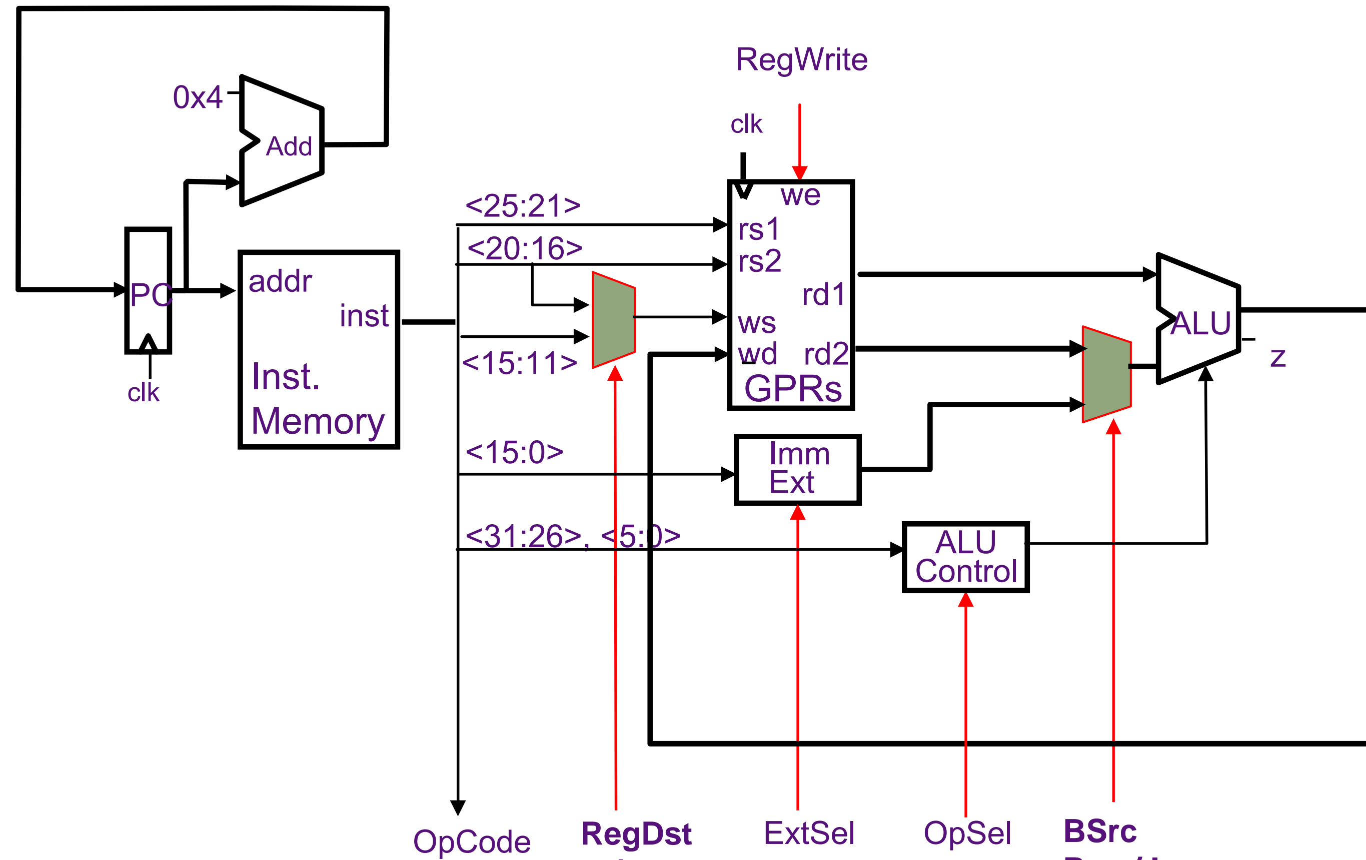
6	5	5	5	5	6
0	rs	rt	rd	0	func
opcode	rs	rt	immediate		

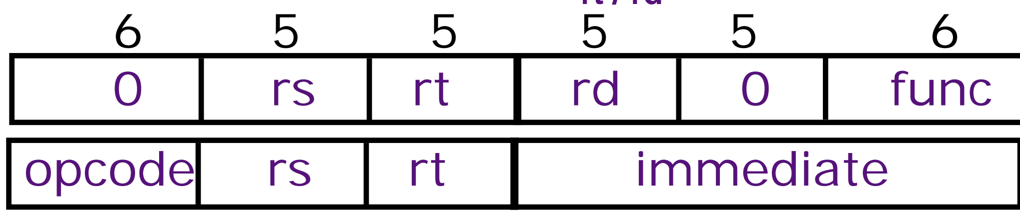
rd ← (rs) func (rt)

rt ← (rs) op immediate

CSAIL

Datapath for ALU Instructions





6	5	5	5	5	6
0	rs	rt	rd	0	func
opcode	rs	rt	immediate		

rd ← (rs) func (rt)

rt ← (rs) op immediate

CSAIL

Datapath for Memory Instructions

Should program and data memory be separate?

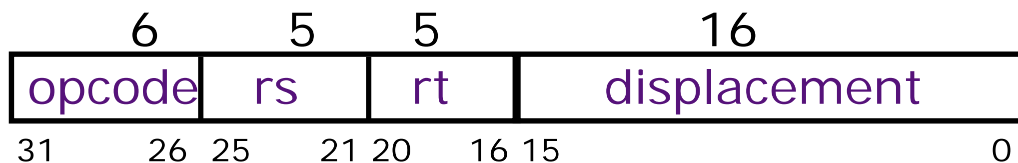
Harvard style: separate (Aiken and Mark 1 influence)
- read-only program memory
- read/write data memory
  at some level the two memories have to be the same

Princeton style: the same (von Neumann's influence)
- A Load or Store instruction requires accessing the memory more than once during its execution

CSAIL

Load/Store Instructions: *Harvard Datapath*

RegWrite
MemWrite
WBSrc
ALU / Mem
0x4
Add
clk
“base”
we
rs1
rs2
rd1
ws
wd
rd2
GPRs
PC
addr
inst
Inst.
Memory
disp
Imm
Ext
ALU
z
ALU
Control
we
addr
rdata
Data
Memory
wdata
OpCode
RegDst
ExtSel
OpSel
BSrc



addressing mode
(rs) + displacement

rs is the base register

rt is the destination of a Load or the source for a Store

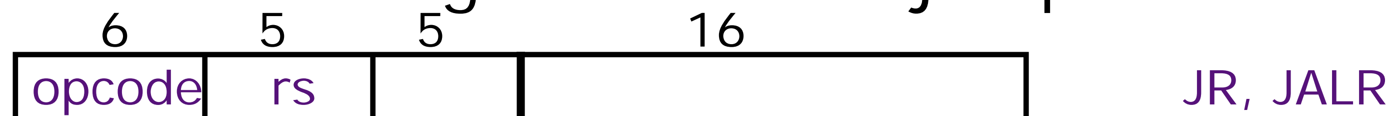
CSAIL

MIPS Control Instructions

Conditional (on GPR) PC-relative branch

BEQZ, BNEZ

Unconditional register-indirect jumps



JR, JALR

Unconditional absolute jumps

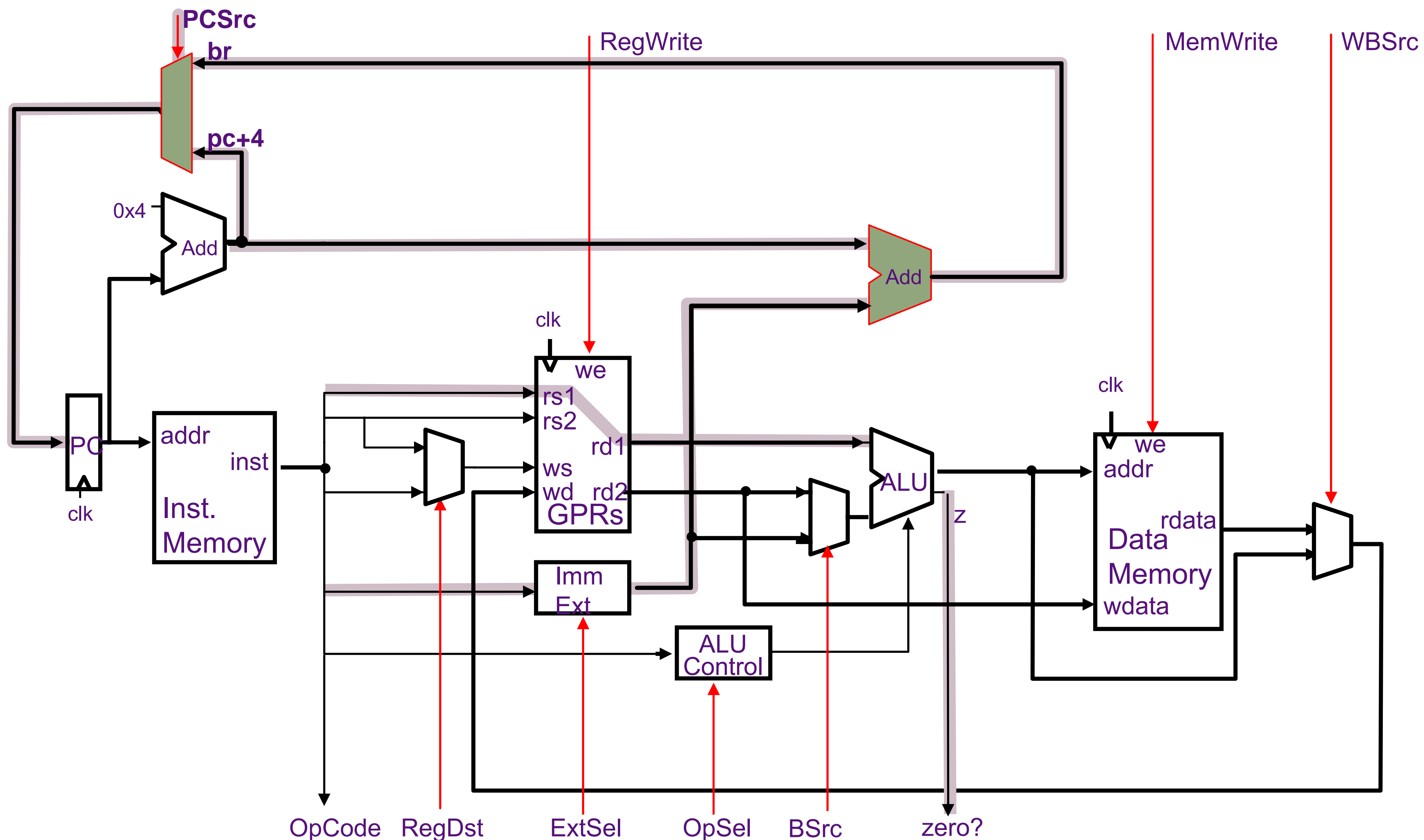
J, JAL

- PC-relative branches add offset×4 to PC+4 to calculate the target address (offset is in words): ±128 KB range
- Absolute jumps append target×4 to PC<31:28> to calculate the target address: 256 MB range
- jump-&-link stores PC+4 into the link register (R31)
- All Control Transfers are delayed by 1 instruction

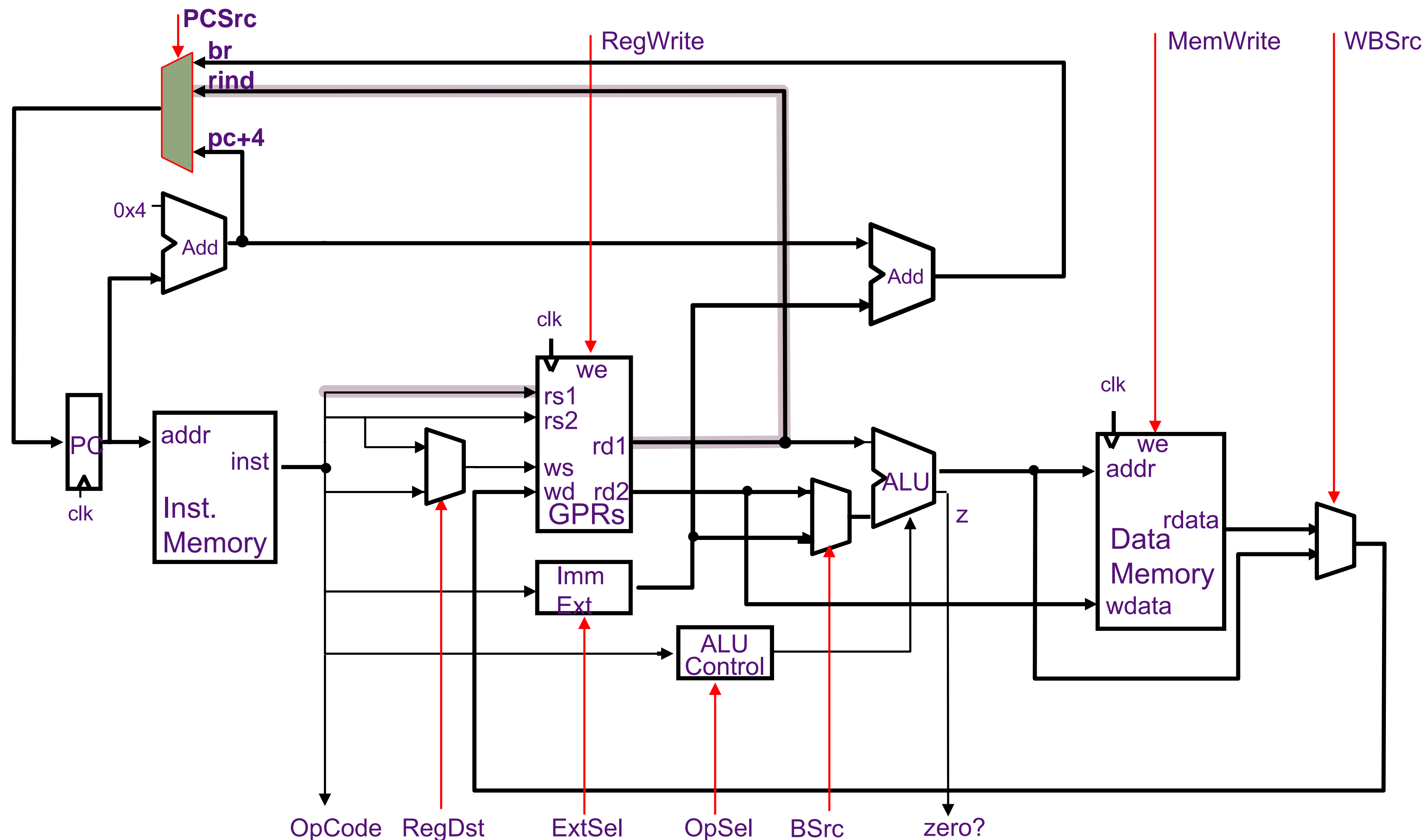
  we will worry about the branch delay slot later

CSAIL

Conditional Branches (BEQZ, BNEZ)

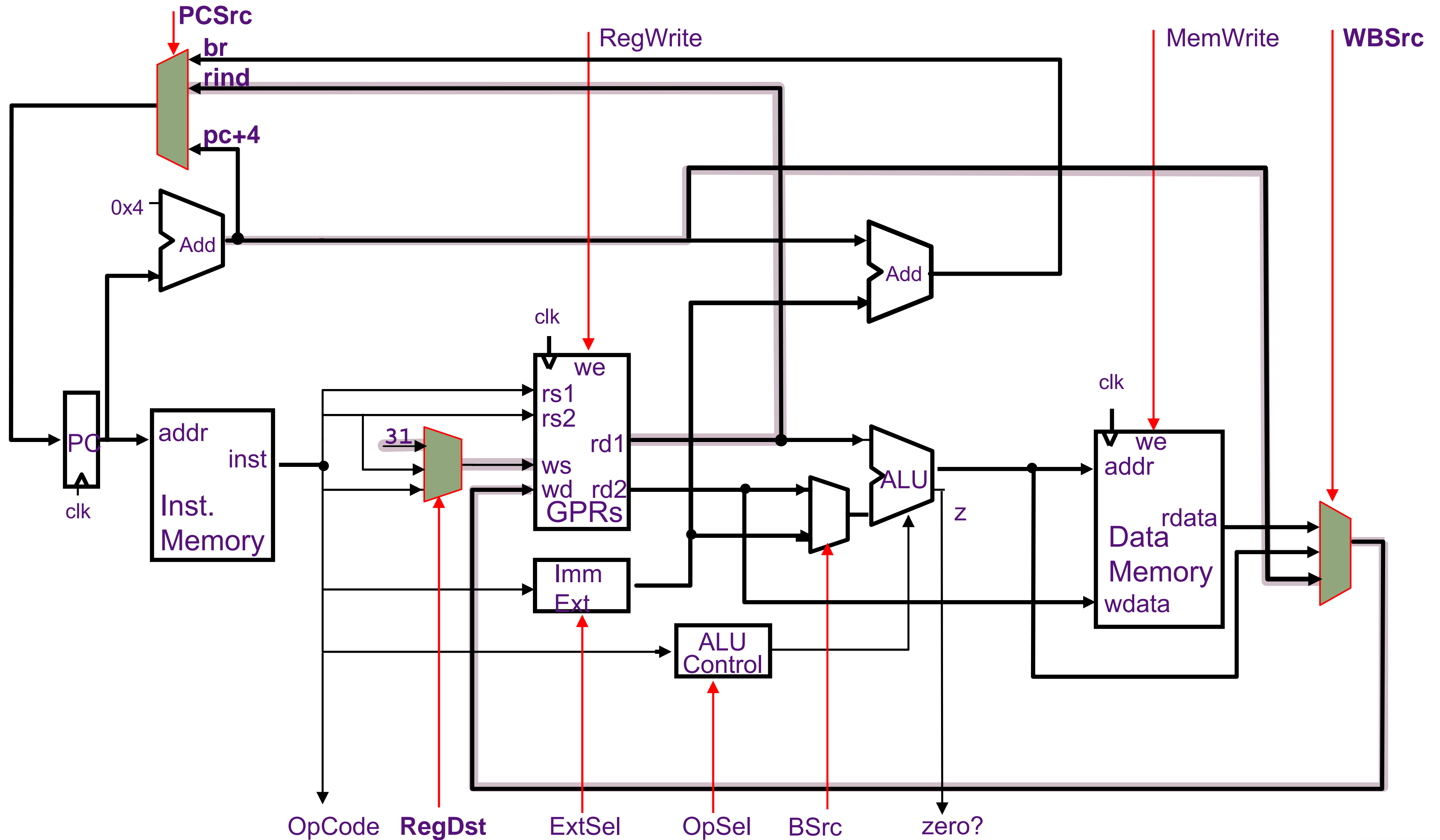


Register-Indirect Jumps (JR)

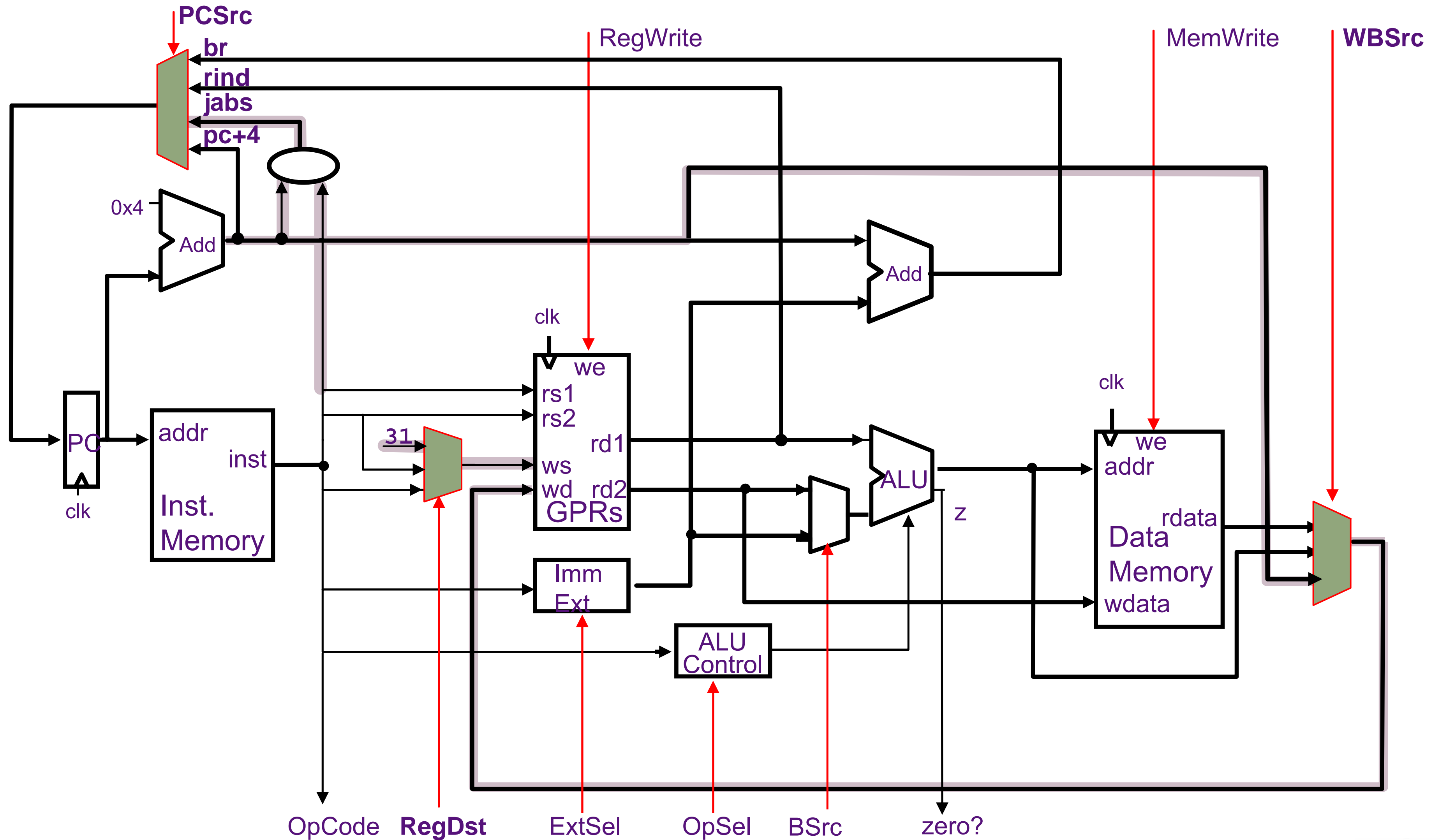


CSAIL

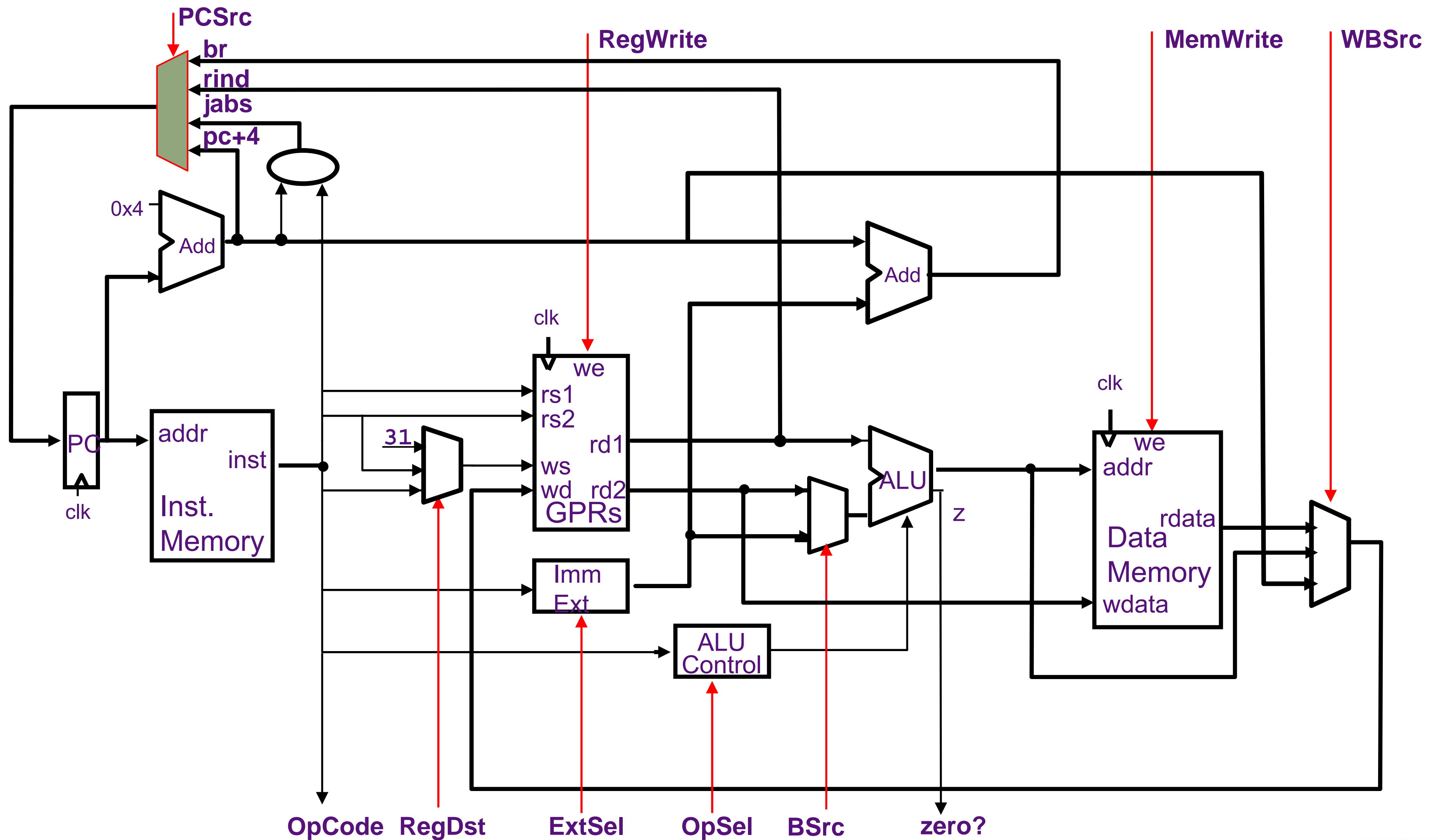
Register-Indirect Jump-&-Link (JALR)



Absolute Jumps (J, JAL)



Harvard-Style Datapath for MIPS



CSAIL

Five-minute break to stretch your legs

Single-Cycle Hardwired Control:
Harvard architecture

We will assume

- clock period is sufficiently long for all of the following steps to be "completed":

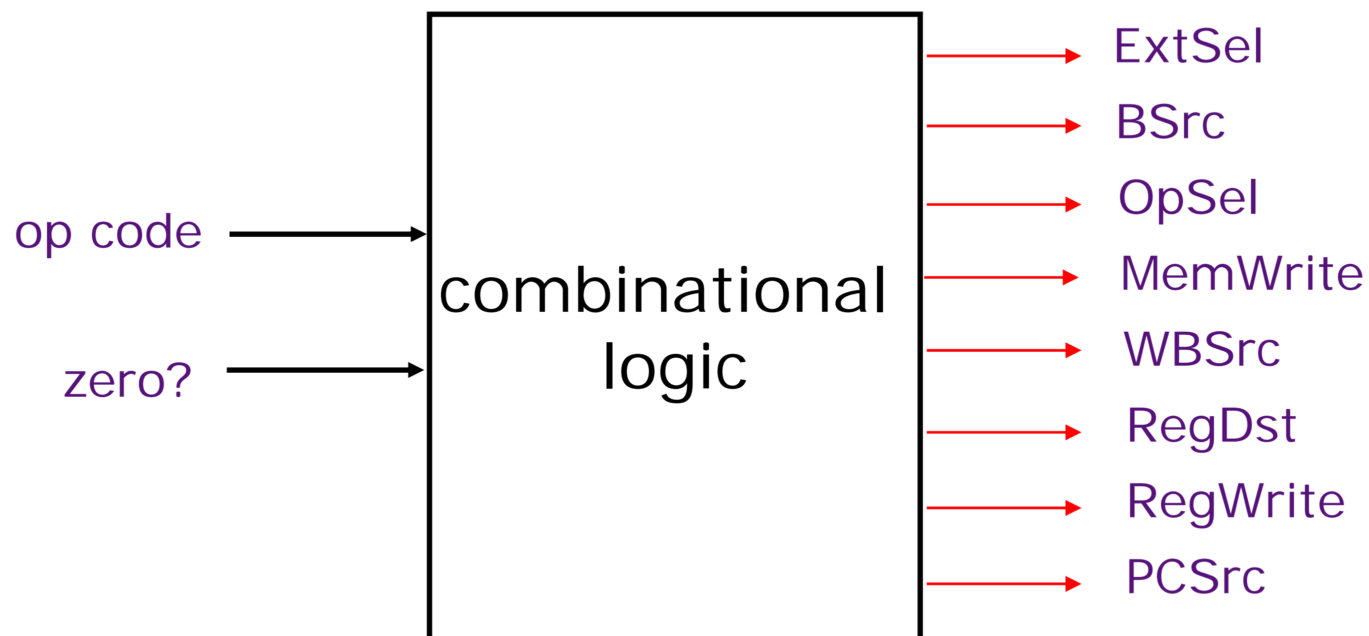
1. instruction fetch
2. decode and register fetch
3. ALU operation
4. data fetch if required
5. register write-back setup time

$$\Rightarrow \quad t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB}$$

- At the rising edge of the following clock, the PC, the register file and the memory are updated

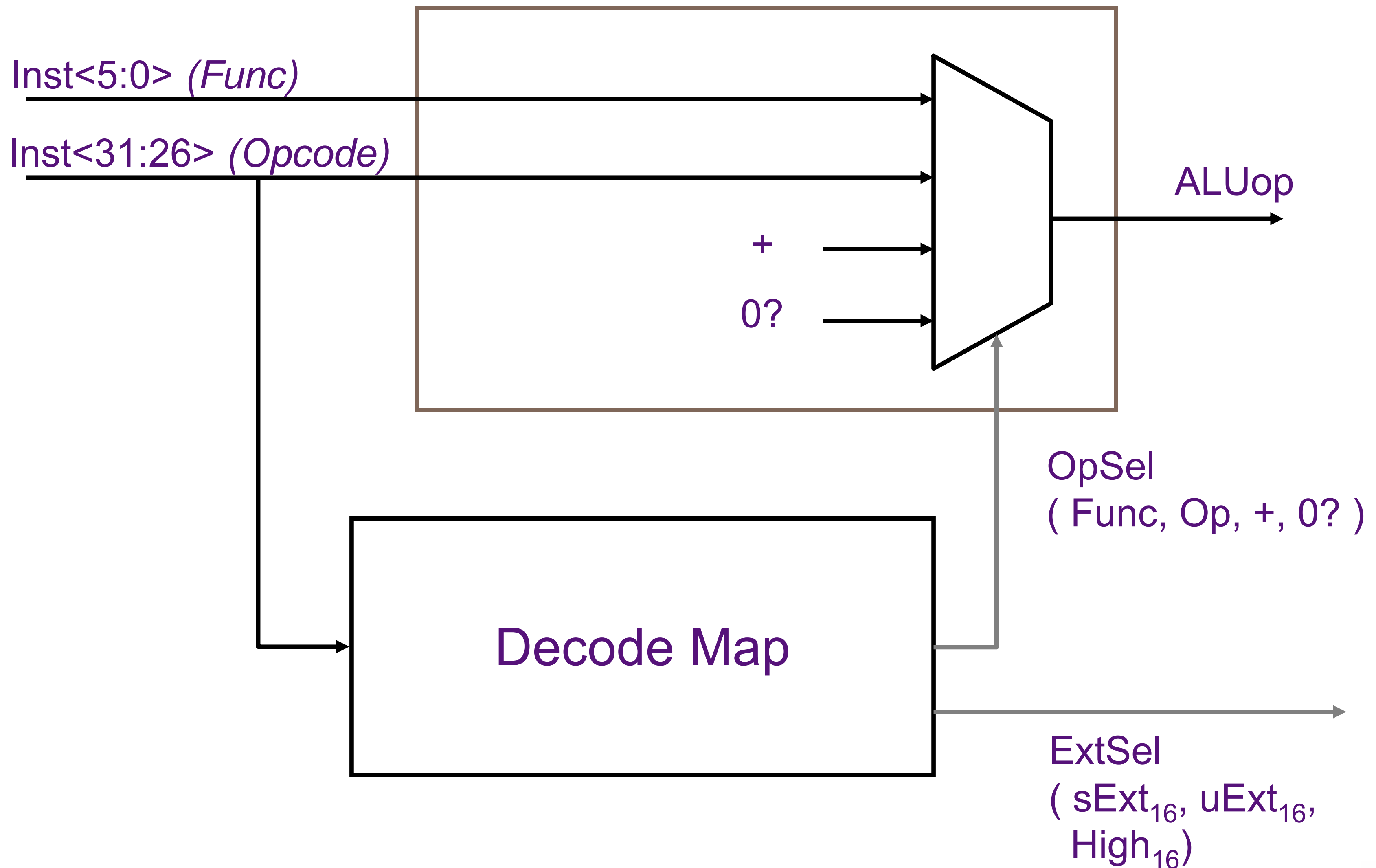
CSAIL

Hardwired Control is pure Combinational Logic



CSAIL

ALU Control & Immediate Extension



Hardwired Control Table

Opcode	ExtSel	BSrc	OpSel	MemW	RegW	WBSrc	RegDst	PCSrc
ALU	*	Reg	Func	no	yes	ALU	rd	pc+4
ALUi	$sExt_{16}$	Imm	Op	no	yes	ALU	rt	pc+4
ALUiu	$uExt_{16}$	Imm	Op	no	yes	ALU	rt	pc+4
LW	$sExt_{16}$	Imm	+	no	yes	Mem	rt	pc+4
SW	$sExt_{16}$	Imm	+	yes	no	*	*	pc+4
$BEQZ_{z=0}$	$sExt_{16}$	*	0?	no	no	*	*	br
$BEQZ_{z=1}$	$sExt_{16}$	*	0?	no	no	*	*	pc+4
J	*	*	*	no	no	*	*	jabs
JAL	*	*	*	no	yes	PC	R31	jabs
JR	*	*	*	no	no	*	*	rind
JALR	*	*	*	no	yes	PC	R31	rind

BSrc = Reg / Imm
RegDst = rt / rd / R31

WBSrc = ALU / Mem / PC
PCSrc = pc+4 / br / rind / jabs

CSAIL

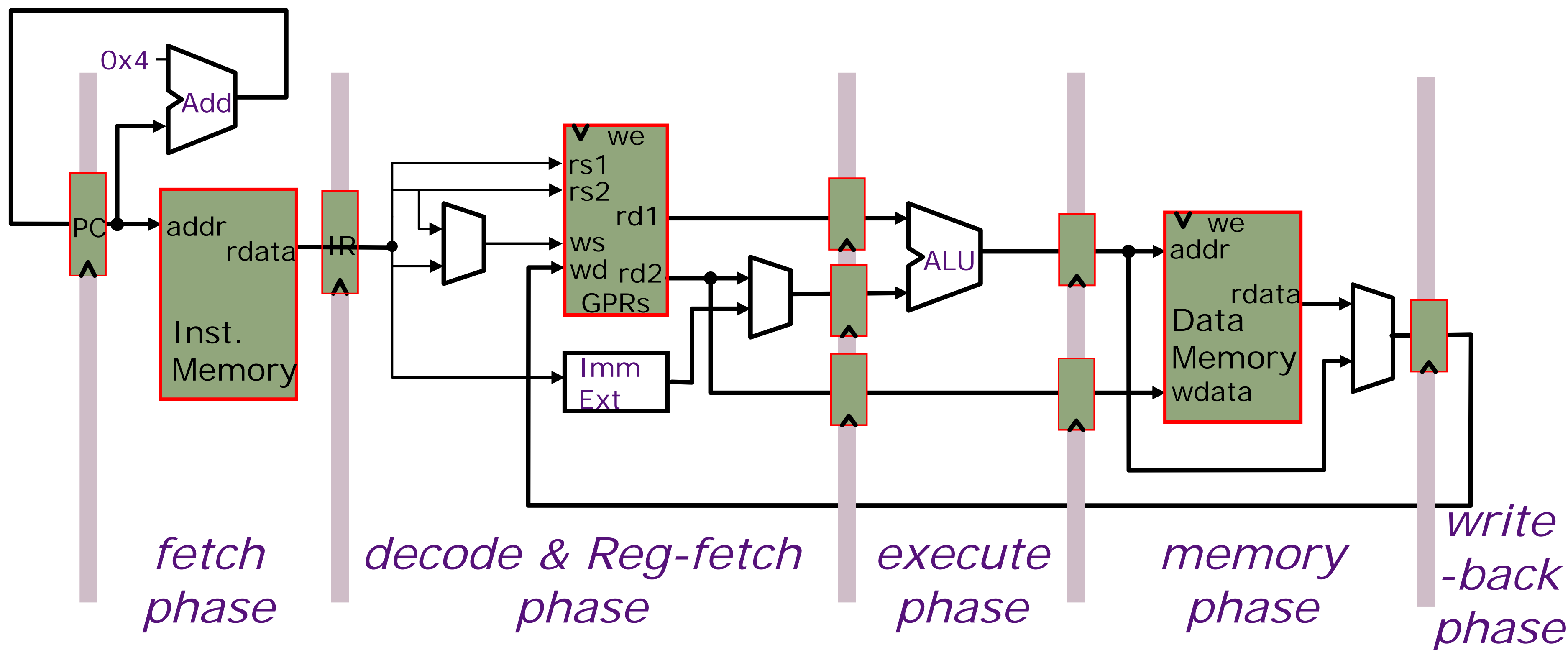
Pipelined MIPS

To pipeline MIPS:

- First build MIPS without pipelining with CPI=1
- Next, add pipeline registers to reduce cycle time while maintaining CPI=1

CSAIL

Pipelined Datapath

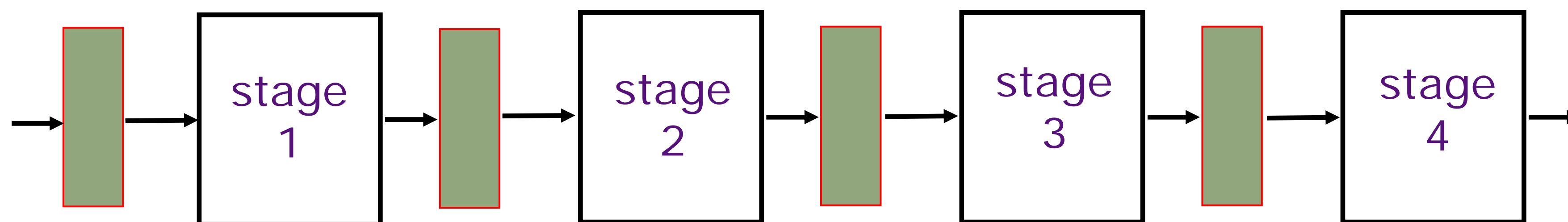


Clock period can be reduced by dividing the execution of an instruction into multiple cycles

$t_C > \max \{t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}\}$ ($= t_{DM}$ *probably*)

However, CPI will increase unless instructions are pipelined

An Ideal Pipeline



- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

These conditions generally hold for industrial assembly lines.
But can an instruction pipeline satisfy the last condition?

CSAIL

How to divide the datapath into stages

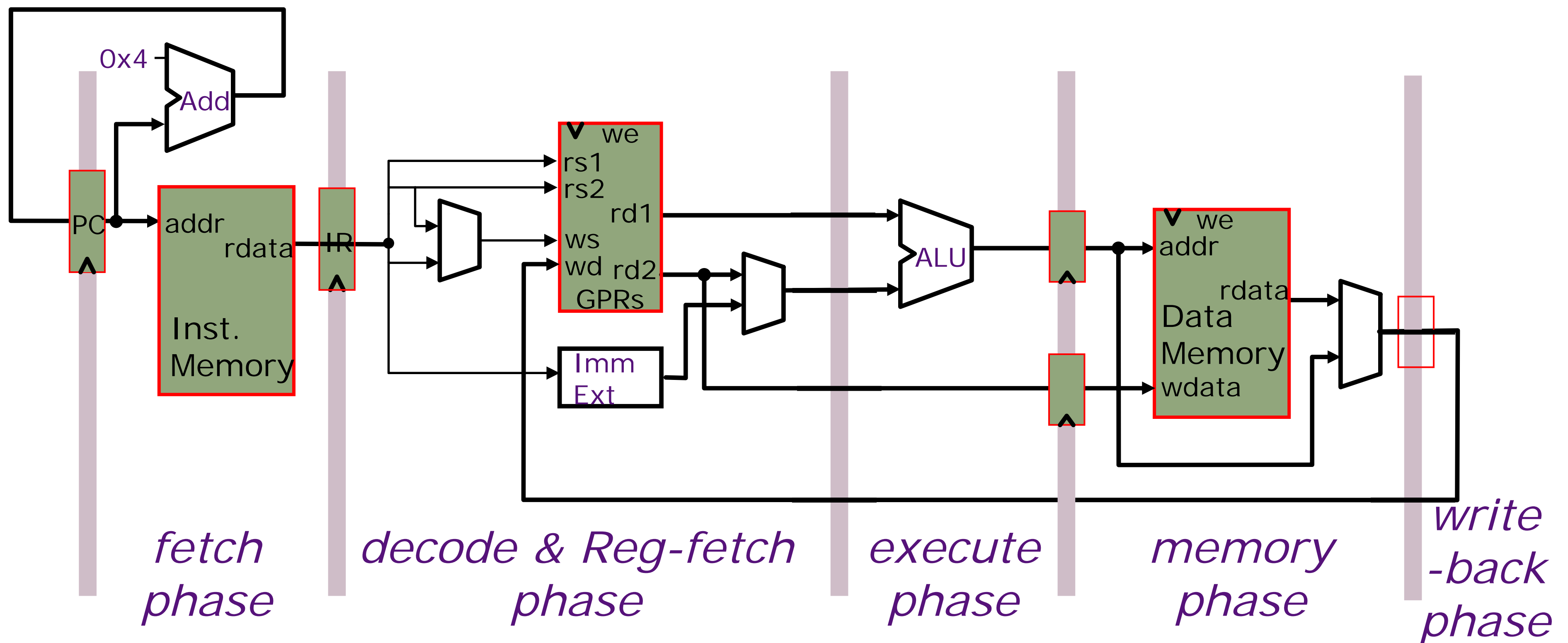
Suppose memory is significantly slower than other stages. In particular, suppose

t_{IM} = 10 units
t_{DM} = 10 units
t_{ALU} = 5 units
t_{RF} = 1 unit
t_{RW} = 1 unit

Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance

CSAIL

Alternative Pipelining



$t_C > \max\{t_{IM}, t_{RF}+t_{ALU}, t_{DM}+t_{RW}\} = t_{DM}+t_{RW}$

$\Rightarrow$ increase the critical path by 10%

Write-back stage takes much less time than other stages. Suppose we combined it with the memory phase

CSAIL

Maximum Speedup by Pipelining

Assumptions	Unpipelined t_C	Pipelined t_C	Speedup
1. $t_{IM} = t_{DM} = 10$, $t_{ALU} = 5$, $t_{RF} = t_{RW} = 1$ 4-stage pipeline	27	10	2.7
2. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$ 4-stage pipeline	25	10	2.5
3. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$ 5-stage pipeline	25	5	5.0

It is possible to achieve higher speedup with more stages in the pipeline.

CSAIL

Thank you !

Pipeline Hazards

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Based on the material prepared by
Arvind and Krste Asanovic

Technology Assumptions

- A small amount of very fast memory (caches) backed up by a large, slower memory
- Fast ALU (at least for integers)
- Multiported Register files (slower!)

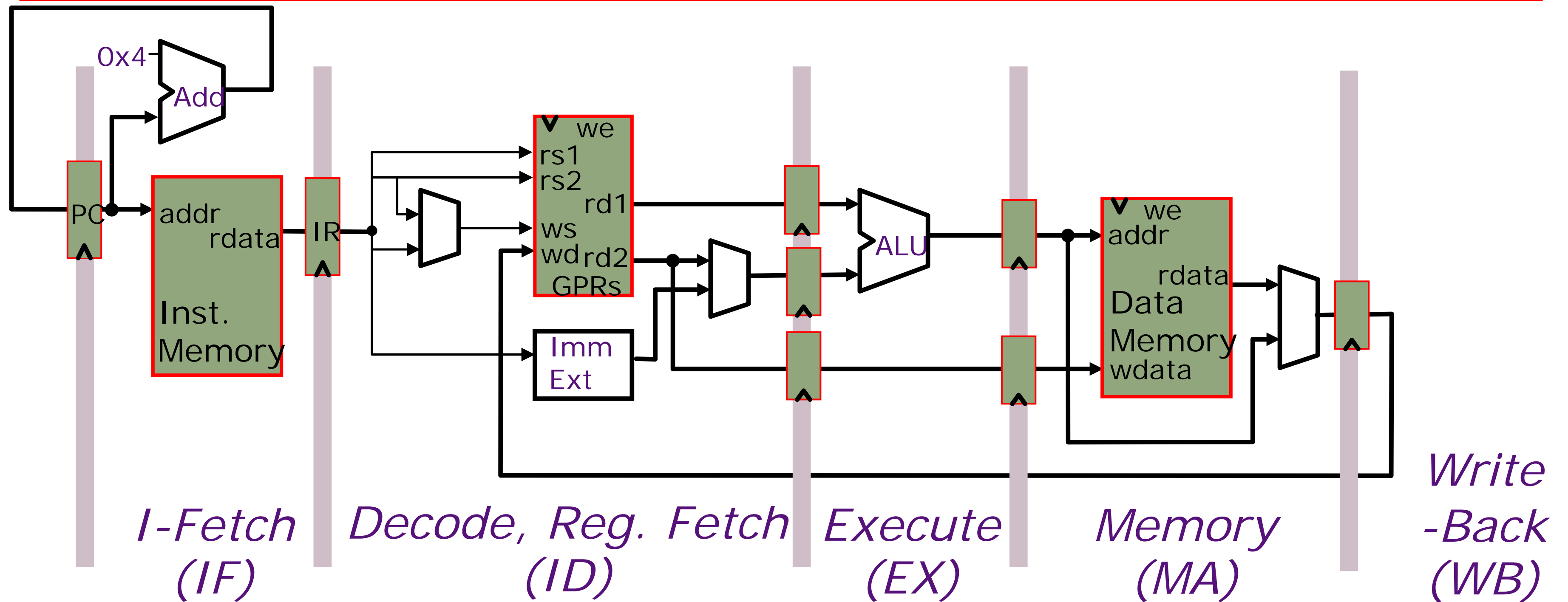
It makes the following timing assumption valid

$$t_{IM} \approx t_{RF} \approx t_{ALU} \approx t_{DM} \approx t_{RW}$$

A 5-stage pipelined Harvard architecture will be the focus of our detailed design

CSAIL

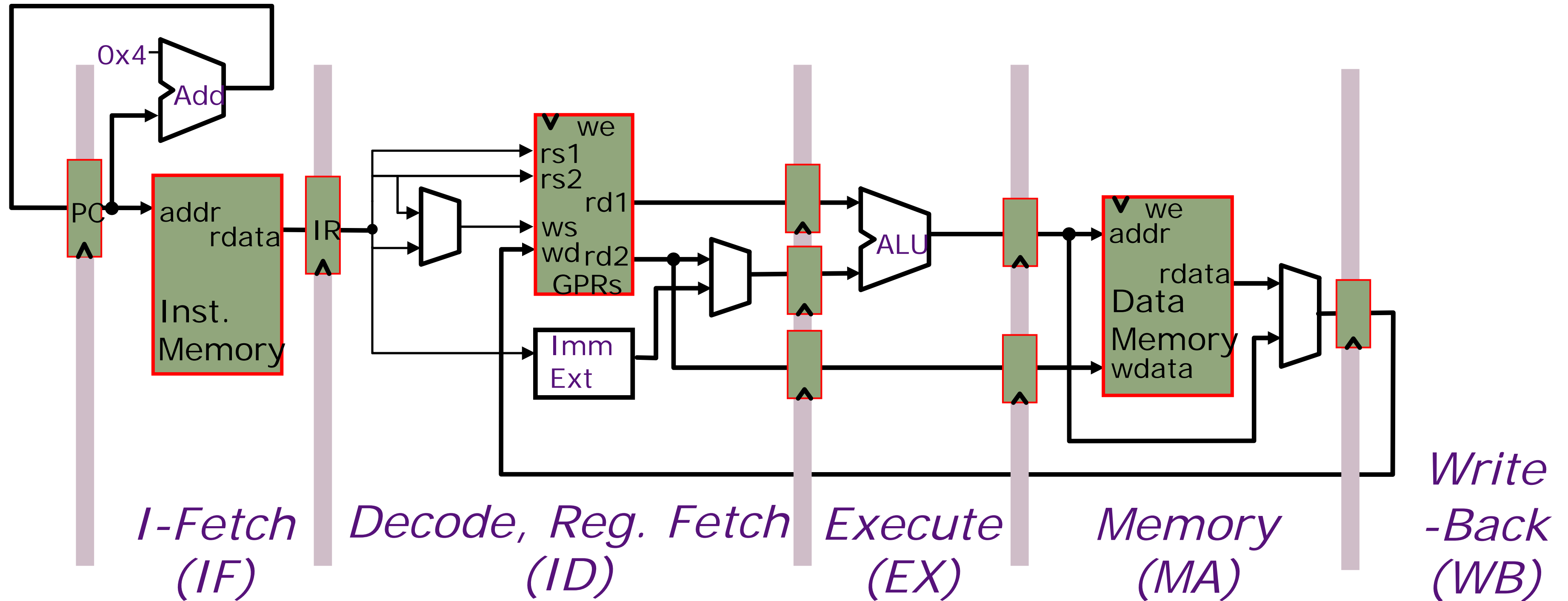
5-Stage Pipelined Execution



time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

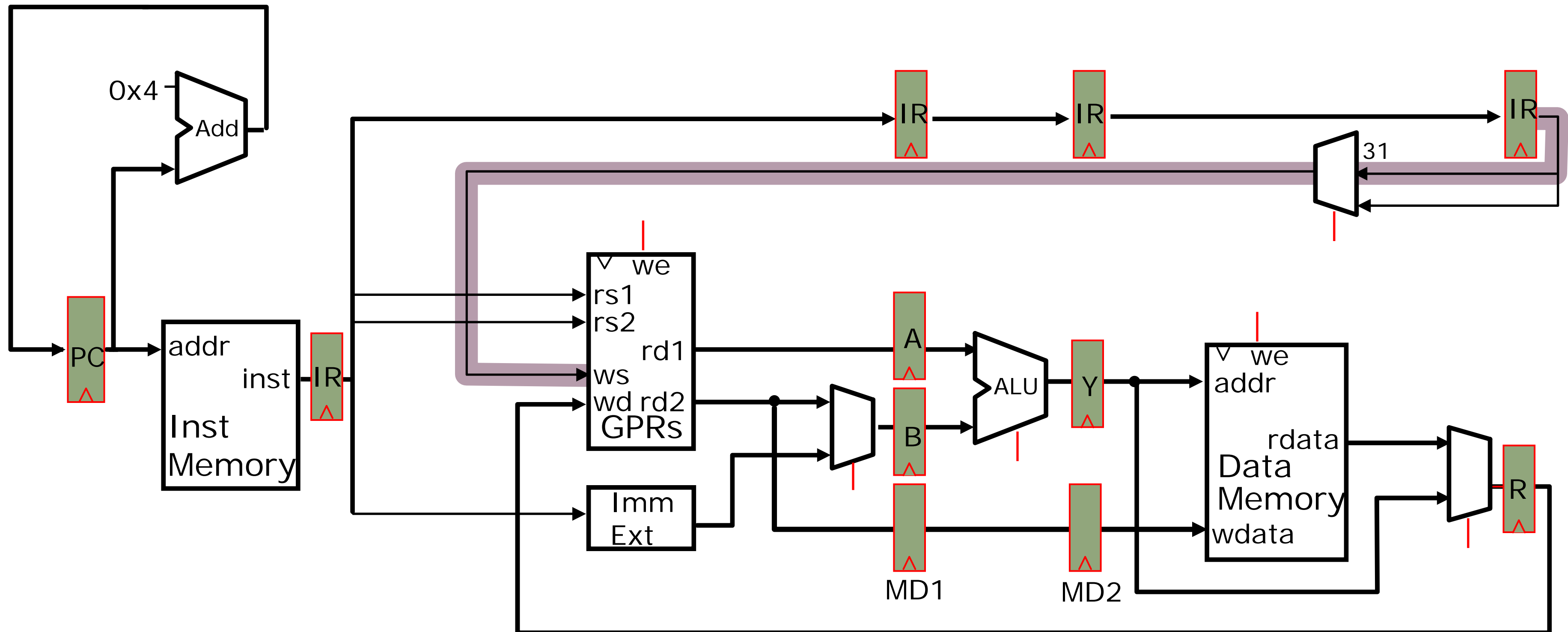
5-Stage Pipelined Execution

Resource Usage Diagram



Resources	*time*	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
	IF	I_1	I_2	I_3	I_4	I_5				
	ID		I_1	I_2	I_3	I_4	I_5			
	EX			I_1	I_2	I_3	I_4	I_5		
	MA				I_1	I_2	I_3	I_4	I_5	
	WB					I_1	I_2	I_3	I_4	I_5

Pipelined Execution: ALU Instructions

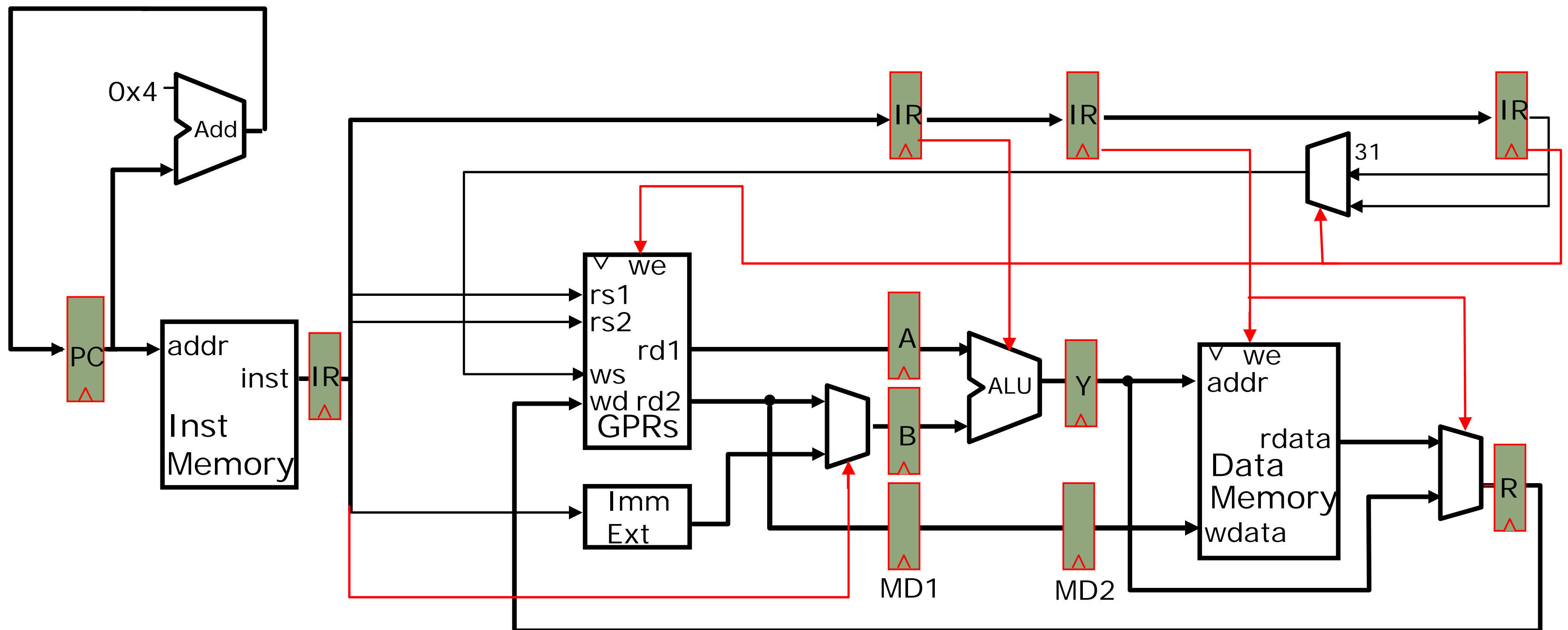


Not quite correct!

We need an Instruction Reg (IR) for each stage

CSAIL

IR's and Control points

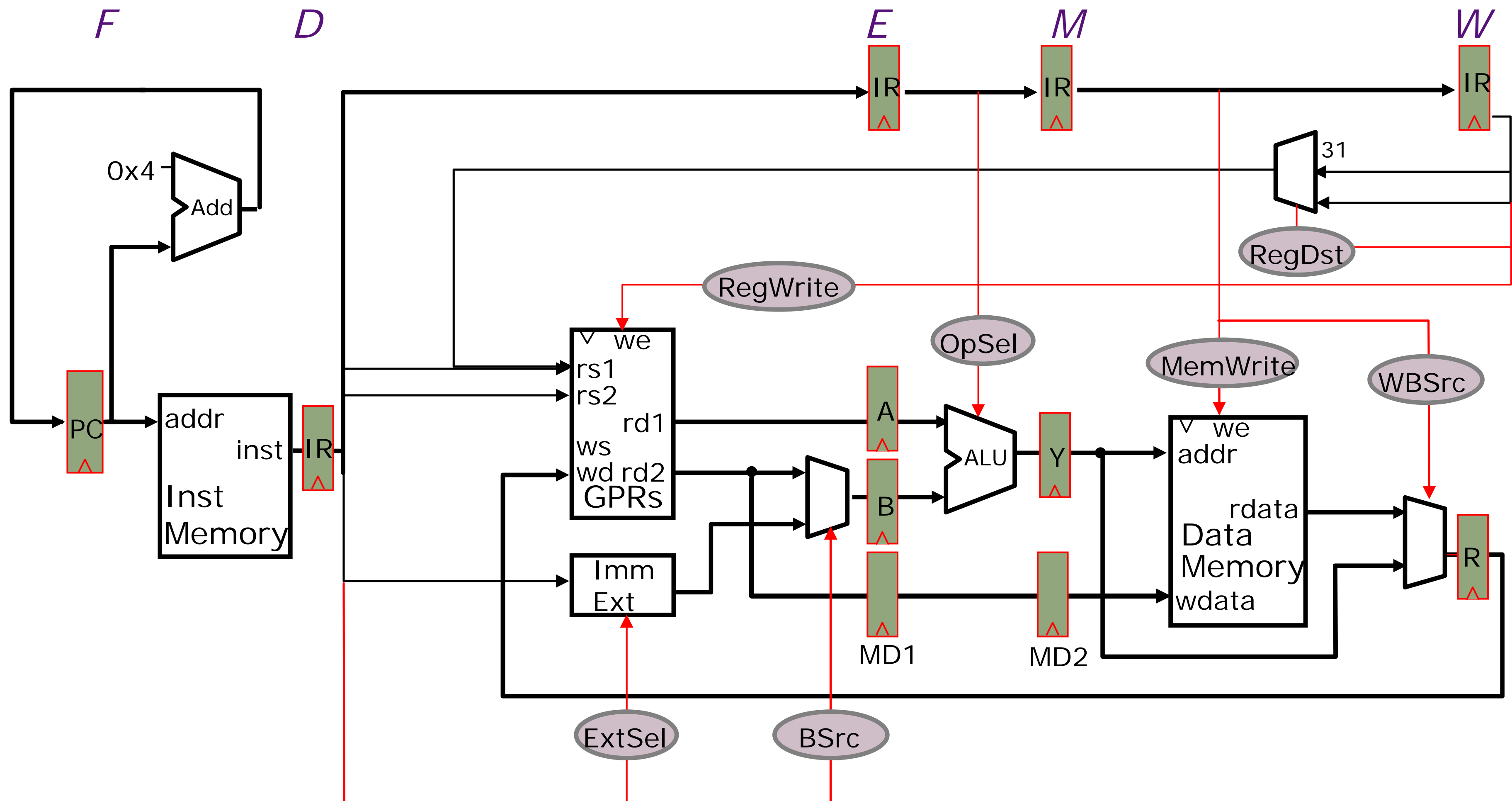


Are control points connected properly?

- ALU instructions
- Load/Store instructions
- Write back

CSAIL

Pipelined MIPS Datapath
without jumps



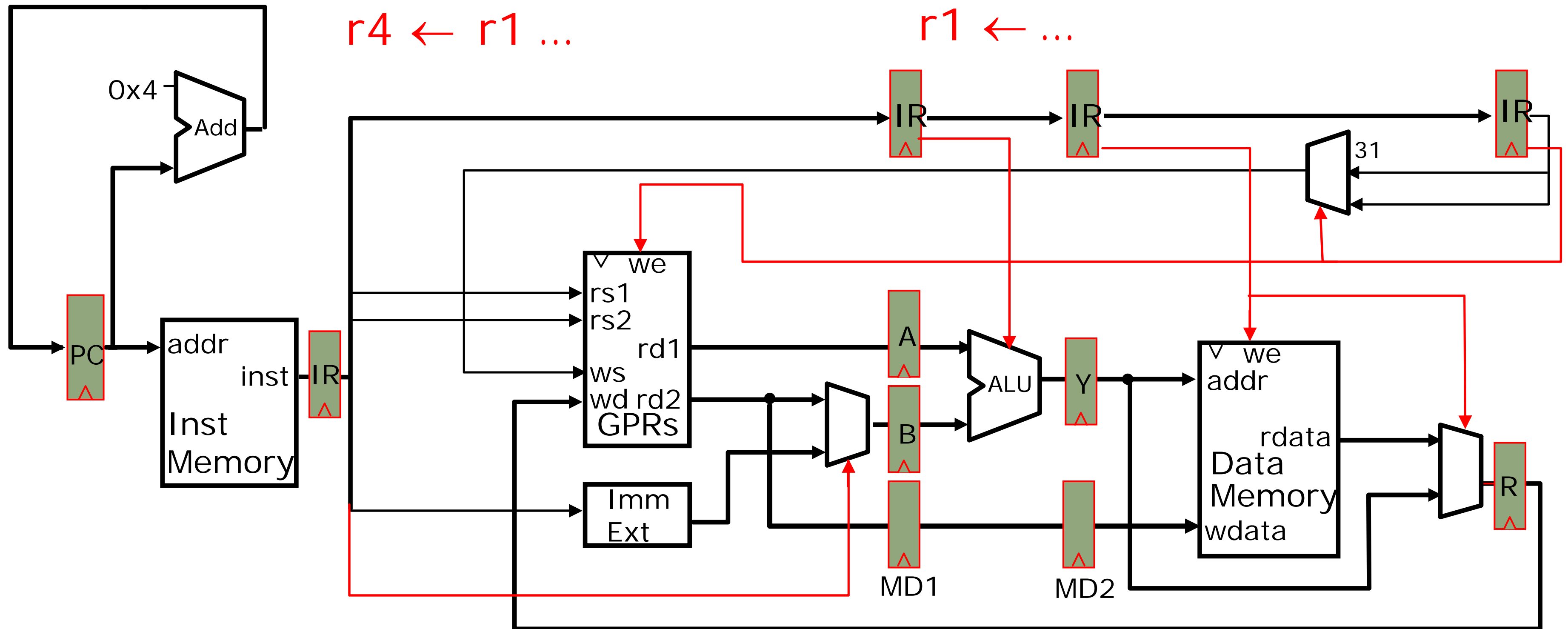
CSAIL

How Instructions can Interact with each other in a pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline
  - *structural hazard*
- An instruction may produce data that is needed by a later instruction
  - *data hazard*
- In the extreme case, an instruction may determine the next instruction to be executed
  - *control hazard (branches, interrupts,...)*

CSAIL

Data Hazards



...
r1 ← r0 + 10
r4 ← r1 + 17
...

r1 is stale. Oops!

Resolving Data Hazards

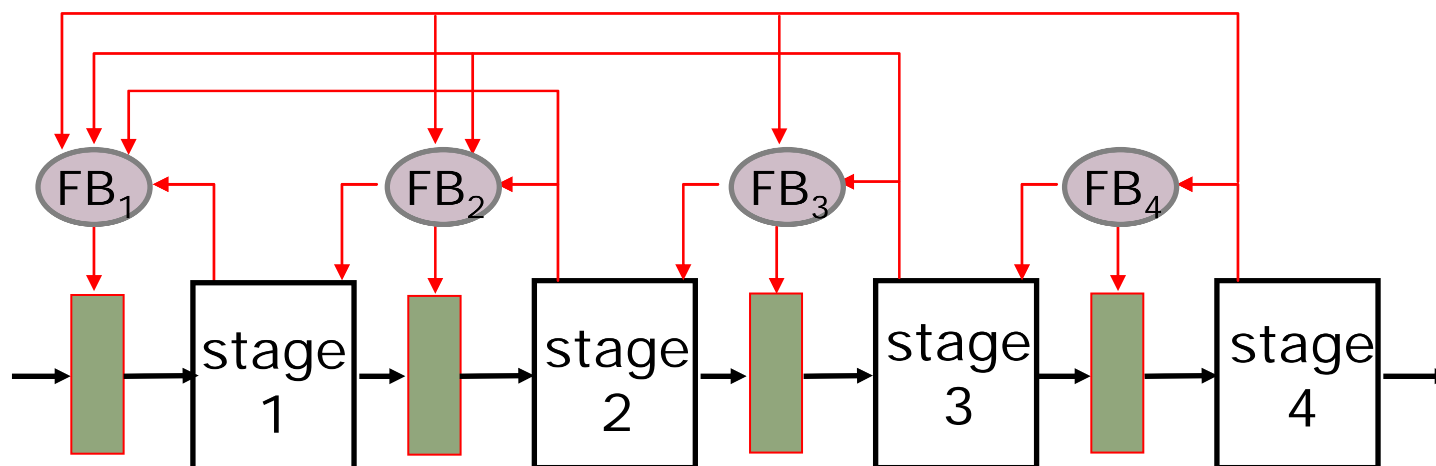
Freeze earlier pipeline stages until the data becomes available ⇒ *interlocks*

If data is available somewhere in the datapath provide a *bypass* to get it to the right stage

Speculate about the hazard resolution and *kill* the instruction later if the speculation is wrong.

CSAIL

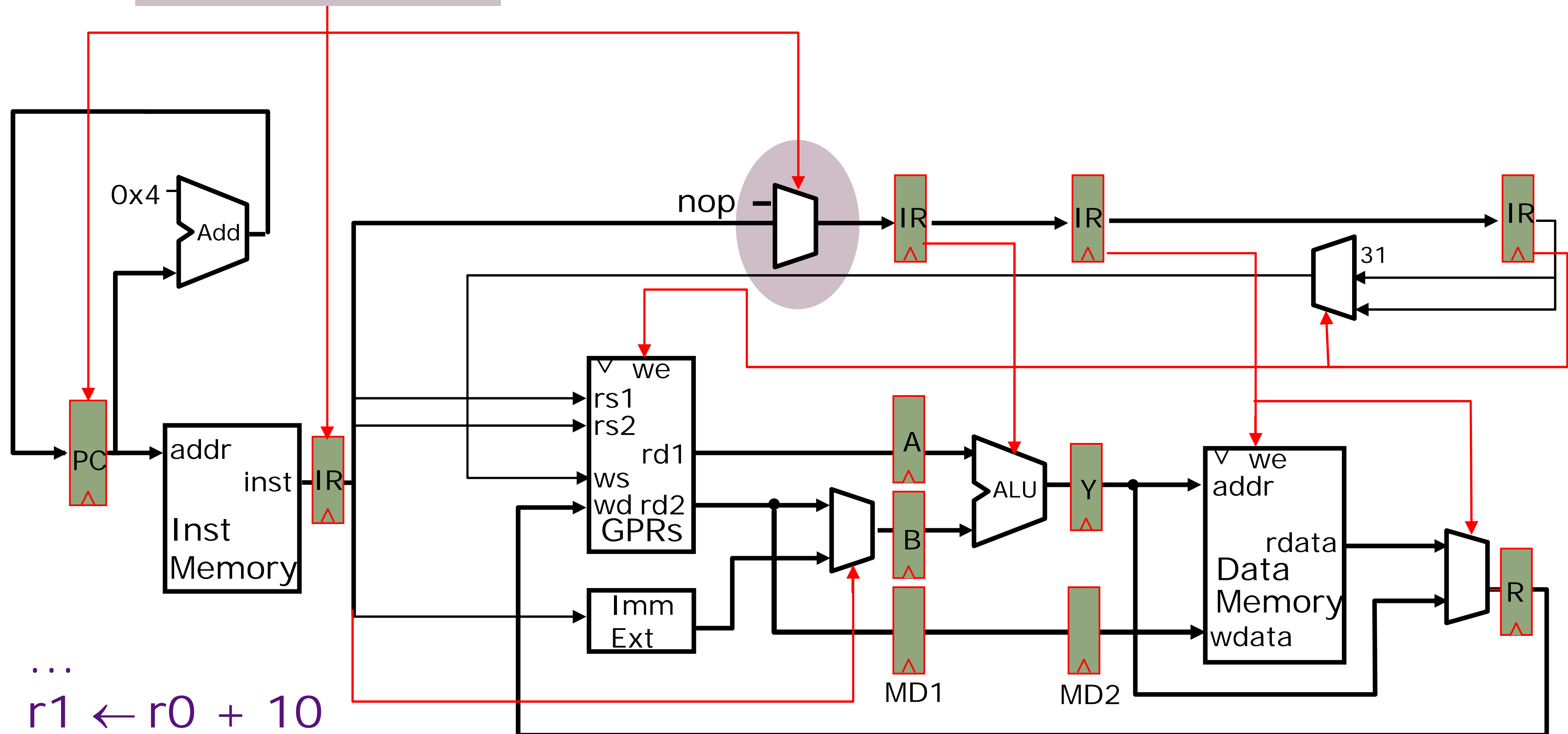
Feedback to Resolve Hazards



- Detect a hazard and provide feedback to previous stages to *stall or kill instructions*

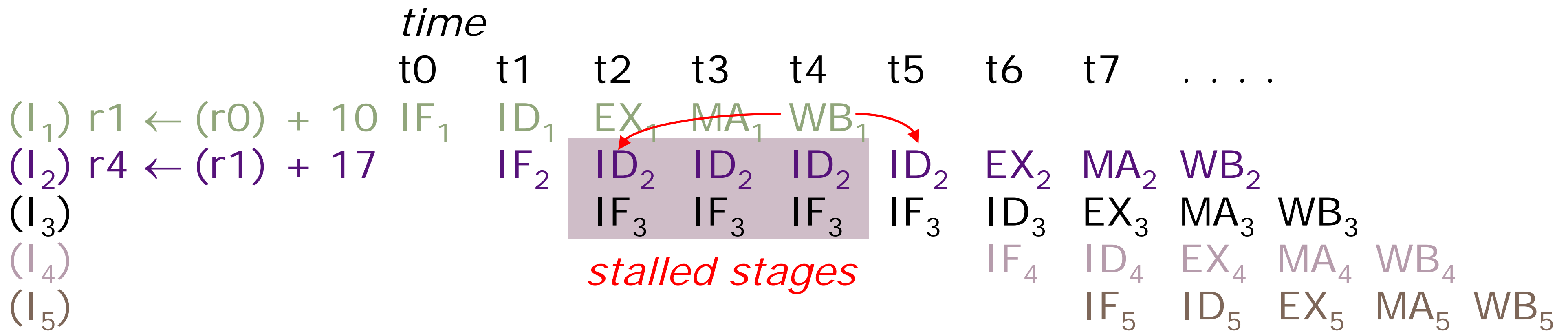
- Controlling a pipeline in this manner works provided *the instruction at stage i+1 can complete without any interference from instructions in stages 1 to i*
  (otherwise deadlocks may occur)

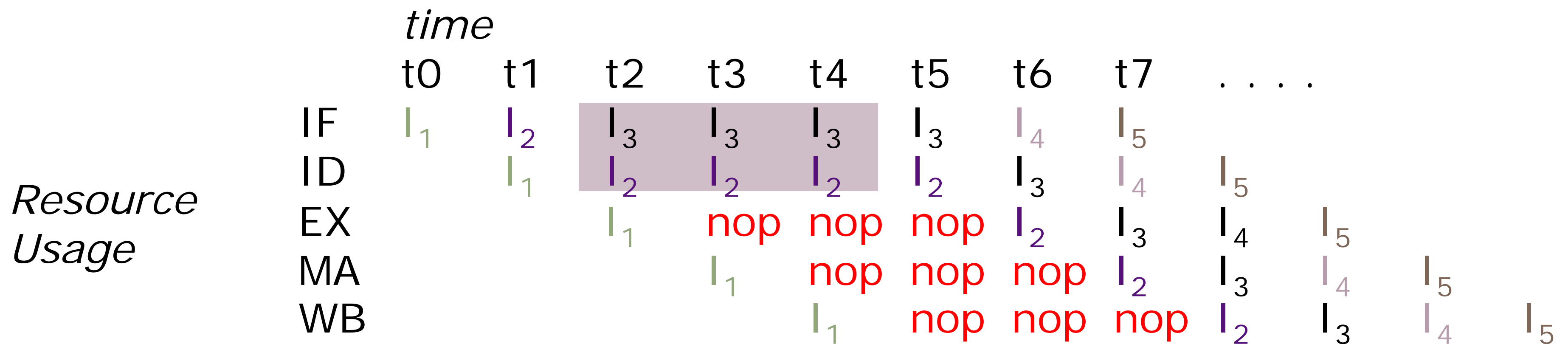
Interlocks to resolve Data Hazards



...
$r1 \leftarrow r0 + 10$
$r4 \leftarrow r1 + 17$
...

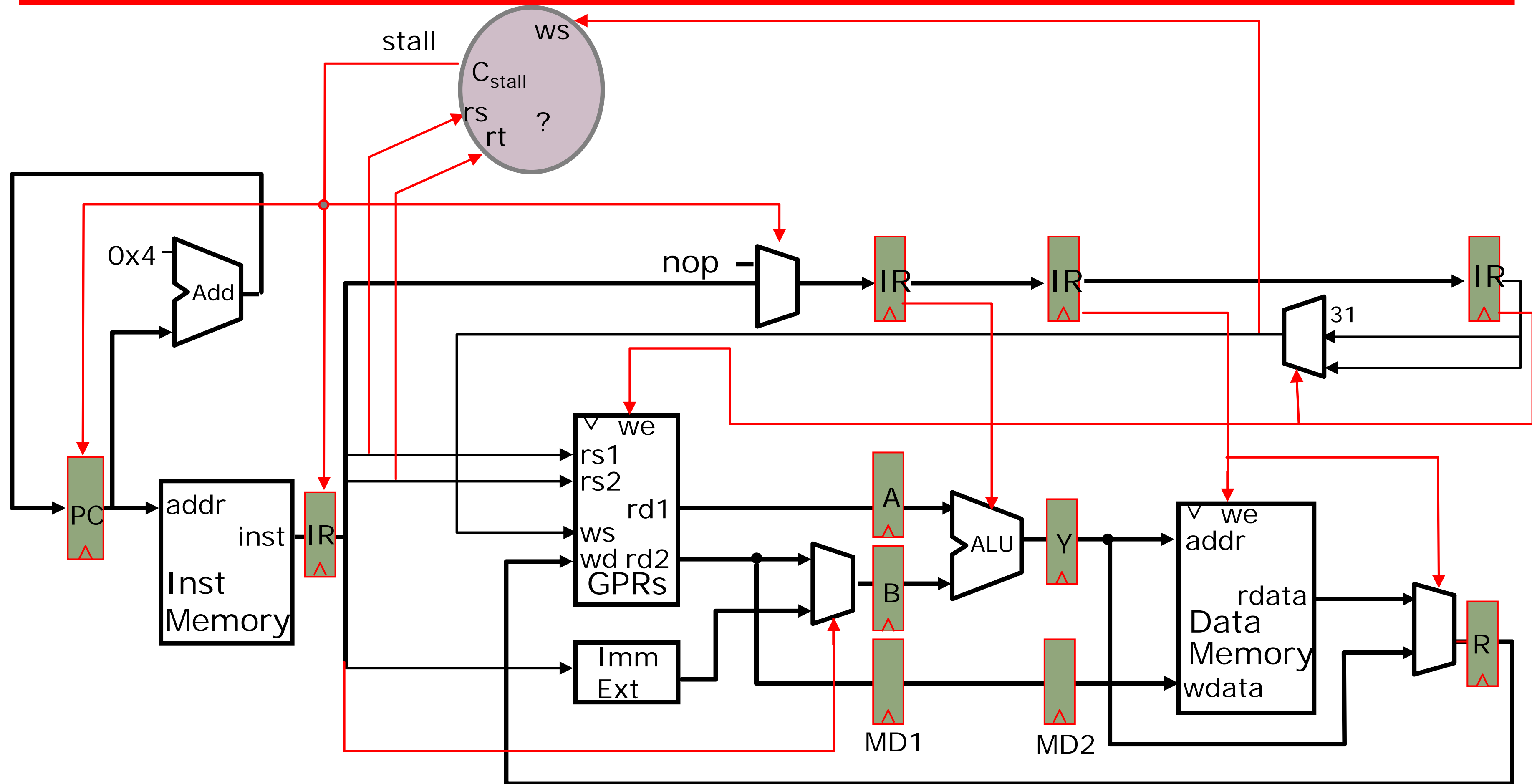
Stalled Stages and Pipeline Bubbles





nop ⇒ *pipeline bubble*

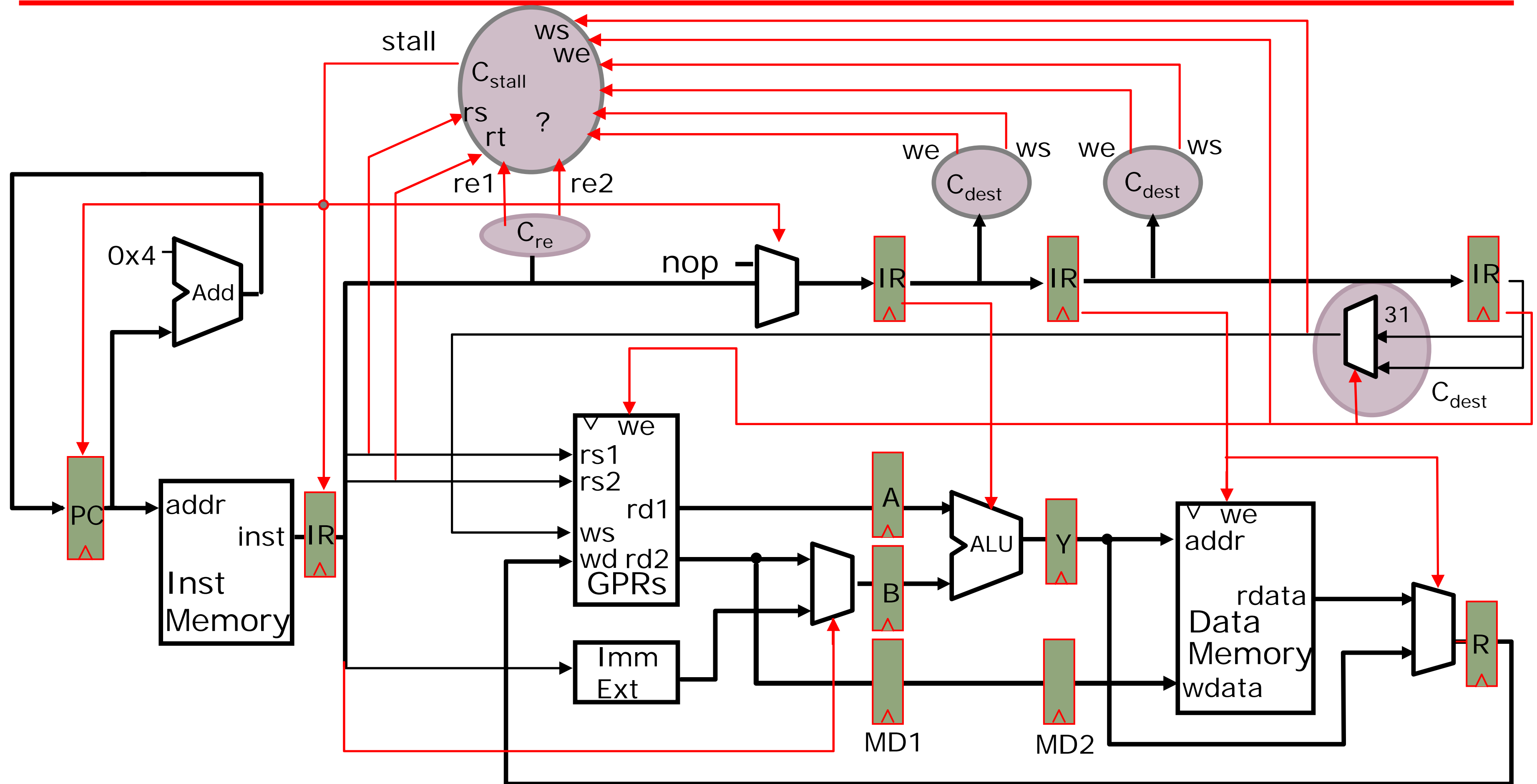
Interlock Control Logic



Compare the *source registers* of the instruction in the decode stage with the *destination register* of the *uncommitted* instructions.

Interlocks Control Logic

ignoring jumps & branches



Should we always stall if the rs field matches some rd?

not every instruction writes a register ⇒ we

not every instruction reads a register ⇒ re

Source & Destination Registers

R-type:

op	rs	rt	rd		func

I-type:

op	rs	rt	immediate16

J-type:

op	immediate26

		source(s)	*destination*
ALU	rd ← (rs) func (rt)	rs, rt	rd
ALUi	rt ← (rs) op imm	rs	rt
LW	rt ← M [(rs) + imm]	rs	rt
SW	M [(rs) + imm] ← (rt)	rs, rt	
BZ	*cond* (rs)		
	true: PC ← (PC) + imm	rs	
	false: PC ← (PC) + 4	rs	
J	PC ← (PC) + imm		
JAL	r31 ← (PC), PC ← (PC) + imm		31
JR	PC ← (rs)	rs	
JALR	r31 ← (PC), PC ← (rs)	rs	31

CSAIL

Deriving the Stall Signal

C_{dest}

ws = *Case* opcode
ALU ⇒ rd
ALUi, LW ⇒ rt
JAL, JALR ⇒ R31

we = *Case* opcode
ALU, ALUi, LW ⇒ (ws ≠ 0)
JAL, JALR ⇒ on
... ⇒ off

C_{re}

re1 = *Case* opcode
ALU, ALUi,
LW, SW, BZ,
JR, JALR ⇒ on
J, JAL ⇒ off

re2 = *Case* opcode
ALU, SW ⇒ on
... ⇒ off

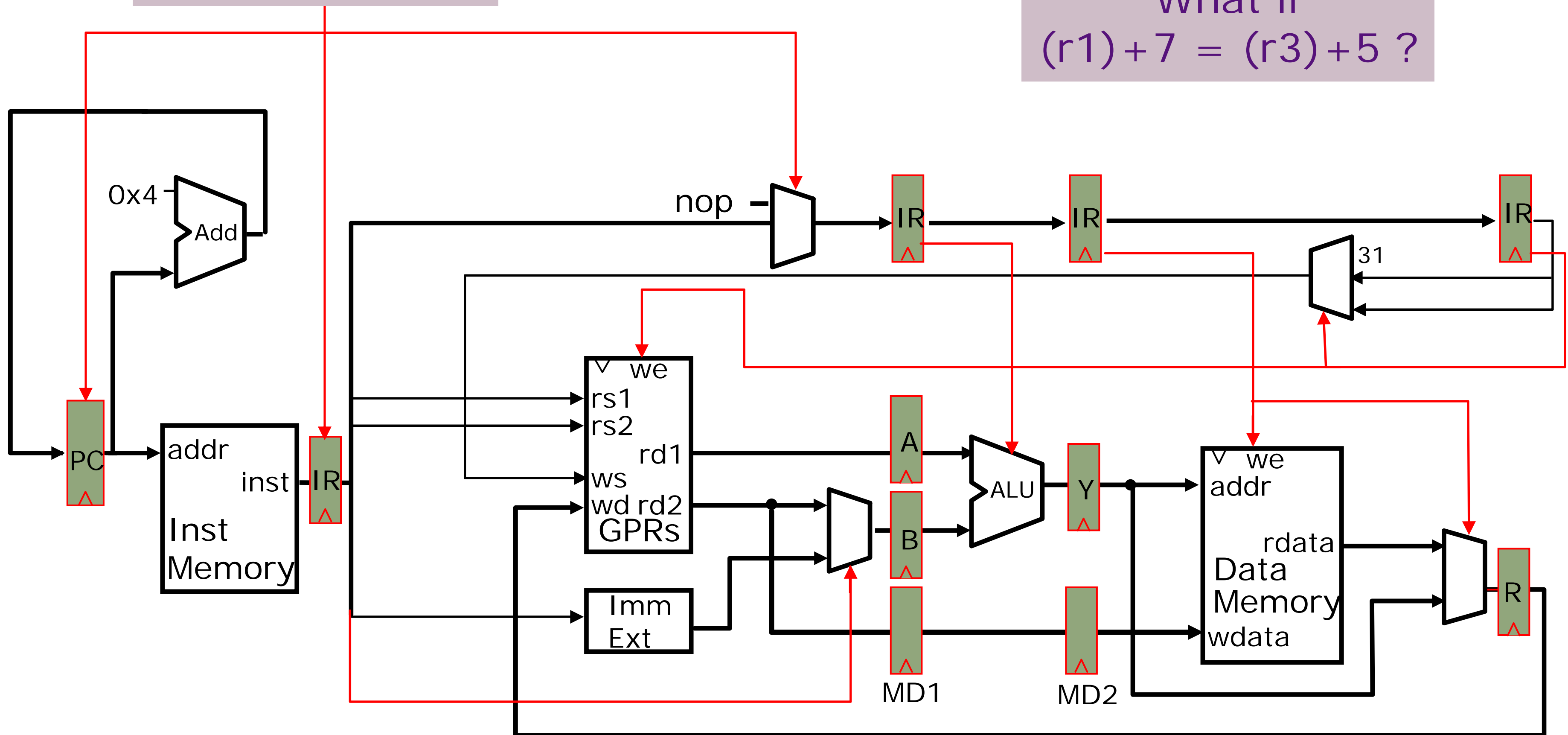
C_{stall}

$$
\begin{aligned}
stall = & ((rs_D = ws_E).we_E + \\
& (rs_D = ws_M).we_M + \\
& (rs_D = ws_W).we_W) \; . \; re1_D \quad + \\
& ((rt_D = ws_E).we_E + \\
& (rt_D = ws_M).we_M + \\
& (rt_D = ws_W).we_W) \; . \; re2_D
\end{aligned}
$$

This is not the full story !

CSAIL

Hazards due to Loads & Stores



...

M[(r1)+7] ← (r2)

r4 ← M[(r3)+5]

...

Is there any possible data hazard in this instruction sequence?

Load & Store Hazards

...
$M[(r1)+7] \leftarrow (r2)$
$r4 \leftarrow M[(r3)+5]$
...

$(r1)+7 = (r3)+5 \Rightarrow$ *data hazard*

However, the hazard is avoided because *our memory system completes writes in one cycle !*

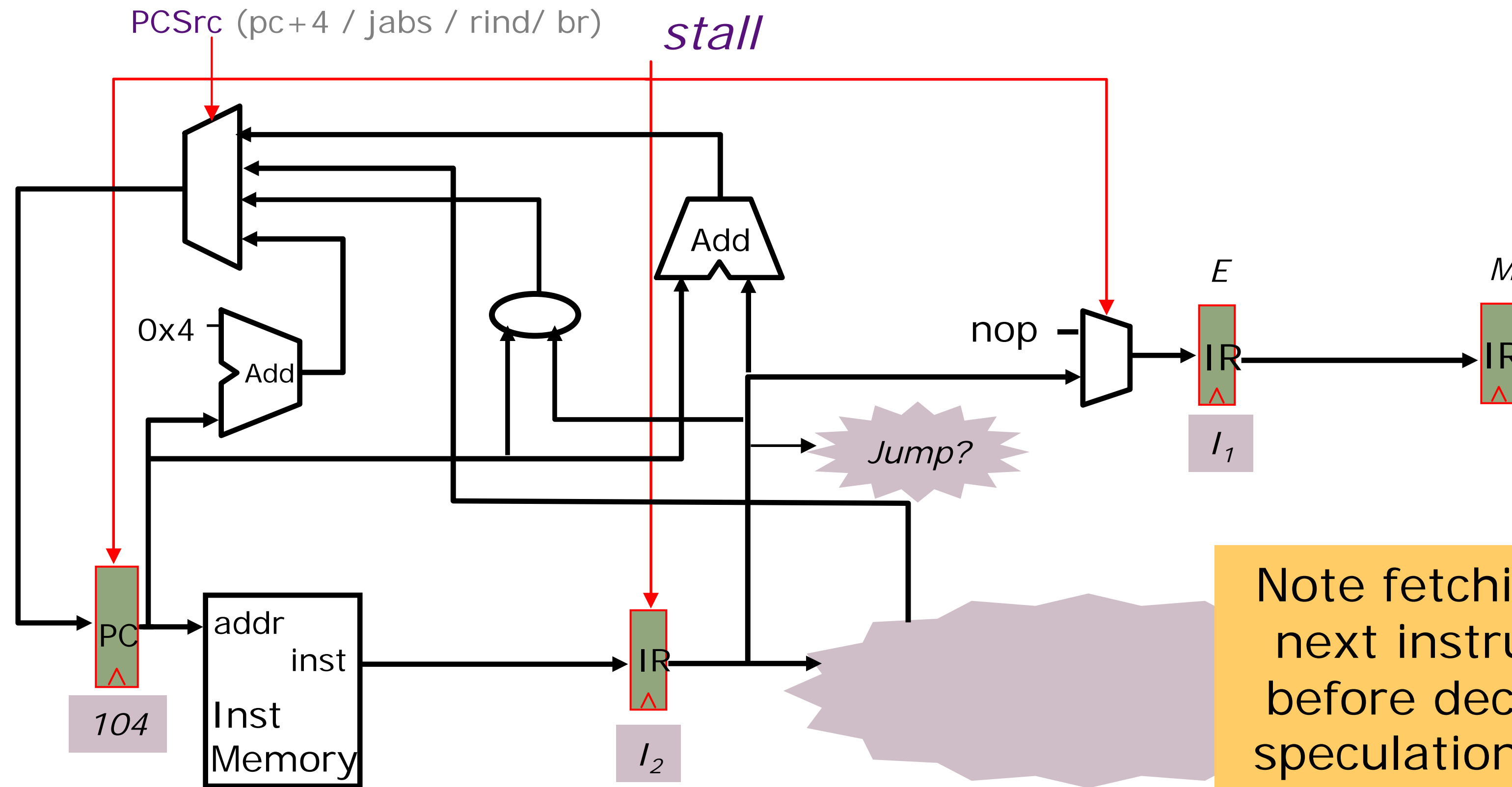
Load/Store hazards, even when they do exist, are often resolved in the memory system itself.

More on this later in the course.

CSAIL

Five-minute break to stretch your legs

Complications due to Jumps

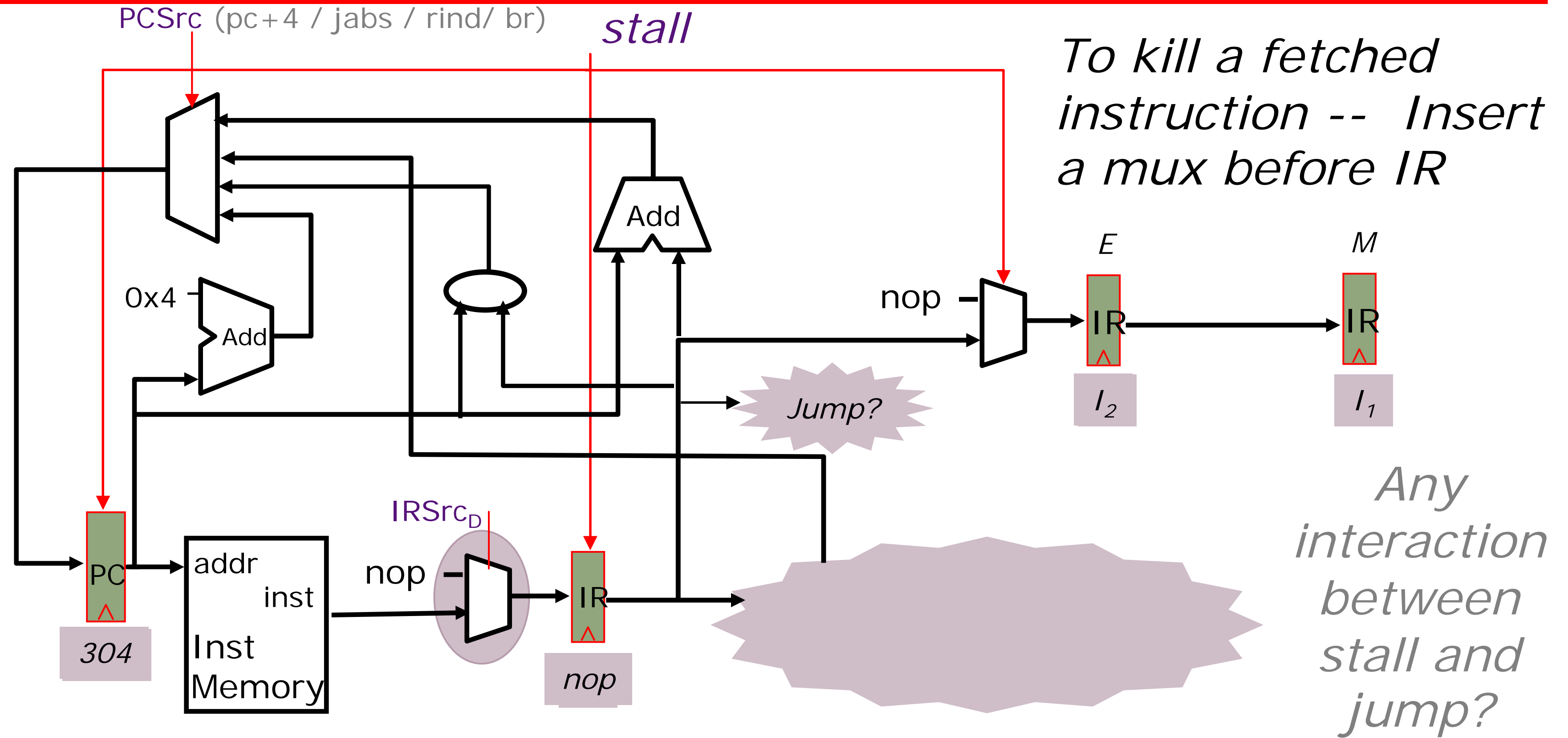


I_1	096	ADD	
I_2	100	J	200
I_3	~~104~~	~~ADD~~	*kill*
I_4	304	ADD	

A jump instruction kills (not stalls) the following instruction

How?

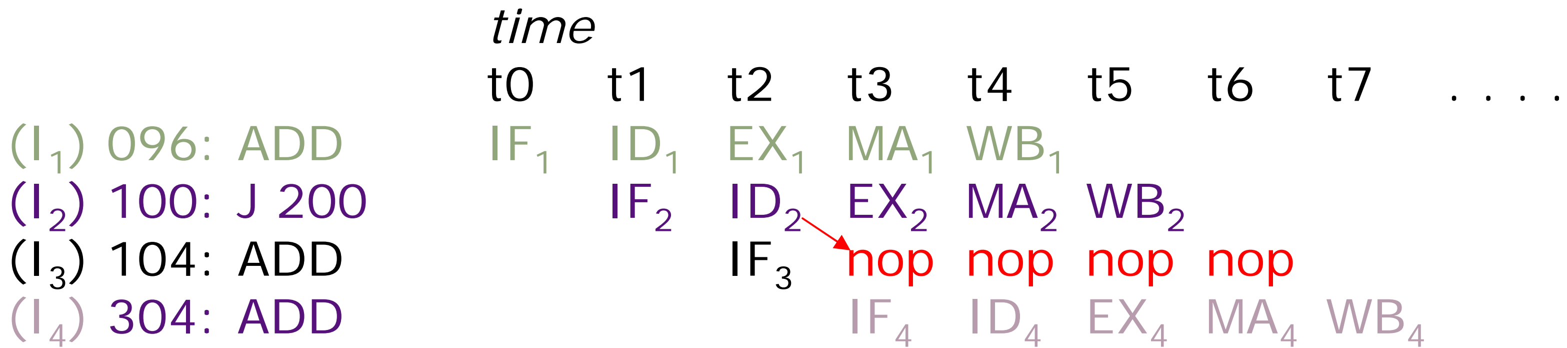
Pipelining Jumps

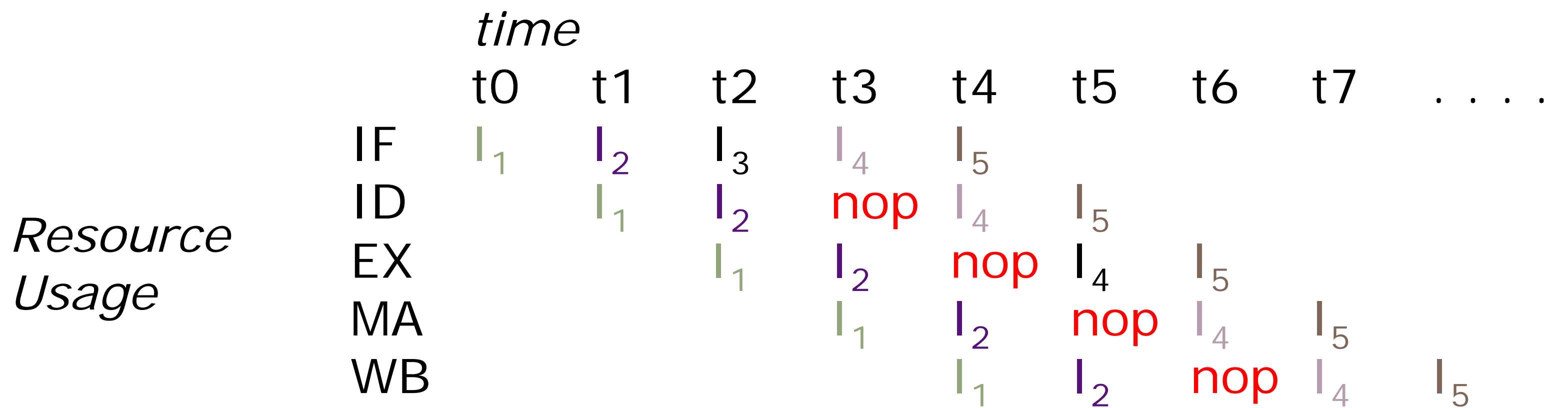


I_1	096	ADD	
I_2	100	J	200
I_3	~~104~~	~~ADD~~	kill
I_4	304	ADD	

$IRSrc_D$ = *Case* $opcode_D$
J, JAL ⇒ nop
... ⇒ IM

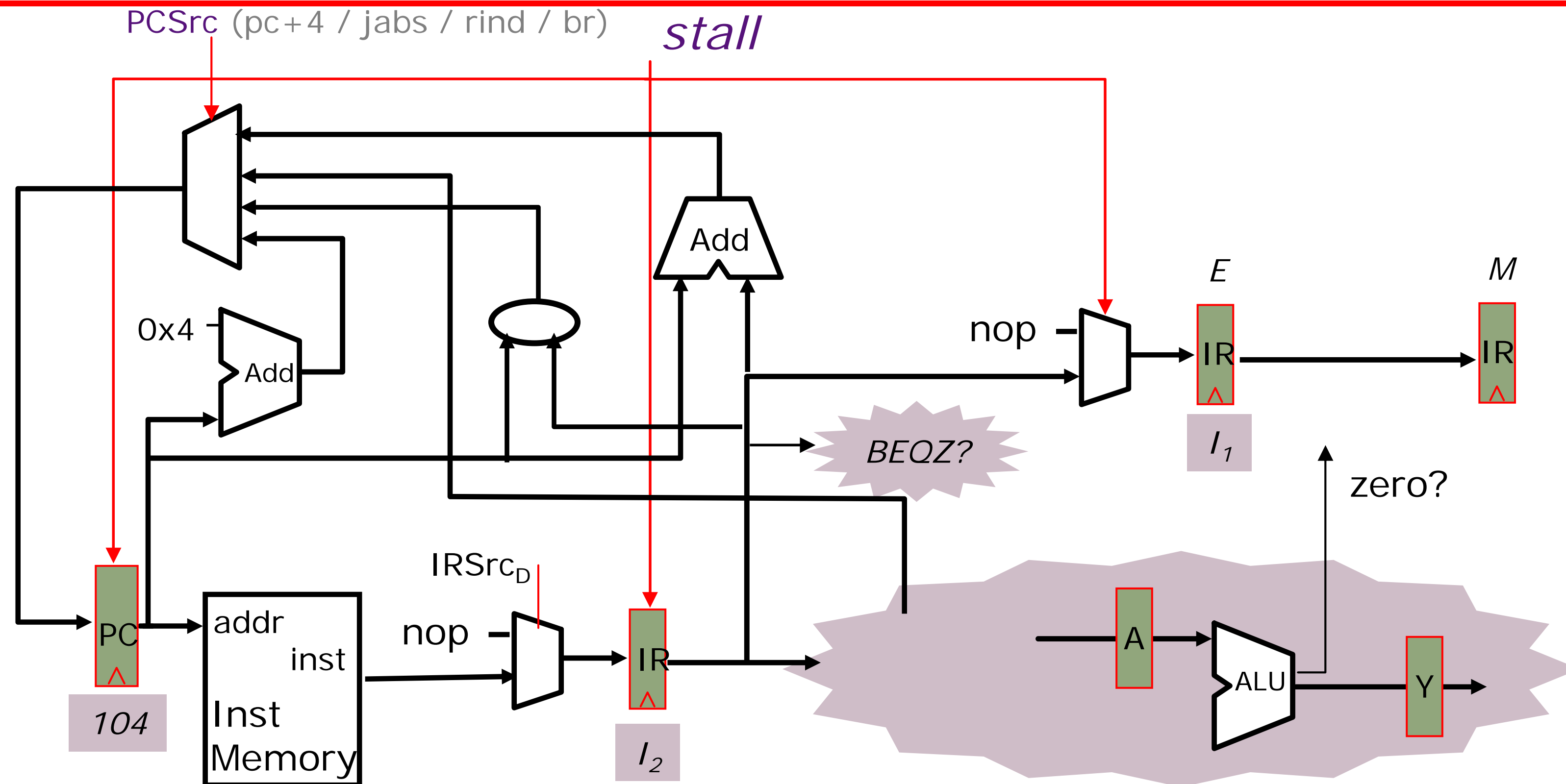
Jump Pipeline Diagrams





nop $\Rightarrow$ *pipeline bubble*

Pipelining Conditional Branches

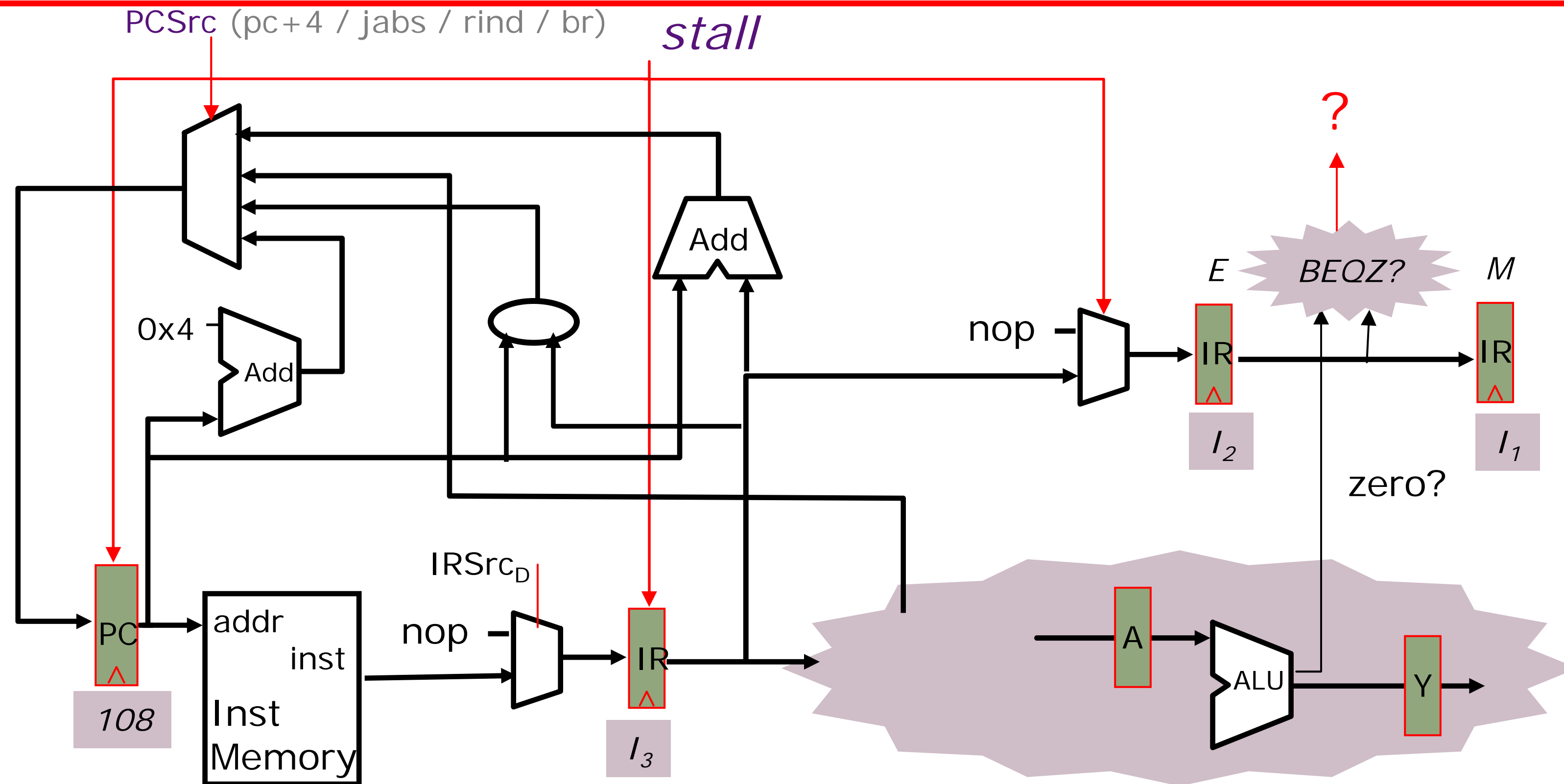


I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

Branch condition is not known until the execute stage

what action should be taken in the decode stage ?

Pipelining Conditional Branches

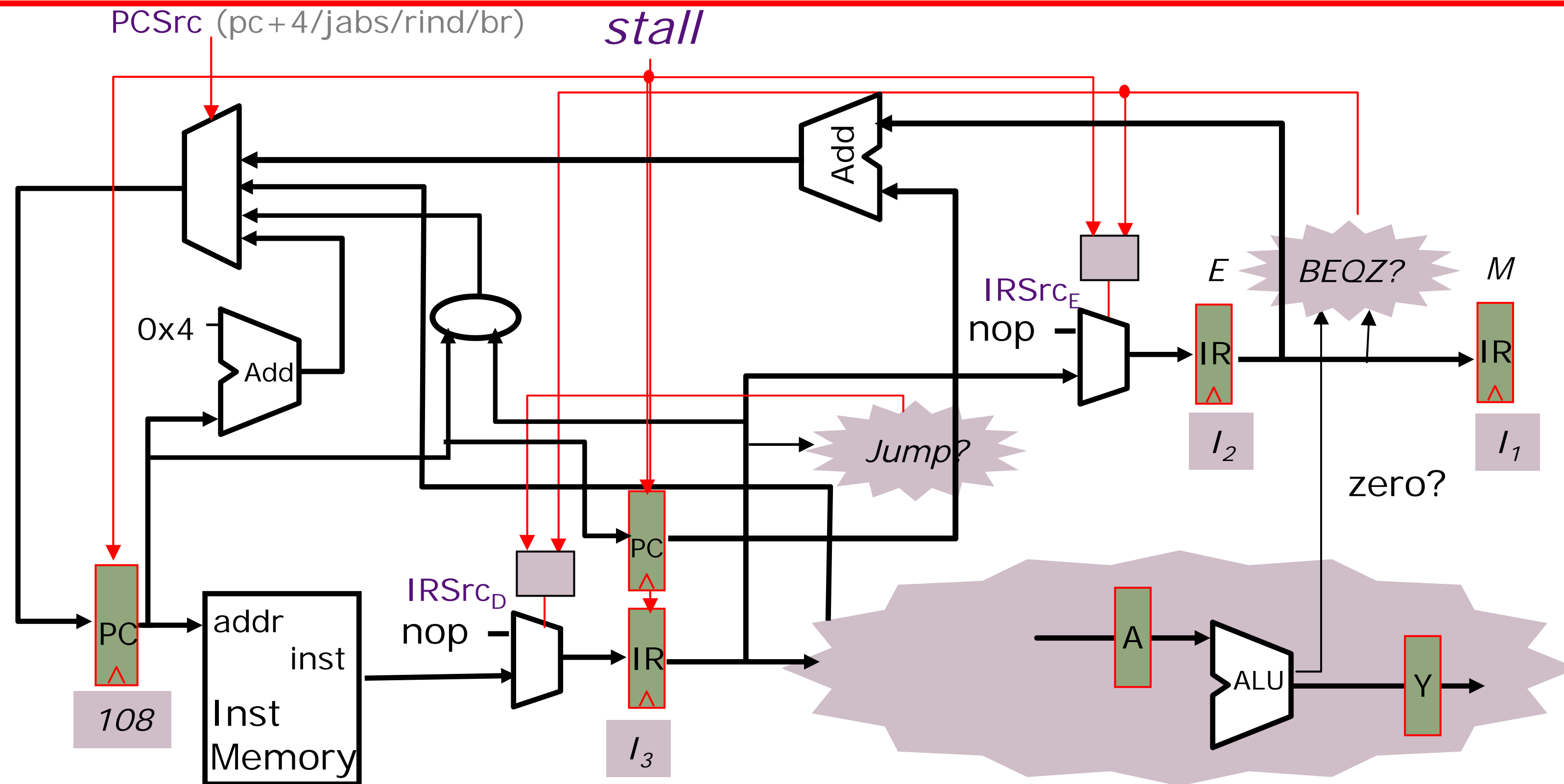


If the branch is taken
- kill the two following instructions
- the instruction at the decode stage is not valid

⇒ *stall signal is not valid*

I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

Pipelining Conditional Branches



I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

If the branch is taken
- kill the two following instructions
- the instruction at the decode stage is not valid

⇒ *stall signal is not valid*

New Stall Signal

$$stall = (((rs_D = ws_E).we_E + (rs_D = ws_M).we_M + (rs_D = ws_W).we_W).re1_D$$
$$+ ((rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W).re2_D$$
$$) . !((opcode_E = BEQZ).z + (opcode_E = BNEZ).!z)$$

Don't stall if the branch is taken. Why?

Instruction at the decode stage is invalid

CSAIL

Control Equations for PC and IR Muxes

PCSrc = *Case* $opcode_E$
  BEQZ.z, BNEZ.!z ⇒ br
  ... ⇒
    Case $opcode_D$
      J, JAL ⇒ jabs
      JR, JALR ⇒ rind
      ... ⇒ pc+4

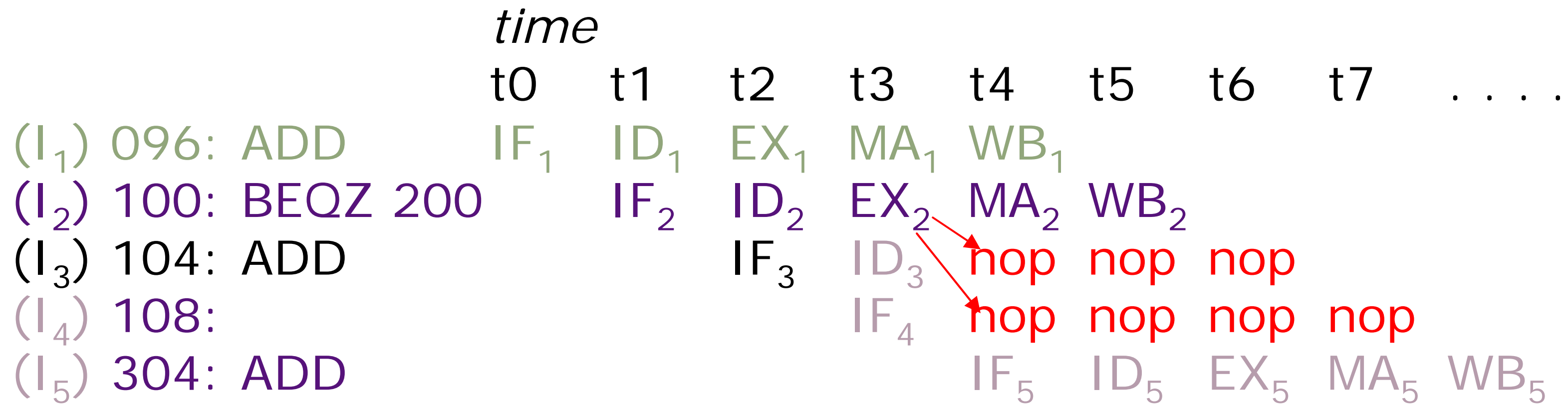
$IRSrc_D$ = *Case* $opcode_E$
  BEQZ.z, BNEZ.!z ⇒ nop
  ... ⇒
    Case $opcode_D$
      J, JAL, JR, JALR ⇒ nop
      ... ⇒ IM

$IRSrc_E$ = *Case* $opcode_E$
  BEQZ.z, BNEZ.!z ⇒ nop
  ... ⇒ stall.nop + !stall.IR_D

Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction

CSAIL

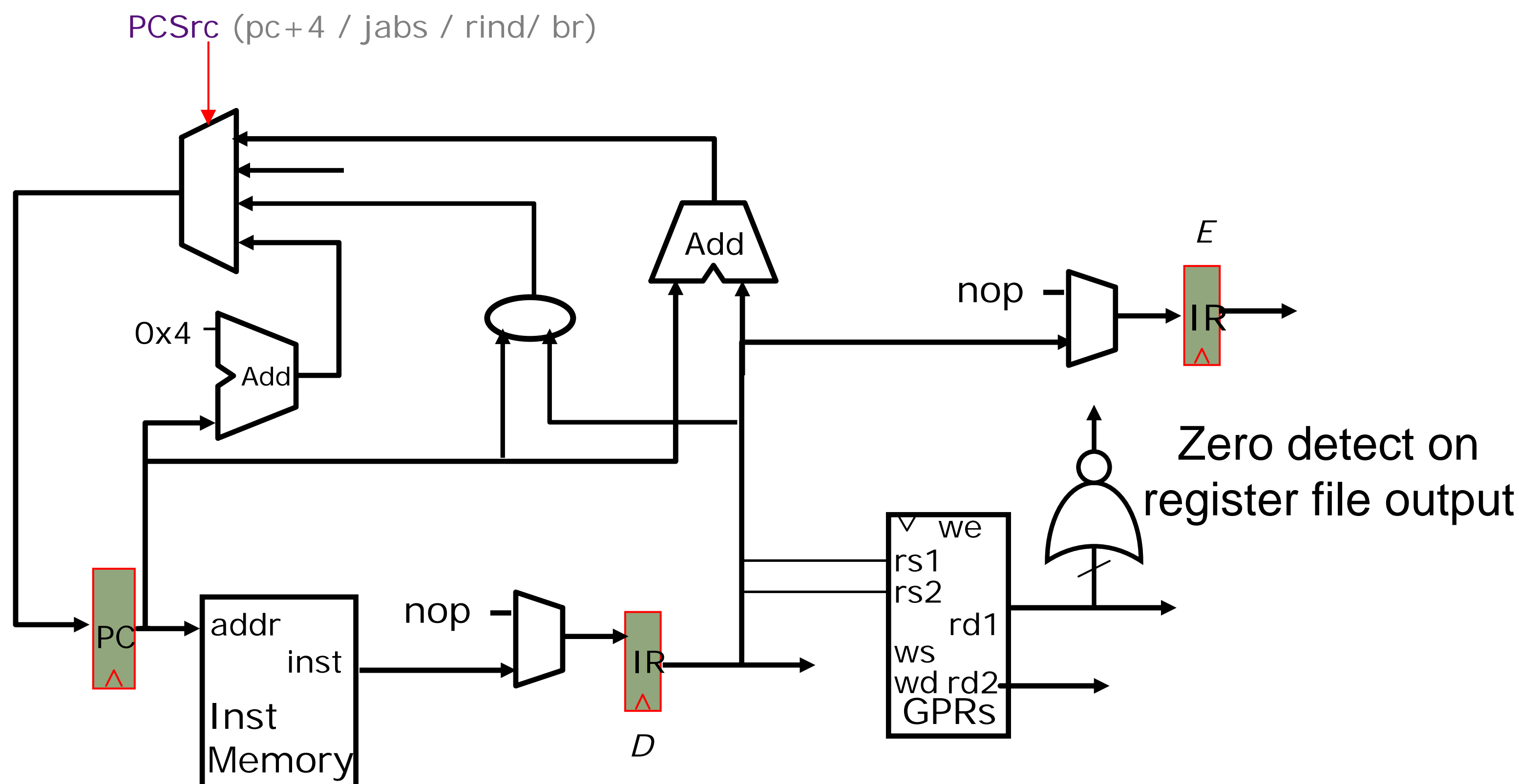
Branch Pipeline Diagrams
(resolved in execute stage)



nop ⇒ *pipeline bubble*

Reducing Branch Penalty
(resolve in decode stage)

- One pipeline bubble can be removed if an extra comparator is used in the Decode stage



Pipeline diagram now same as for jumps

Branch Delay Slots (expose control hazard to software)

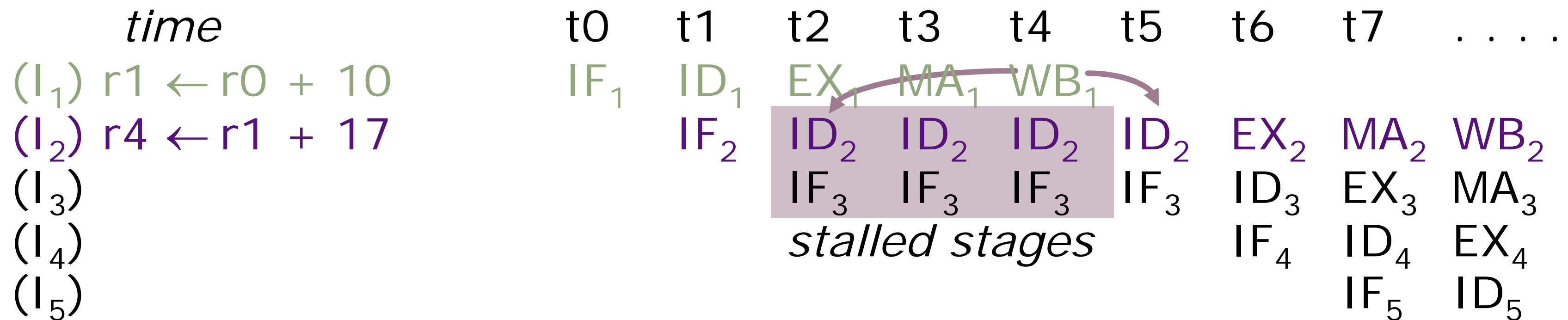
- Change the ISA semantics so that the instruction that follows a jump or branch is always executed
  - gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.

I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD ← *Delay slot instruction executed regardless of branch outcome*
I_4	304	ADD

- Other techniques include branch prediction, which can dramatically reduce the branch penalty... *to come later*

CSAIL

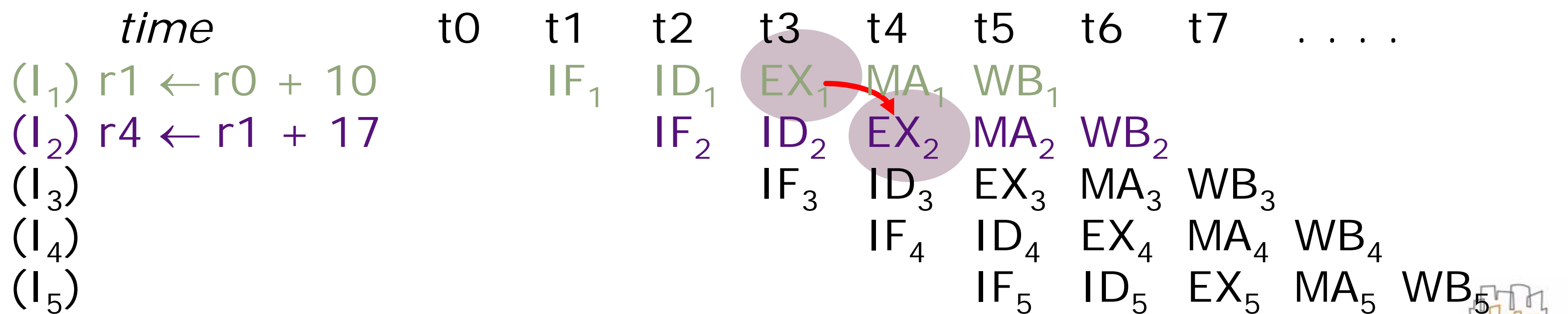
Bypassing



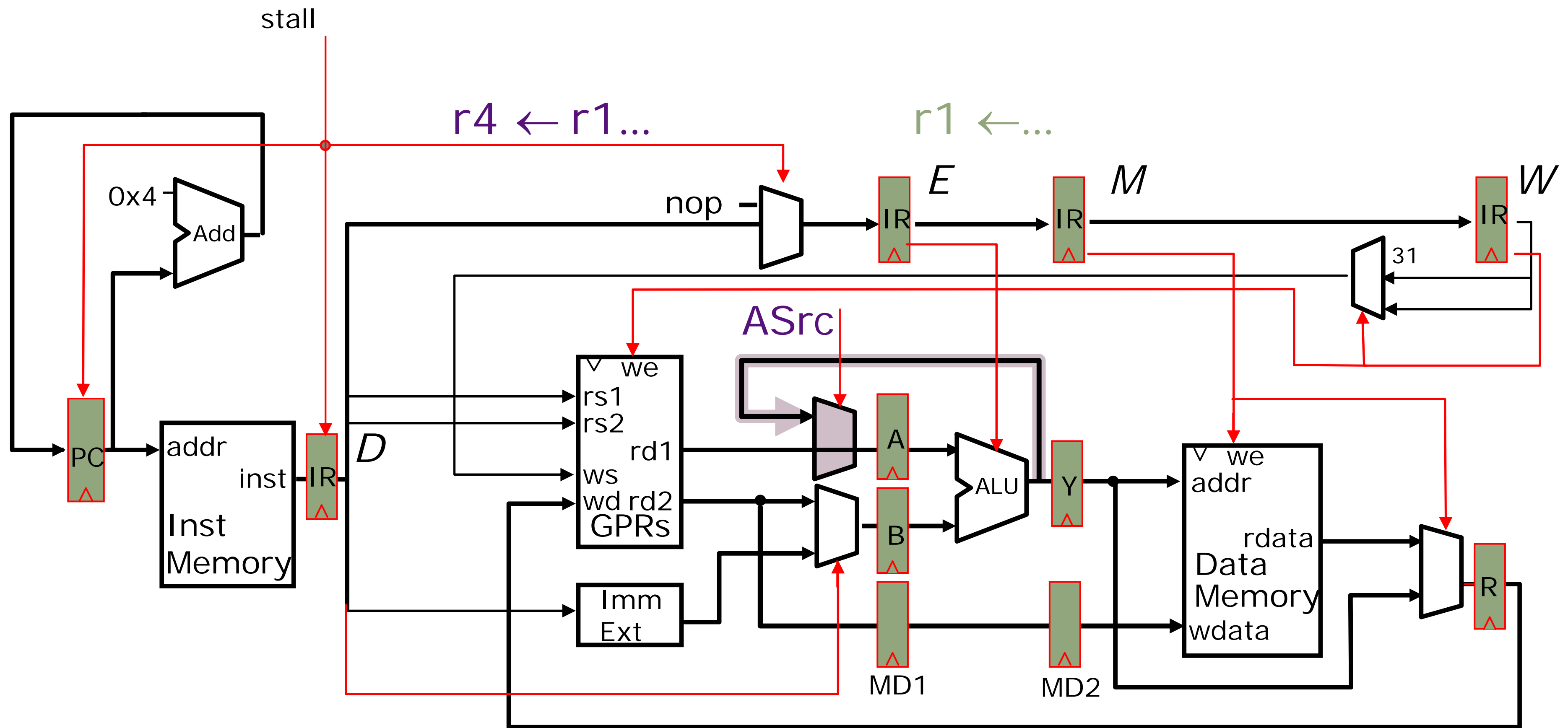
Each *stall or kill* introduces a bubble in the pipeline

$\Rightarrow$ *CPI > 1*

A new datapath, i.e., *a bypass*, can get the data from the output of the ALU to its input



Adding a Bypass



When does this bypass help?

...

(I_1) r1 ← r0 + 10
(I_2) r4 ← r1 + 17
yes

r1 ← M[r0 + 10]
r4 ← r1 + 17
no

JAL 500
r4 ← r31 + 17
no

CSAIL

The Bypass Signal

Deriving it from the Stall Signal

stall = (~~$((rs_D = ws_E).we_E$~~ + $(rs_D = ws_M).we_M$ + $(rs_D = ws_W).we_W).re1_D$

+$((rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W).re2_D$)

ws = *Case* opcode
- ALU ⇒ rd
- ALUi, LW ⇒ rt
- JAL, JALR ⇒ R31

we = *Case* opcode
- ALU, ALUi, LW ⇒ $(ws \neq 0)$
- JAL, JALR ⇒ on
- ... ⇒ off

$ASrc = (rs_D = ws_E).we_E.re1_D$

Is this correct?

No because only ALU and ALUi instructions can benefit from this bypass

Split we_E into two components: we-bypass, we-stall

CSAIL

Bypass and Stall Signals

Split we_E into two components: we-bypass, we-stall

$$\text{we-bypass}_E = \textit{Case}\ \text{opcode}_E$$
$$\text{ALU, ALUi} \Rightarrow (ws \neq 0)$$
$$\ldots \Rightarrow \text{off}$$

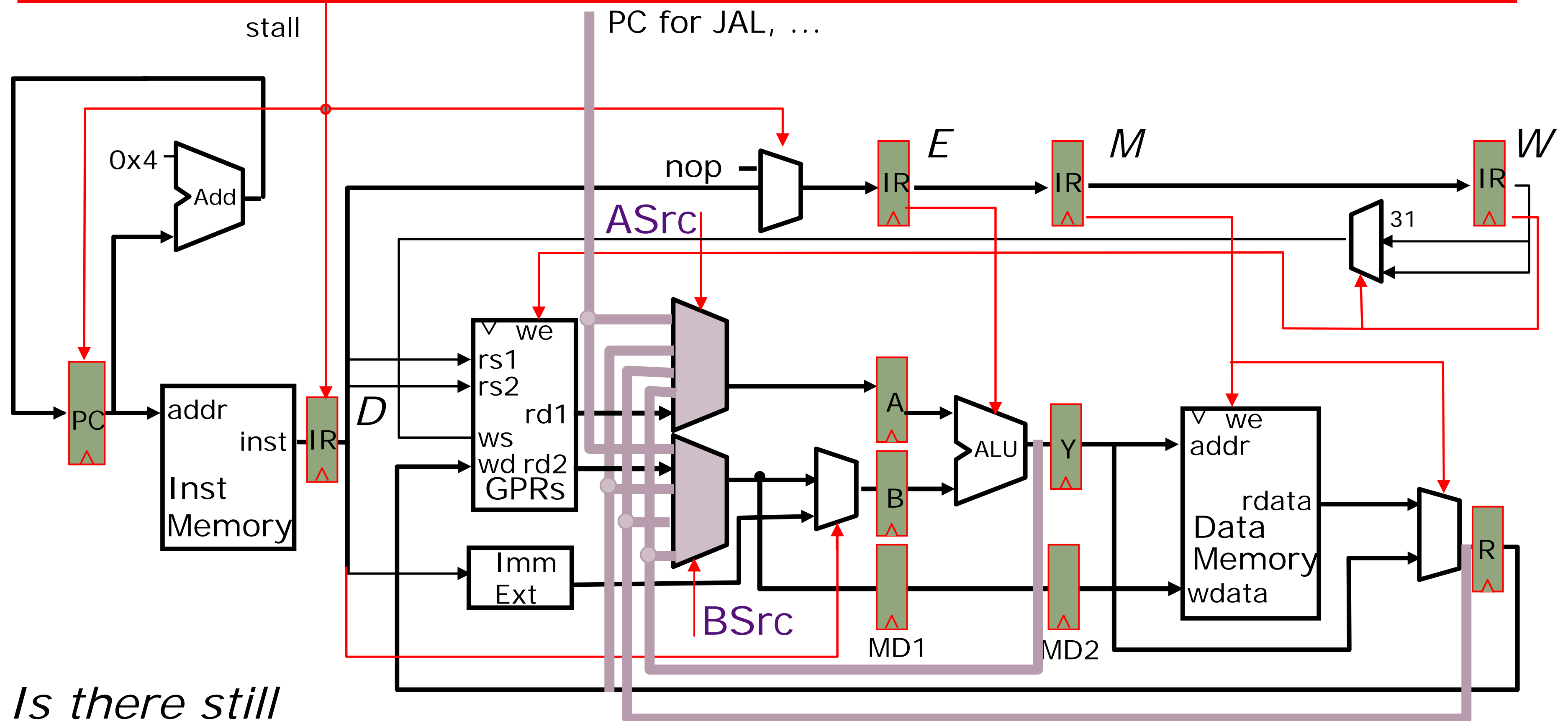
$$\text{we-stall}_E = \textit{Case}\ \text{opcode}_E$$
$$\text{LW} \Rightarrow (ws \neq 0)$$
$$\text{JAL, JALR} \Rightarrow \text{on}$$
$$\ldots \Rightarrow \text{off}$$

$$\text{ASrc} = (rs_D = ws_E).\text{we-bypass}_E \;.\; re1_D$$

$$\text{stall} = ((rs_D = ws_E).\text{we-stall}_E + (rs_D = ws_M).we_M + (rs_D = ws_W).we_W).\ re1_D + ((rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W).\ re2_D$$

CSAIL

Fully Bypassed Datapath



Is there still a need for the stall signal ?

$$\text{stall} = (rs_D = ws_E).(opcode_E = LW_E).(ws_E \neq 0).re1_D + (rt_D = ws_E).(opcode_E = LW_E).(ws_E \neq 0).re2_D$$

Why an Instruction may not be dispatched every cycle (CPI>1)

- Full bypassing may be too expensive to implement
  - typically all frequently used paths are provided
  - some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads have two cycle latency
  - Instruction after load cannot use load result
  - MIPS-I ISA defined *load delay slots*, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II.
- Conditional branches may cause bubbles
  - kill following instruction(s) if no delay slots

Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler.

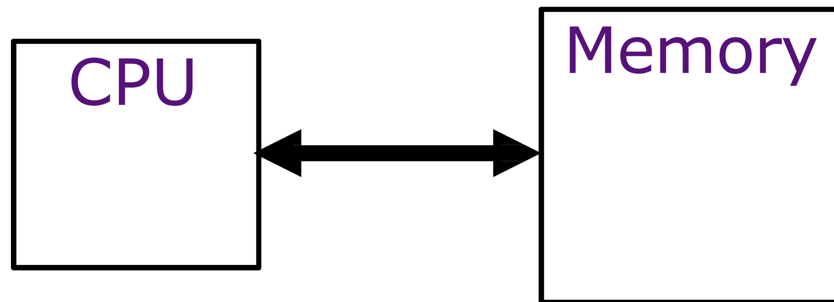
Thank you !

Multilevel Memories

Dheya Mustafa

Based on the material prepared by
Krste Asanovic and Arvind

CPU-Memory Bottleneck



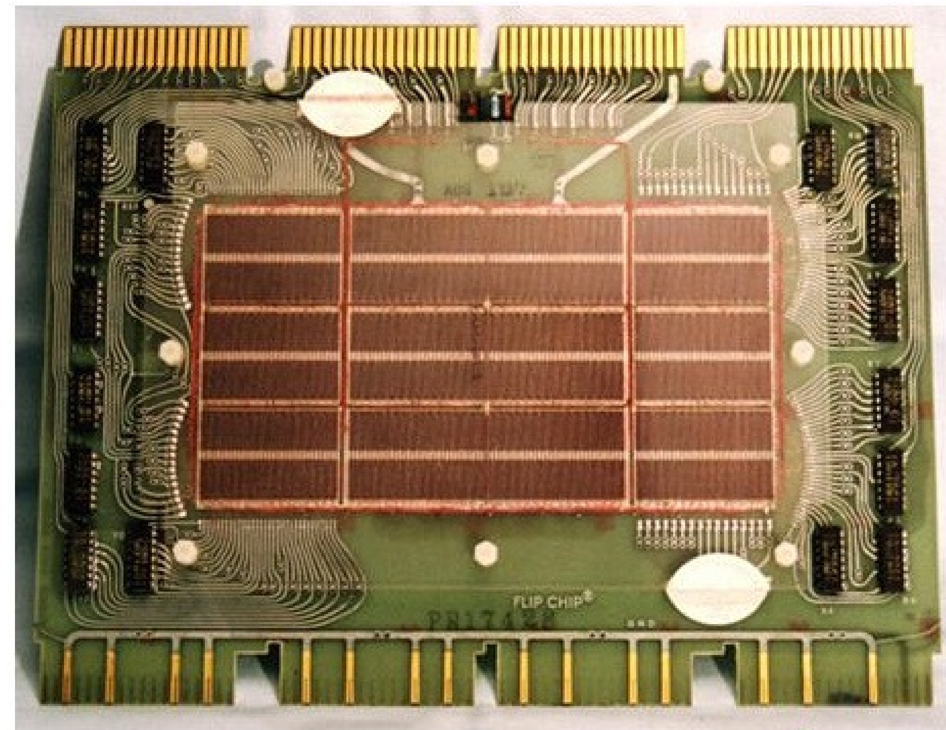
Performance of high-speed computers is usually limited by memory *bandwidth* &*latency*

- Latency)time for a single access(
  Memory access time >> Processor cycle time

- Bandwidth)number of accesses per unit time(
  if fraction *m* of instructions access memory,
  ⇒1+*m* memory references / instruction
  ⇒ CPI = 1 requires 1+*m* memory refs / cycle

Core Memory

- Core memory was first large scale reliable main memory
  - invented by Forrester in late 40s at MIT for Whirlwind project
- Bits stored as magnetization polarity on small ferrite cores threaded onto 2dimensional grid of wires
- Coincident current pulses on X and Y wires would write cell and also sense original state)destructive reads(
- Robust, non-volatile storage
- Used on space shuttle computers until recently
- Cores threaded onto wires by hand (25 billion a year at peak production)
- Core access time ~ 1μs

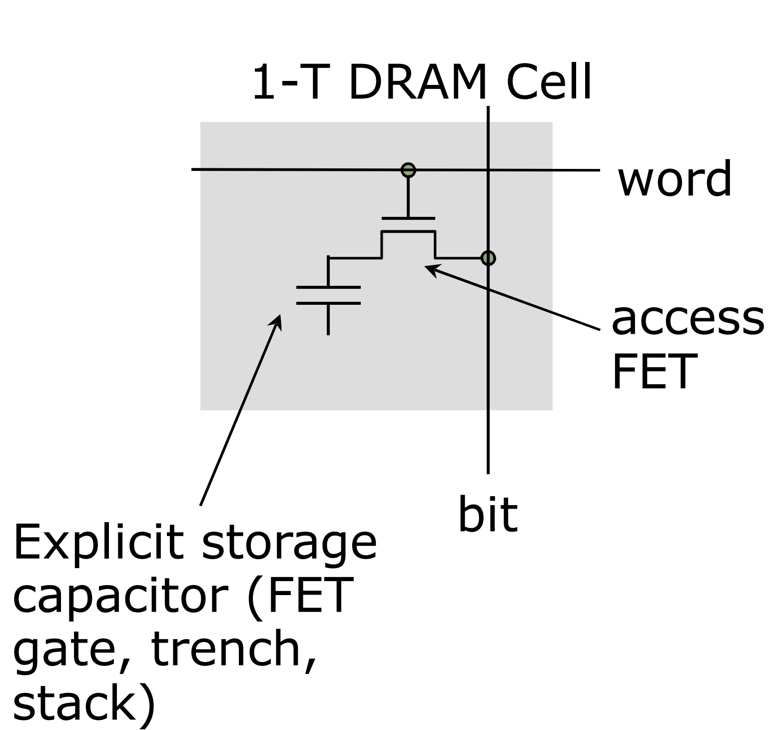


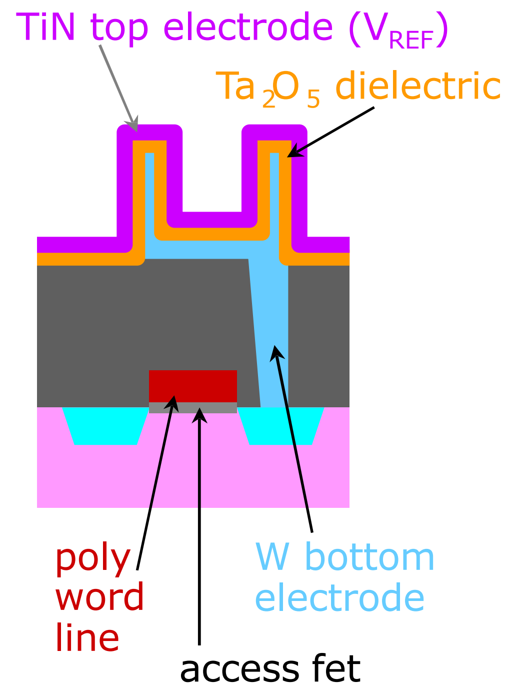
DEC PDP-8/E Board,
4K words x 12bits, ((1968

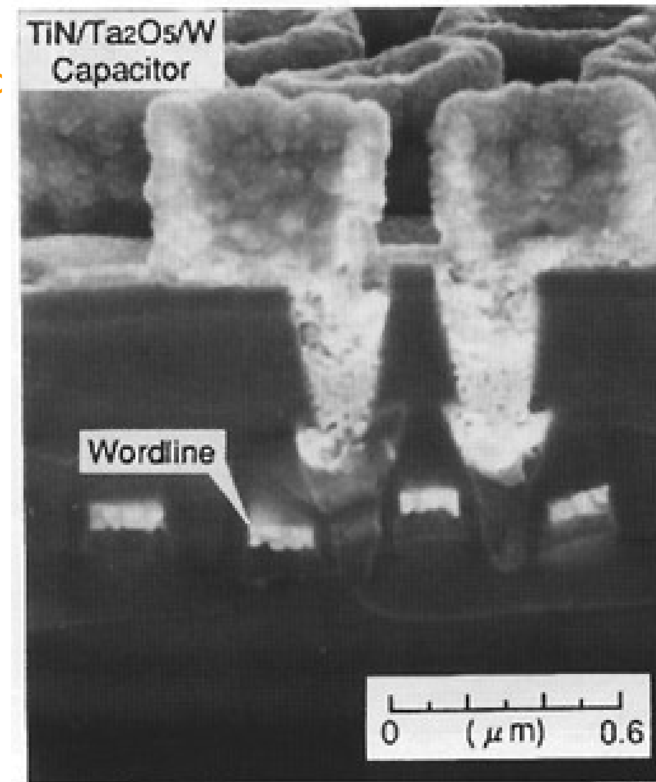
Semiconductor Memory, DRAM

- Semiconductor memory began to be competitive in early 1970s
  - Intel formed to exploit market for semiconductor memory
- First commercial DRAM was Intel 1103
  - 1Kbit of storage on single chip
  - charge on a capacitor used to hold value
- Semiconductor memory quickly replaced core in 1970s

One Transistor Dynamic RAM

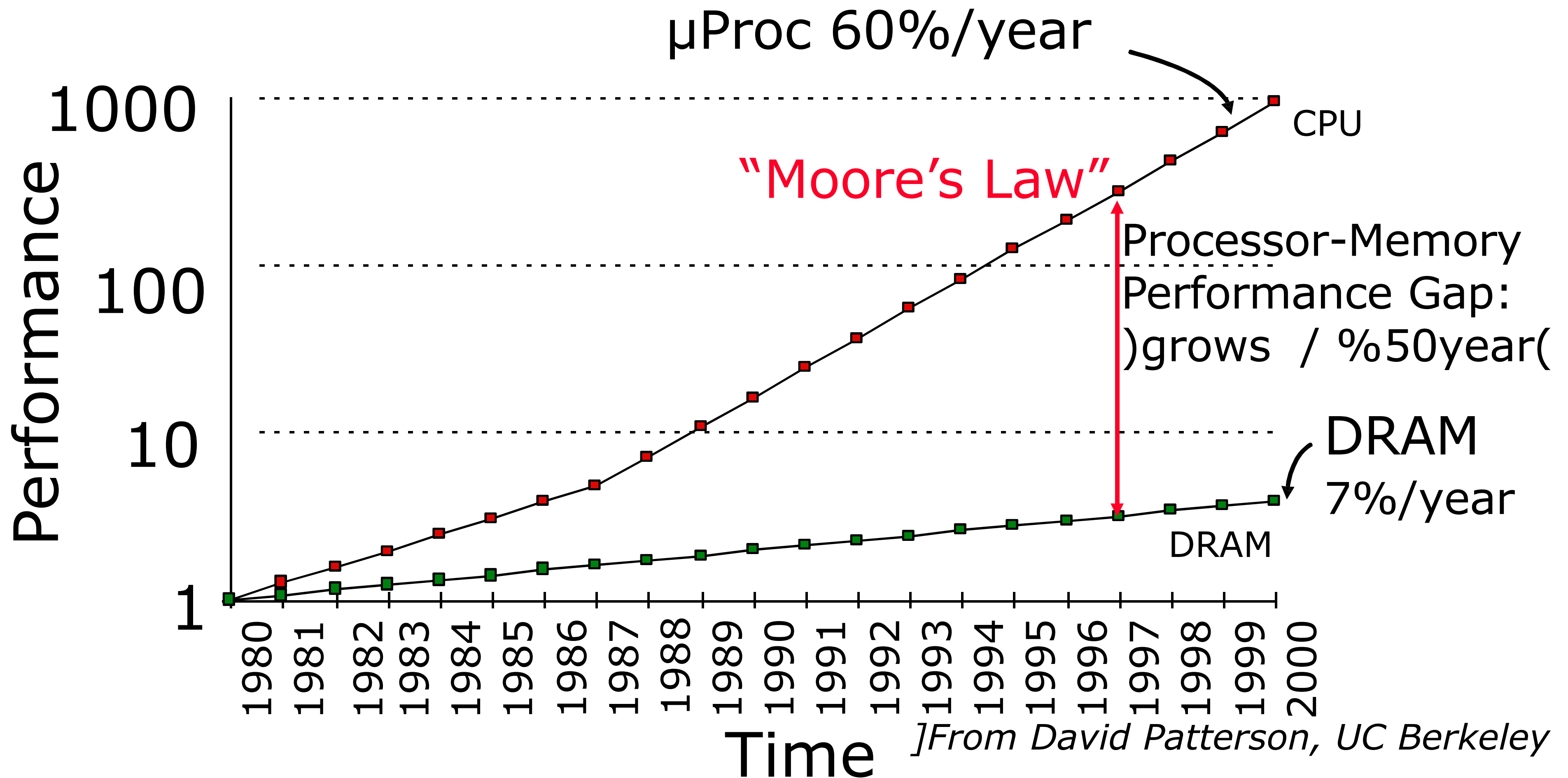






TiN/Ta2O5/W Capacitor

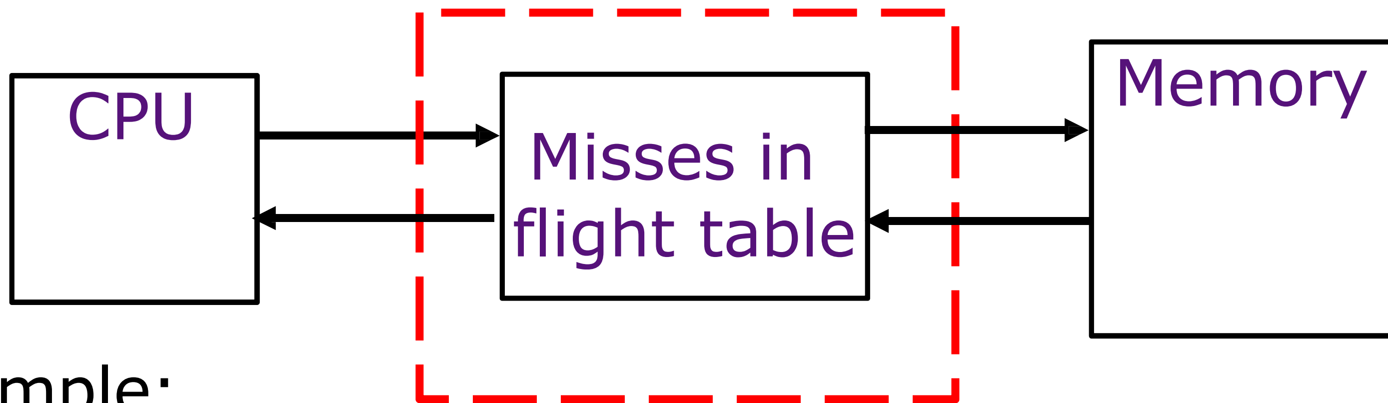
Processor-DRAM Gap (latency(



Four-issue superscalar could execute 800 instructions during cache miss!

Little's Law

Throughput (T) = Number in Flight (N) / Latency (L)



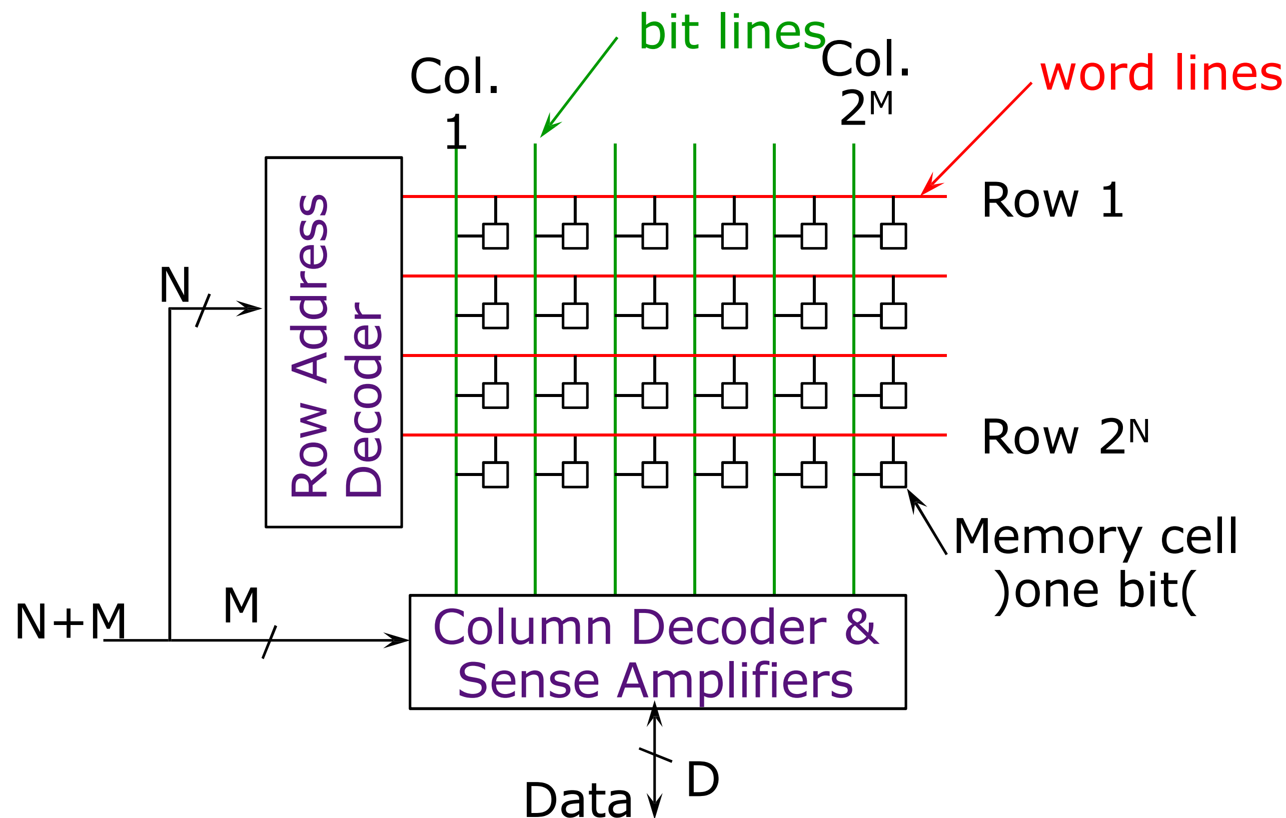
Example:

---Assume infinite bandwidth memory
100 ---cycles /memory reference
0.2 + 1 ---memory references /instruction

*⇒ Table size= 100 * 1.2 = 120 entries*

120 independent memory operations in flight!

DRAM Architecture



- Bits stored in -2dimensional arrays on chip
- Modern chips have around 4logical banks on each chip
  - –each logical bank physically implemented as many smaller arrays

DRAM Operation

Three steps in read/write access to a given bank

- Row access (RAS)
  - decode row address, enable addressed row (often multiple Kb in row)
  - bitlines share charge with storage cell
  - small change in voltage detected by sense amplifiers which latch whole row of bits
  - sense amplifiers drive bitlines full rail to recharge storage cells
- Column access (CAS)
  - decode column address to select small number of sense amplifier latches (4, 8, 16, or 32 bits depending on DRAM package)
  - on read, send latched bits out to chip pins
  - on write, change sense amplifier latches which then charge storage cells to required value
  - can perform multiple column accesses on same row without another row access (burst mode)
- Precharge
  - charges bit lines to known value, required before next row access

Each step has a latency of around 20ns in modern DRAMs

Various DRAM standards (DDR, RDRAM) have different ways of encoding the signals for transmission to the DRAM, but all share the same core architecture

Multilevel Memory

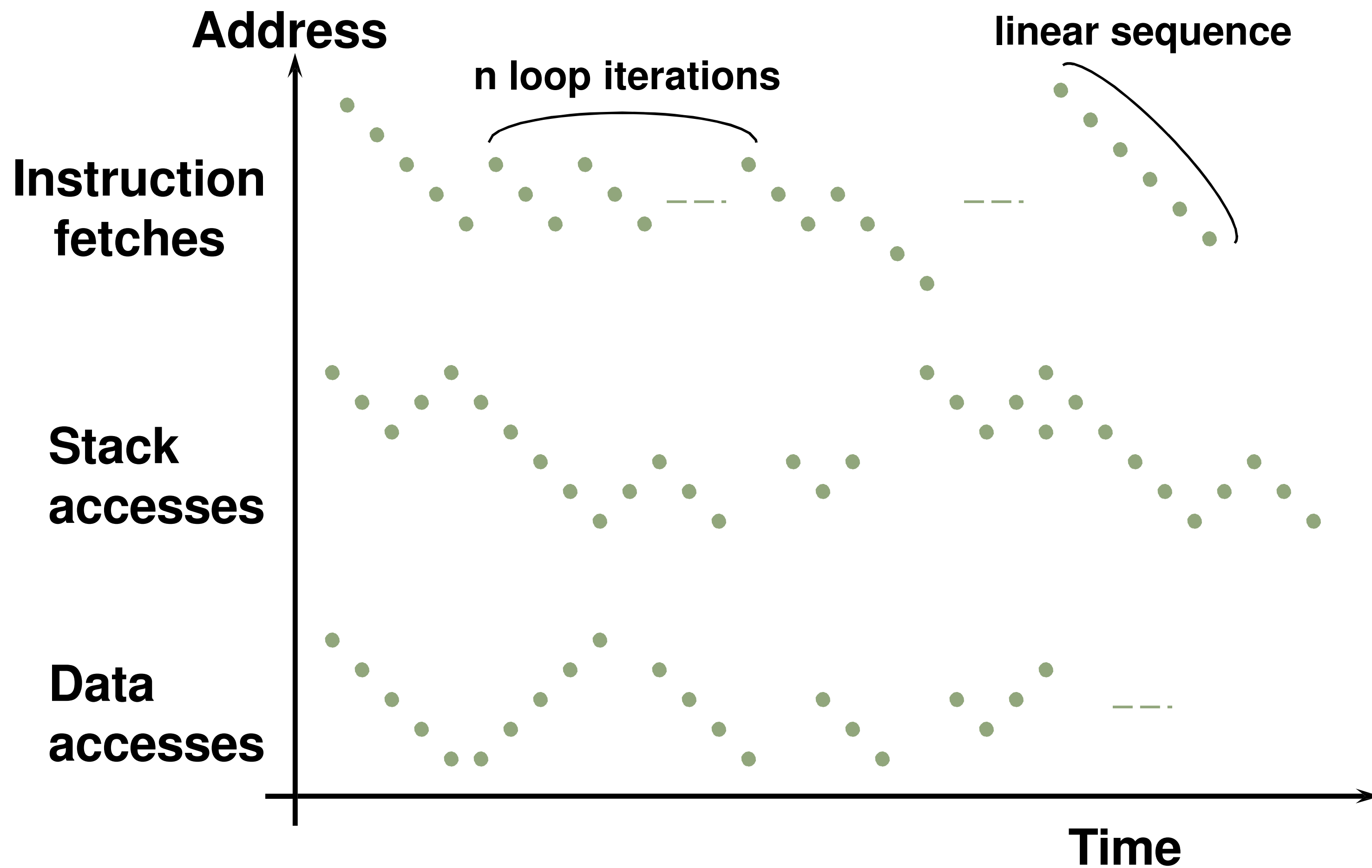
Strategy: <u>Hide</u> latency using small, fast memories called caches.

Caches are a mechanism to hide memory latency based on the empirical observation that the patterns of memory references made by a processor are often highly predictable:

	PC
...	*96*
loop: ADD r2, r1, r1	*100*
SUBI r3, r3, #1	*104*
BNEZ r3, loop	*108*
...	*112*

What is the pattern of instruction memory addresses?

Typical Memory Reference Patterns



Common Predictable Patterns

Two predictable properties of memory references:

- Temporal Locality: If a location is referenced it is likely to be referenced again in the near future.

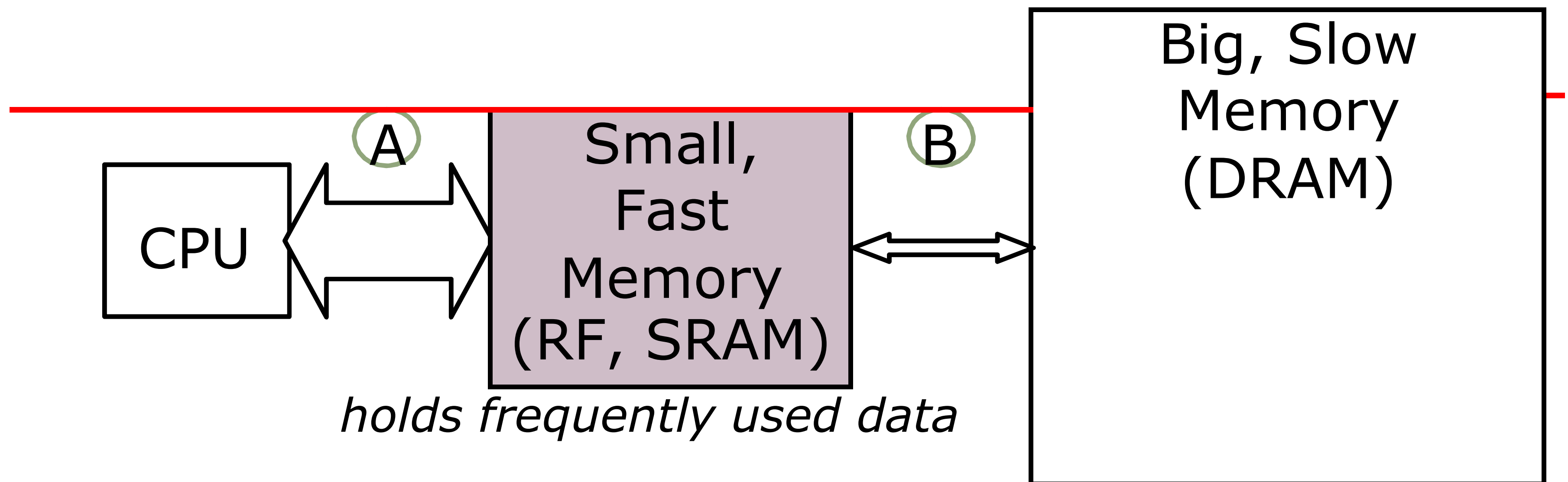
- Spatial Locality: If a location is referenced it is likely that locations near it will be referenced in the near future.

Caches

Caches exploit both types of predictability:

- Exploit temporal locality by remembering the contents of recently accessed locations.
- Exploit spatial locality by fetching blocks of data around recently accessed locations.

Memory Hierarchy



- *size*: Register << SRAM << DRAM *why?*
- *latency:* Register << SRAM << DRAM *why?*
- *bandwidth:* on-chip >>off-chip *why?*

On a data access:

hit (data ∈ fast memory) ⇒ low latency access

miss)data ∉ fast memory) ⇒ long latency access *(DRAM(*

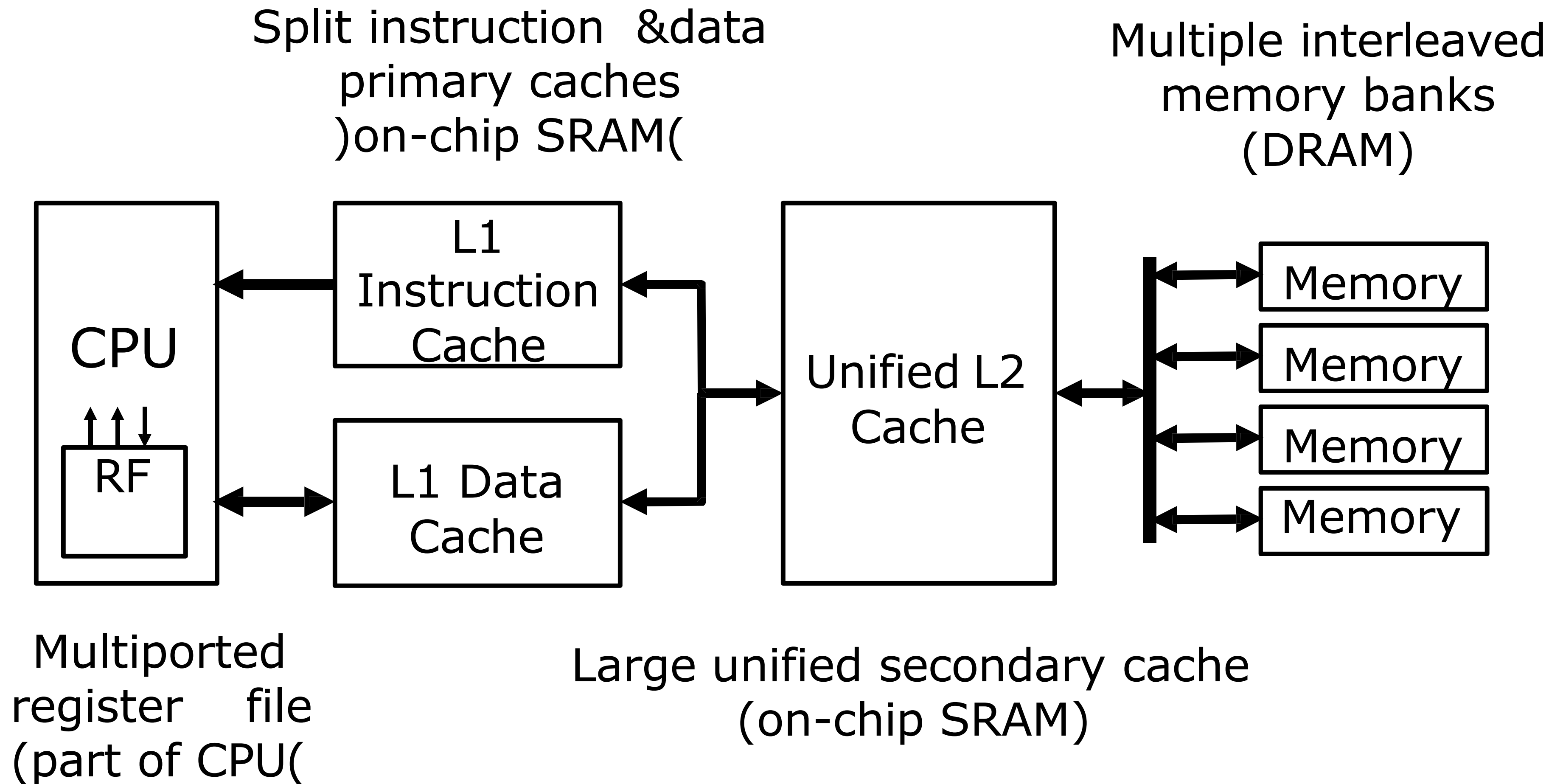
Fast mem. effective only if bandwidth requirement at B >>A

Management of Memory Hierarchy

- Small/fast storage, e.g., registers
  - Address usually specified in instruction
  - Generally implemented directly as a register file
    - but hardware might do things behind software's back, e.g., stack management, register renaming

- Large/slower storage, e.g., memory
  - Address usually computed from values in register
  - Generally implemented as a cache hierarchy
    - hardware decides what is kept in fast memory
    - but software may provide "hints", e.g., don't cache or prefetch

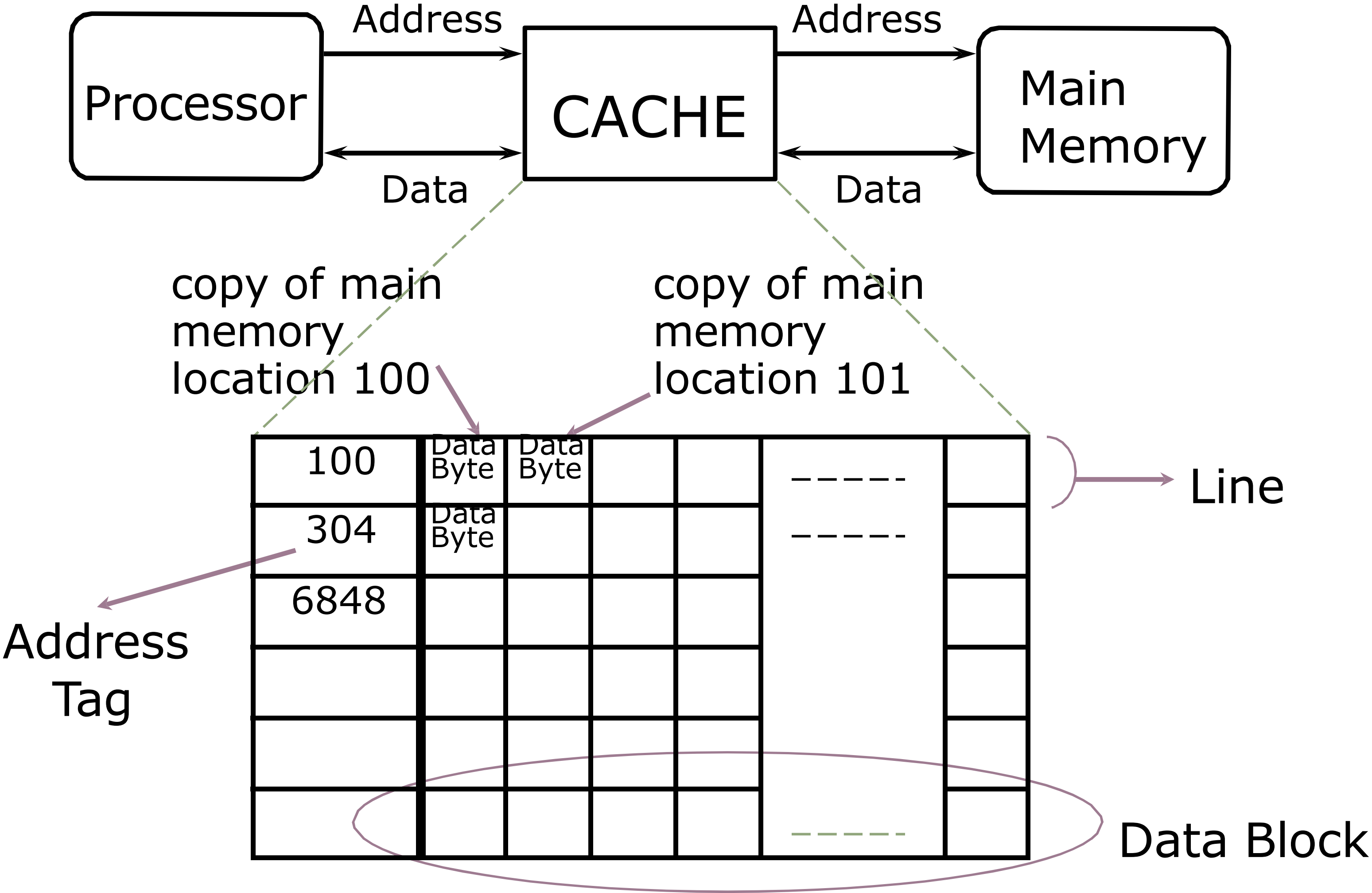
A Typical Memory Hierarchy c.2003



Workstation Memory System
)Apple PowerMac G5, (2003

- Dual 2GHz processors, each with 64KB I- cache, 32KB D-cache, and 512KB L2 unified cache
- 1GB/s1GHz, 2x32-bit bus, 16GB/s
- North Bridge Chip
- Up to 8GB DRAM, 400MHz, -128bit bus, 6.4GB/s
- AGP Graphics Card, 533MHz, 32-bit bus, .2
- PCI-X Expansion, 133MHz, -64bit bus, 1 GB/s

Inside a Cache



Cache Algorithm (Read)

Look at Processor Address, search cache tags to find match. Then either

Found in cache
a.k.a. HIT

Return copy of data from cache

Not in cache
a.k.a. MISS

Read block of data from Main Memory

Wait …

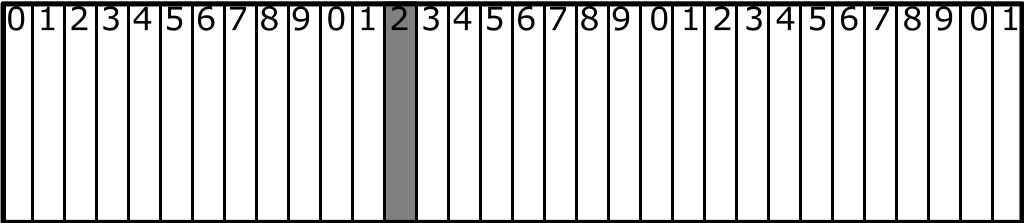
Return data to processor and update cache

Q: Which line do we replace?

Placement Policy

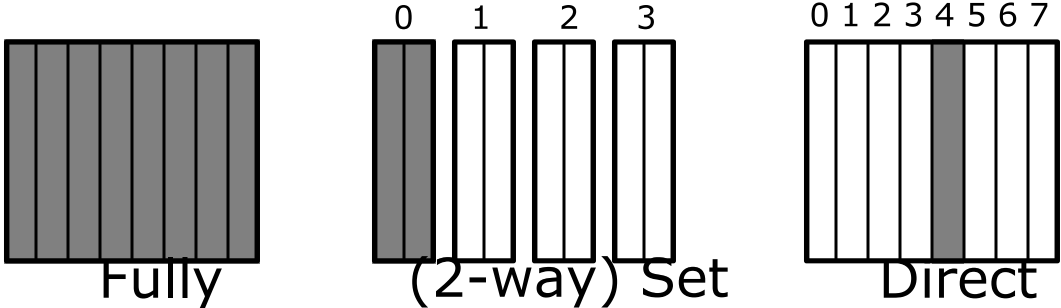
Block Number

Memory



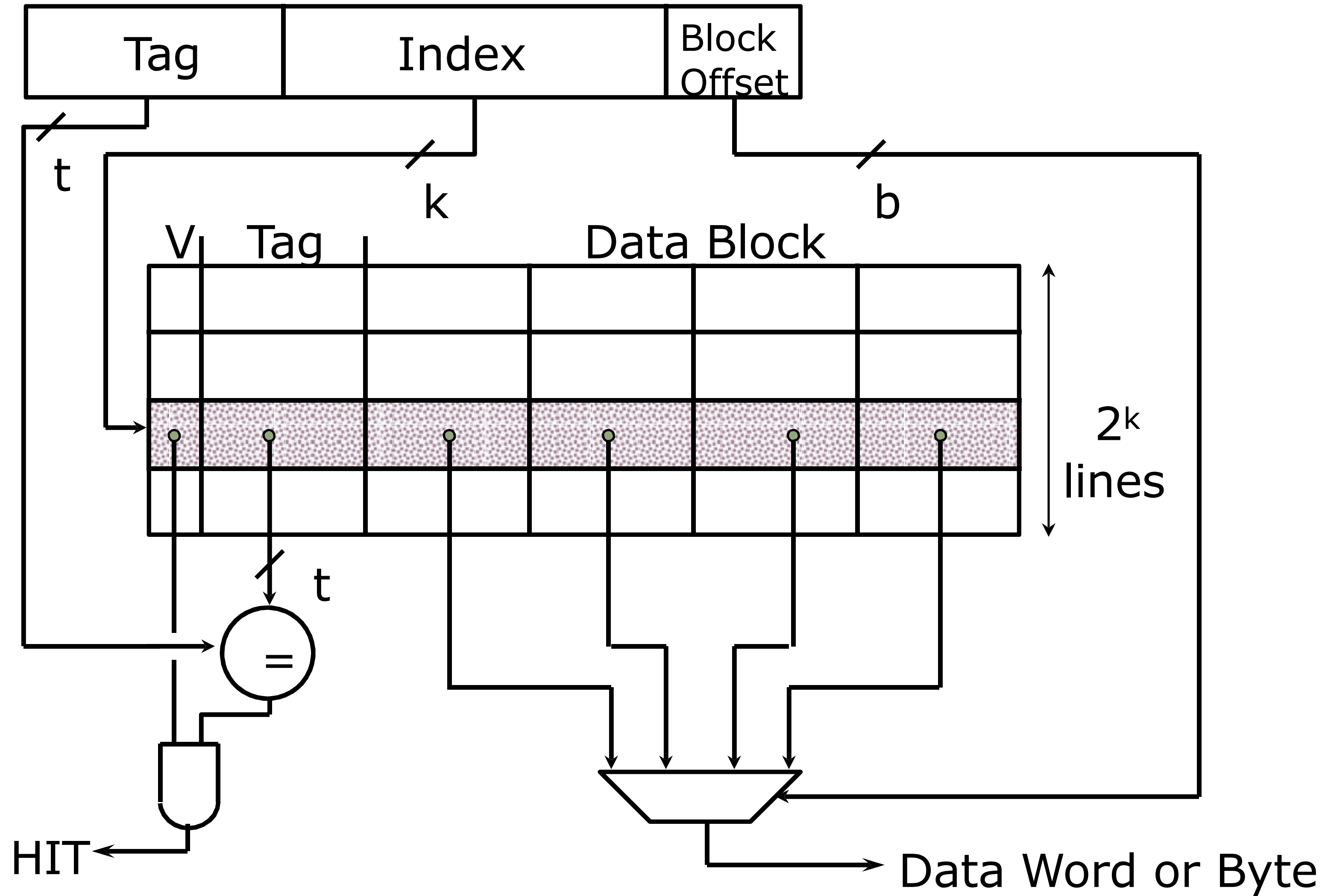
Set Number

Cache



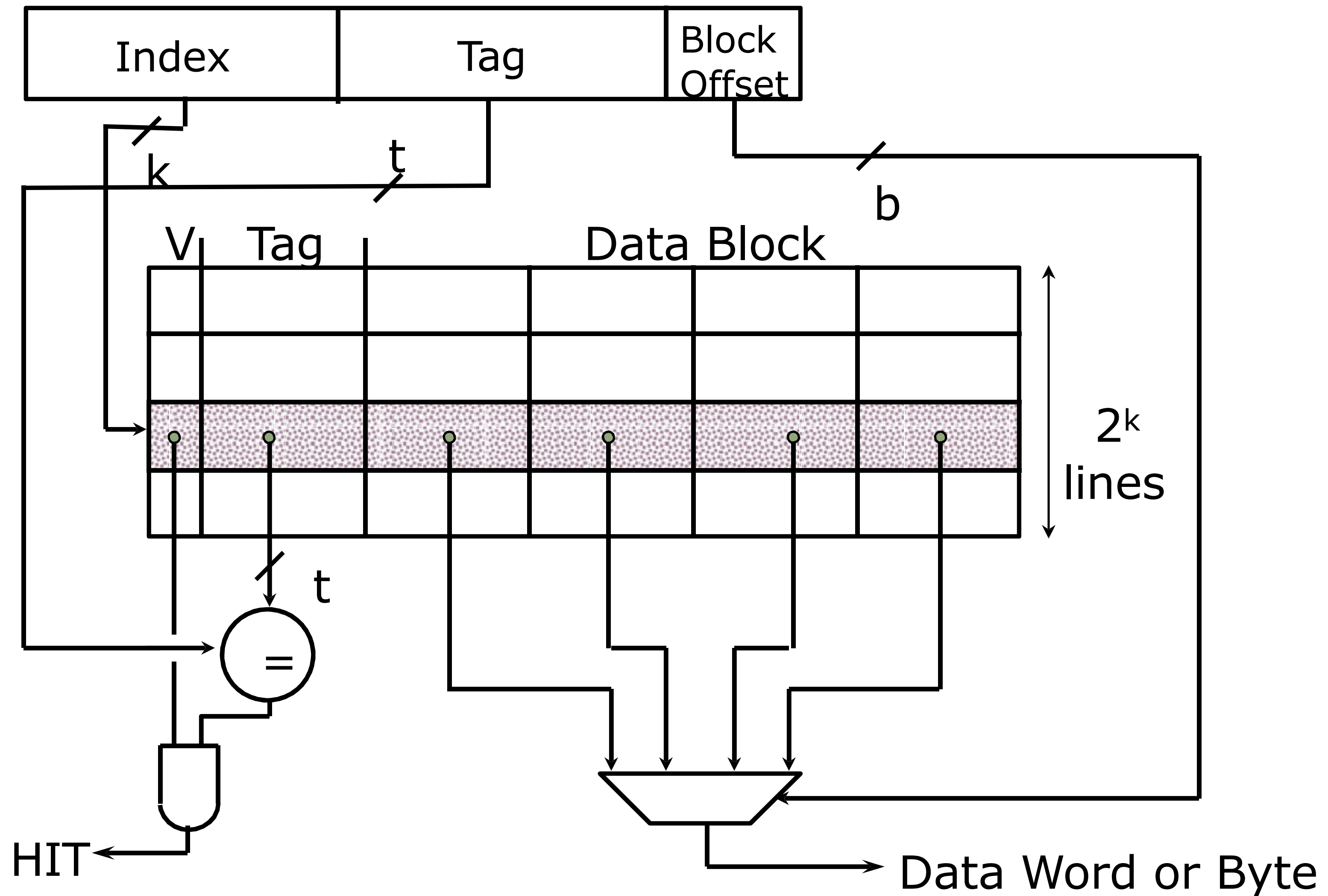
	Fully Associative	(2-way) Set Associative	Direct Mapped
block 12 can be placed	anywhere	anywhere in set 0 *(12 mod 4)*	only into block 4 *12)mod (8*

Direct-Mapped Cache

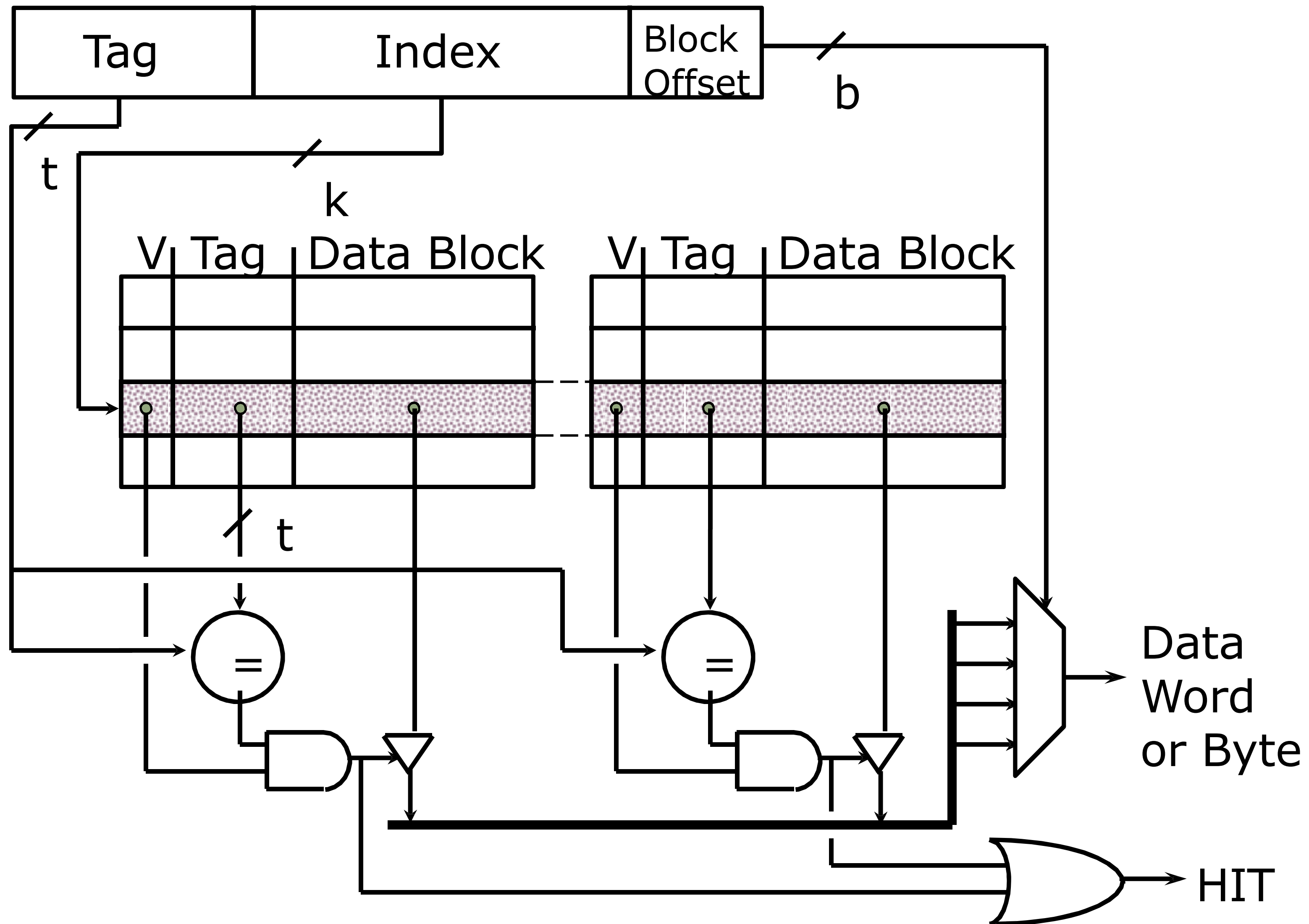


Direct Map Address Selection

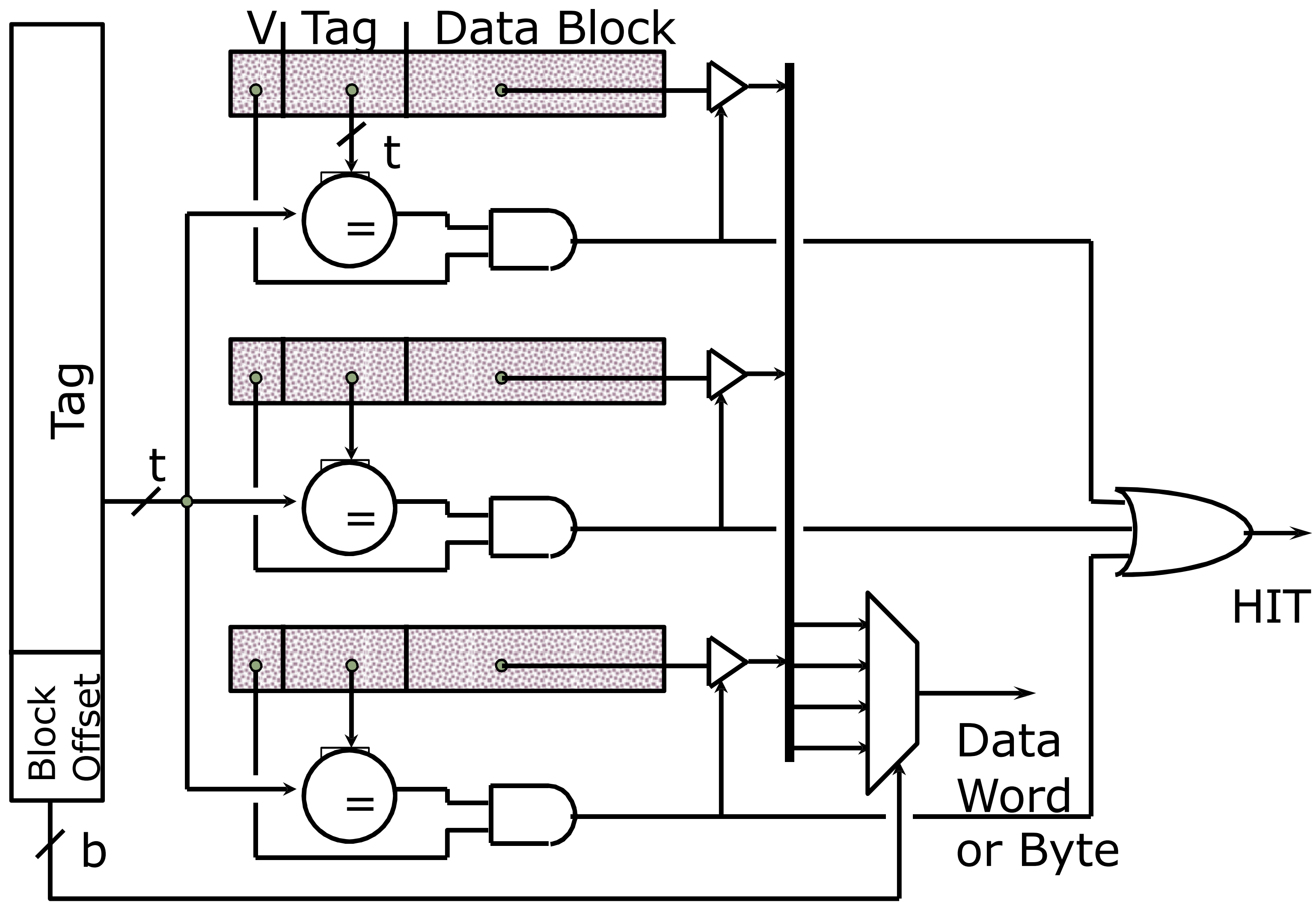
higher-order vs. lower-order address bits



2-Way Set-Associative Cache



Fully Associative Cache



Replacement Policy

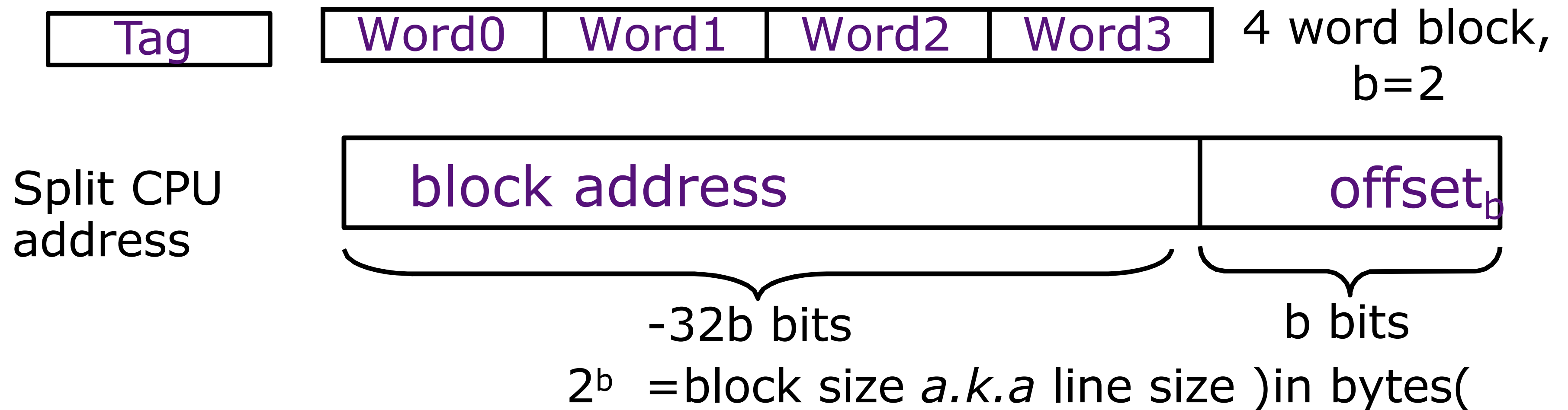
In an associative cache, which block from a set should be evicted when the set becomes full?

- Random
- Least Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way)
  - pseudo-LRU binary tree often used for 4-8 way
- First In, First Out (FIFO) a.k.a. Round-Robin
  - used in highly associative caches
- Not Least Recently Used (NLRU)
  - FIFO with exception for most recently used block

This is a second-order effect. Why?

Block Size and Spatial Locality

Block is unit of transfer between the cache and memory



Larger block size has distinct hardware advantages

- less tag overhead
- exploit fast burst transfers from DRAM
- exploit fast burst transfers over wide busses

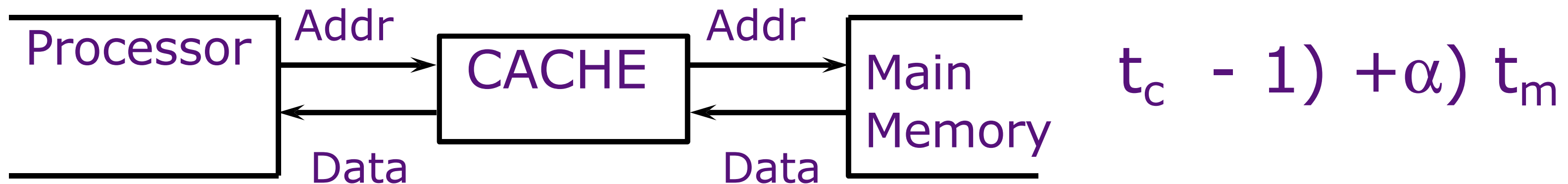
What are the disadvantages of increasing block size?

Average Cache Read Latency

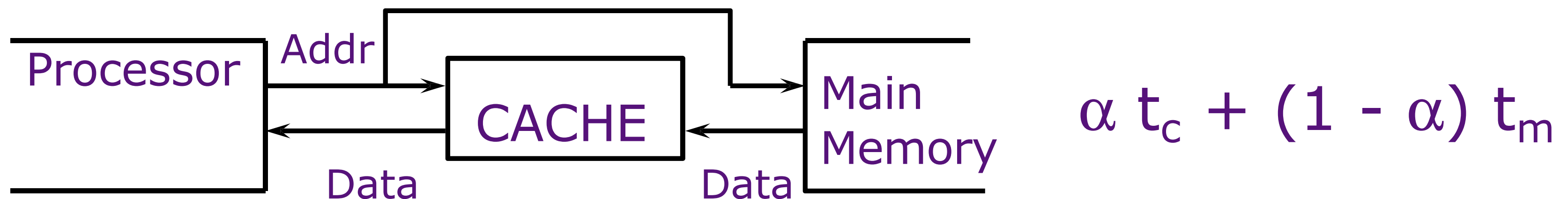
α is HIT RATIO: Fraction of references in cache

- 1α is MISS RATIO: Remaining references

Average access time for serial search:



Average access time for parallel search:



t_c *is smallest for which type of cache?*

Improving Cache Performance

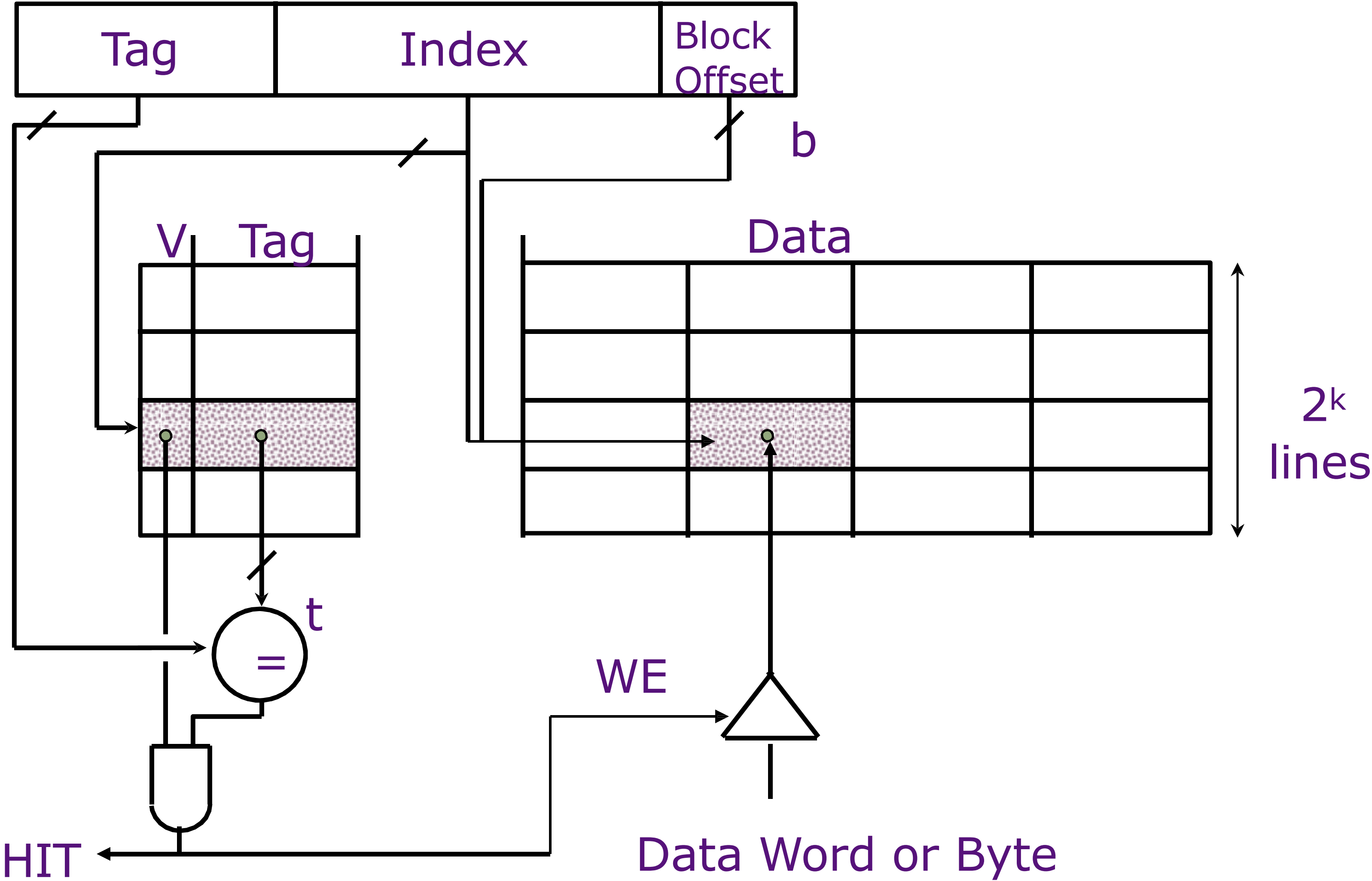
Average memory access time=
Hit time +Miss rate x Miss penalty

To improve performance:

- reduce the miss rate)e.g., larger cache(
- reduce the miss penalty)e.g., L2 cache(
- reduce the hit time

What is the simplest design strategy?

Write Performance



Write Policy

- Cache hit:
  - write through*:* write both cache & memory
    - generally higher traffic but simplifies cache coherence
  - write back*:* write cache only
)memory is written only when the entry is evicted(
    - a dirty bit per block can further reduce the traffic
- Cache miss:
  - no write allocate*:* only write to main memory
  - write allocate *(aka fetch on write):* fetch into cache

- Common combinations:
  - write through and no write allocate
  - write back with write allocate

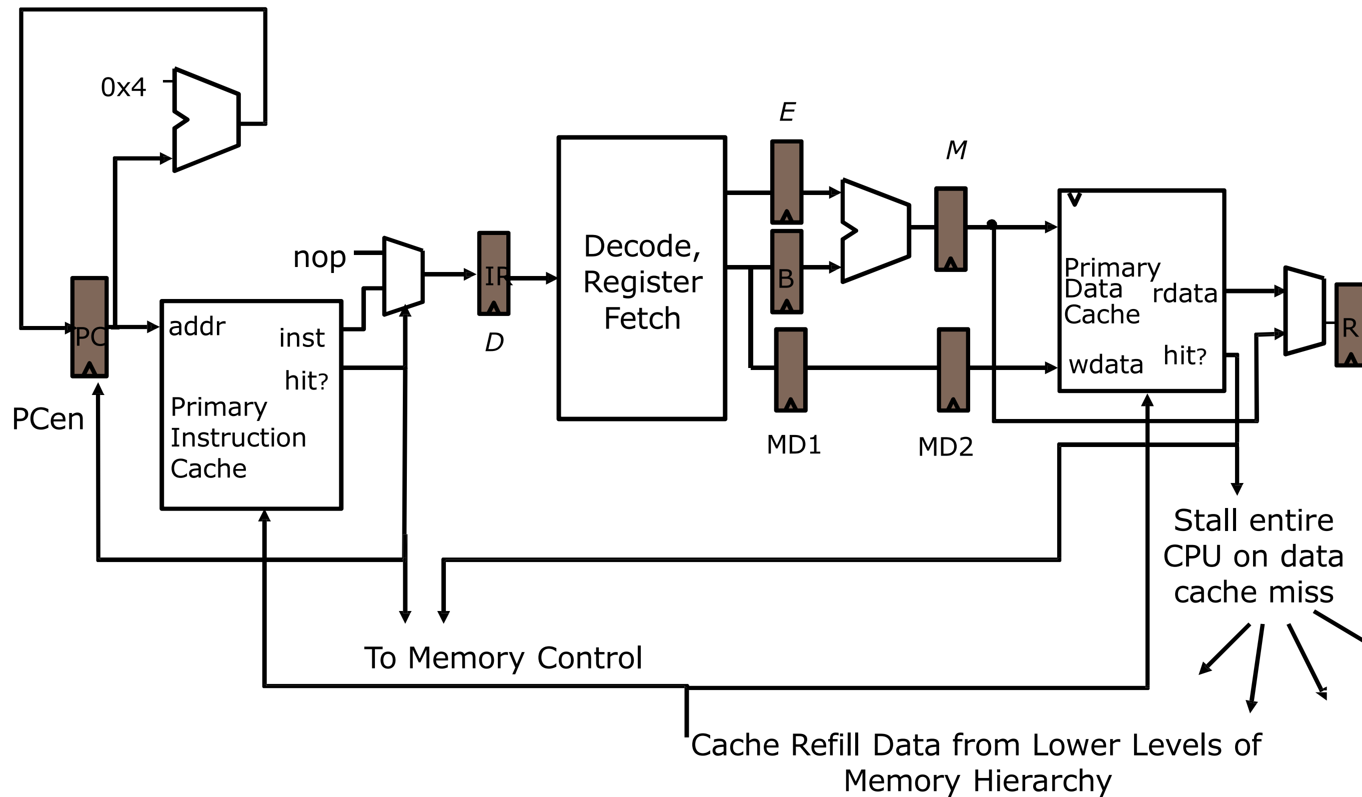
1

Cache Optimizations

Dheya Mustafa

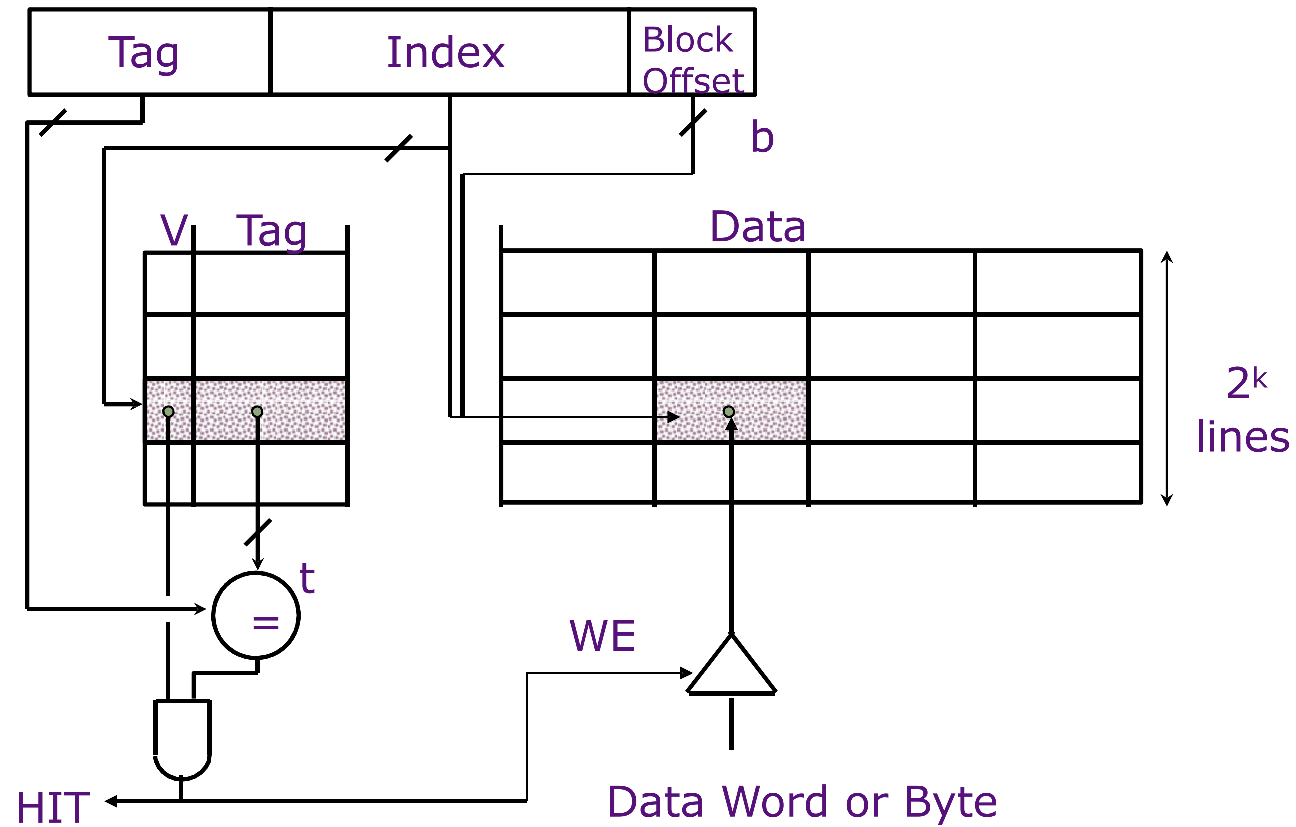
Based on the material prepared by Krste Asanovic and Arvind

CPU-Cache Interaction
(5-stage pipeline)



What about Instruction miss or writes to i-stream ?

Write Performance



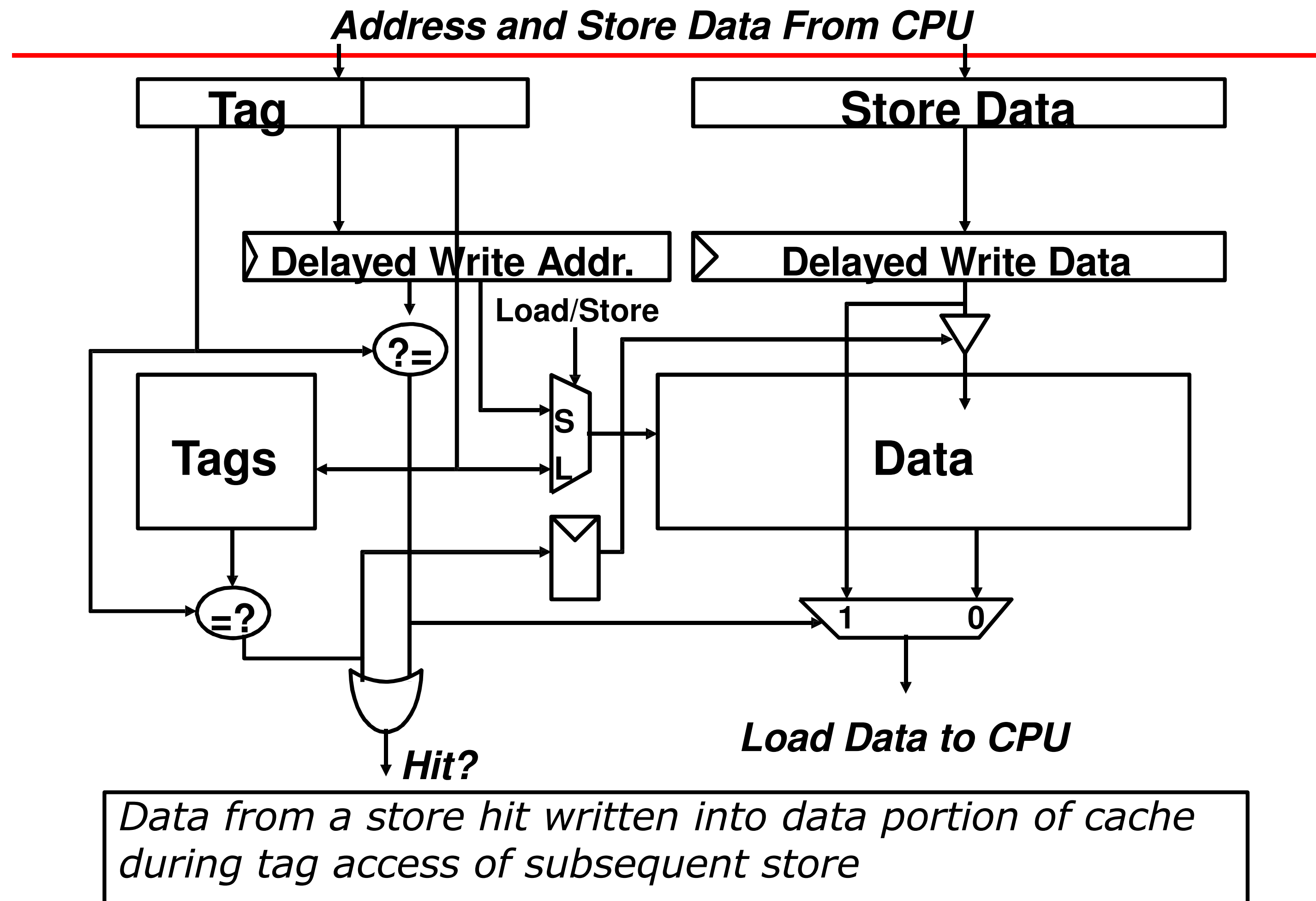
Reducing Write Hit Time

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

Solutions:

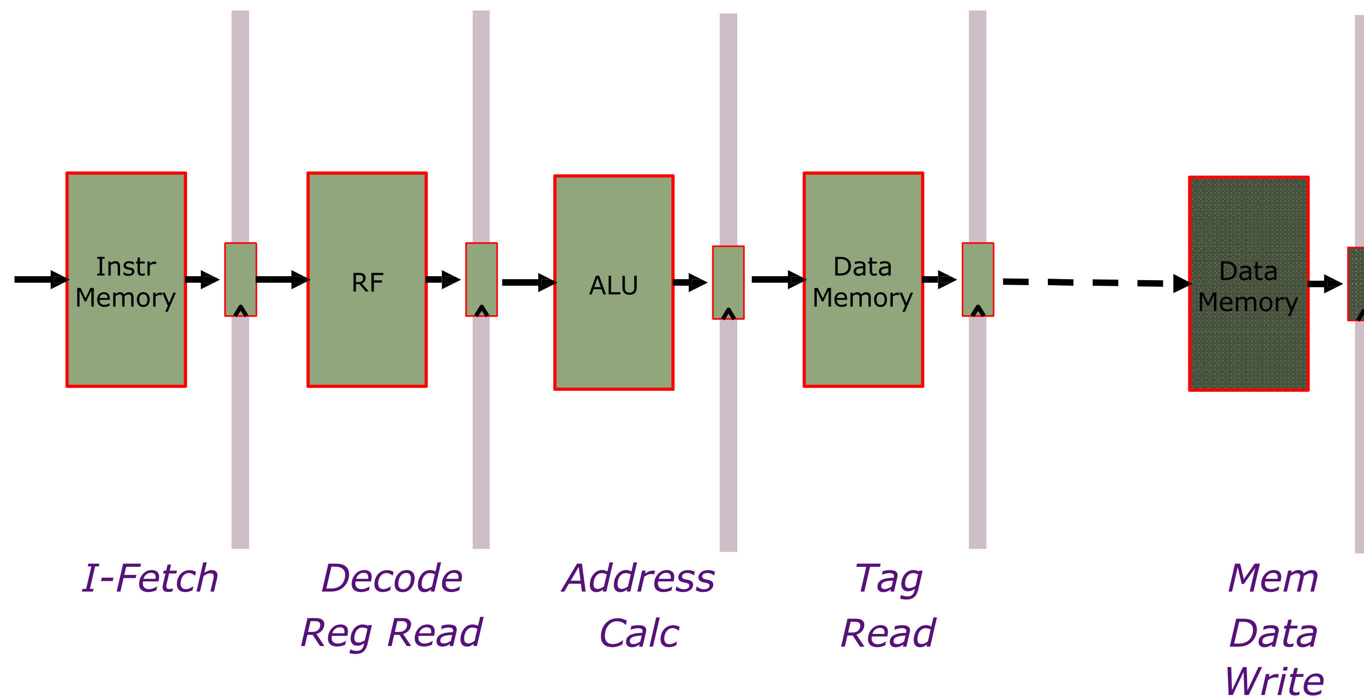
- Design data RAM that can perform read and write in one cycle, restore old value after tag miss
- CAM-Tag caches: Word line only enabled if hit
- Pipelined writes: Hold write data for store in single buffer ahead of cache, write cache data during next store's tag check

Pipelining Cache Writes



Data from a store hit written into data portion of cache during tag access of subsequent store

Write pipeline



What hazard has been introduced in this pipeline?

Write Policy

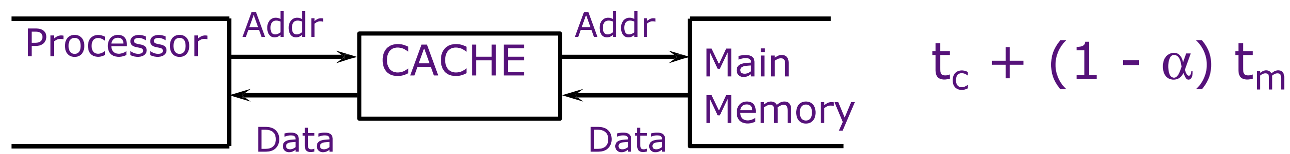
- Cache hit:
  - write through*:* write both cache & memory
    - generally higher traffic but simplifies cache coherence
  - write back*:* write cache only
)memory is written only when the entry is evicted(
    - a dirty bit per block can further reduce the traffic
- Cache miss:
  - no write allocate*:* only write to main memory
  - write allocate *(aka fetch on write):* fetch into cache
- Common combinations:
  - write through and no write allocate
  - write back with write allocate

Average Cache Read Latency

α is HIT RATIO: Fraction of references in cache

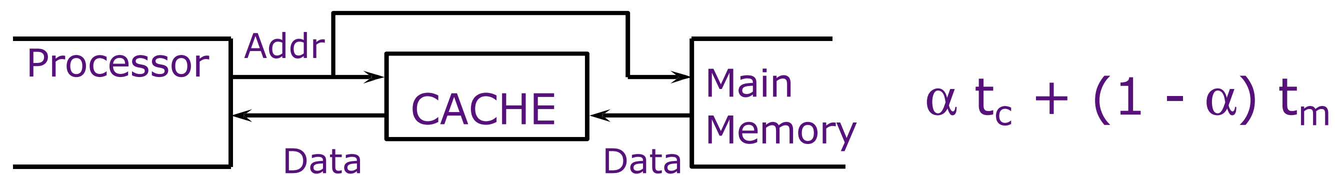
$1 - \alpha$ is MISS RATIO: Remaining references

Average access time for serial search:



$t_c + (1 - \alpha)\, t_m$

Average access time for parallel search:



$\alpha\, t_c + (1 - \alpha)\, t_m$

t_c *is smallest for which type of cache?*

Improving Cache Performance

Average memory access time =
Hit time + Miss rate x Miss penalty

To improve performance:

- reduce the miss rate (e.g., larger cache)
- reduce the miss penalty (e.g., L2 cache)
- reduce the hit time

What is the simplest design strategy?

Improving Cache Performance

Average memory access time =
Hit time + Miss rate x Miss penalty

To improve performance:

- reduce the miss rate (e.g., larger cache)
- reduce the miss penalty (e.g., L2 cache)
- reduce the hit time

The simplest design strategy is to design the largest primary cache without slowing down the clock or adding pipeline stages

(but design decisions are more complex with out-of-order or highly pipelined CPUs)

Causes for Cache Misses

- *Compulsory:* first-reference to a block *a.k.a.* cold start misses
  -misses that would occur even with infinite cache
- *Capacity:* cache is too small to hold all data needed by the program
  - misses that would occur even under perfect placement &replacement policy
- *Conflict:* misses that occur because of collisions due to block-placement strategy
  - misses that would not occur with full associativity

Effect of Cache Parameters on Performance

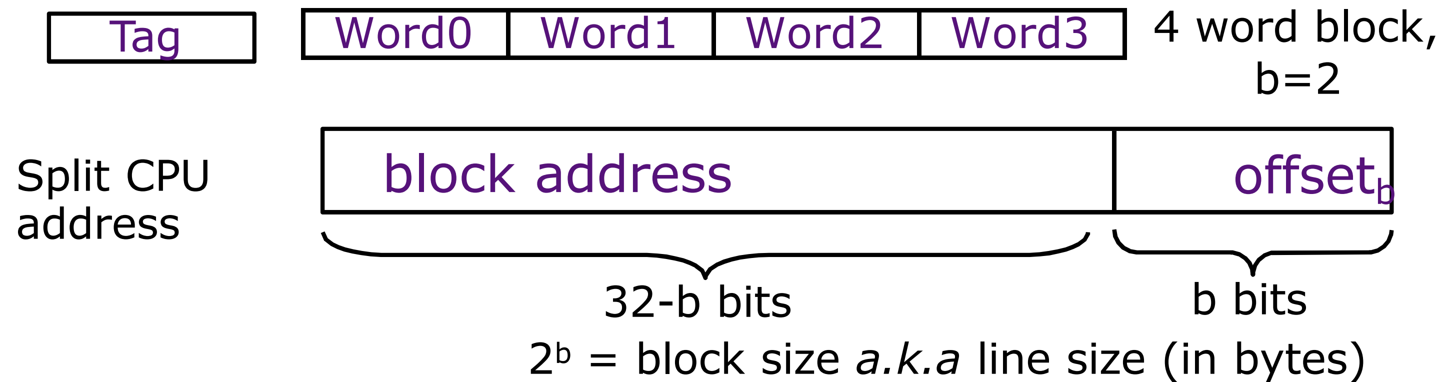
- Larger cache size
  + reduces capacity and conflict misses
  - hit time will increase

- Higher associativity
  + reduces conflict misses (up to around 4-8 way)
  - may increase access time

- Larger block size

Block Size and Spatial Locality

Block is unit of transfer between the cache and memory



2^b = block size *a.k.a* line size (in bytes)

Larger block size has distinct hardware advantages

- less tag overhead
- exploit fast burst transfers from DRAM
- exploit fast burst transfers over wide busses

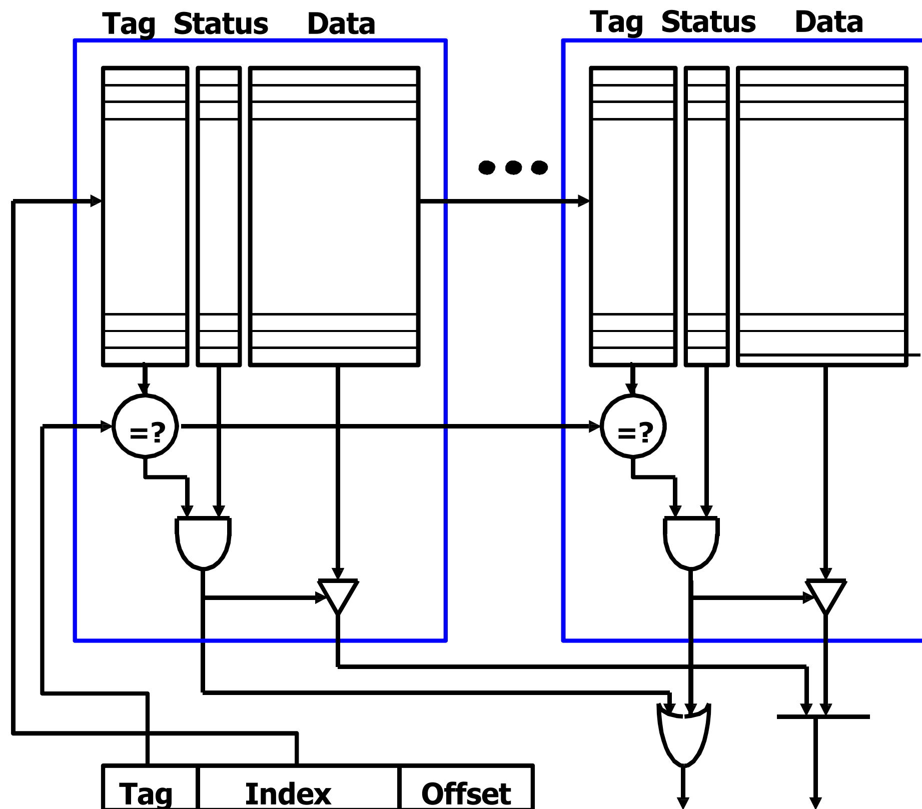
What are the disadvantages of increasing block size?

Block-level Optimizations

- Tags are too large, i.e., too much overhead
  - Simple solution: Larger blocks, but miss penalty could be large.
- Sub-block placement (aka sector cache)
  - A valid bit added to units smaller than the full block, called sub-blocks
  - Only read a sub-block on a miss
  - *If a tag matches, is the word in the cache?*

Tag								
100	1		1		1		1	
300	1		1		0		0	
204	0		1		0		1	

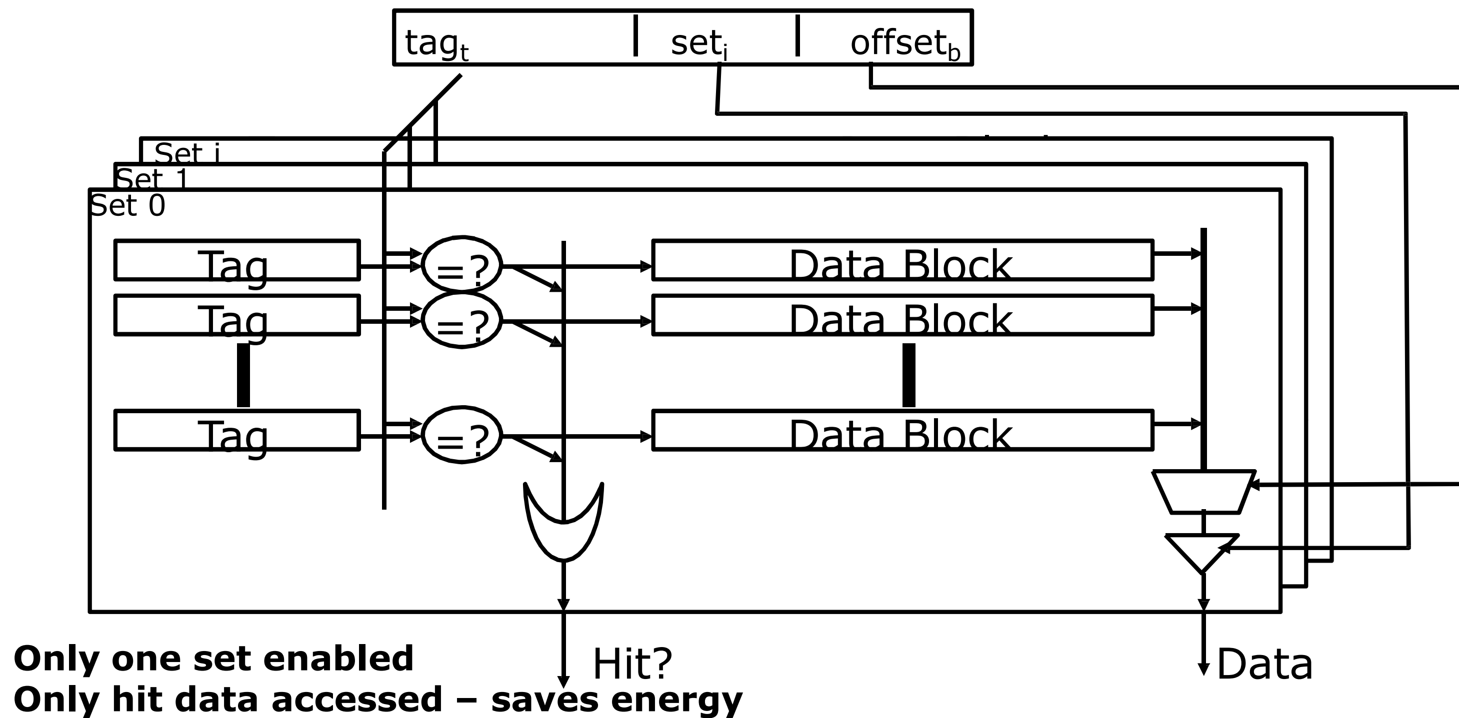
Set-Associative RAM-Tag Cache



- Not energy-efficient
  - A tag and data word is read from every way
- Two-phase approach
  - First read tags, then just read data from selected way
  - More energy-efficient
  - Doubles latency in L1
  - OK, for L2 and above, why?

Highly-Associative CAM-Tag Caches

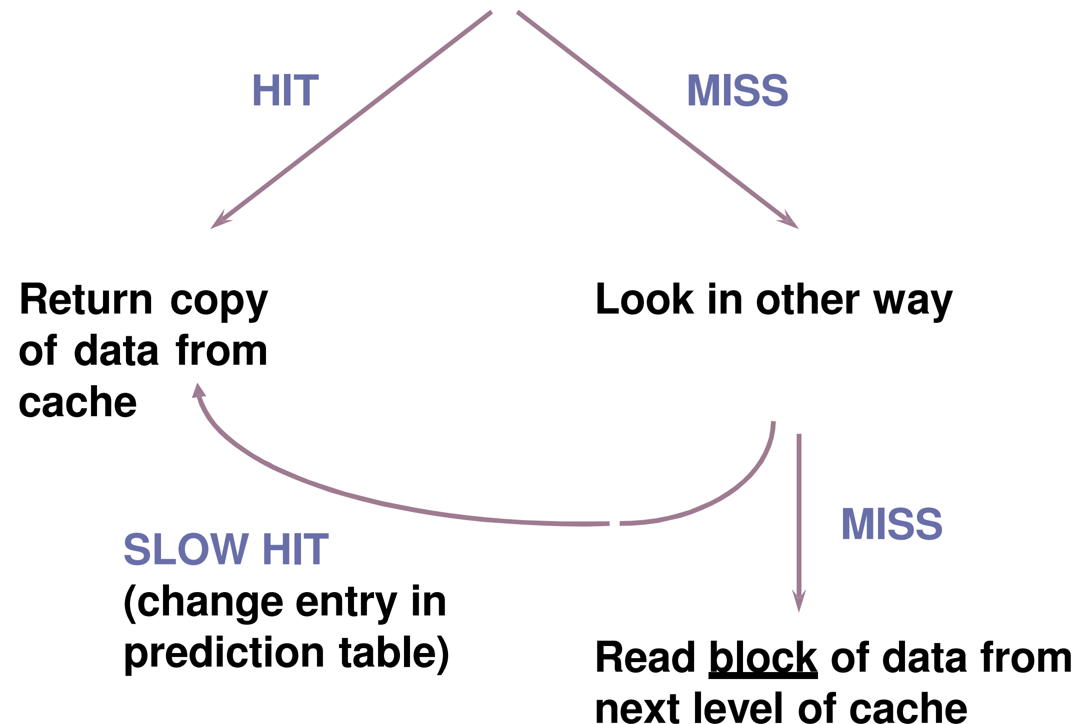
- For high associativity (e.g., 32-way), use content-addressable memory (CAM) for tags *(Intel XScale)*
- *Overhead: Tag+comparator bit 2-4x area of plain RAM-tag bit*



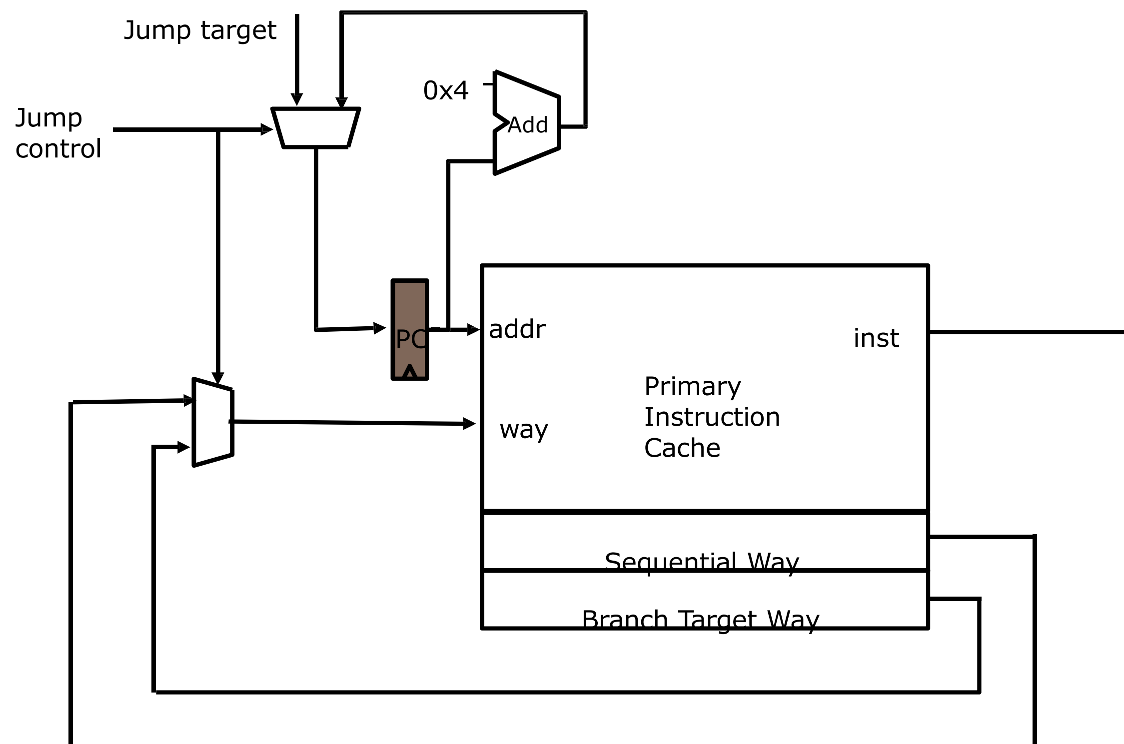
Only one set enabled
Only hit data accessed – saves energy

Way Predicting Caches
(MIPS R10000 L2 cache)

- Use processor address to index into way prediction table
- Look in predicted way at given index, then:

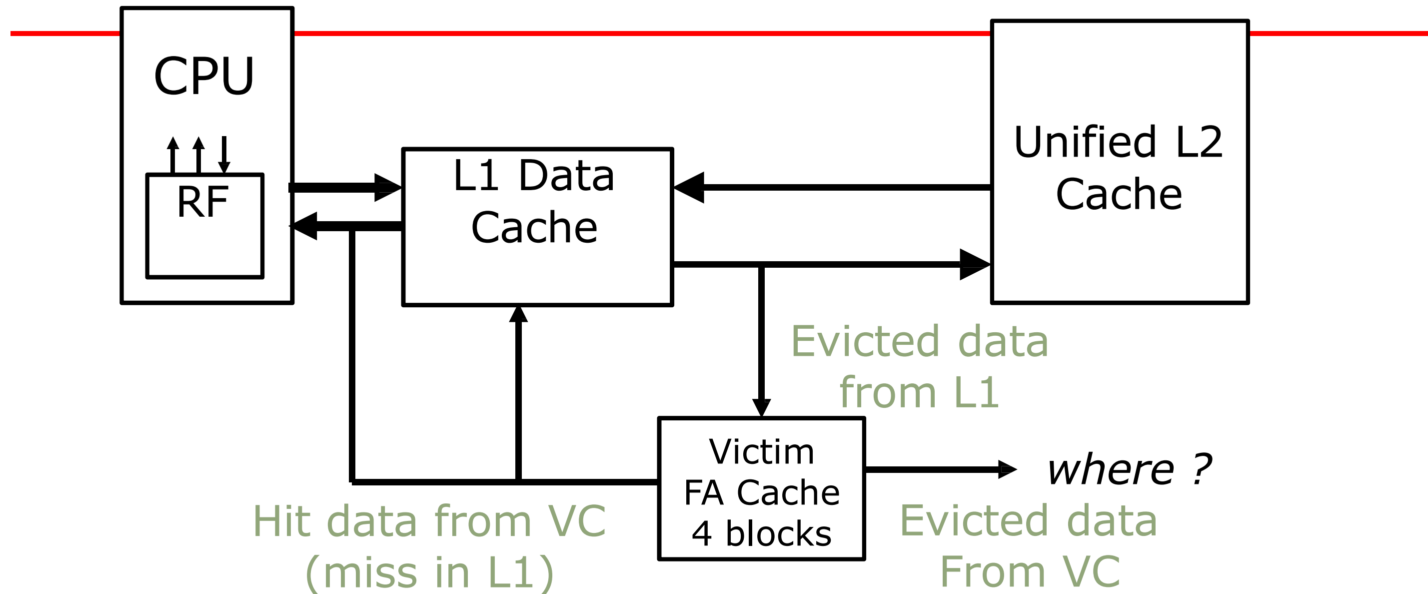


Way Predicting Instruction Cache
(Alpha 21264-like)



20

Victim Caches (HP 7200)



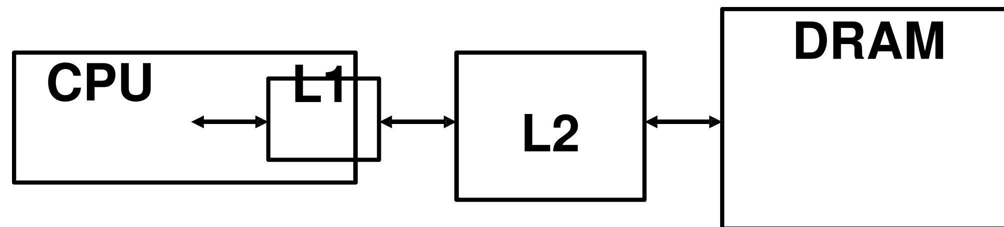
Victim cache is a small associative back up cache, added to a direct mapped cache, which holds recently evicted lines

- First look up in direct mapped cache
- If miss, look in victim cache
- If hit in victim cache, swap hit line with line now evicted from L1
- If miss in victim cache, L1 victim -> VC, VC victim->?

Fast hit time of direct mapped but with reduced conflict misses

Multilevel Caches

- A memory cannot be large and fast
- Increasing sizes of cache at each level



Local miss rate = misses in cache / accesses to cache Global

miss rate = misses in cache / CPU memory accesses

Misses per instruction = misses in cache / number of instructions

Inclusion Policy

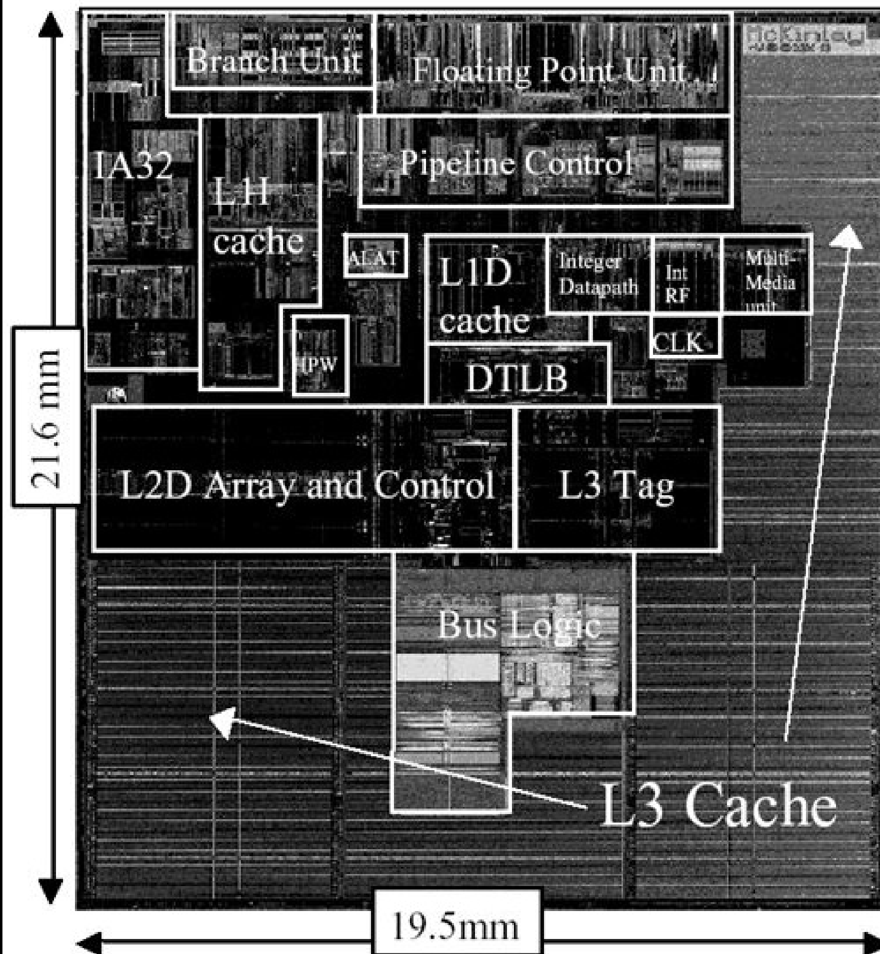
- Inclusive multilevel cache:
  - Inner cache holds copies of data in outer cache
  - Extra-CPU access needs only check outer cache
  - Most common case

- *Exclusive* multilevel caches:
  - Inner cache may hold data not in outer cache
  - Swap lines between inner/outer caches on miss
  - Used in Athlon with 64KB primary and 256KB secondary cache

Why choose one type of the other?

[Source: K. Asanovic, 2008]

Itanium-2 On-Chip Caches

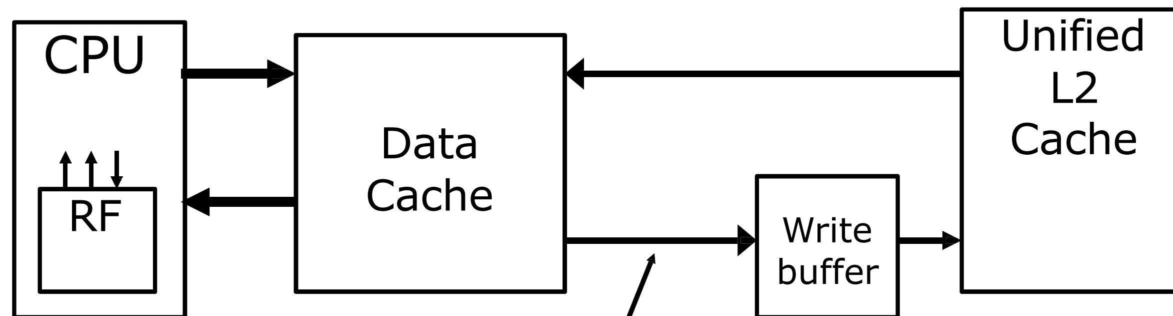


002)

- Level 1, 16KB, 4-way s.a., 64B line, quad-port (2 load+2 store), single cycle latency
- Level 2, 256KB, 4-way s.a, 128B line, quad-port (4 load or 4 store), five cycle latency
- Level 3, 3MB, 12-way s.a., 128B line, single 32B port, twelve cycle latency

L3 and L2 caches occupy more than 2/3 of total area!

Reducing Read Miss Penalty



Evicted dirty lines for writeback cache
OR
All writes in writethru cache

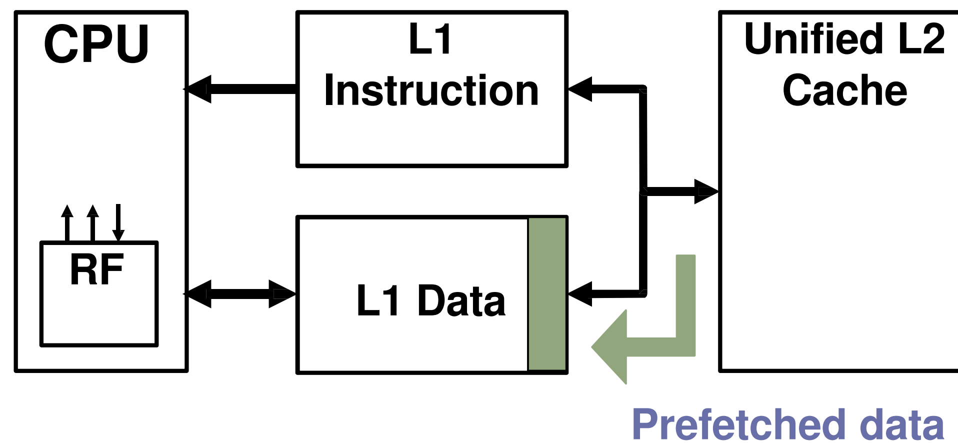
- Write buffer may hold updated value of location needed by a read miss
- Simple scheme: on a read miss, wait for the write buffer to go empty
- Faster scheme: Check write buffer addresses against read miss addresses, if no match, allow read miss to go ahead of writes, else, return value in write buffer

Prefetching

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- *What types of misses does prefetching affect?*

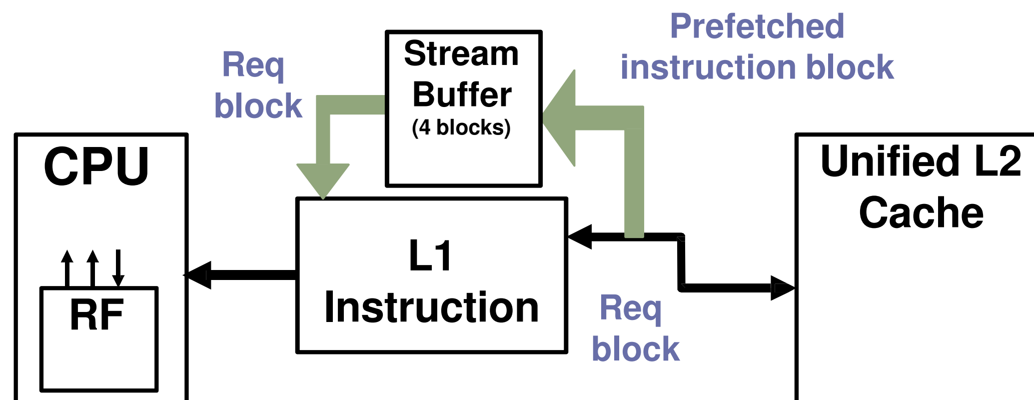
Issues in Prefetching

- Usefulness – should produce hits
- Timeliness – not late and not too early
- Cache and bandwidth pollution



Hardware Instruction Prefetching

- Instruction prefetch in Alpha AXP 21064
  - Fetch two blocks on a miss; the requested block and the next consecutive block
  - Requested block placed in cache, and next block in instruction stream buffer



Hardware Data Prefetching

- Prefetch-on-miss:
  - Prefetch b + 1 upon miss on b
- One Block Lookahead (OBL) scheme
  - Initiate prefetch for block b + 1 when block b is accessed
  - *Why is this different from doubling block size?*
  - Can extend to N block lookahead
- Strided prefetch
  - If sequence of accesses to block b, b+N, b+2N, then prefetch b+3N etc.

Software Prefetching

```
for(i=0; i < N; i++) {
    prefetch( &a[i + 1] );
    prefetch( &b[i + 1] );
    SUM = SUM + a[i] * b[i];
}
```

- *What property do we require of the cache for prefetching to work ?*

Software Prefetching Issues

- Timing is the biggest issue, not predictability
  - If you prefetch very close to when the data is required, you might be too late
  - Prefetch too early, cause pollution
  - Estimate how long it will take for the data to come into L1, so we can set P appropriately
  - *Why is this hard to do?*

```
for(i=0; i < N; i++) {
    prefetch( &a[i + P] );
    prefetch( &b[i + P] );
    SUM = SUM + a[i] * b[i];
}
```

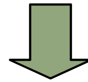
Must consider cost of prefetch instructions

Compiler Optimizations

- Restructuring code affects the data block access sequence
  - Group data accesses together to improve spatial locality
  - Re-order data accesses to improve temporal locality
- Prevent data from entering the cache
  - Useful for variables that will only be accessed once before being replaced
  - Needs mechanism for software to tell hardware not to cache data (instruction hints or page table bits)
- Kill data that will never be used again
  - Streaming data exploits spatial locality but not temporal locality
  - Replace into dead cache locations

Loop Interchange

```
for(j=0; j < N; j++) {
    for(i=0; i < M; i++) {
        x[i][j] = 2 * x[i][j];
    }
}
```



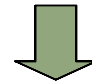
```
for(i=0; i < M; i++) {
    for(j=0; j < N; j++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

What type of locality does this improve?

Loop Fusion

```
for(i=0; i < N; i++)
    for(j=0; j < M; j++)
        a[i][j] = b[i][j] * c[i][j];

for(i=0; i < N; i++)
    for(j=0; j < M; j++)
        d[i][j] = a[i][j] * c[i][j];
```

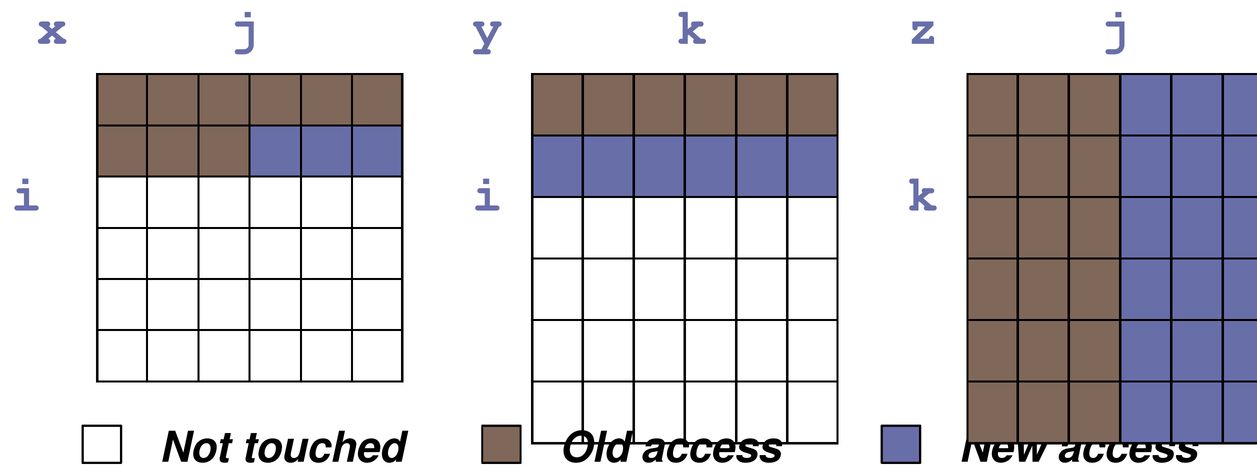


```
for(i=0; i < M; i++)
    for(j=0; j < N; j++) {
        a[i][j] = b[i][j] * c[i][j];
        d[i][j] = a[i][j] * c[i][j];
    }
```

What type of locality does this improve?

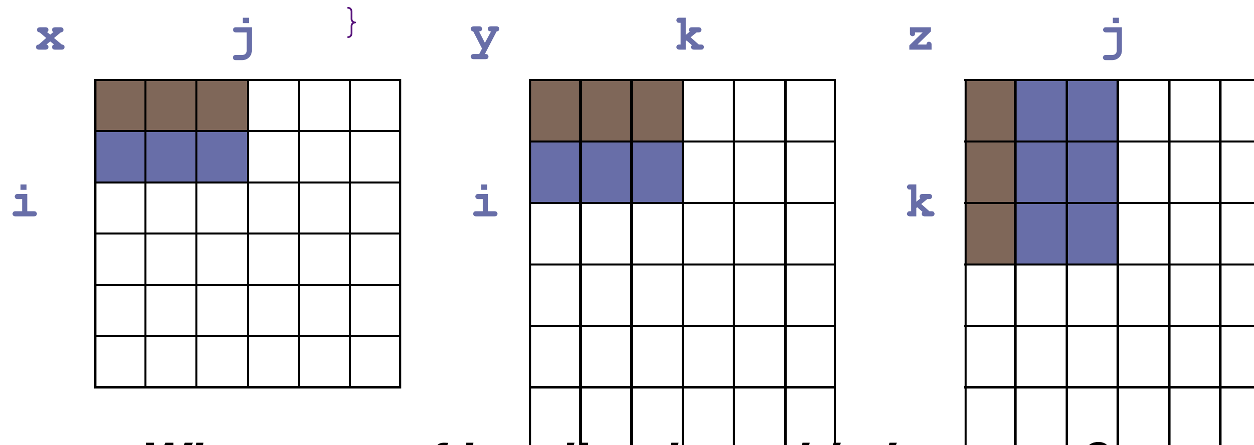
Blocking

```
for(i=0; i < N; i++)
    for(j=0; j < N; j++) {
        r = 0;
        for(k=0; k < N; k++)
            r = r + y[i][k] * z[k][j];
        x[i][j] = r;
    }
```



Blocking

```
for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=jj; j < min(jj+B,N); j++) {
                r = 0;
                for(k=kk; k < min(kk+B,N); k++)
                    r = r + y[i][k] * z[k][j];
                x[i][j] = x[i][j] + r;
            }
```



What type of locality does this improve?

Memory Management: *From Absolute Addresses to Demand Paging*

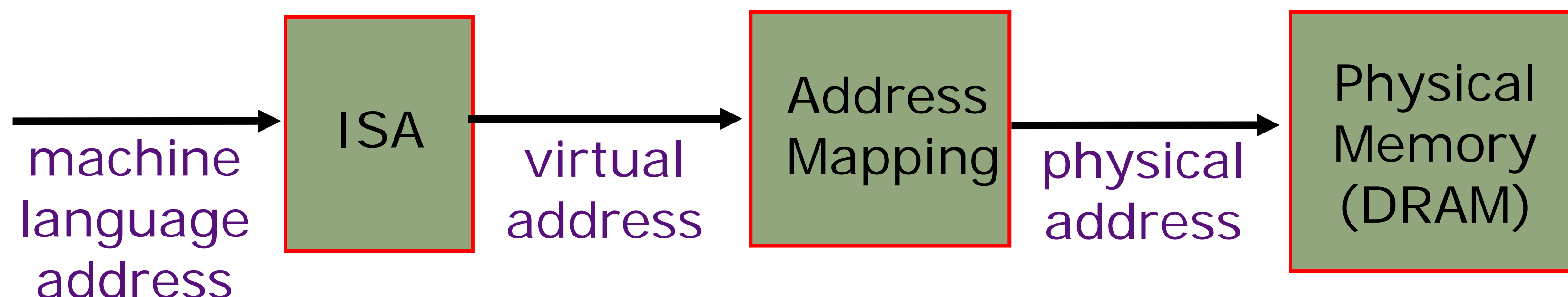
Joel Emer
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Based on the material prepared by Arvind and Krste Asanovic

Memory Management

- The Fifties
  - Absolute Addresses
  - Dynamic address translation
- The Sixties
  - Paged memory systems and TLBs
  - Atlas' Demand paging
- Modern Virtual Memory Systems

Names for Memory Locations



- Machine language address
  - as specified in machine code
- Virtual address
  - ISA specifies translation of machine code address into virtual address of program variable (sometime called *effective* address)
- Physical address
  ⇒ operating system specifies mapping of virtual address into name for a physical memory location

Absolute Addresses

EDSAC, early 50's

virtual address = physical memory address

- Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)
- Addresses in a program depended upon where the program was to be loaded in memory
- *But* it was more convenient for programmers to write location-independent subroutines

How could location independence be achieved?

CSAIL

Dynamic Address Translation

Motivation

In the early machines, I/O operations were slow and each word transferred involved the CPU

Higher throughput if CPU and I/O of 2 or more programs were overlapped. *How?*
⇒ *multiprogramming*

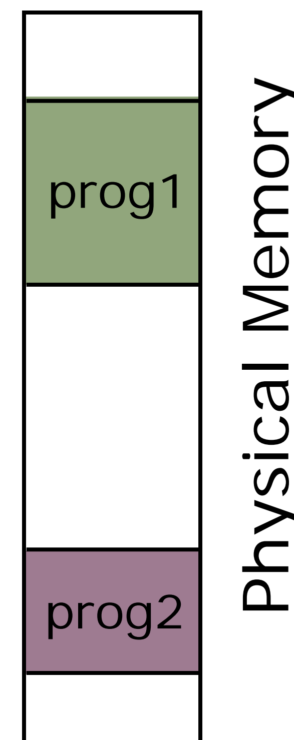
Location independent programs

Programming and storage management ease
⇒ need for a *base register*

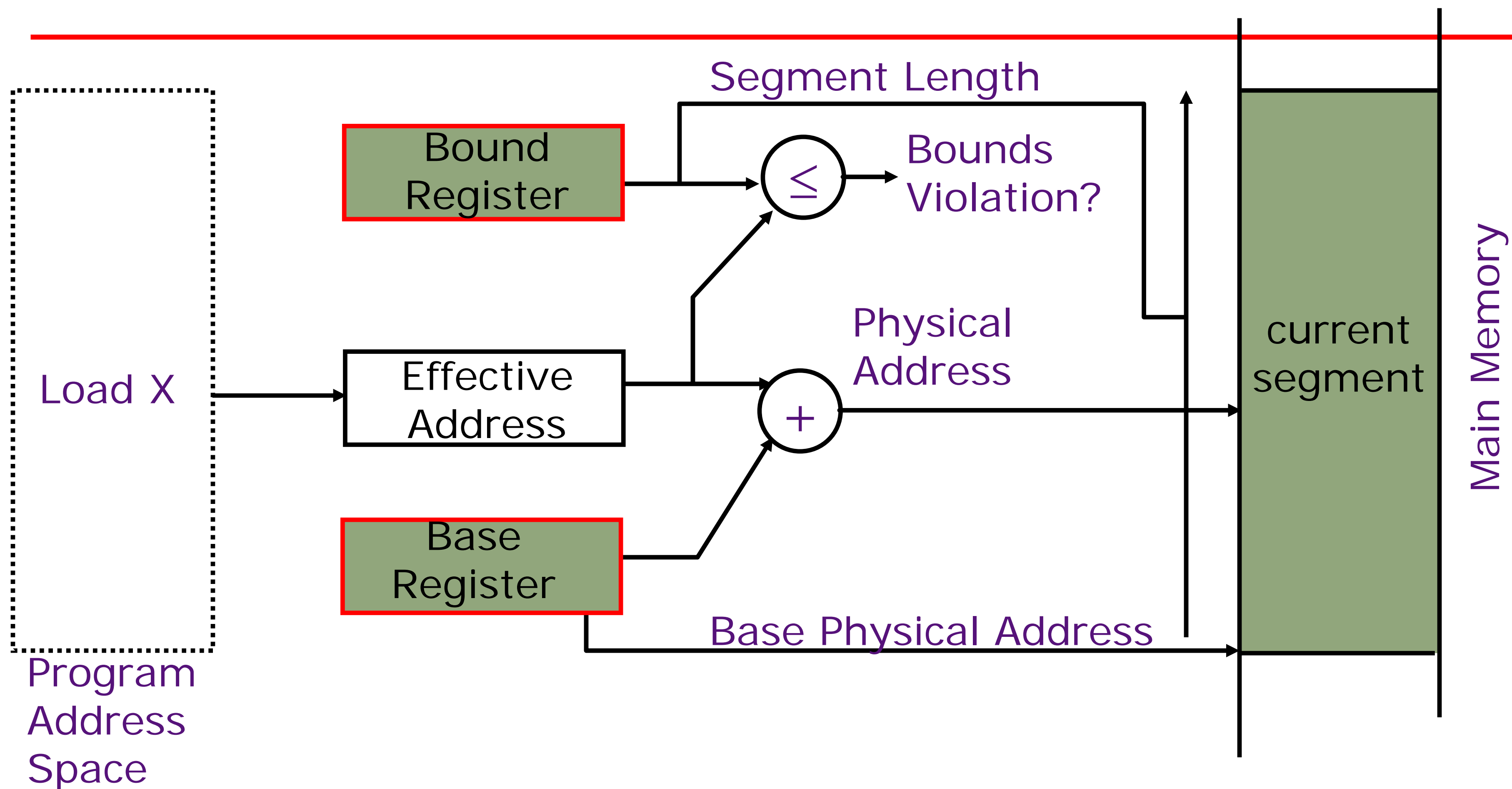
Protection

Independent programs should not affect each other inadvertently

⇒ need for a *bound register*



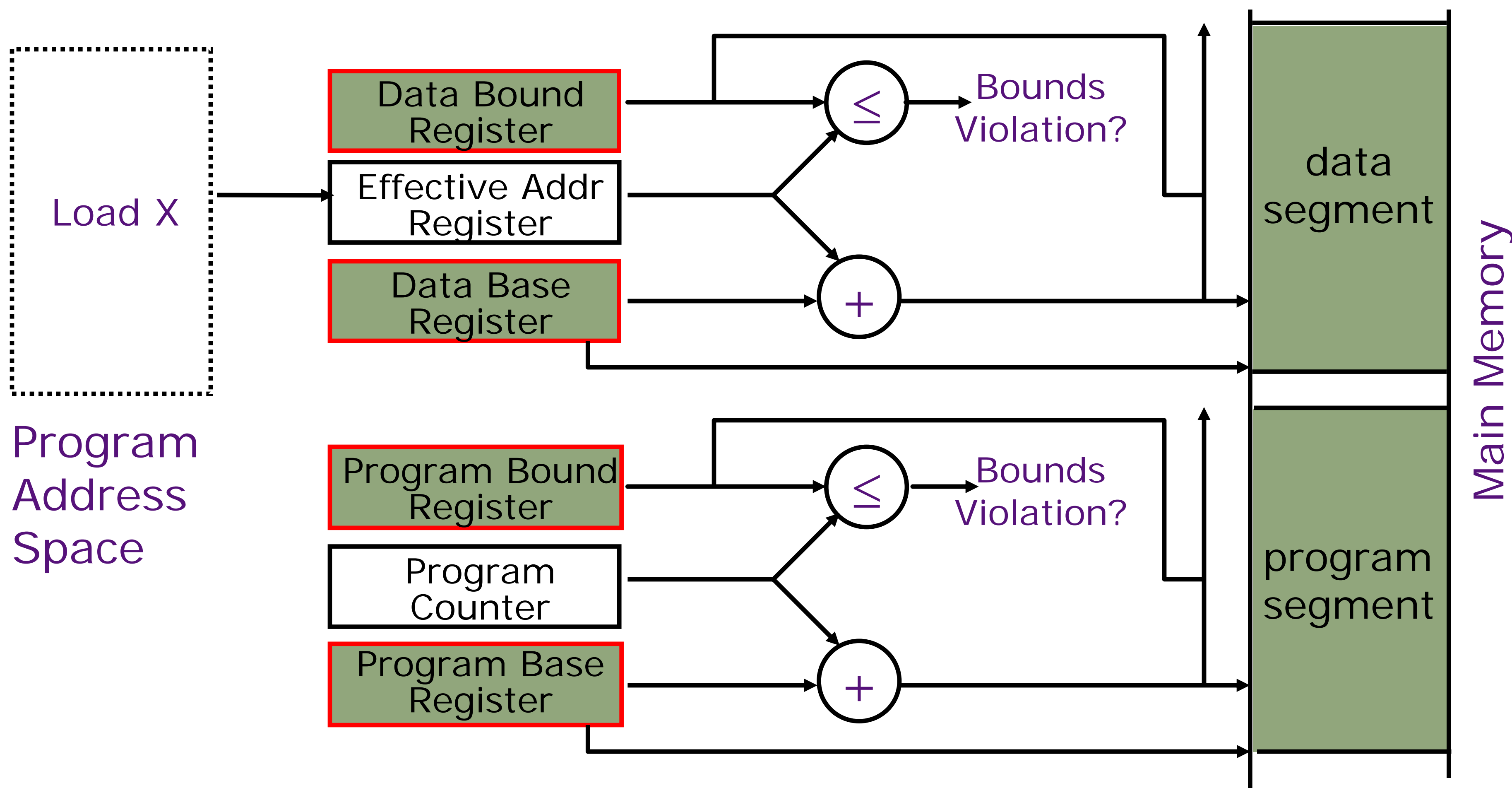
Simple Base and Bound Translation



Base and bounds registers are visible/accessible only when processor is running in the *supervisor mode*

CSAIL

Separate Areas for Program and Data

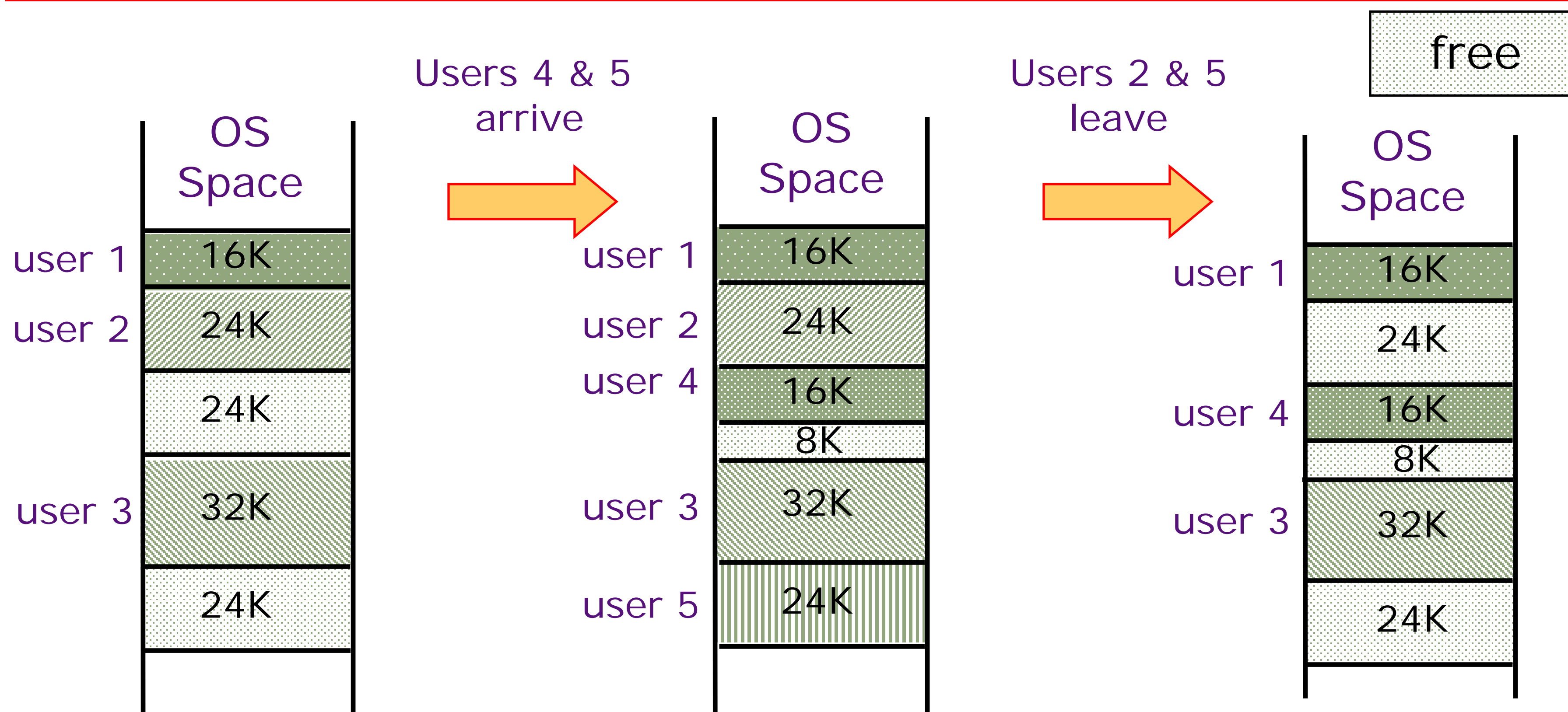


What is an advantage of this separation?

(Scheme still used today on Cray vector supercomputers)

CSAIL

Memory Fragmentation



As users come and go, the storage is "fragmented". Therefore, at some stage programs have to be moved around to compact the storage.

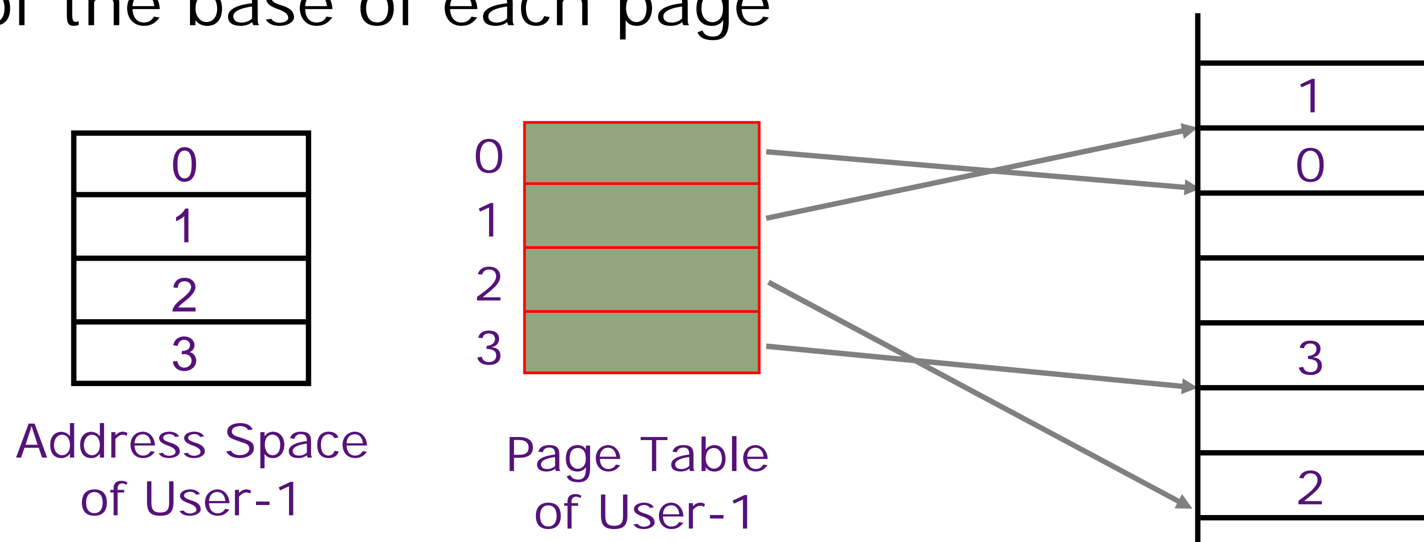
CSAIL

Paged Memory Systems

- Processor generated address can be interpreted as a pair <page number, offset>

page number	offset

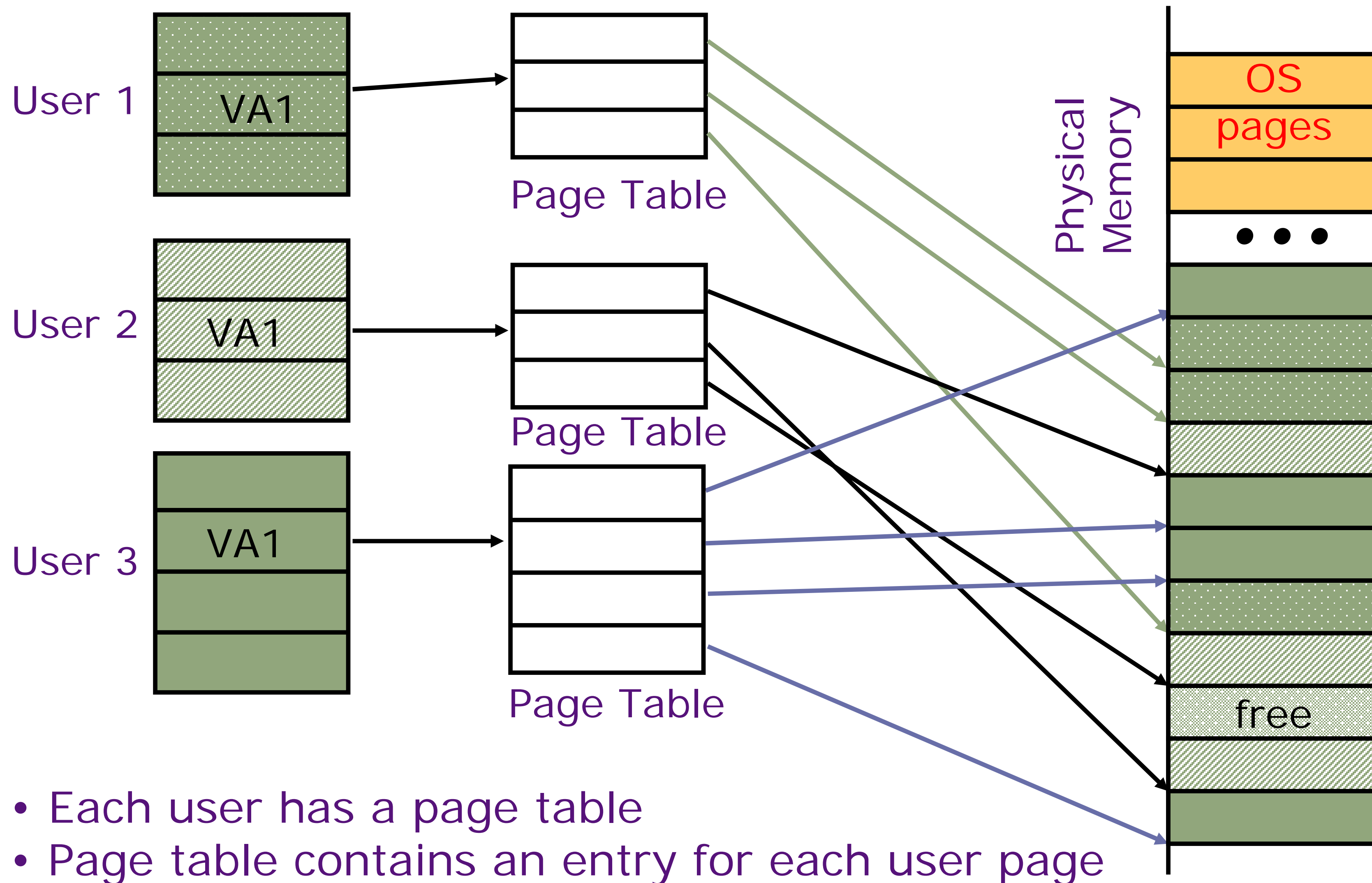
- A page table contains the physical address of the base of each page



Page tables make it possible to store the pages of a program non-contiguously.

CSAIL

Private Address Space per User



- Each user has a page table
- Page table contains an entry for each user page

Where Should Page Tables Reside?

- Space required by the page tables (PT) is proportional to the address space, number of users, ...
  - ⇒ Space requirement is large
  - ⇒ Too expensive to keep in registers

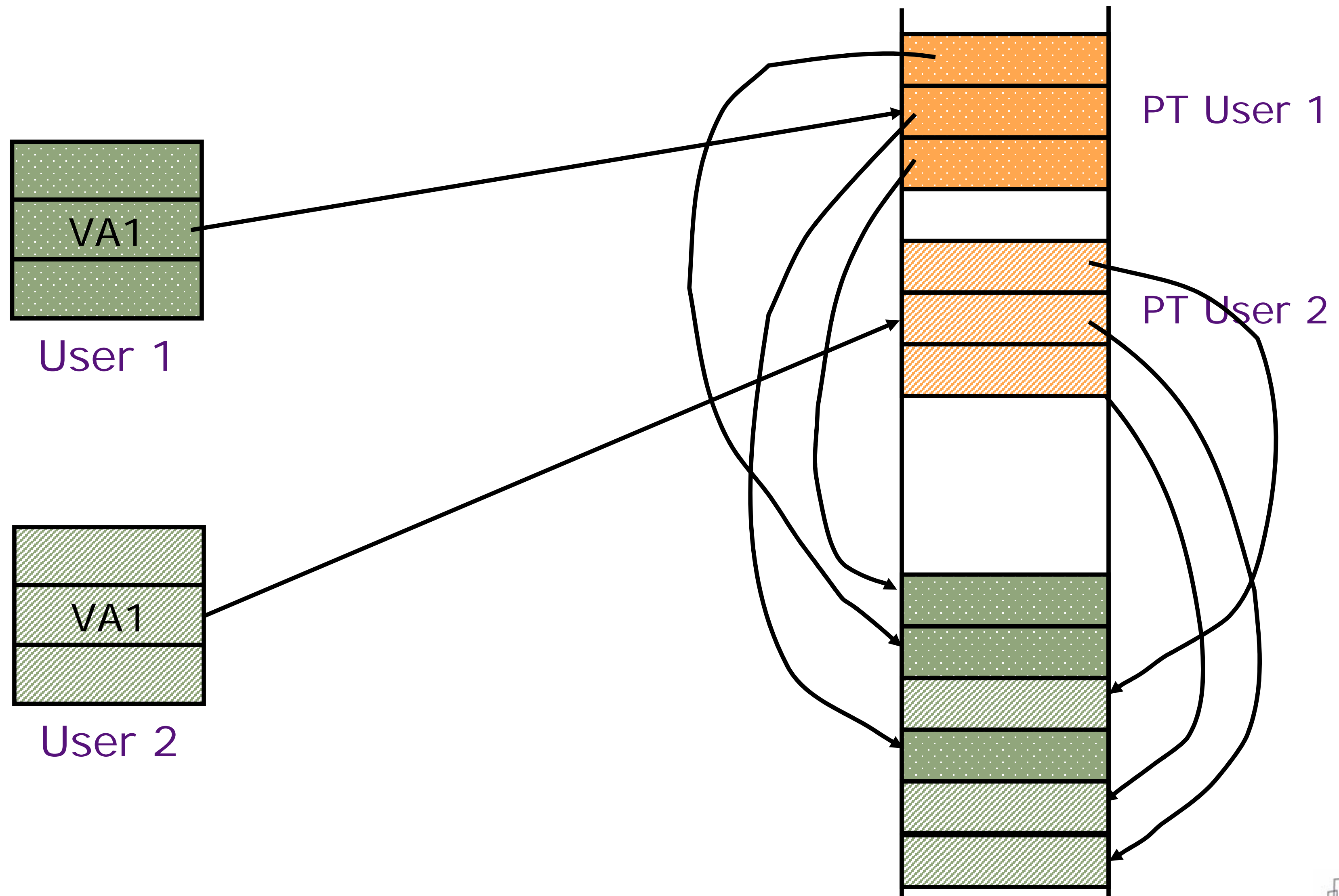
- Idea: Keep PT of the current user in special registers
  - may not be feasible for large page tables
  - Increases the cost of context swap

- Idea: Keep PTs in the main memory
  - needs one reference to retrieve the page base address and another to access the data word

    ⇒ *doubles the number of memory references!*

CSAIL

Page Tables in Physical Memory



A Problem in Early Sixties

- There were many applications whose data could not fit in the main memory, e.g., payroll
  - *Paged memory system reduced fragmentation but still required the whole program to be resident in the main memory*

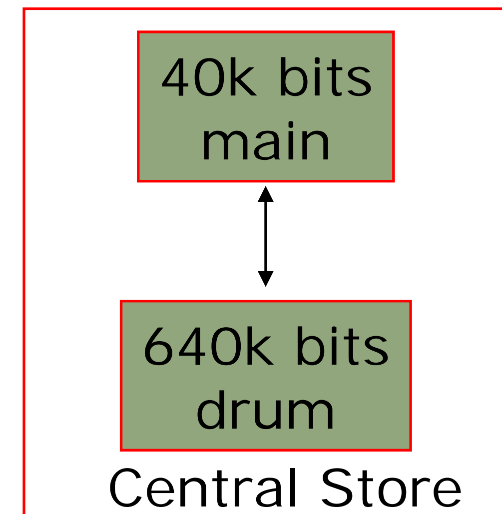
- Programmers moved the data back and forth from the secondary store by *overlaying* it repeatedly on the primary store

tricky programming!

Manual Overlays

- Assume an instruction can address all the storage on the drum
- *Method 1:* programmer keeps track of addresses in the main memory and initiates an I/O transfer when required
- *Method 2:* automatic initiation of I/O transfers by software address translation

Brooker's interpretive coding, 1960



Ferranti Mercury
1956

Problems?

Method1: Difficult, error prone
Method2: Inefficient

CSAIL

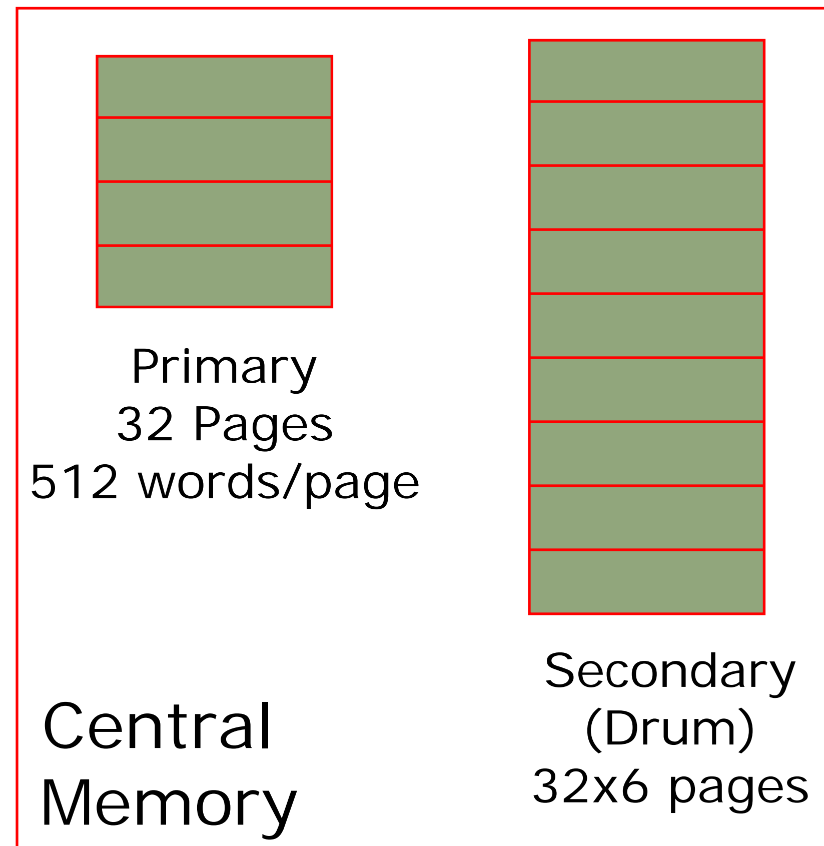
Demand Paging in Atlas (1962)

"A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor."

Tom Kilburn

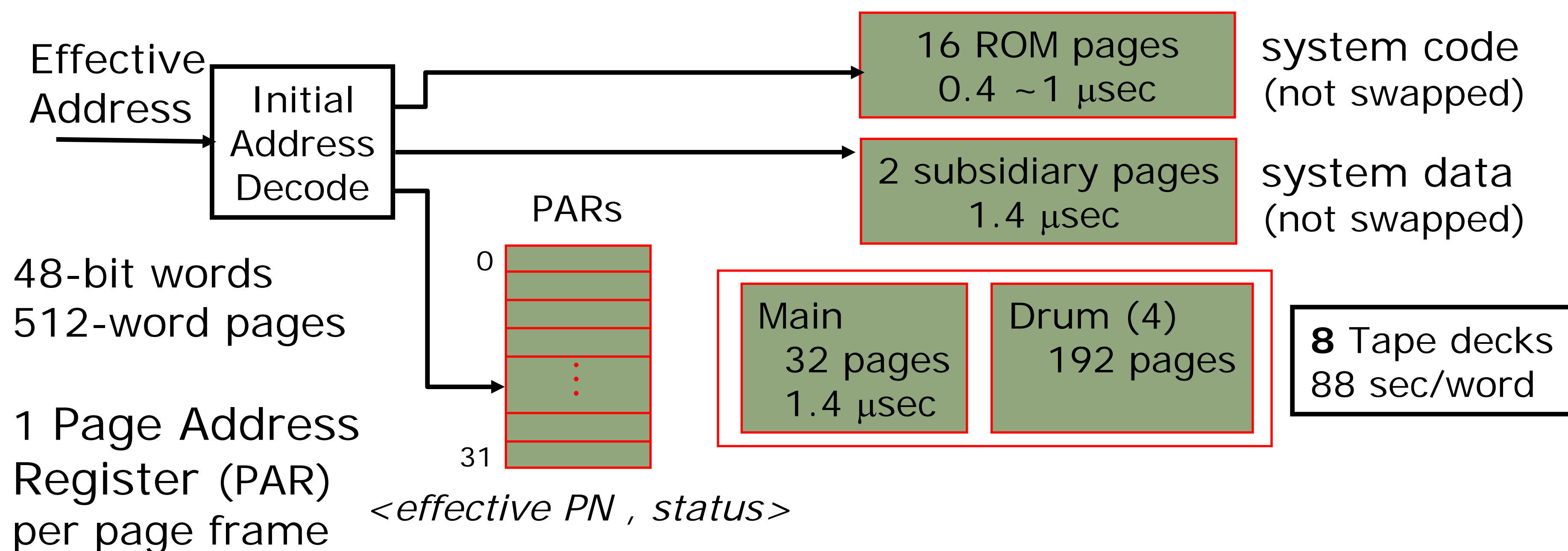
Primary memory as a *cache* for secondary memory

User sees 32 x 6 x 512 words of storage



CSAIL

Hardware Organization of Atlas



Compare the effective page address against all 32 PARs

match ⇒ normal access

no match ⇒ *page fault*

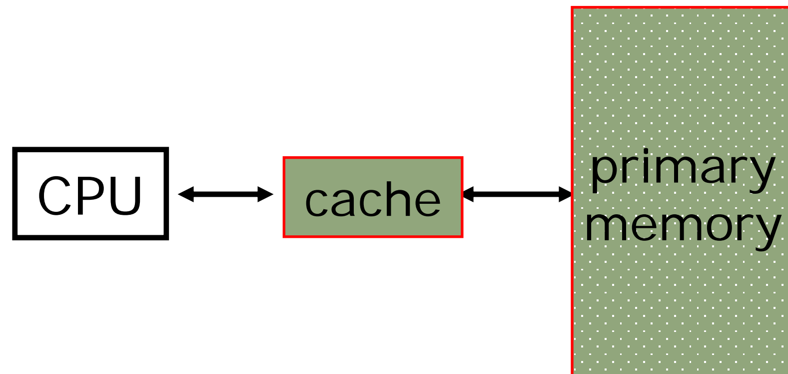
save the state of the partially executed instruction

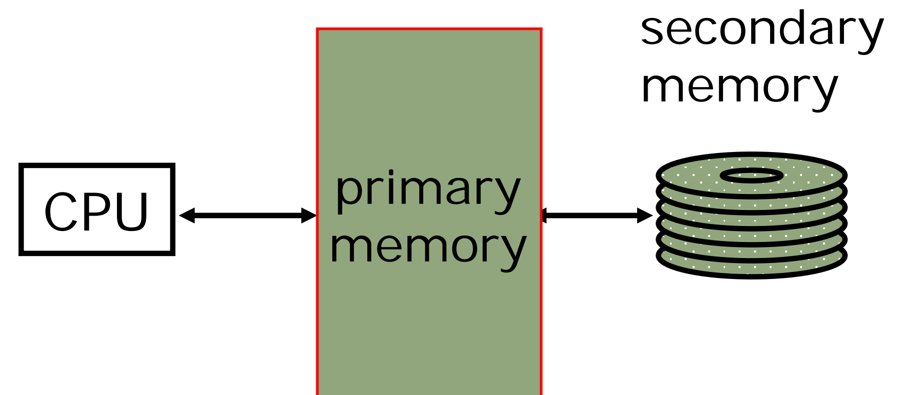
Atlas Demand Paging Scheme

- On a page fault:
  - Input transfer into a free page is initiated
  - The Page Address Register (PAR) is updated
  - If no free page is left, a *page is selected to be replaced* (based on usage)
  - The replaced page is written on the drum
    - to minimize drum latency effect, the first empty page on the drum was selected
  - The *page table is updated* to point to the new location of the page on the drum

CSAIL

Caching vs. Demand Paging





Caching	*Demand paging*
cache entry	page-frame
cache block (~32 bytes)	page (~4K bytes)
cache miss (1% to 20%)	page miss (<0.001%)
cache hit (~1 cycle)	page hit (~100 cycles)
cache miss (~100 cycles)	page miss(~5M cycles)
a miss is handled in *hardware*	a miss is handled mostly in *software*

Five-minute break to stretch your legs

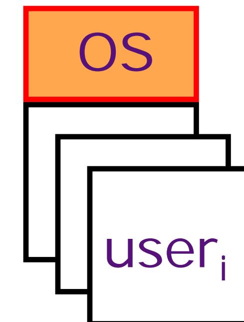
Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy

several users, each with their private address space and one or more shared address spaces

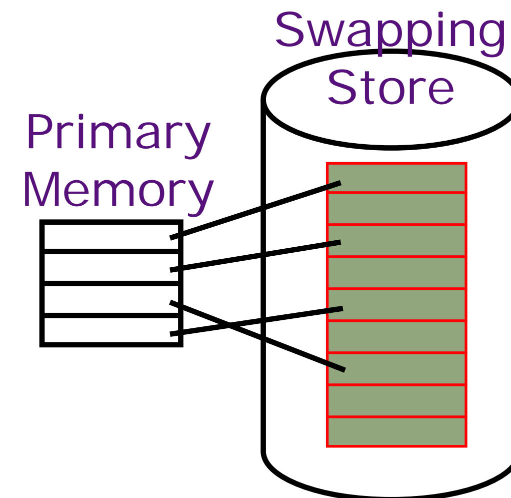
page table ≡ name space



Demand Paging

Provides the ability to run programs larger than the primary memory

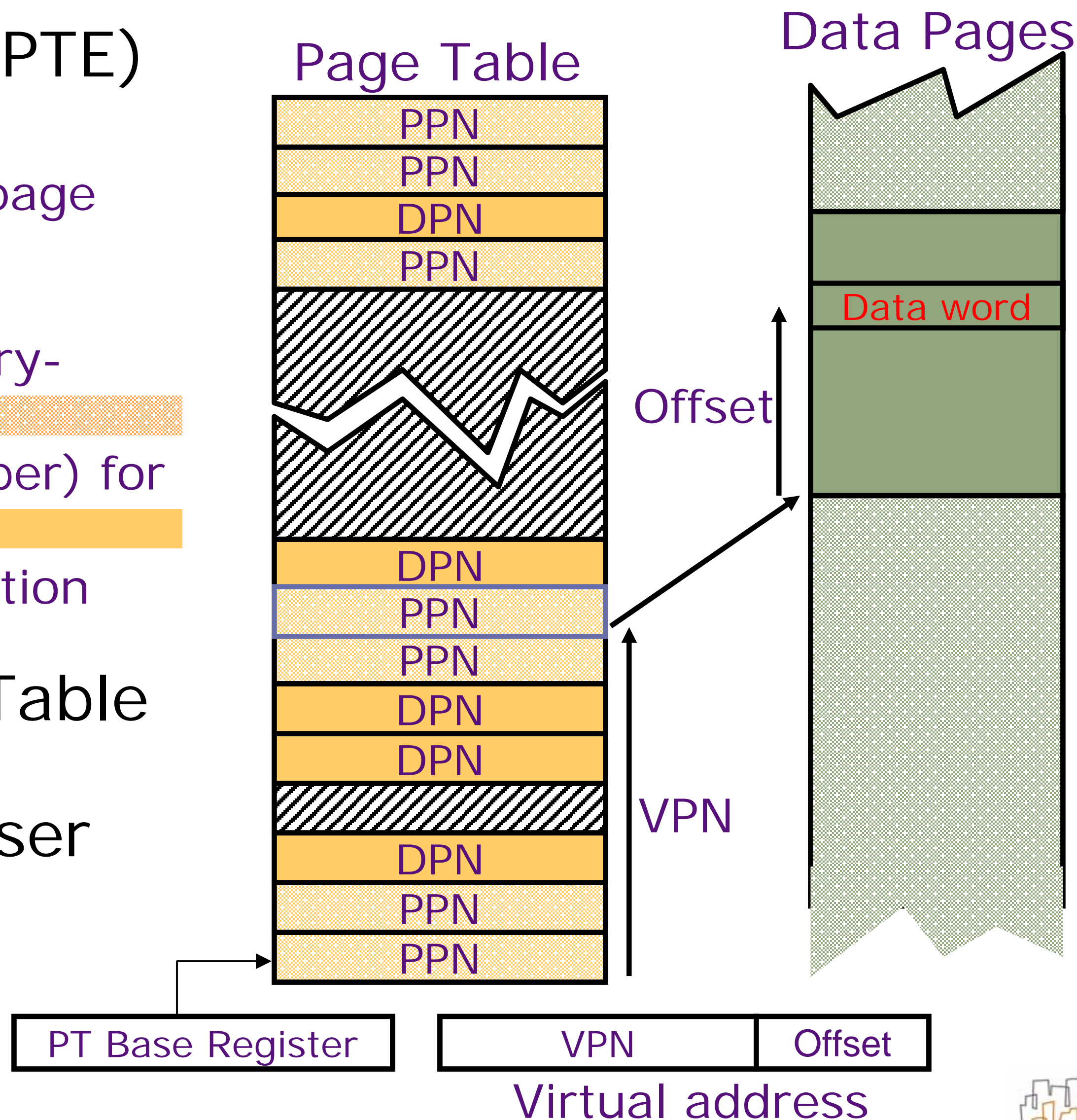
Hides differences in machine configurations



The price is address translation on each memory reference

Linear Page Table

- Page Table Entry (PTE) contains:
  - A bit to indicate if a page exists
  - PPN (physical page number) for a memory-resident page
  - DPN (disk page number) for a page on the disk
  - Status bits for protection and usage
- OS sets the Page Table Base Register whenever active user process changes



Size of Linear Page Table

With 32-bit addresses, 4-KB pages & 4-byte PTEs:

⇒ 2^{20} PTEs, i.e, 4 MB page table per user

⇒ 4 GB of swap needed to back up full virtual address space

Larger pages?

- Internal fragmentation (Not all memory in a page is used)
- Larger page fault penalty (more time to read from disk)

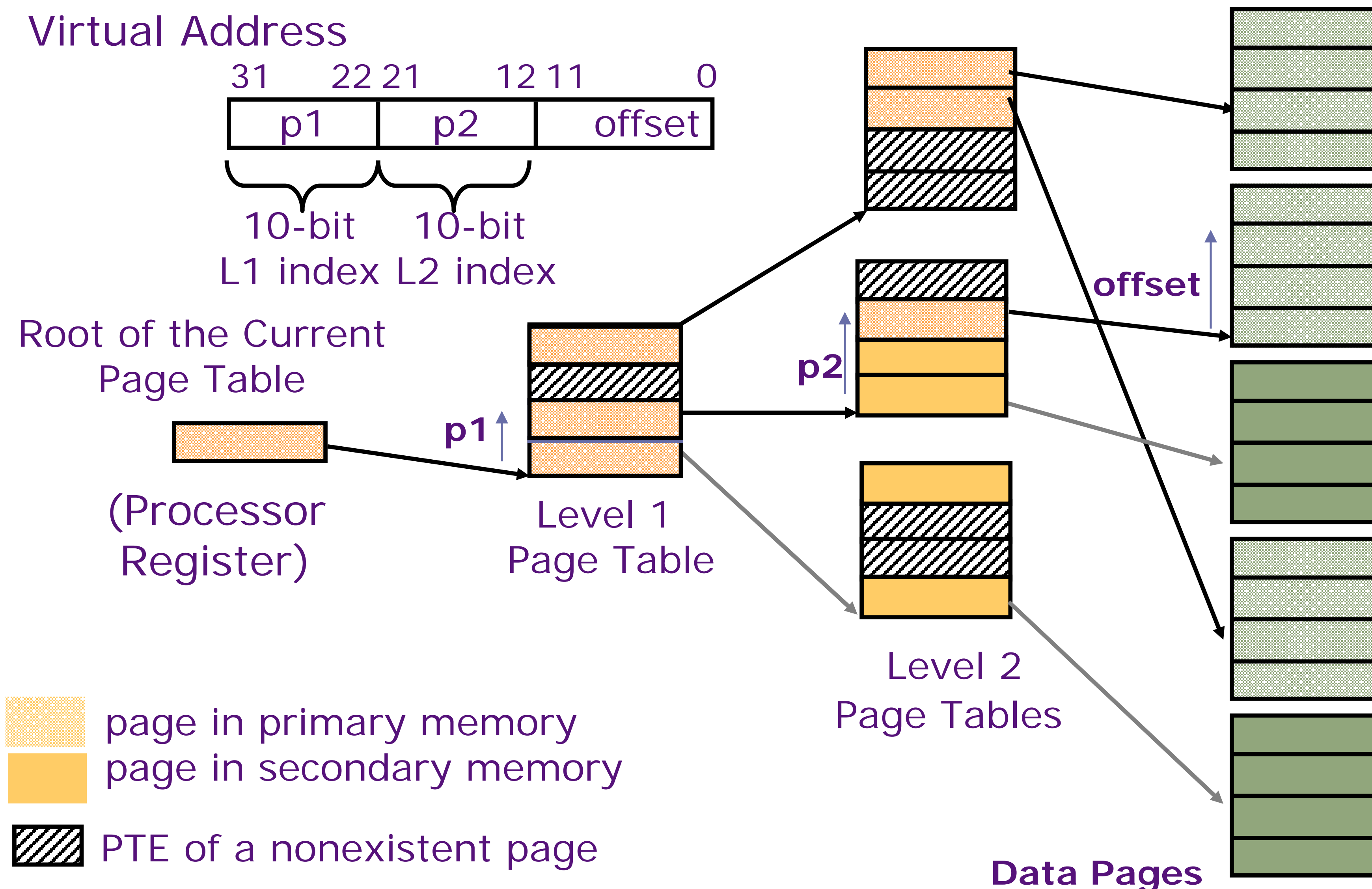
What about 64-bit virtual address space???

- Even 1MB pages would require 2^{44} 8-byte PTEs (35 TB!)

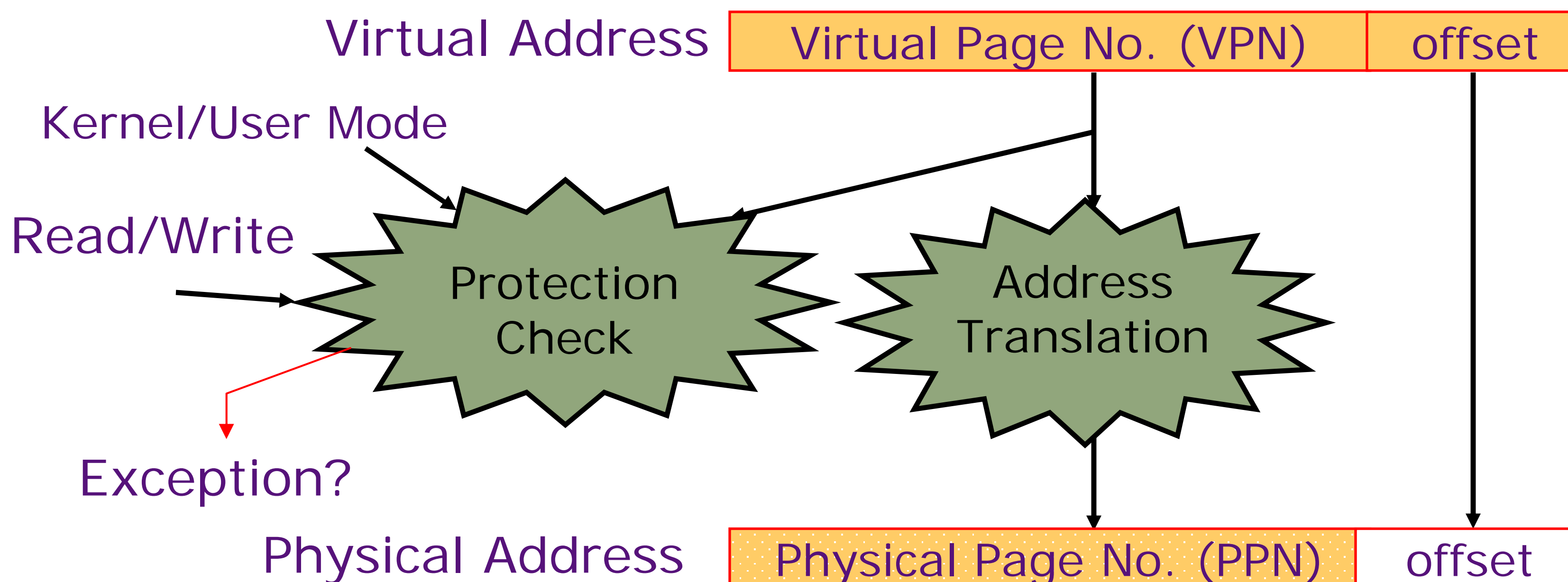
What is the "saving grace" ?

CSAIL

Hierarchical Page Table



Address Translation & Protection



- Every instruction and data access needs address translation and protection checks

A good VM design needs to be fast (~ one cycle) and space efficient

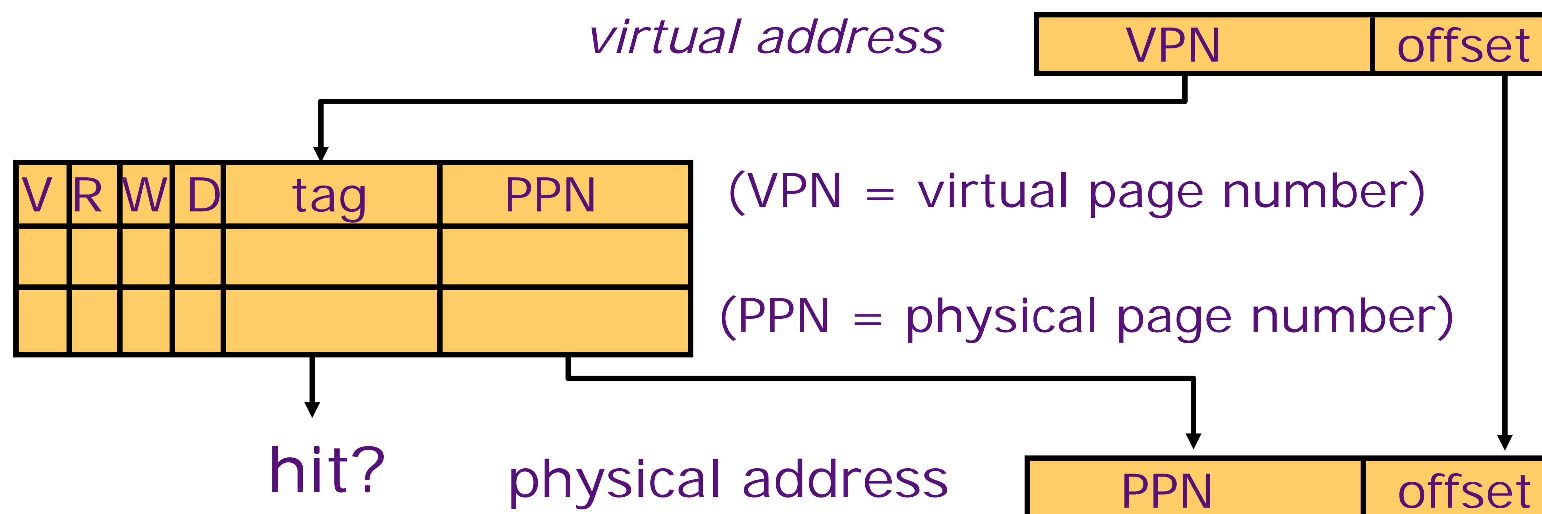
CSAIL

Translation Lookaside Buffers

Address translation is very expensive!
In a two-level page table, each reference becomes several memory accesses

Solution: *Cache translations in TLB*

TLB hit ⇒ *Single Cycle Translation*
TLB miss ⇒ *Page Table Walk to refill*



TLB Designs

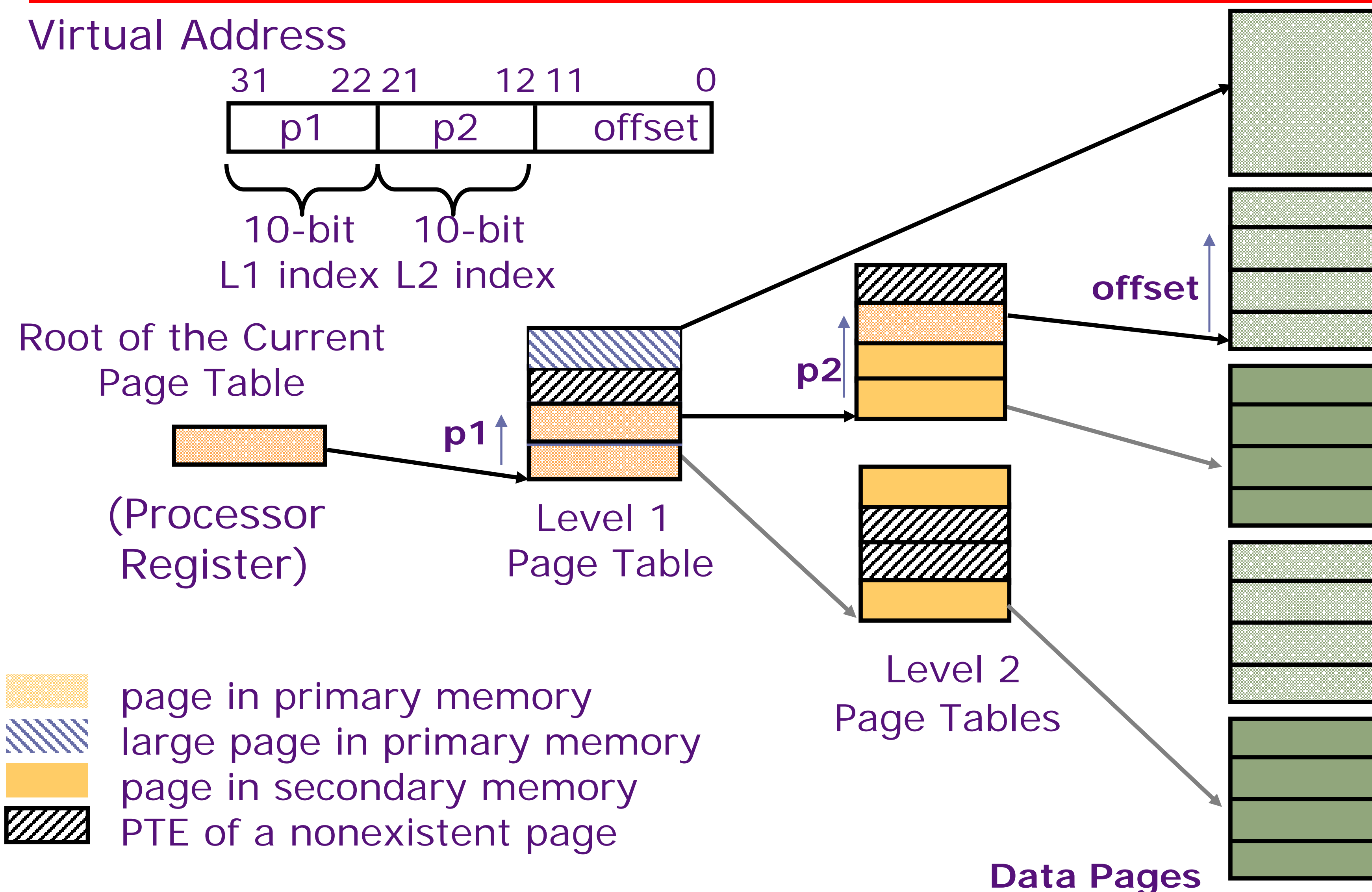
- Typically 32-128 entries, usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages → more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
- Random or FIFO replacement policy
- No process information in TLB?
- TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB

Example: 64 TLB entries, 4KB pages, one page per entry

TLB Reach = _______________________________?

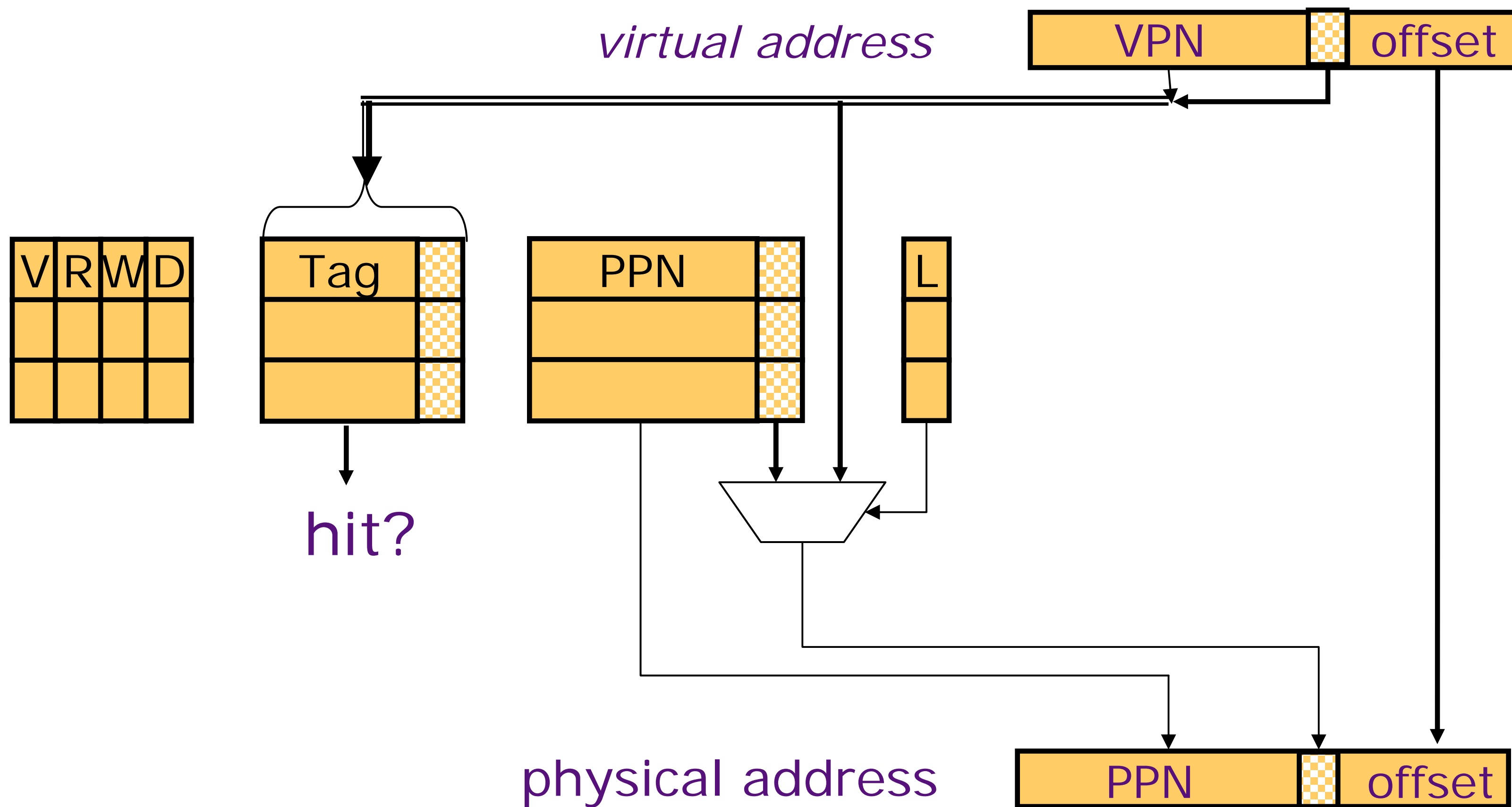
CSAIL

Variable Sized Page Support



Variable Size Page TLB

Some systems support multiple page sizes.



CSAIL

Handling A TLB Miss

Software (MIPS, Alpha)

TLB miss causes an exception and the operating system walks the page tables and reloads TLB. *A privileged "untranslated" addressing mode used for walk*

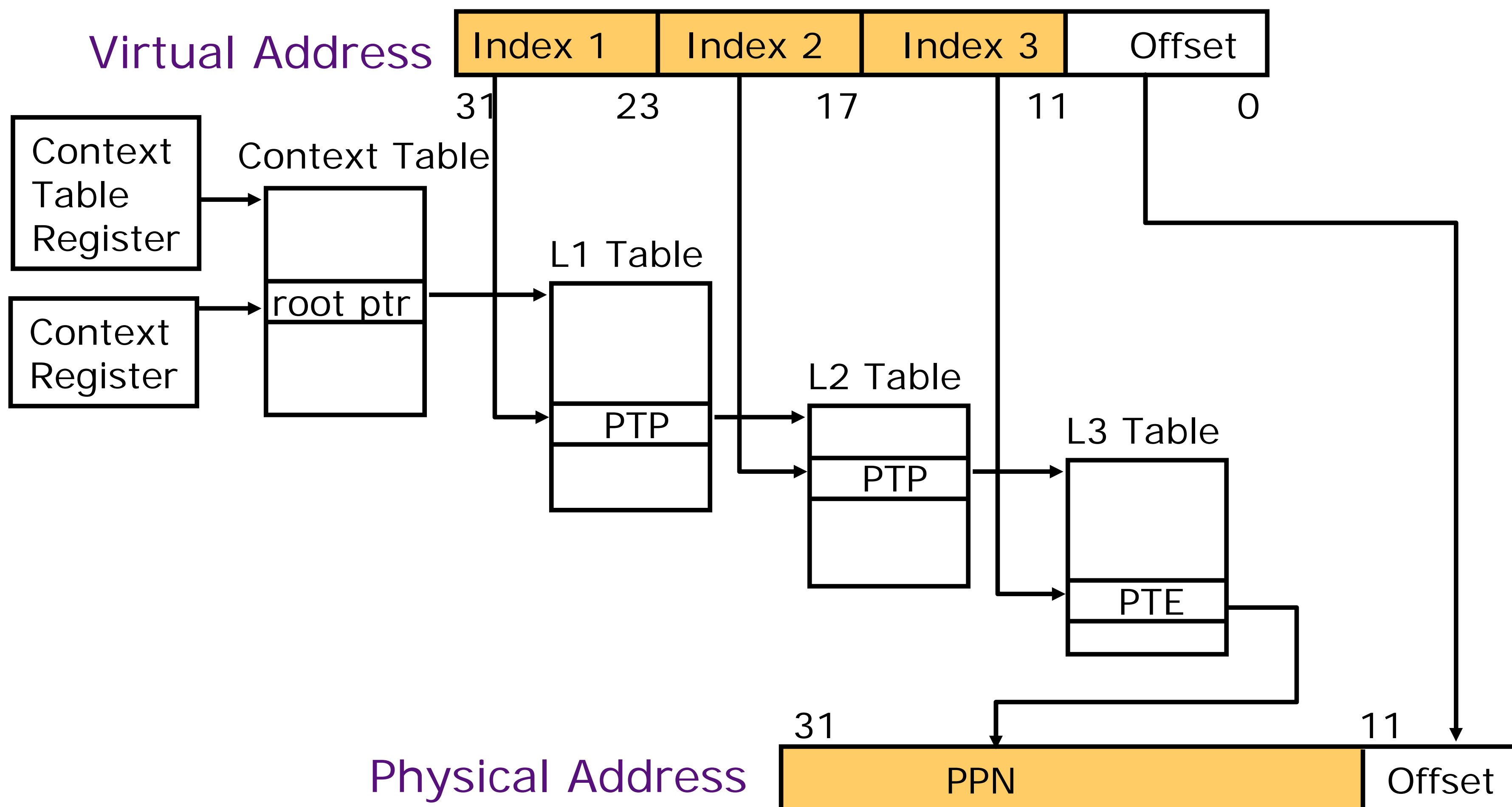
Hardware (SPARC v8, x86, PowerPC)

A memory management unit (MMU) walks the page tables and reloads the TLB

If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page-Fault exception for the original instruction

CSAIL

Hierarchical Page Table Walk: SPARC v8

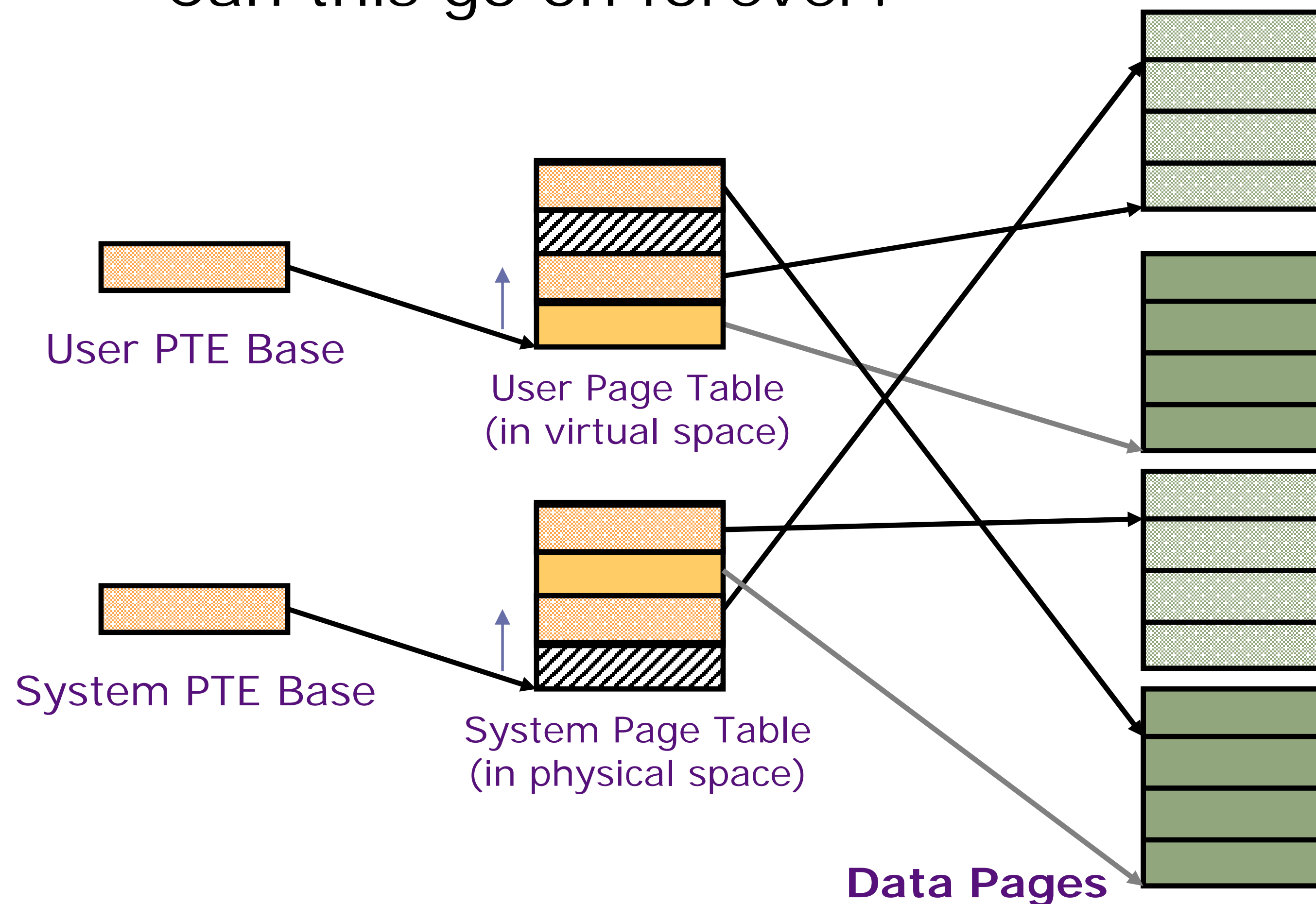


MMU does this table walk in hardware on a TLB miss

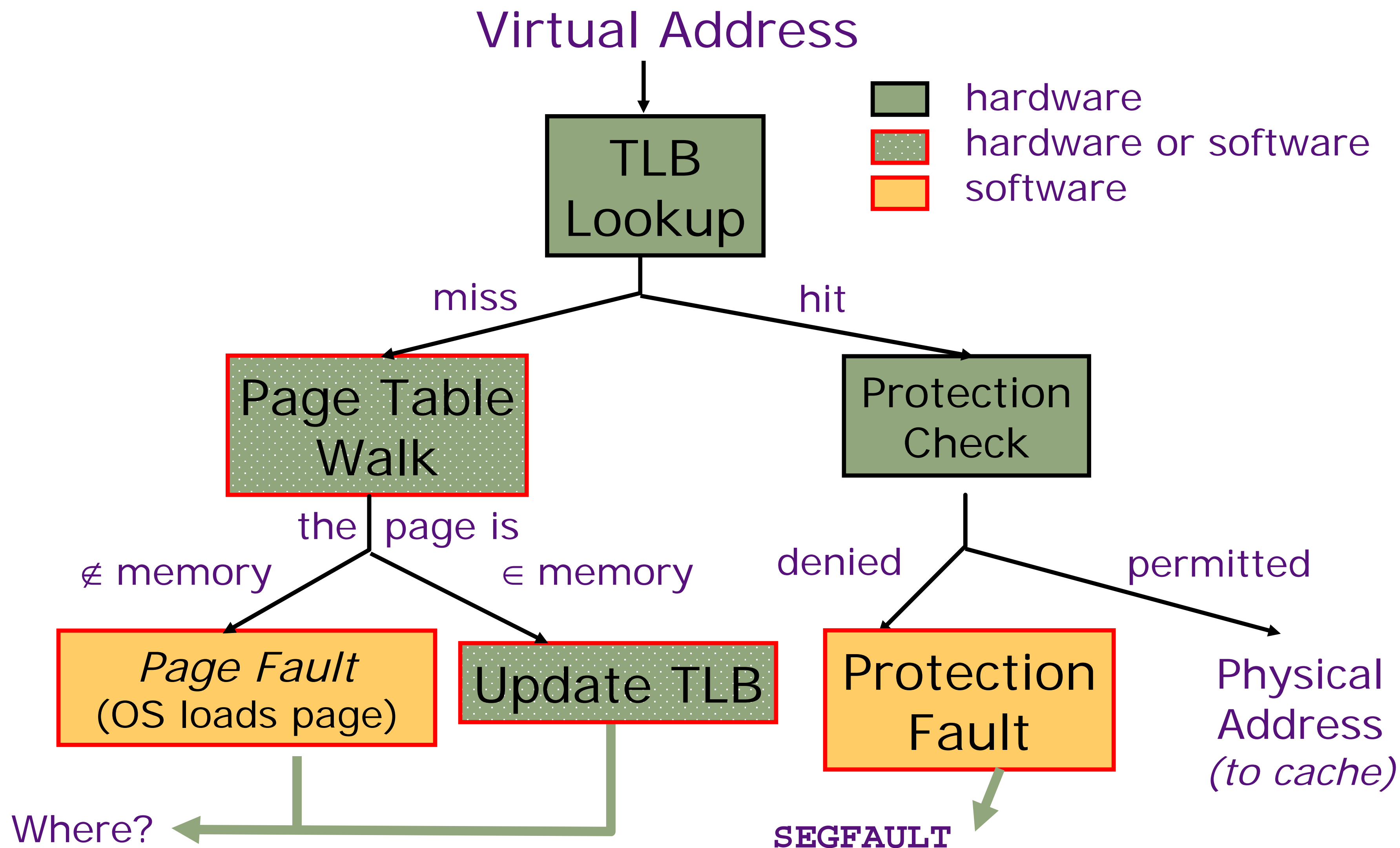
CSAIL

Translation for Page Tables

- Can references to page tables TLB miss
- Can this go on forever?



Address Translation: *putting it all together*



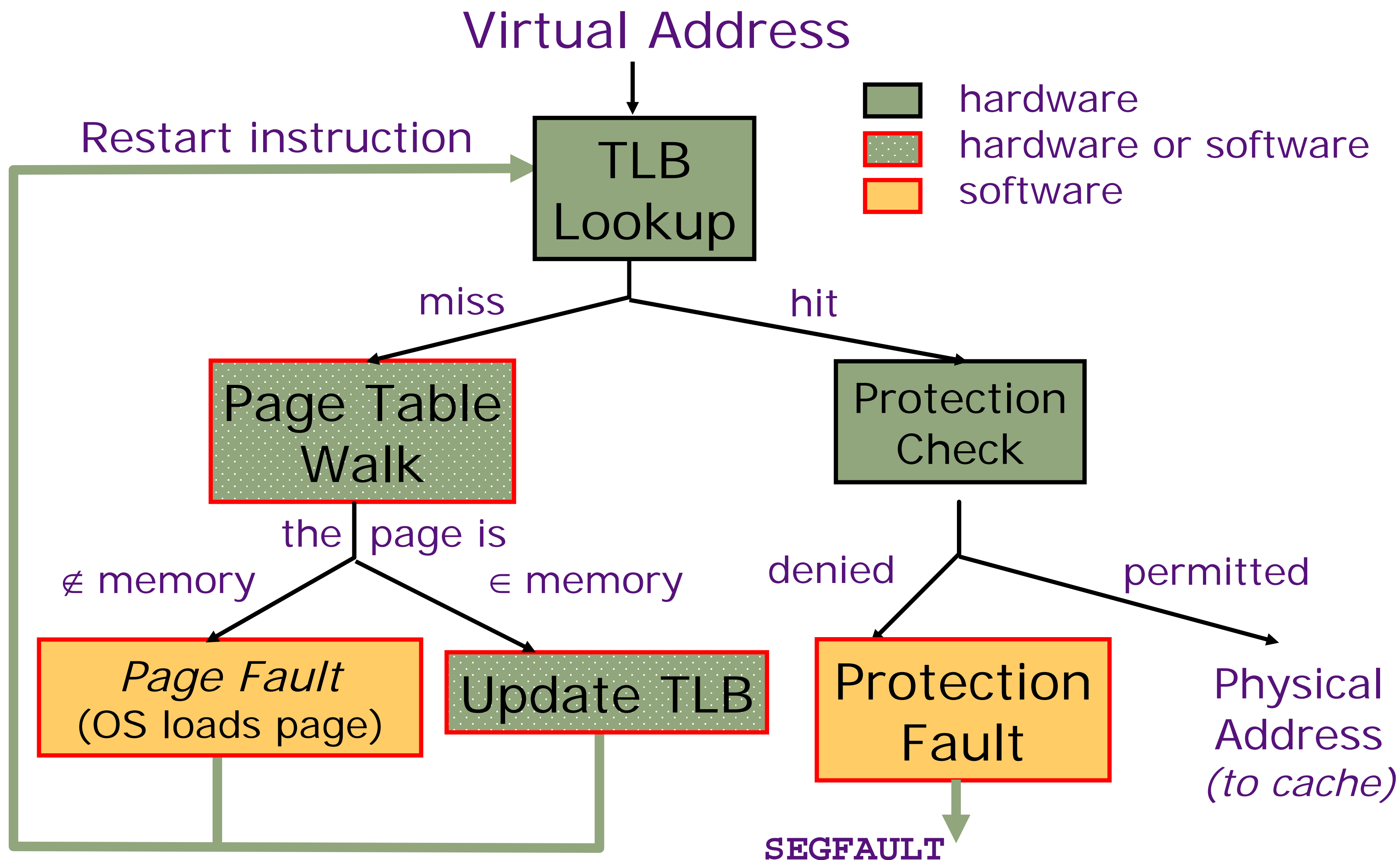
Thank you !

Modern Virtual Memory Systems

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

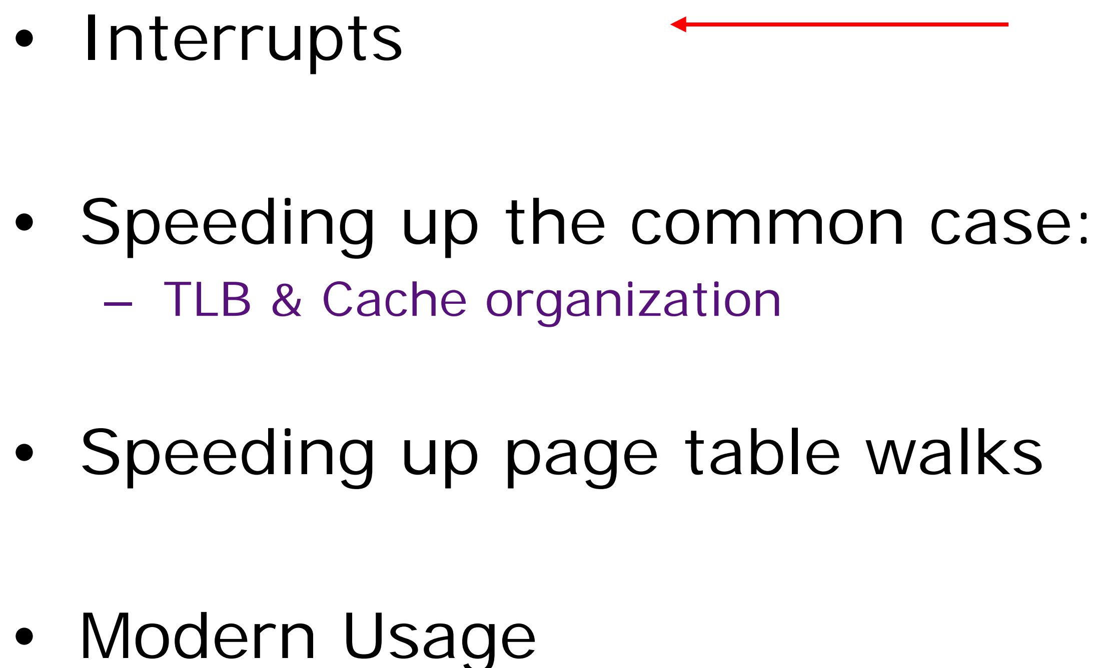
Based on the material prepared by
Arvind and Krste Asanovic

Address Translation: *putting it all together*



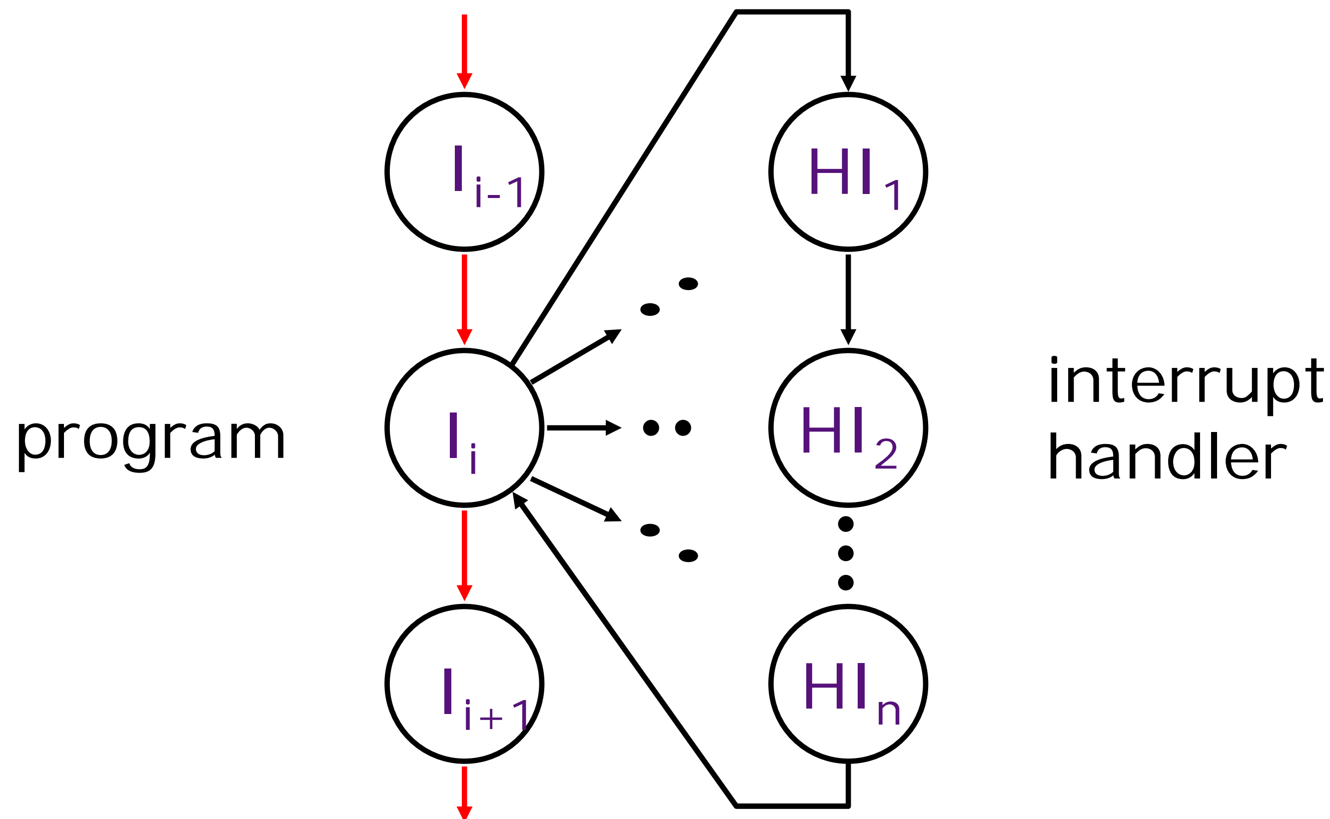
CSAIL

Topics



- Interrupts
- Speeding up the common case:
  - TLB & Cache organization
- Speeding up page table walks
- Modern Usage

Interrupts:
altering the normal flow of control



An *external or internal event* that needs to be processed by another (system) program. The event is usually unexpected or rare from program's point of view.

Causes of Interrupts

Interrupt: an *event* that requests the attention of the processor

- Asynchronous: an *external event*
  - input/output device service-request
  - timer expiration
  - power disruptions, hardware failure
- Synchronous: an *internal event (a.k.a exceptions)*
  - undefined opcode, privileged instruction
  - arithmetic overflow, FPU exception
  - misaligned memory access
  - *virtual memory exceptions:* page faults, TLB misses, protection violations
  - *traps:* system calls, e.g., jumps into kernel

Asynchronous Interrupts:
invoking the interrupt handler

- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*

- When the processor decides to process the interrupt
  - It stops the current program at instruction I_i, completing all the instructions up to I_{i-1} (*precise interrupt)*
  - It saves the PC of instruction I_i in a special register (EPC)
  - It disables interrupts and transfers control to a designated interrupt handler running in the kernel mode

CSAIL

Interrupt Handler

- Saves EPC before enabling interrupts to allow nested interrupts ⇒
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the cause of the interrupt
- Uses a special indirect jump instruction RFE (*return-from-exception*) which
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state

Synchronous Interrupts

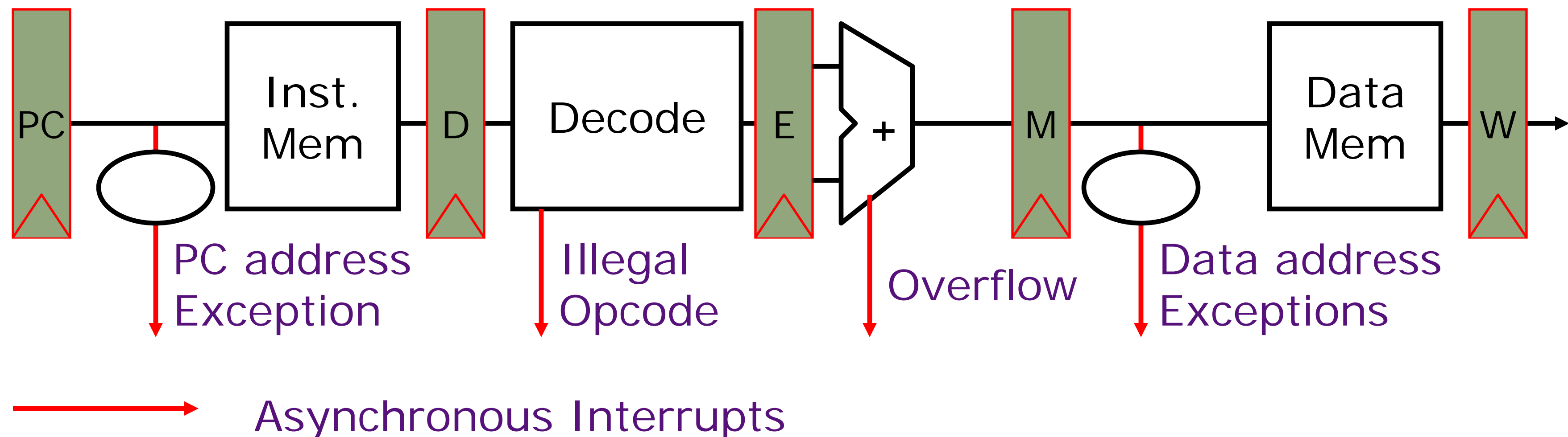
- A synchronous interrupt (exception) is caused by a *particular instruction*

- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - requires undoing the effect of one or more partially executed instructions

- In case of a trap (system call), the instruction is considered to have been completed
  - a special jump instruction involving a change to privileged kernel mode

CSAIL

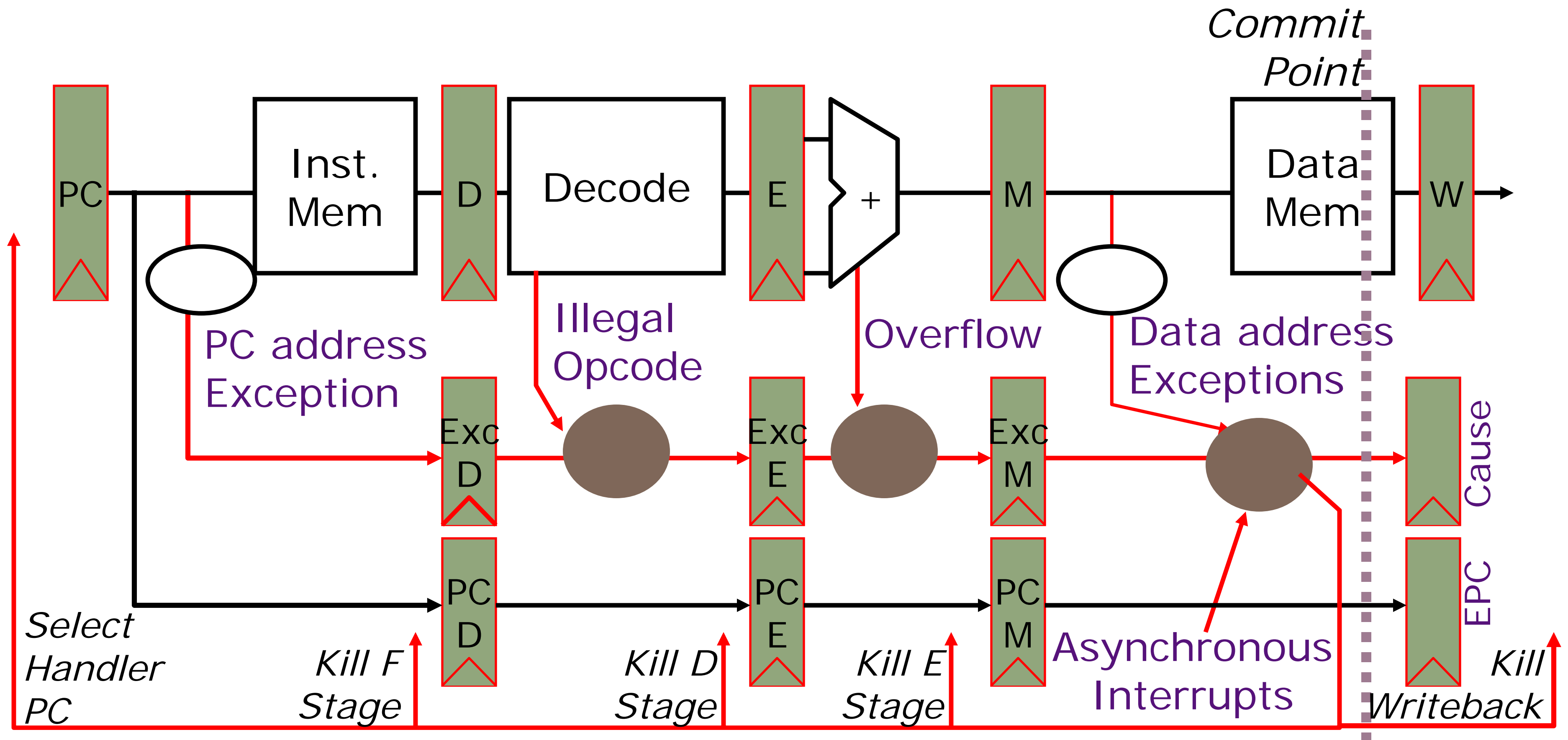
Exception Handling 5-Stage Pipeline



- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

CSAIL

Exception Handling 5-Stage Pipeline



CSAIL

Exception Handling 5-Stage Pipeline

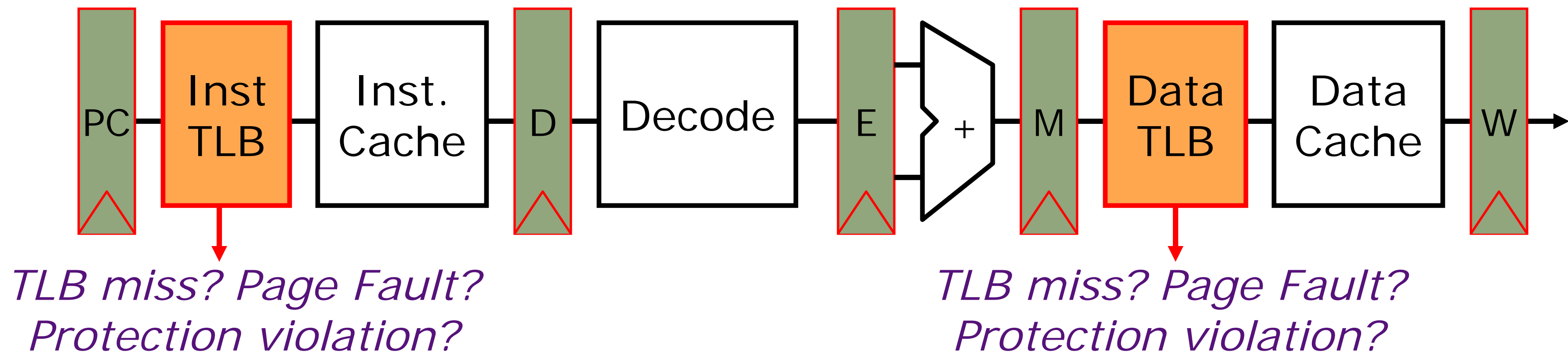
- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

CSAIL

Topics

- Interrupts
- Speeding up the common case:
  - TLB & Cache organization
- Speeding up page table walks
- Modern Usage

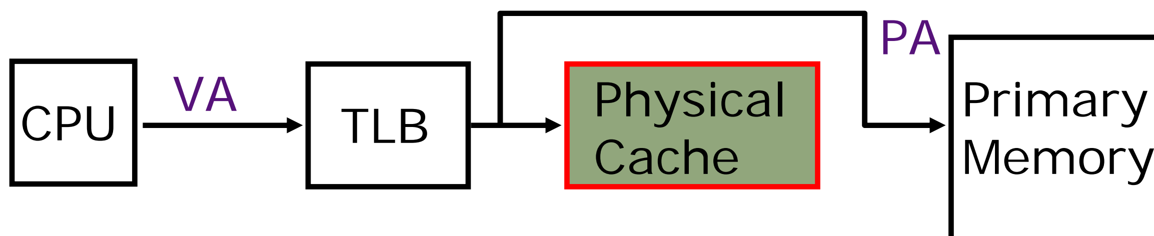
Address Translation in CPU Pipeline



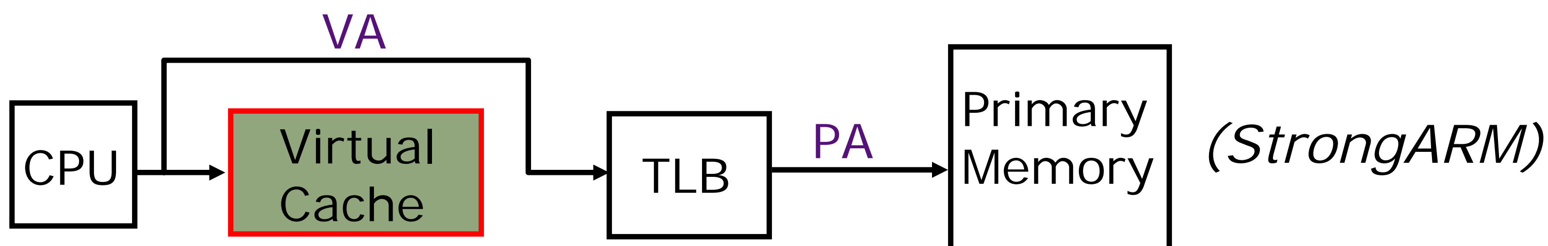
- Software handlers need a *restartable* exception on page fault or protection violation
- Handling a TLB miss needs a *hardware* or *software* mechanism to refill TLB
- Need mechanisms to cope with the additional latency of a TLB:
  - slow down the clock
  - pipeline the TLB and cache access
  - virtual address caches
  - parallel TLB/cache access

CSAIL

Virtual Address Caches



Alternative: place the cache before the TLB

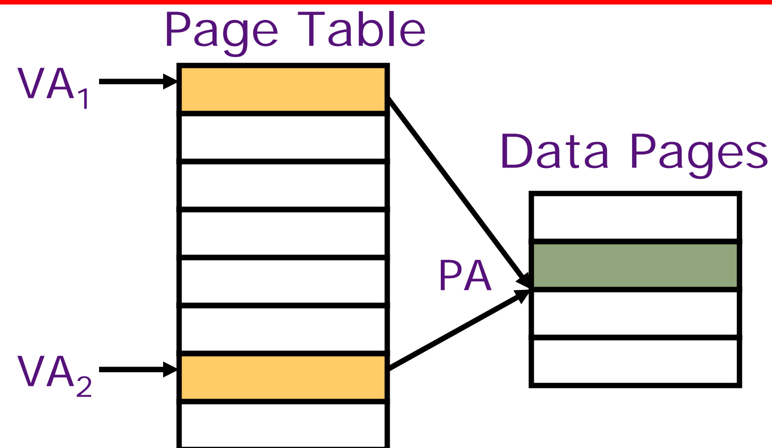


(StrongARM)

- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- *aliasing problems* due to the sharing of pages (-)

CSAIL

Aliasing in Virtual-Address Caches



Two virtual pages share one physical page

Tag	Data
VA_1	1st Copy of Data at PA
VA_2	2nd Copy of Data at PA

Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

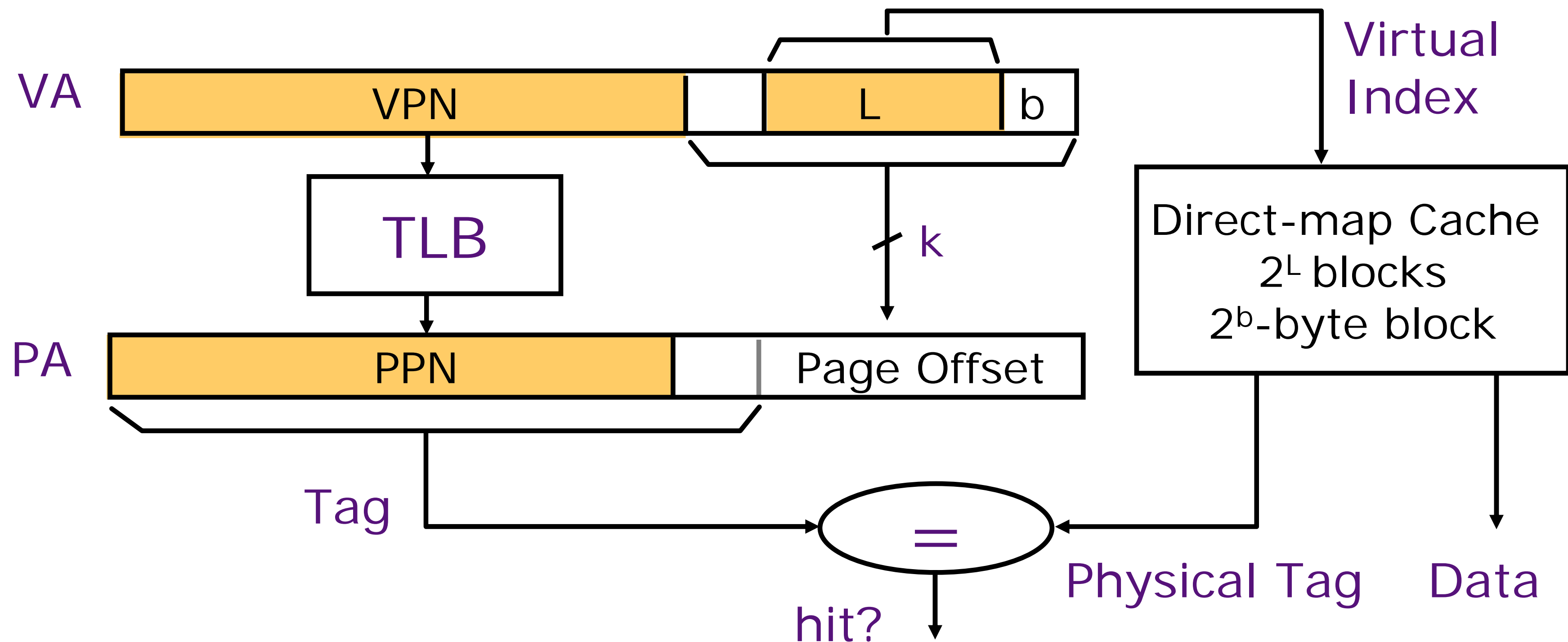
General Solution: *Disallow aliases to coexist in cache*

Software (i.e., OS) solution for direct-mapped cache

VAs of shared pages must agree in cache index bits; this ensures all VAs accessing same PA will conflict in direct-mapped cache (early SPARCs)

CSAIL

Concurrent Access to TLB & Cache



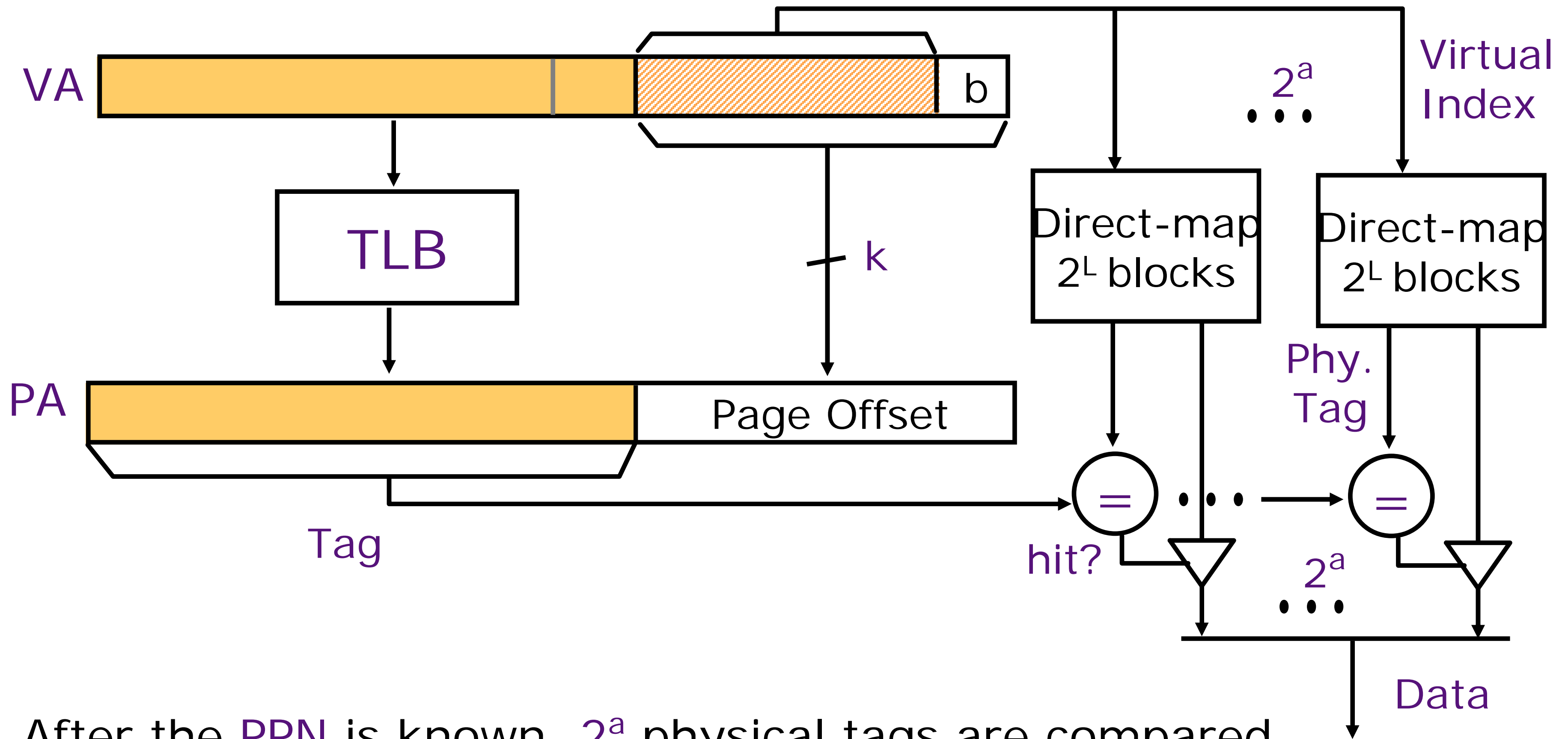
Index L is available without consulting the TLB

⇒ *cache and TLB accesses can begin simultaneously*

Tag comparison is made after both accesses are completed

Cases: $L + b = k$ $L + b < k$ $L + b > k$

Virtual-Index Physical-Tag Caches: Associative Organization

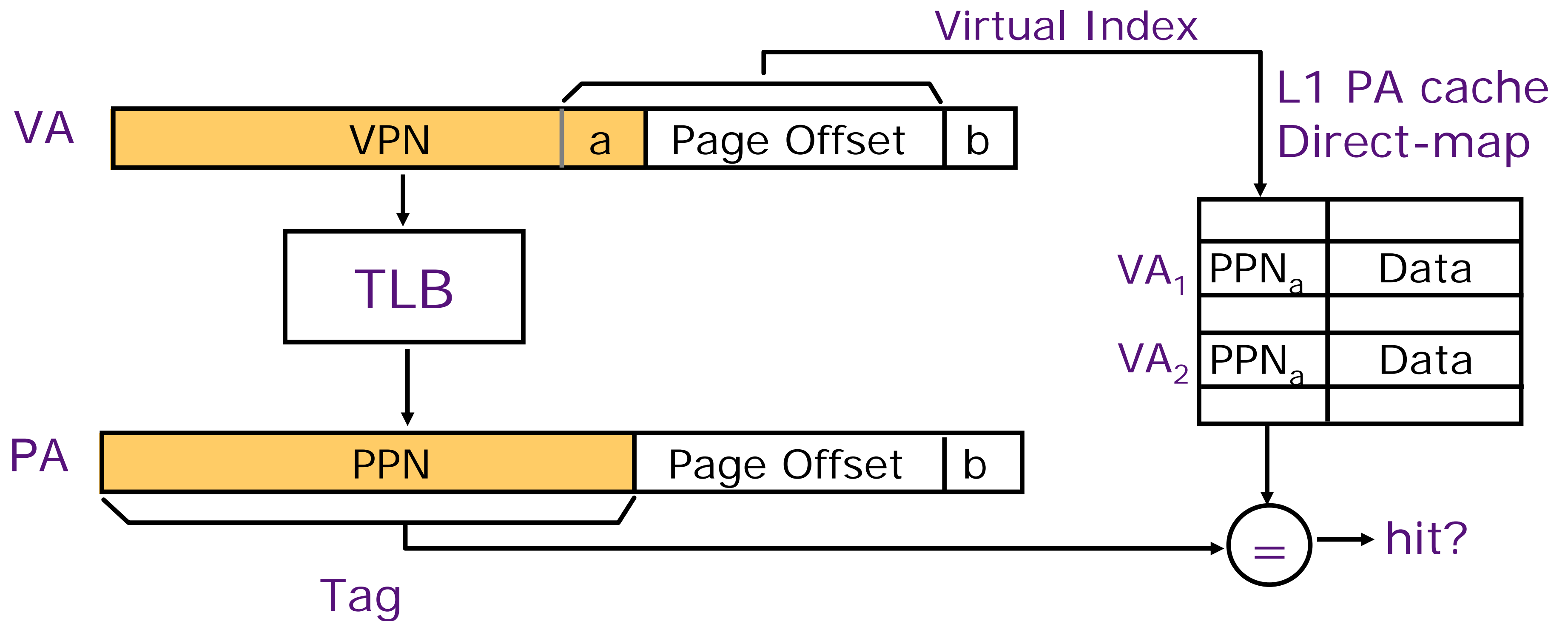


After the PPN is known, 2^a physical tags are compared

Is this scheme realistic?

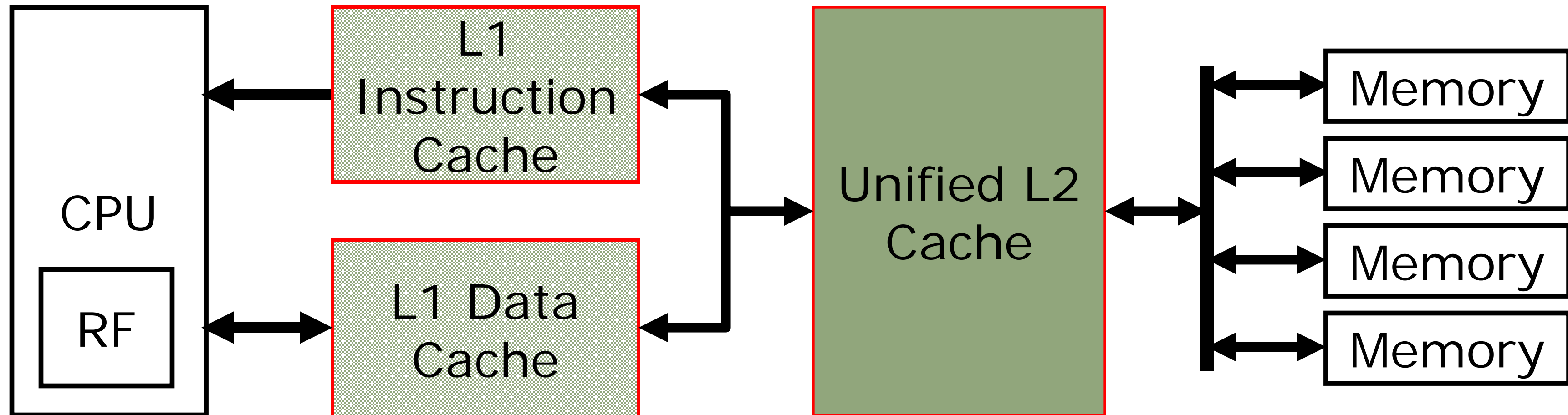
Concurrent Access to TLB & Large L1

The problem with L1 > Page size



Can VA$_1$ *and* VA$_2$ *both map to* PA *?*

A solution via Second Level Cache

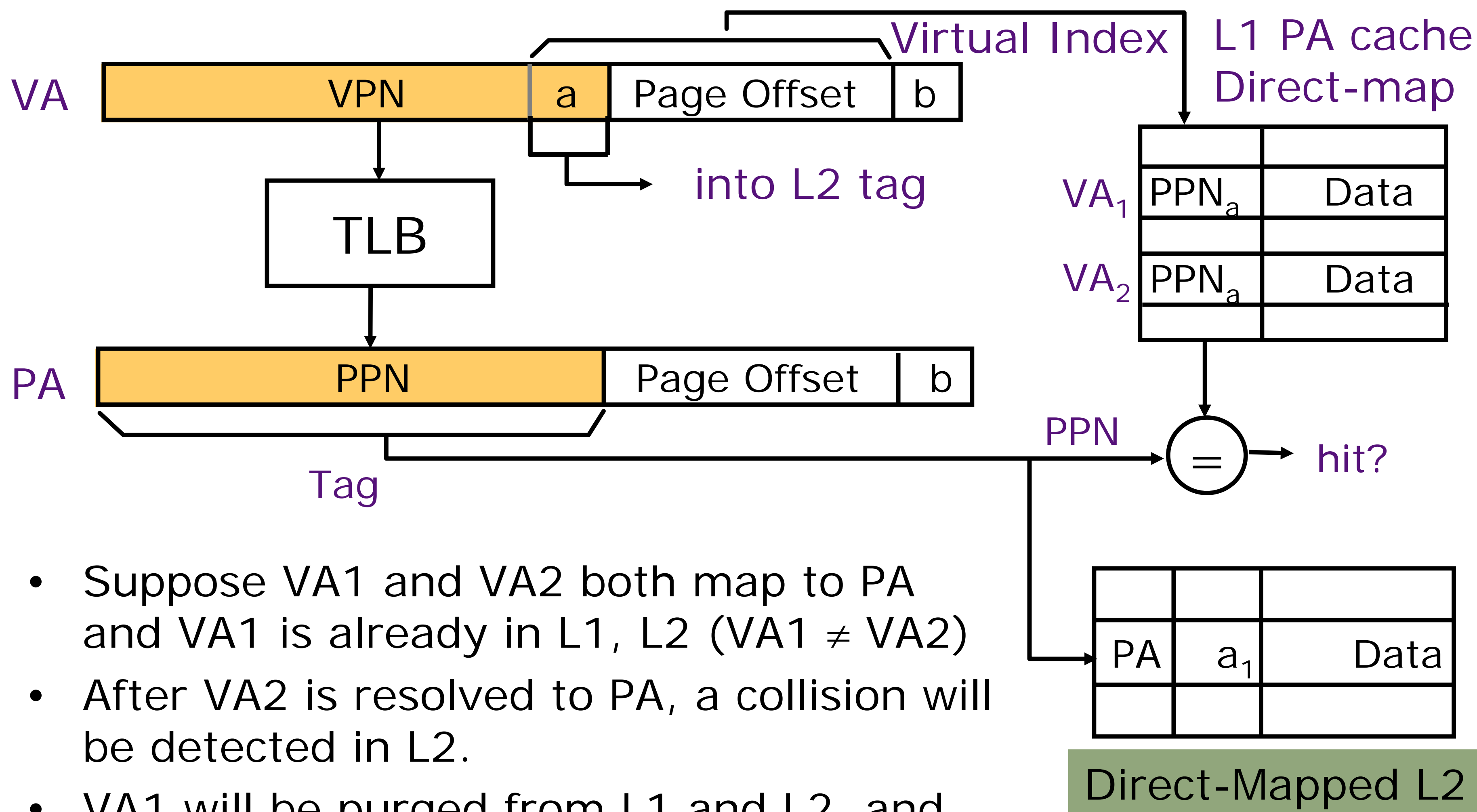


Usually a common L2 cache backs up both Instruction and Data L1 caches

L2 is "inclusive" of both Instruction and Data caches

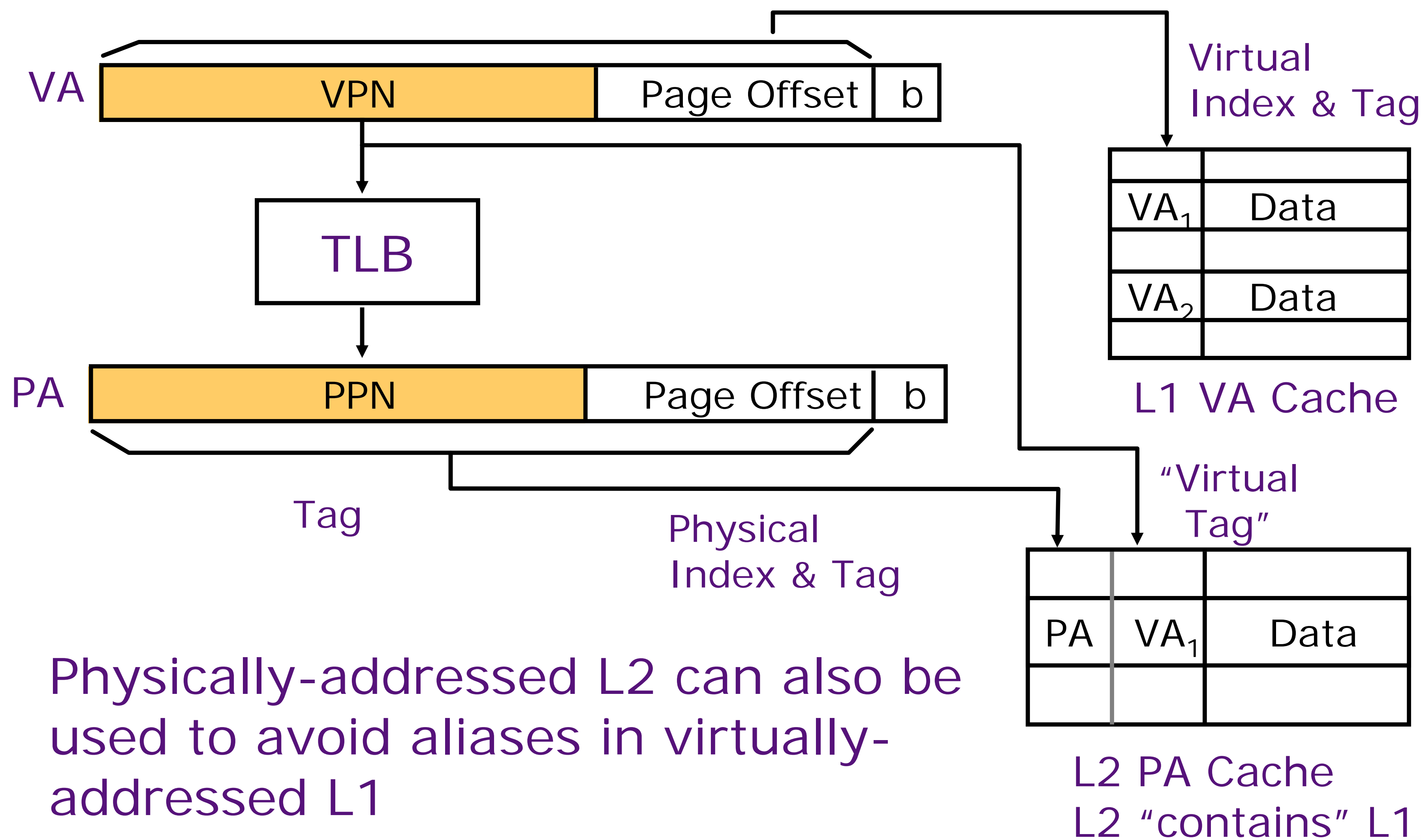
CSAIL

Anti-Aliasing Using L2: *MIPS R10000*



- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 ≠ VA2)
- After VA2 is resolved to PA, a collision will be detected in L2.
- VA1 will be purged from L1 and L2, and VA2 will be loaded ⇒ *no aliasing !*

Virtually-Addressed L1:
Anti-Aliasing using L2



Physically-addressed L2 can also be used to avoid aliases in virtually-addressed L1

Five-minute break to stretch your legs

Topics

- Interrupts
- Speeding up the common case:
  - TLB & Cache organization
- Speeding up page table walks
- Modern Usage

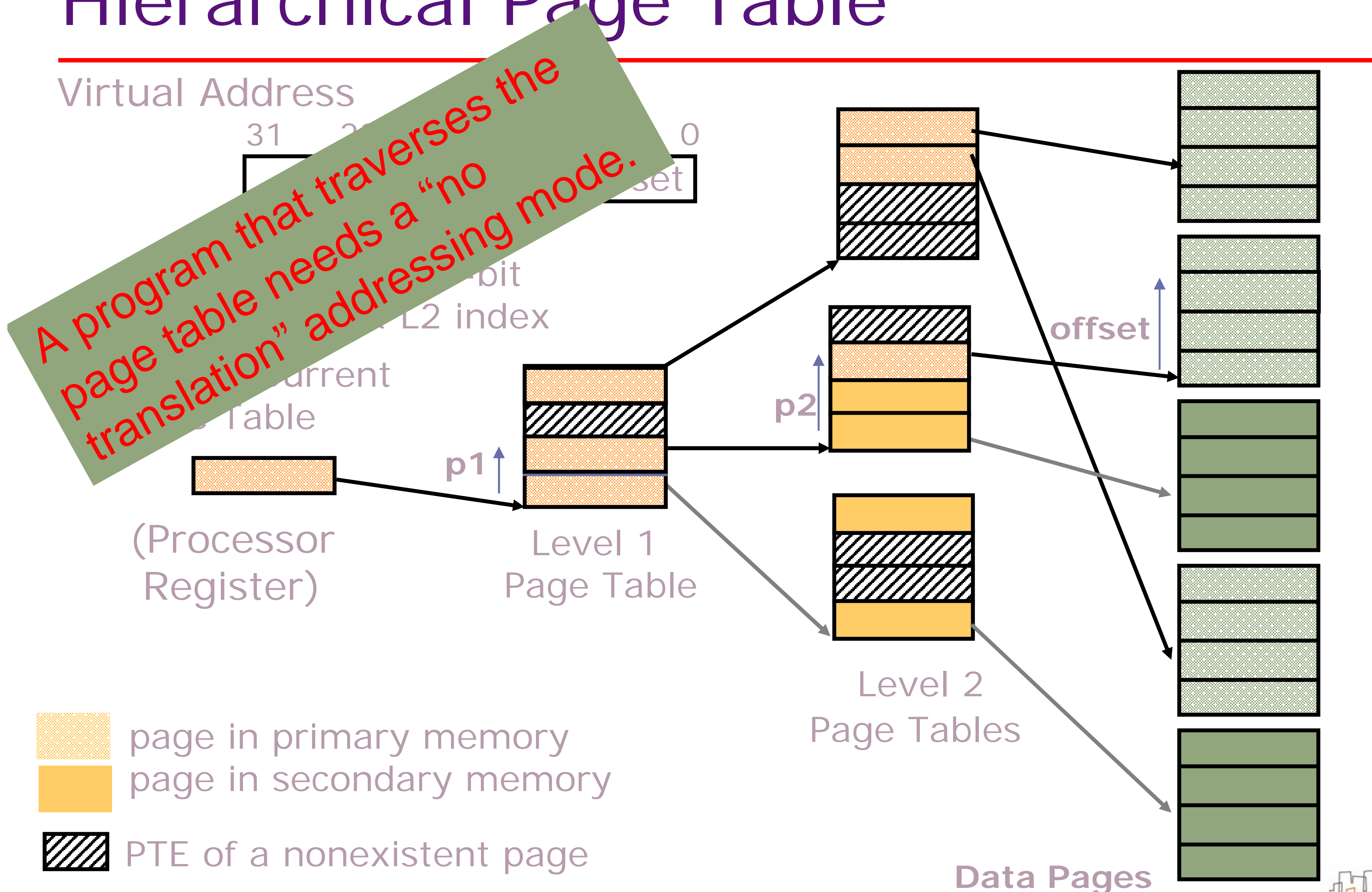
Page Fault Handler

- When the referenced page is not in DRAM:
  - The missing page is located (or created)
  - It is brought in from disk, and page table is updated

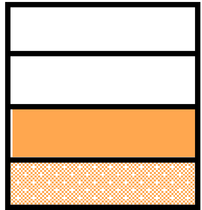
    Another job may be run on the CPU while the first job waits for the requested page to be read from disk
  - If no free pages are left, a page is swapped out

    Pseudo-LRU replacement policy
- Since it takes a long time to transfer a page (msecs), page faults are handled completely in software by the OS
  - Untranslated addressing mode is essential to allow kernel to access page tables

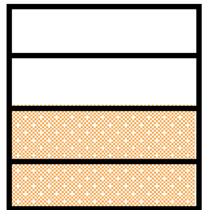
Hierarchical Page Table



Swapping a Page of a Page Table



A PTE in primary memory contains
primary or secondary memory addresses



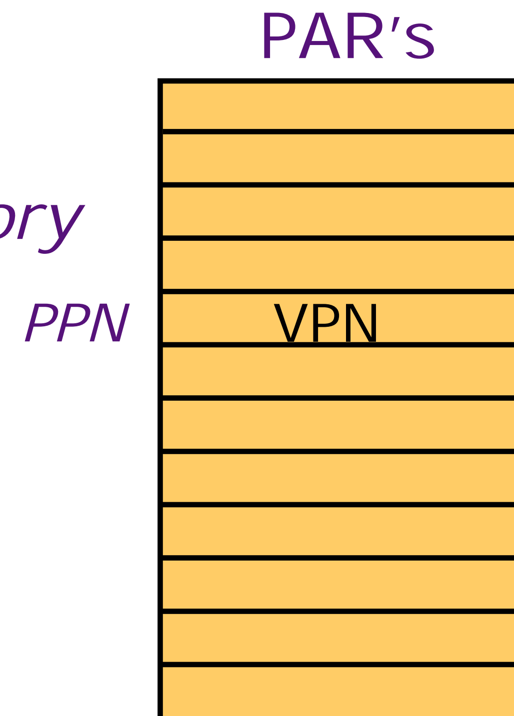
A PTE in secondary memory contains
only secondary memory addresses

⇒ a page of a PT can be swapped out only if none its PTE's point to pages in the primary memory

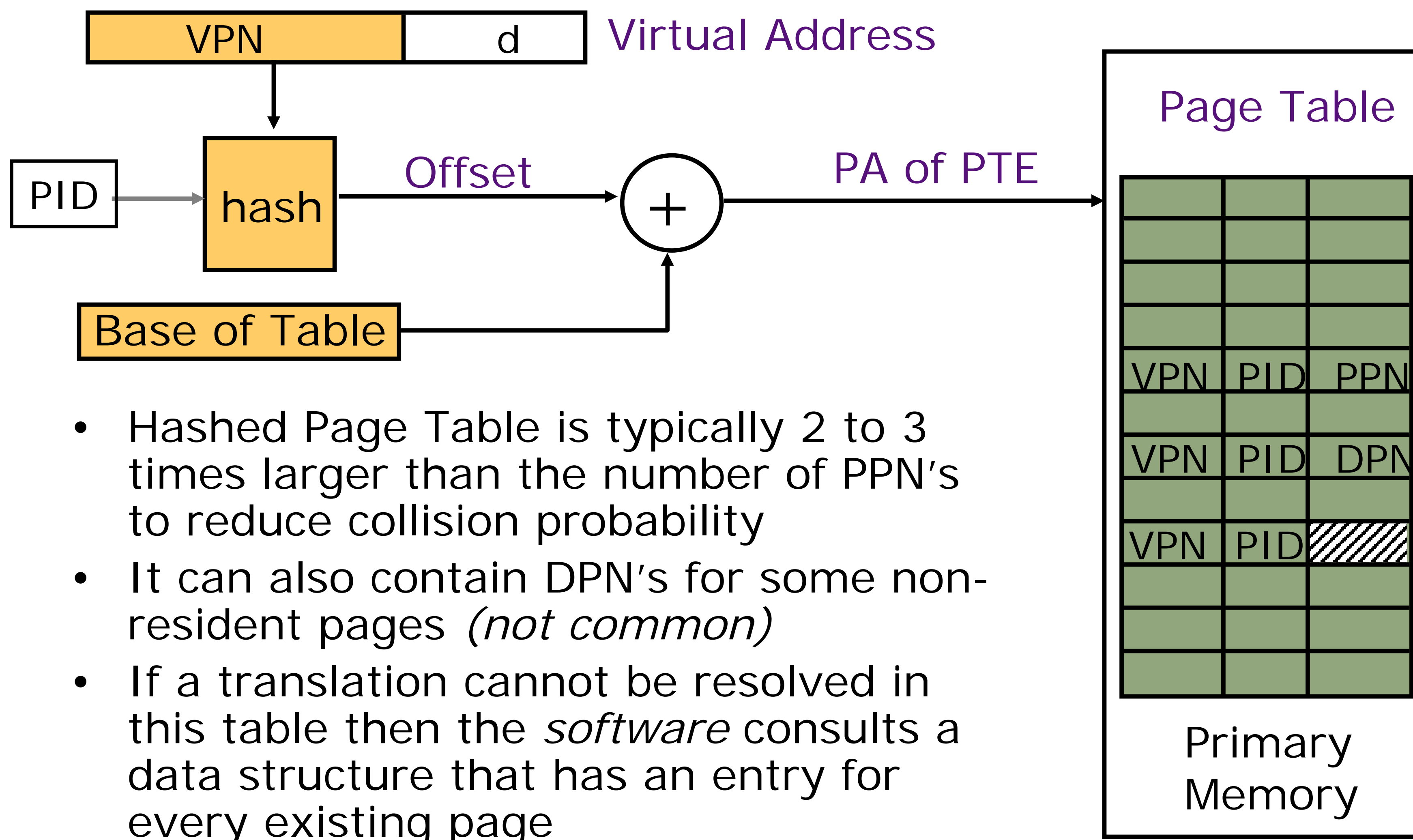
*Why?*_________________________________

Atlas Revisited

- One PAR for each physical page
- PAR′s contain the VPN′s of the pages *resident in primary memory*
- *Advantage:* The size is proportional to the size of the primary memory
- *What is the disadvantage ?*



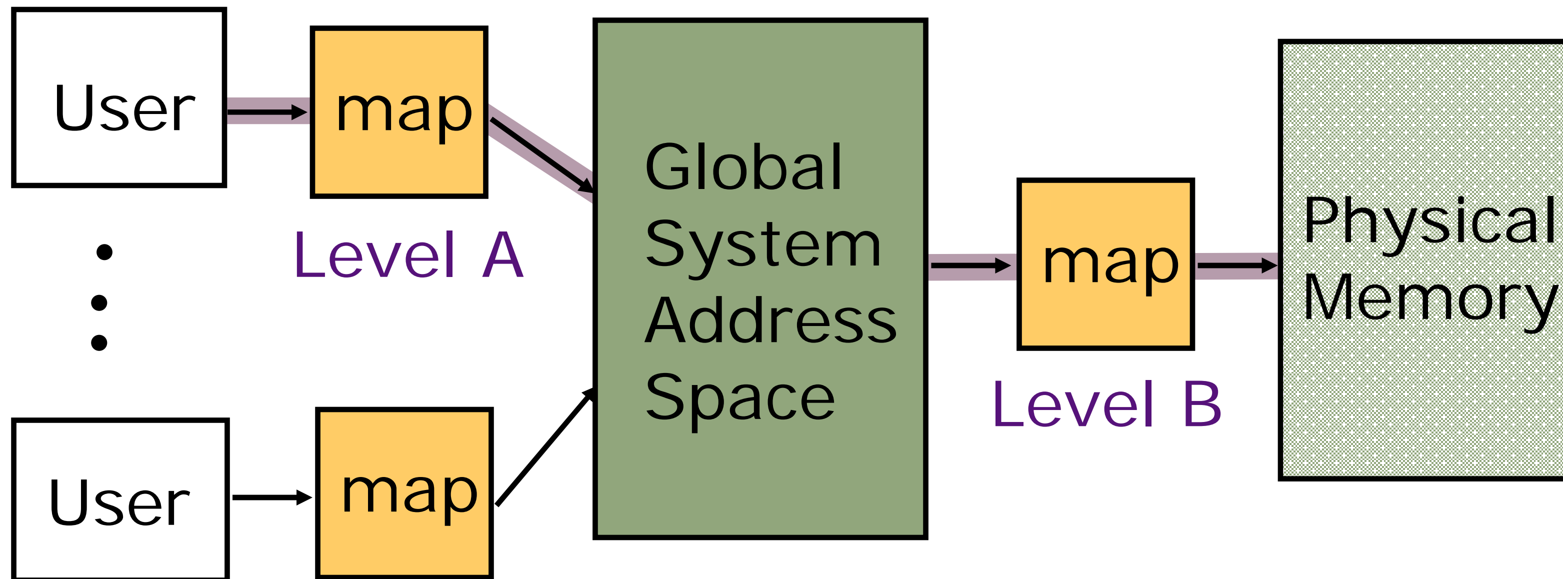
Hashed Page Table:
Approximating Associative Addressing



- Hashed Page Table is typically 2 to 3 times larger than the number of PPN's to reduce collision probability
- It can also contain DPN's for some non-resident pages *(not common)*
- If a translation cannot be resolved in this table then the *software* consults a data structure that has an entry for every existing page

CSAIL

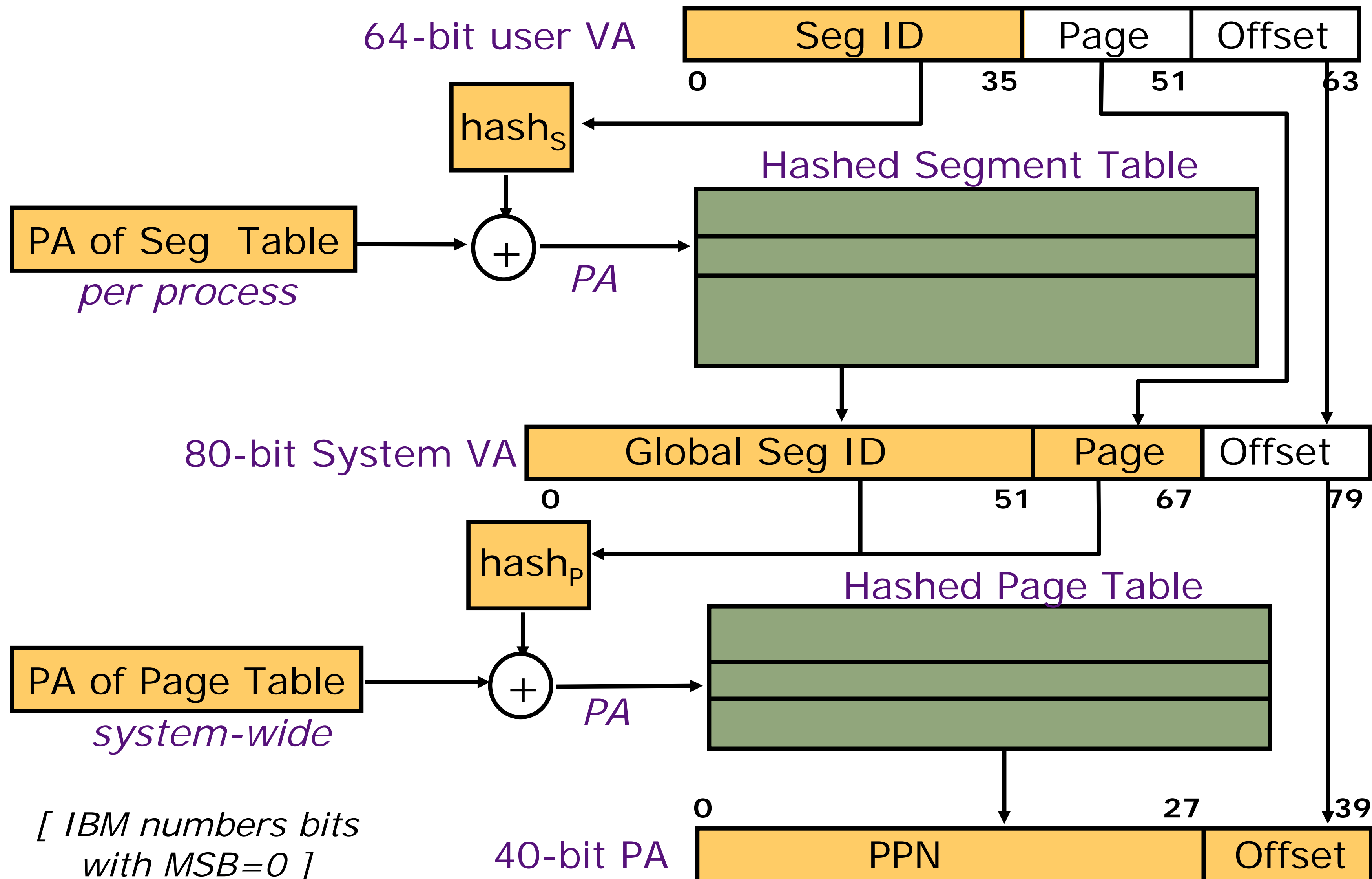
Global System Address Space



- Level A maps users' address spaces into the global space providing privacy, protection, sharing etc.
- Level B provides demand-paging for the large global system address space
- Level A and Level B translations may be kept in separate TLB's

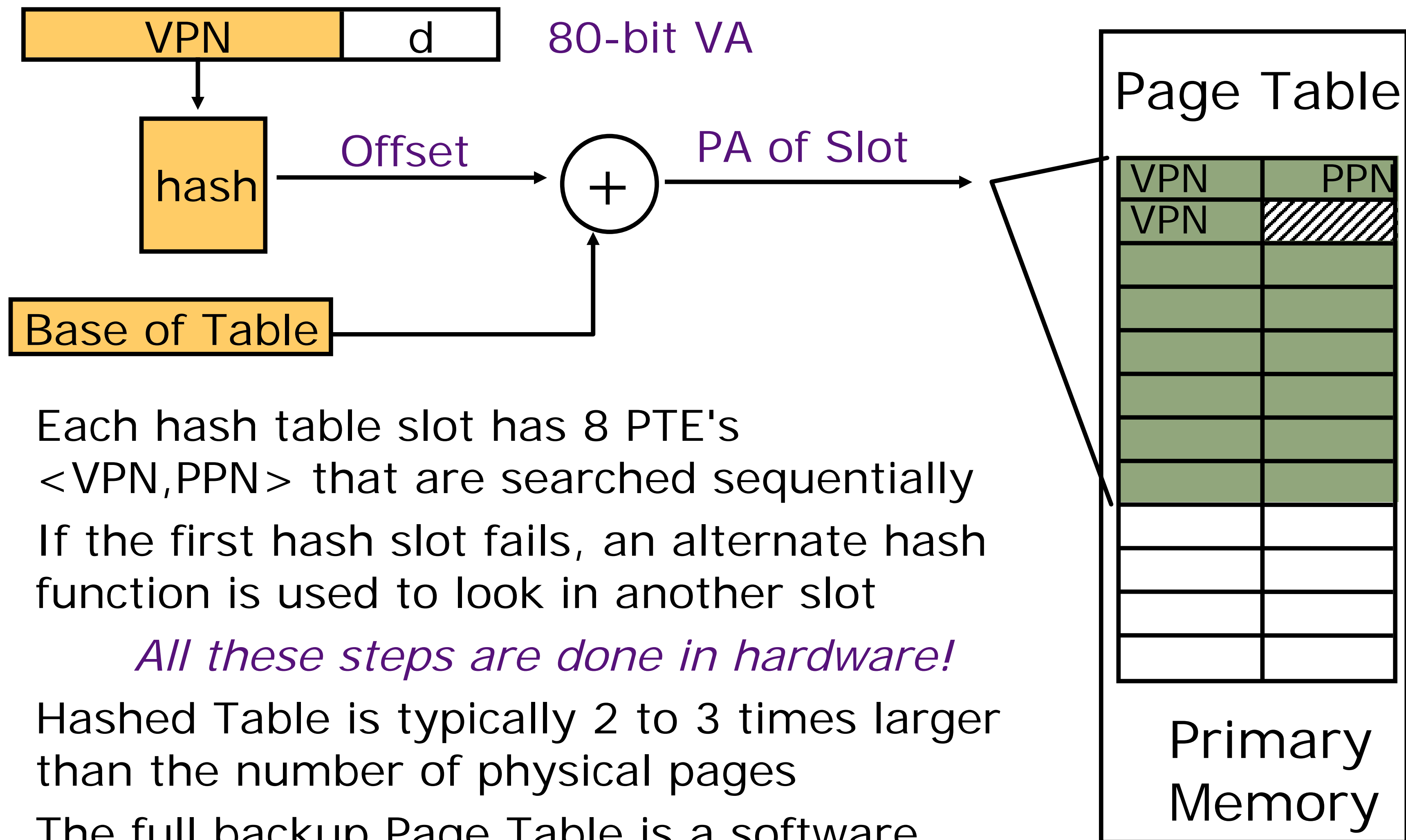
Hashed Page Table Walk:

PowerPC Two-level, Segmented Addressing



[IBM numbers bits with MSB=0]

Power PC: Hashed Page Table



- Each hash table slot has 8 PTE's <VPN,PPN> that are searched sequentially
- If the first hash slot fails, an alternate hash function is used to look in another slot

All these steps are done in hardware!

- Hashed Table is typically 2 to 3 times larger than the number of physical pages
- The full backup Page Table is a software data structure

CSAIL

Virtual Memory Use Today - 1

- Desktops/servers have full demand-paged virtual memory
  - Portability between machines with different memory sizes
  - Protection between multiple users or multiple tasks
  - Share small physical memory among active tasks
  - Simplifies implementation of some OS features
- Vector supercomputers have translation and protection but not demand-paging
  (Crays: base&bound, Japanese: pages)
  - Don't waste expensive CPU time thrashing to disk (make jobs fit in memory)
  - Mostly run in batch mode (run set of jobs that fits in memory)
  - Difficult to implement restartable vector instructions

Virtual Memory Use Today - 2

- Most embedded processors and DSPs provide physical addressing only
  - Can't afford area/speed/power budget for virtual memory support
  - Often there is no secondary storage to swap to!
  - Programs custom written for particular memory configuration in product
  - Difficult to implement restartable instructions for exposed architectures

Given the software demands of modern embedded devices (e.g., cell phones, PDAs) all this may change in the near future!

Thank you !

Branch Prediction and Speculative Execution

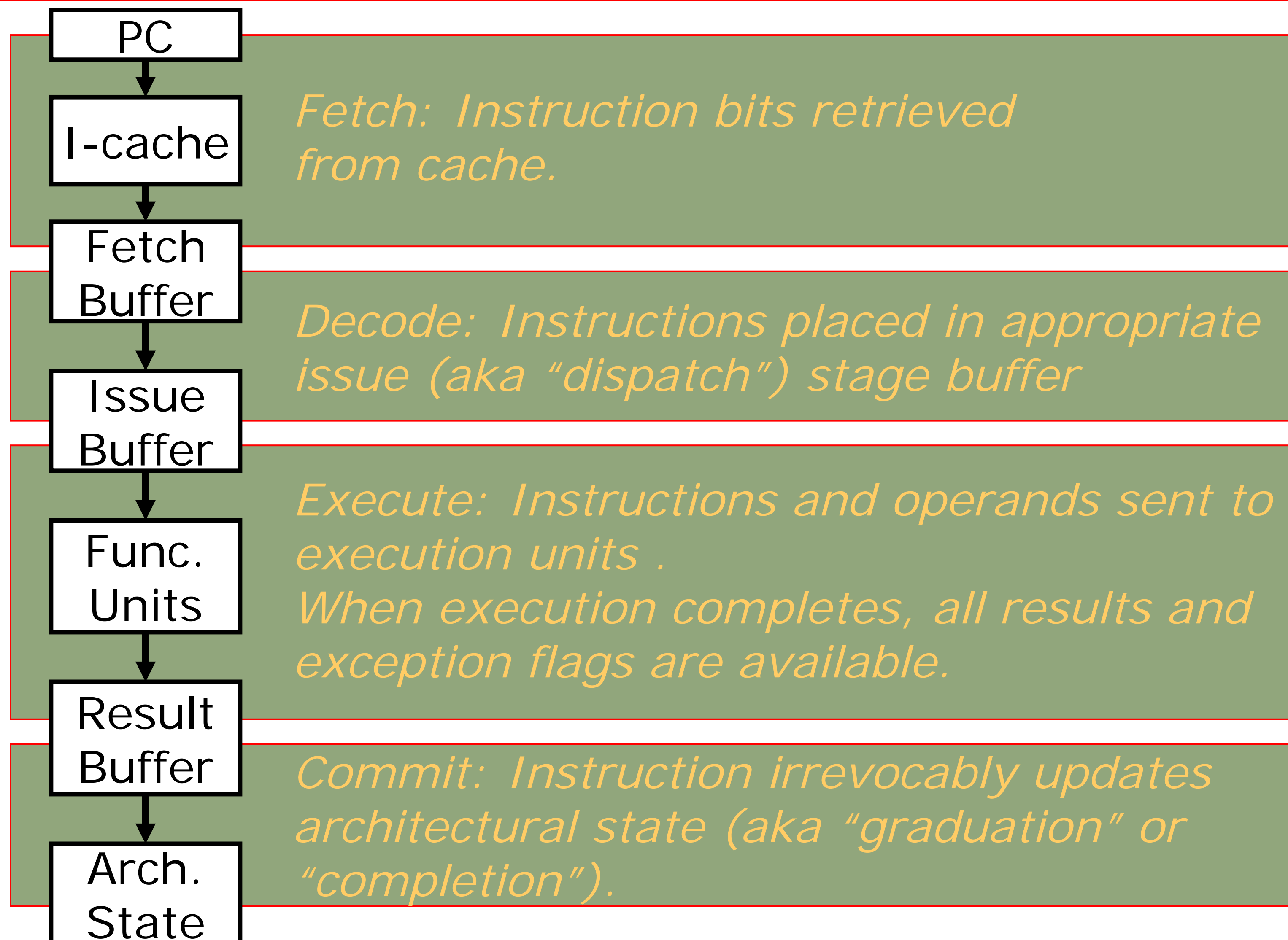
Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Based on the material prepared by Krste Asanovic and Arvind

Outline

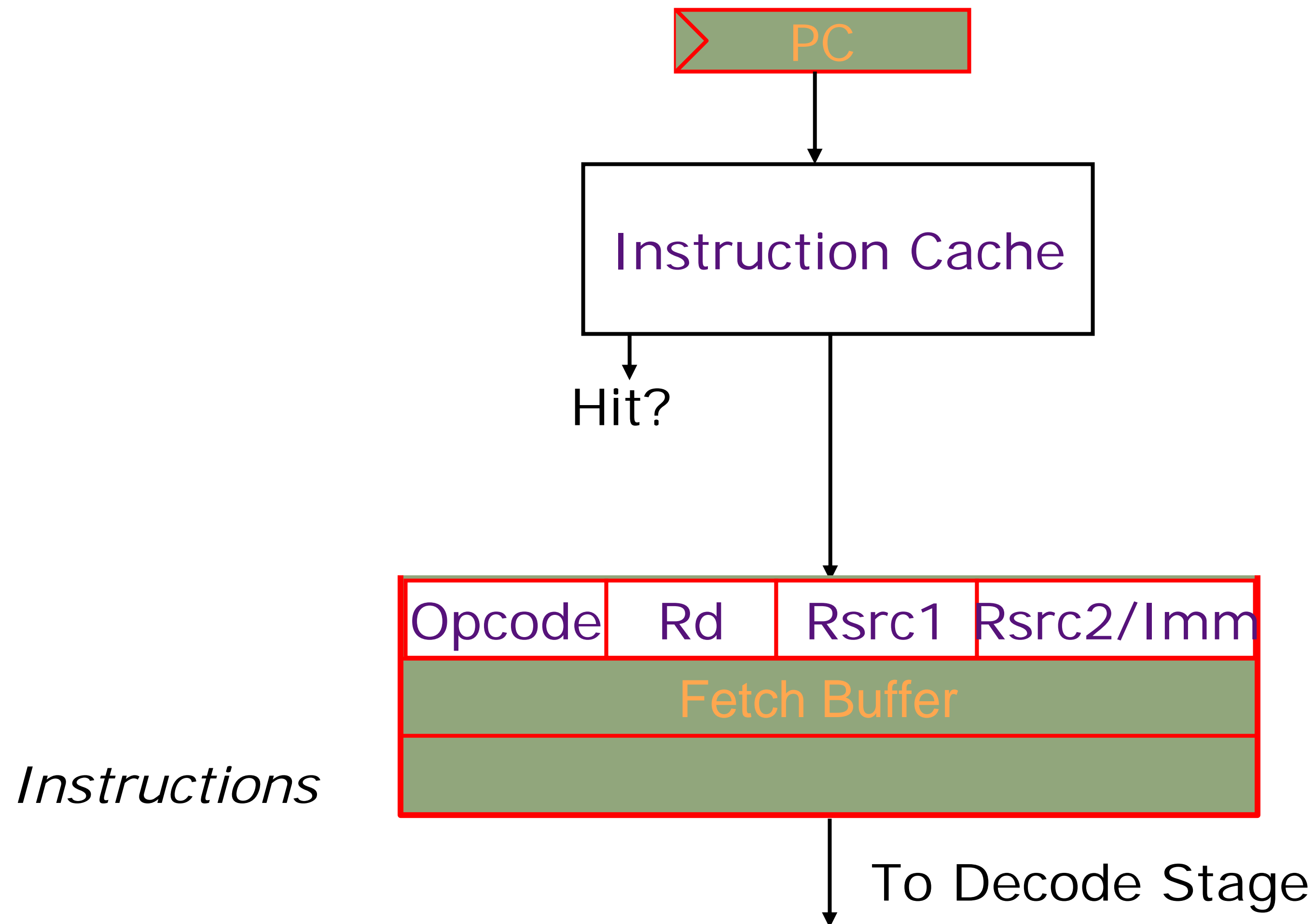
- Control transfer penalty
- Branch prediction schemes
- Branch misprediction recovery schemes

Phases of Instruction Execution



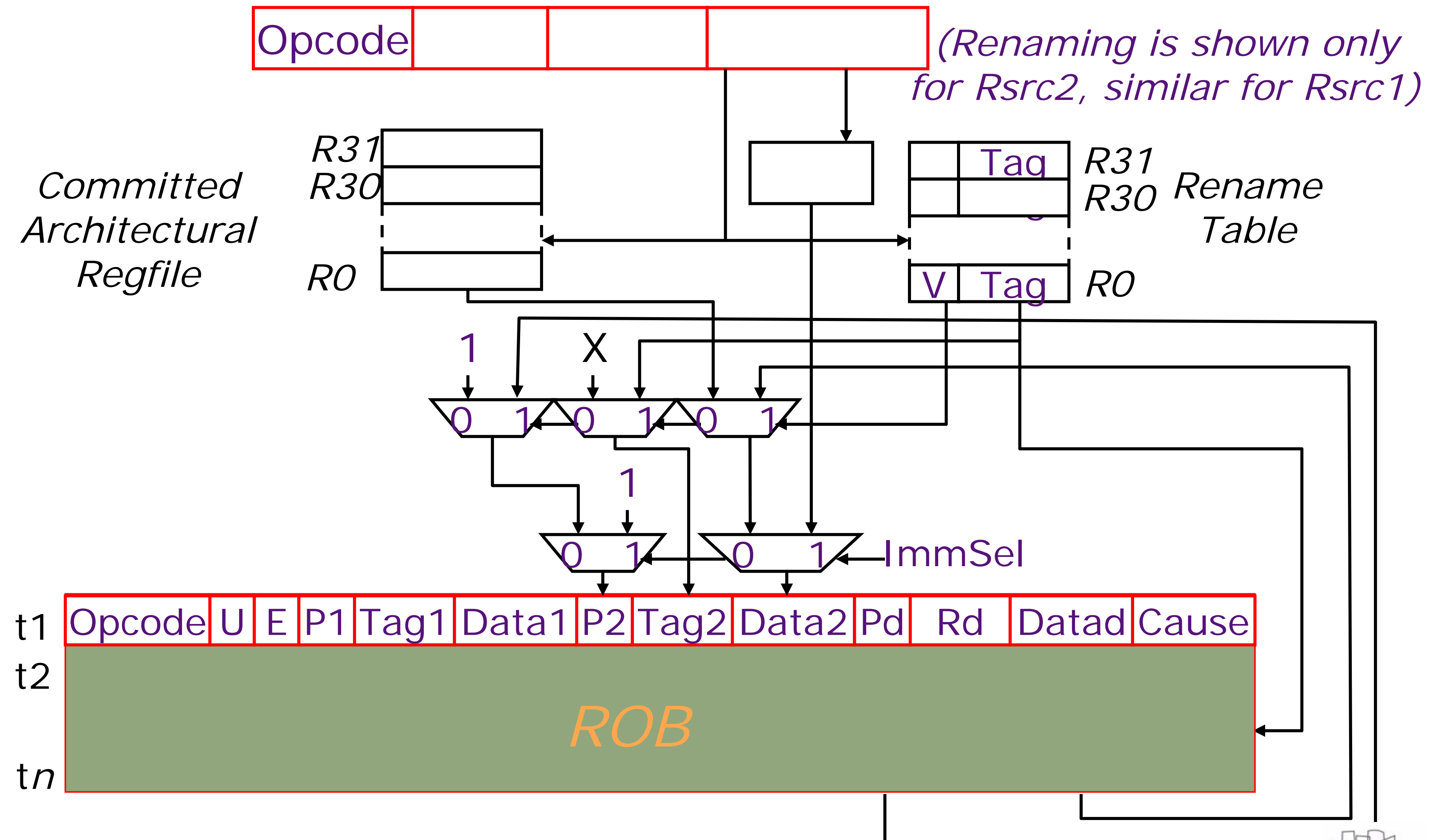
CSAIL

Fetch Stage



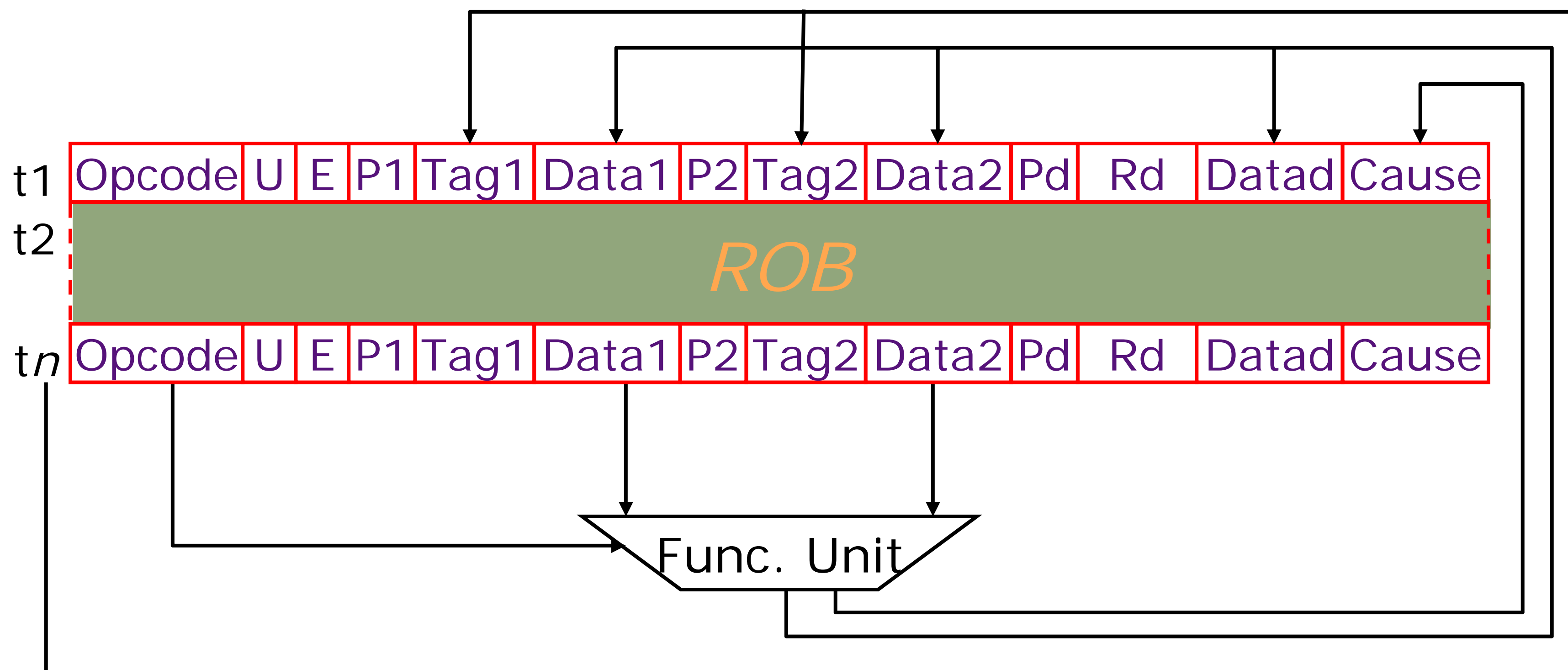
CSAIL

Decode & Rename Stage



Execute Stage

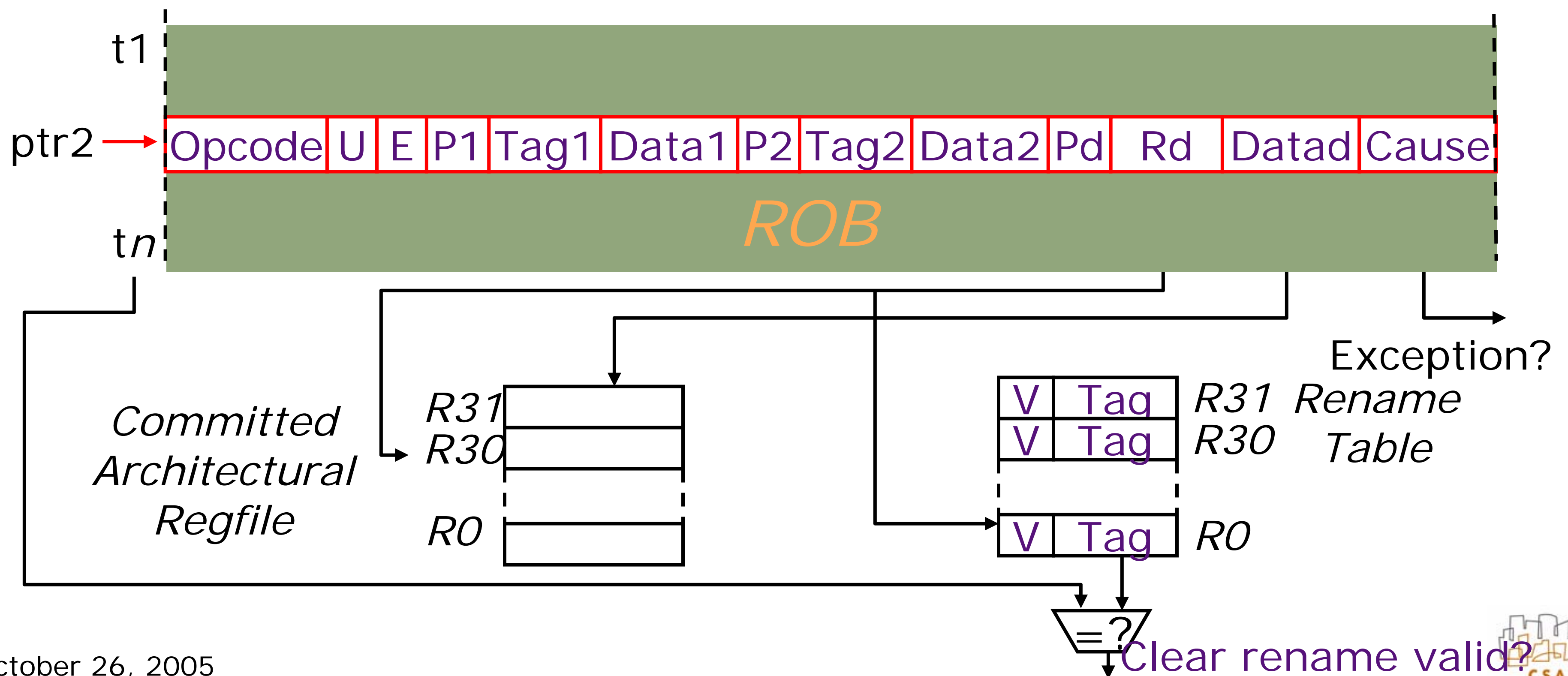
- Arbiter selects one ready instruction (P1=1 AND P2=1) to execute
- Instruction reads operands from ROB, executes, and broadcasts tag and result to waiting instructions in ROB. Also saves result and exception flags for commit.



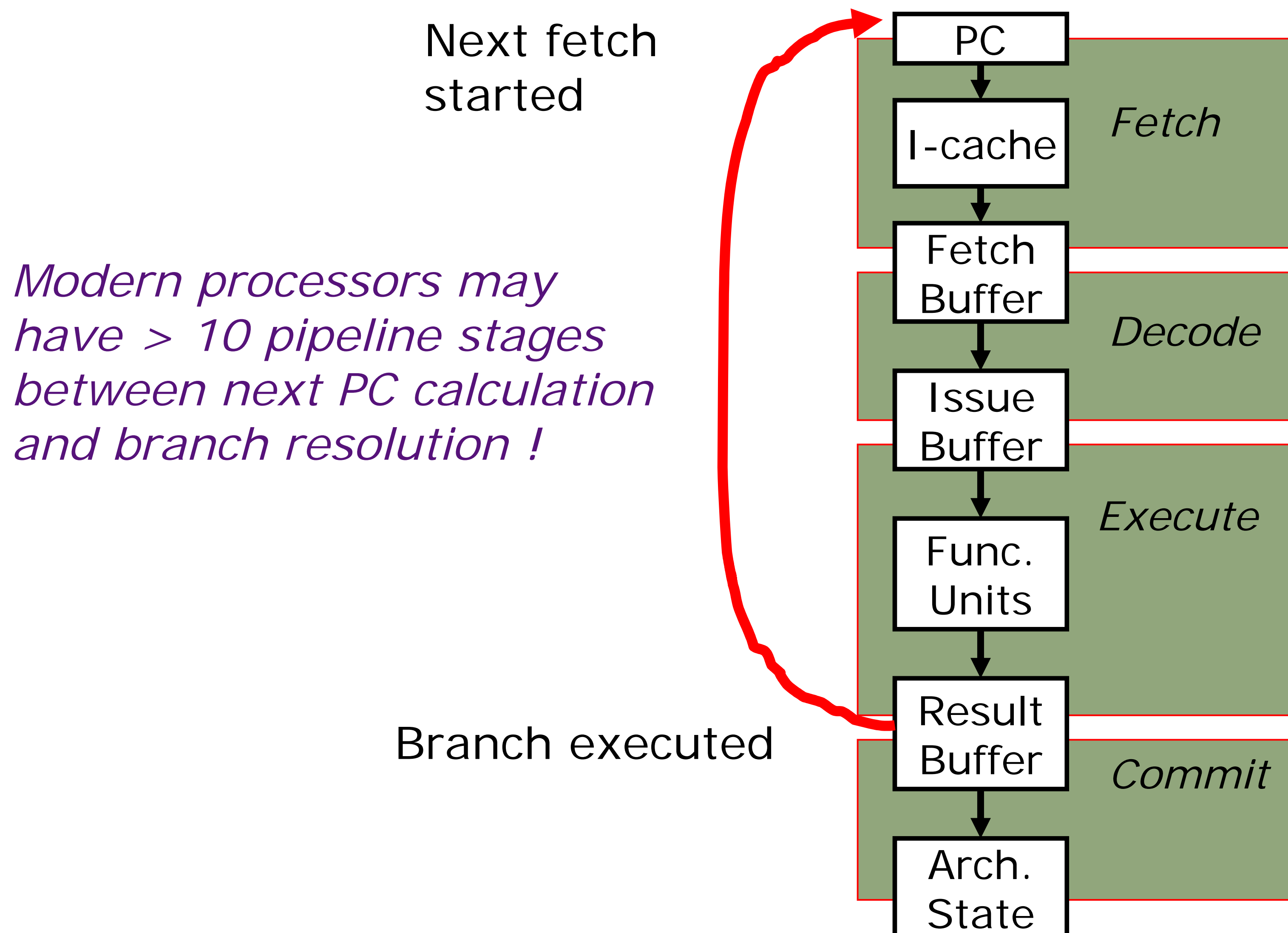
CSAIL

Commit Stage

- When instruction at ptr2 (commit point) has completed, write back result to architectural state and check for exceptions
- Check if rename table entry for architectural register written matches tag, if so, clear valid bit in rename table



Branch Penalty



Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution !

Average Run-Length between Branches

Average dynamic instruction mix from SPEC92:

	SPECint92	SPECfp92
ALU	39 %	13 %
FPU Add		20 %
FPU Mult		13 %
load	26 %	23 %
store	9 %	9 %
branch	16 %	8 %
other	10 %	12 %

SPECint92: *compress, eqntott, espresso, gcc, li*
SPECfp92: *doduc, ear, hydro2d, mdijdp2, su2cor*

What is the average *run length* between branches

CSAIL

Reducing Control Transfer Penalties

Software solution

- *loop unrolling*
  Increases the run length
- *instruction scheduling*
  Compute the branch condition as early as possible (limited)

Hardware solution

- *delay slots*
  replaces pipeline bubbles with useful work (requires software cooperation)
- *branch prediction & speculative execution* of instructions beyond the branch

CSAIL

MIPS Branches and Jumps

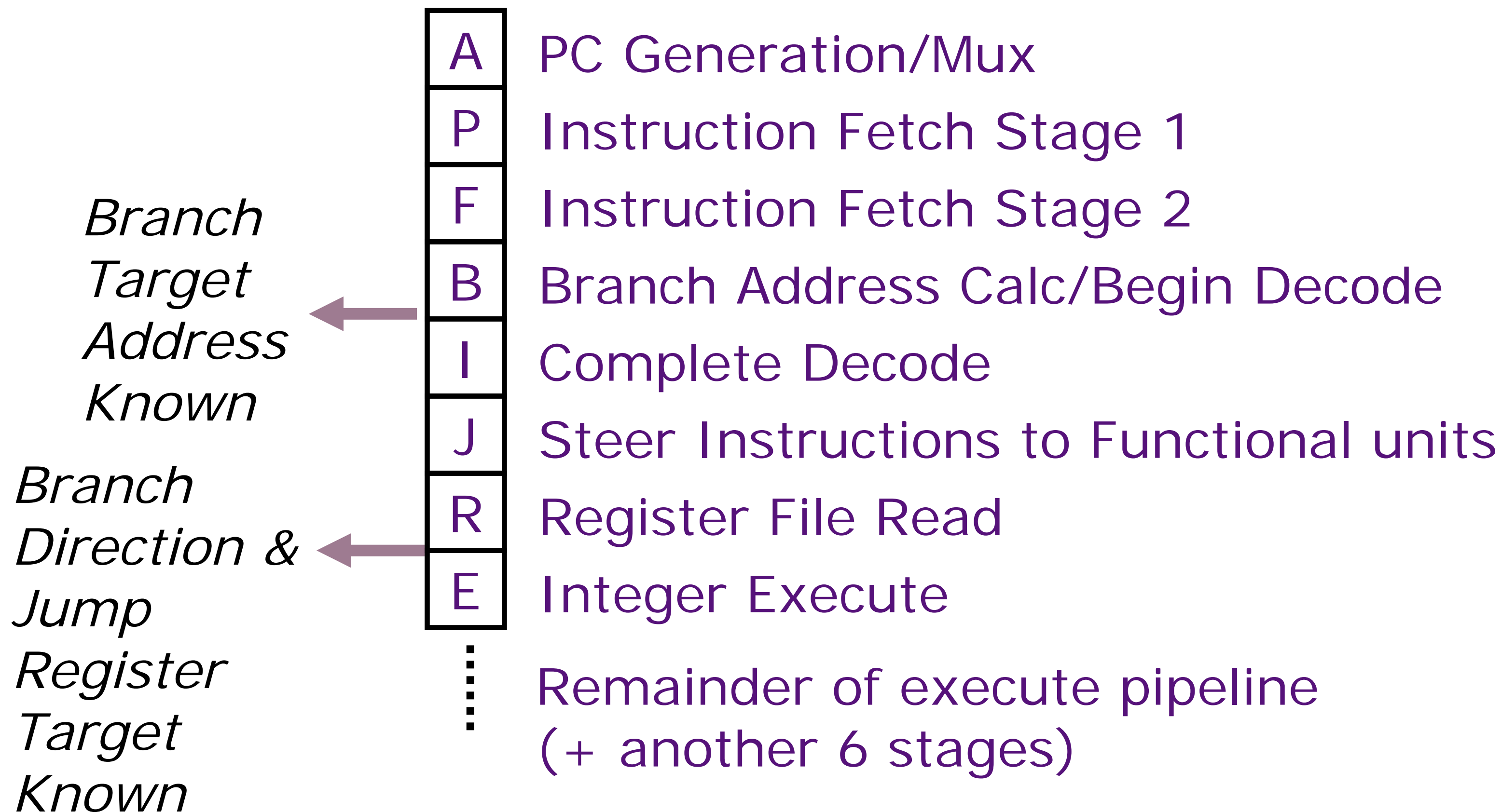
Need to know (or guess) both target address and whether the branch/jump is taken or not

Instruction	*Taken known?*	*Target known?*
BEQZ/BNEZ	After Reg. Fetch	After Inst. Fetch
J	Always Taken	After Inst. Fetch
JR	Always Taken	After Reg. Fetch

CSAIL

Branch Penalties in Modern Pipelines

UltraSPARC-III instruction fetch pipeline stages
(in-order issue, 4-way superscalar, 750MHz, 2000)



CSAIL

Outline

- Control transfer penalty
- Branch prediction schemes
- Branch misprediction recovery schemes

CSAIL

Branch Prediction

Motivation: branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

Required hardware support:

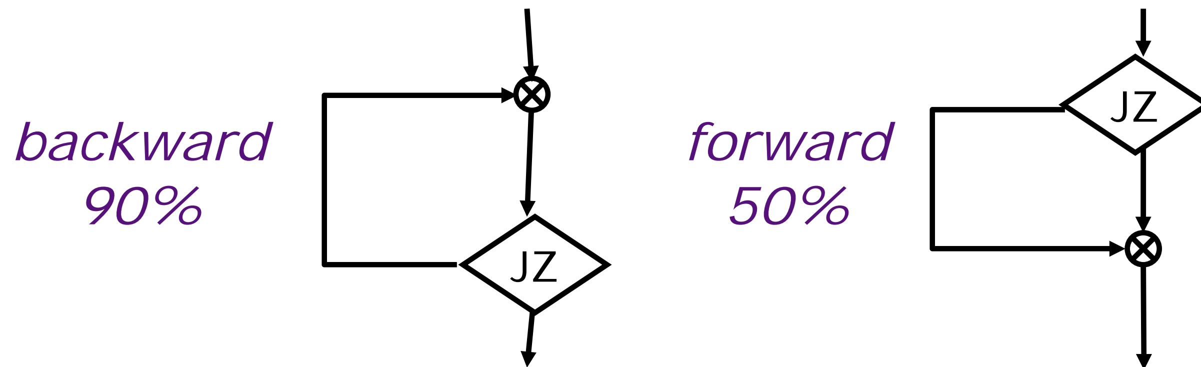
Prediction structures: branch history tables, branch target buffers, etc.

Mispredict recovery mechanisms:

- In-order machines: kill instructions following branch in pipeline
- Out-of-order machines: shadow registers and memory buffers for each speculated branch

Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:



ISA can attach additional semantics to branches about *preferred direction,* e.g., Motorola MC88110
bne0 *(preferred taken)* beq0 *(not taken)*

ISA can allow arbitrary choice of statically predicted direction (HP PA-RISC, Intel IA-64)

CSAIL

Dynamic Branch Prediction
learning based on past behavior

Temporal correlation

The way a branch resolves may be a good predictor of the way it will resolve at the next execution

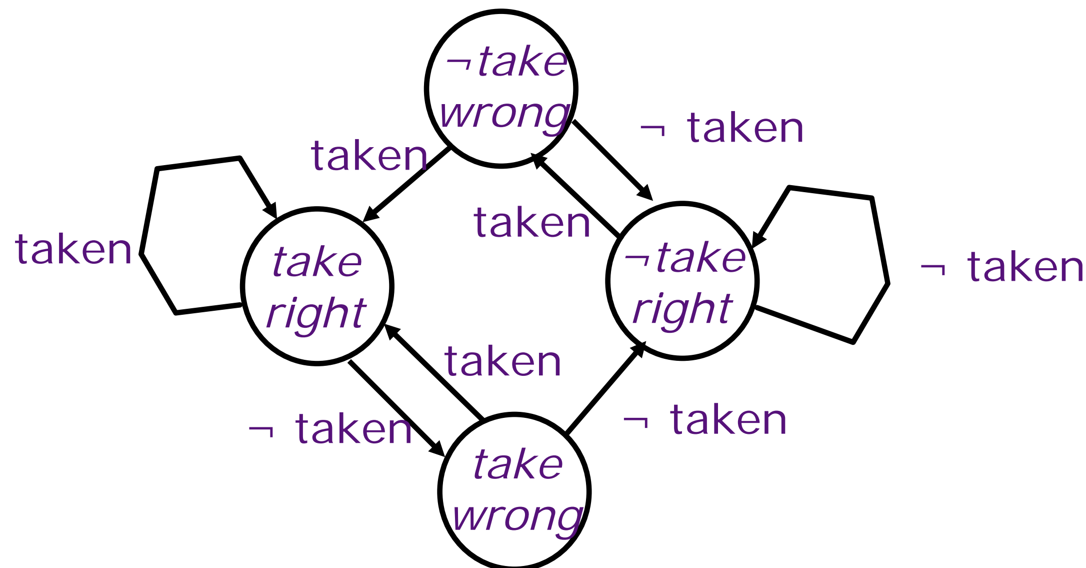
Spatial correlation

Several branches may resolve in a highly correlated manner *(a preferred path of execution)*

CSAIL

Branch Prediction Bits

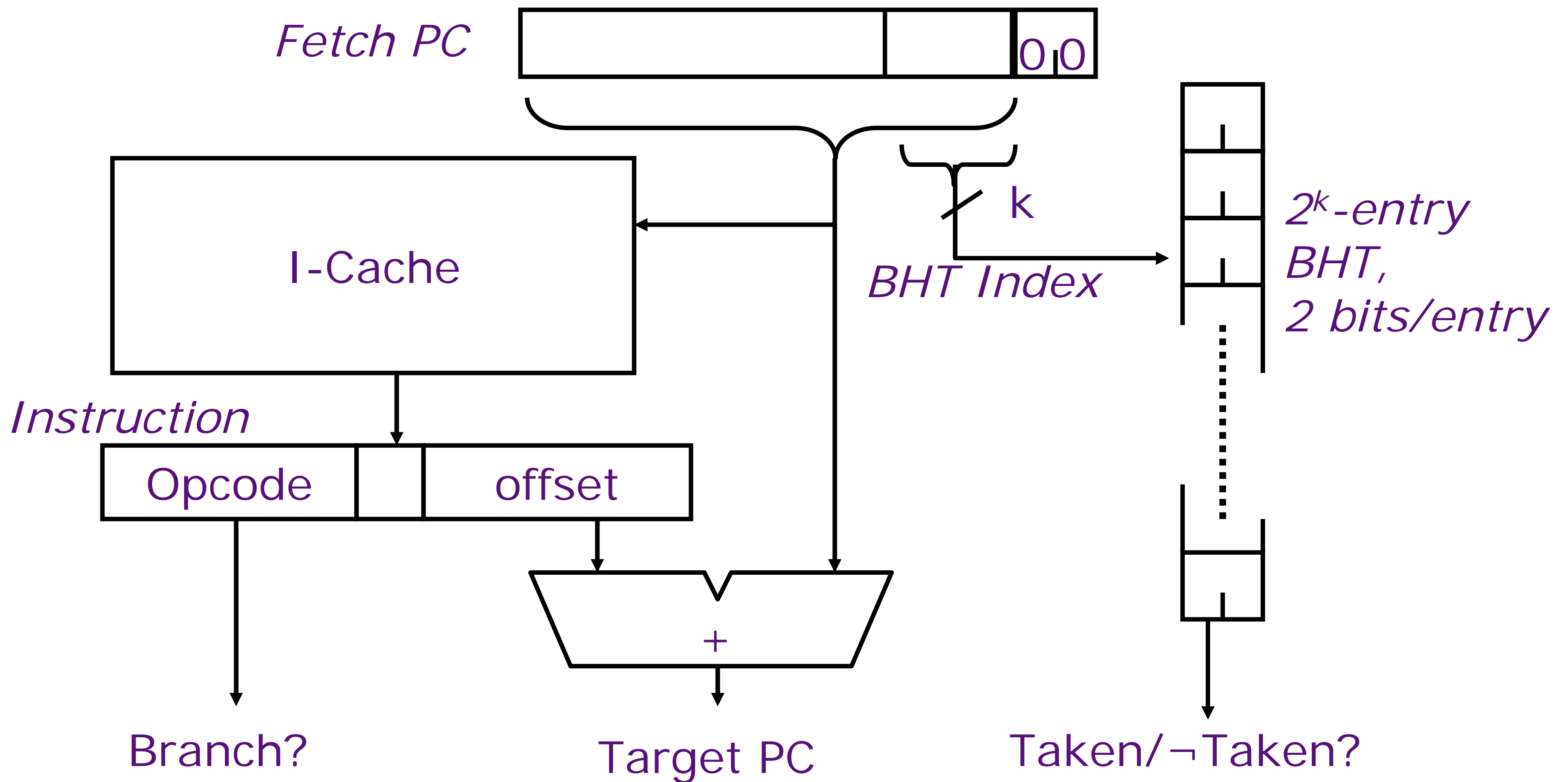
- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!



BP state:

(*predict* take/¬take) x (*last prediction* right/wrong)

Branch History Table

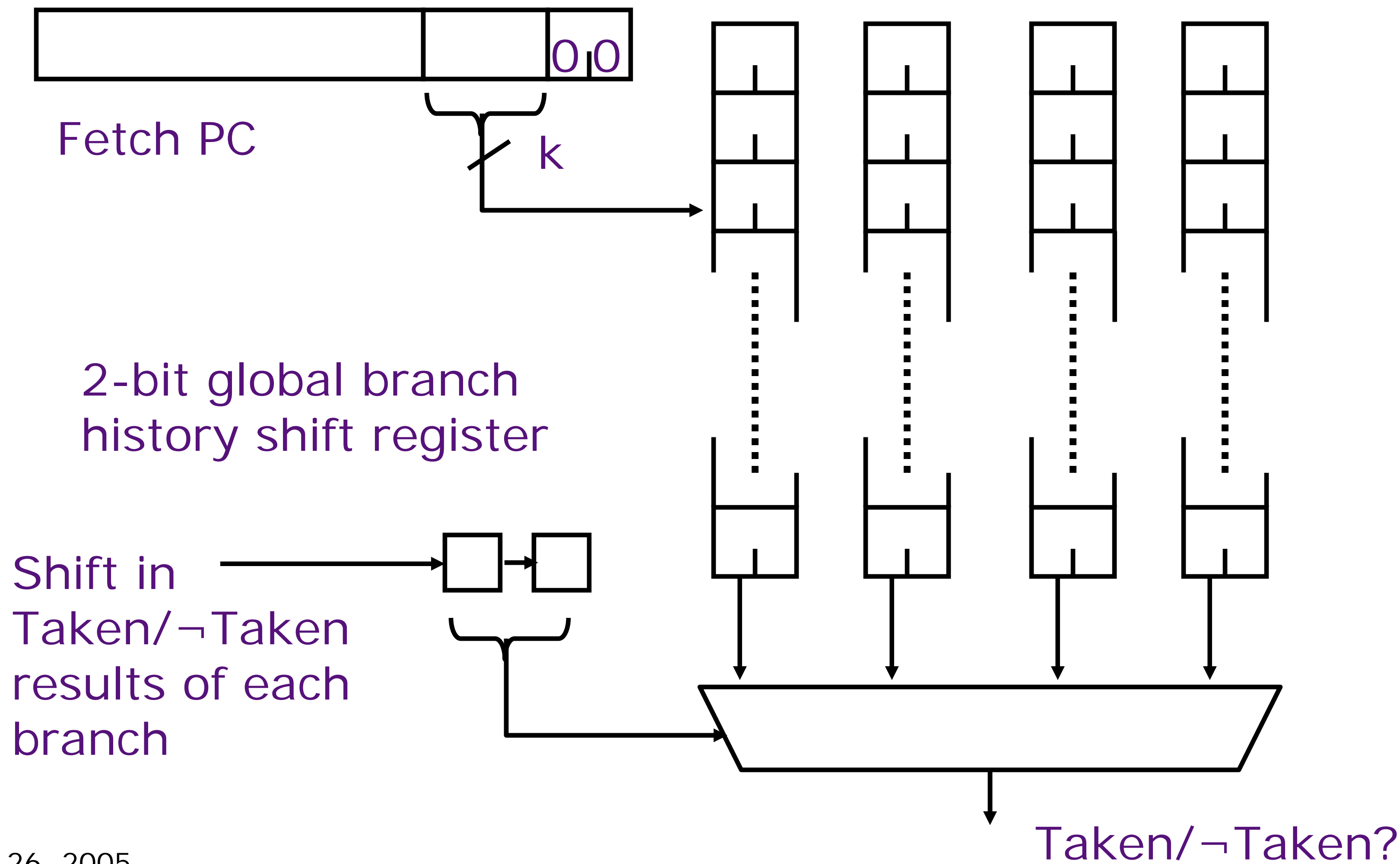


4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

CSAIL

Two-Level Branch Predictor

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)



Exploiting Spatial Correlation
Yeh and Patt, 1992

```
if (x[i] < 7) then
      y += 1;
if (x[i] < 5) then
      c -= 4;
```

If first condition false, second condition also false

History bit: H records the direction of the last branch executed by the processor

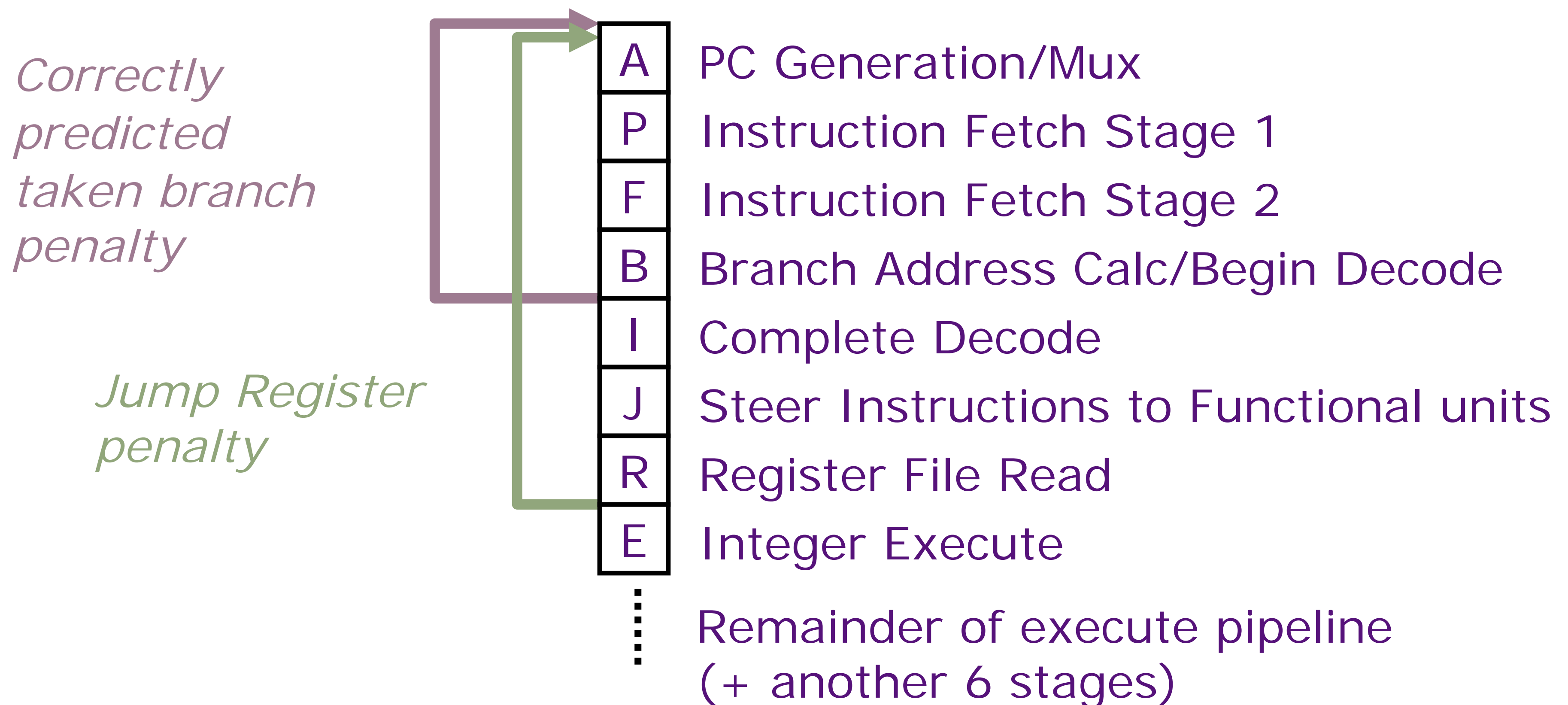
Two sets of BHT bits (BHT0 & BHT1) per branch instruction

H = 0 (not taken) ⇒ consult BHT0
H = 1 (taken) ⇒ consult BHT1

CSAIL

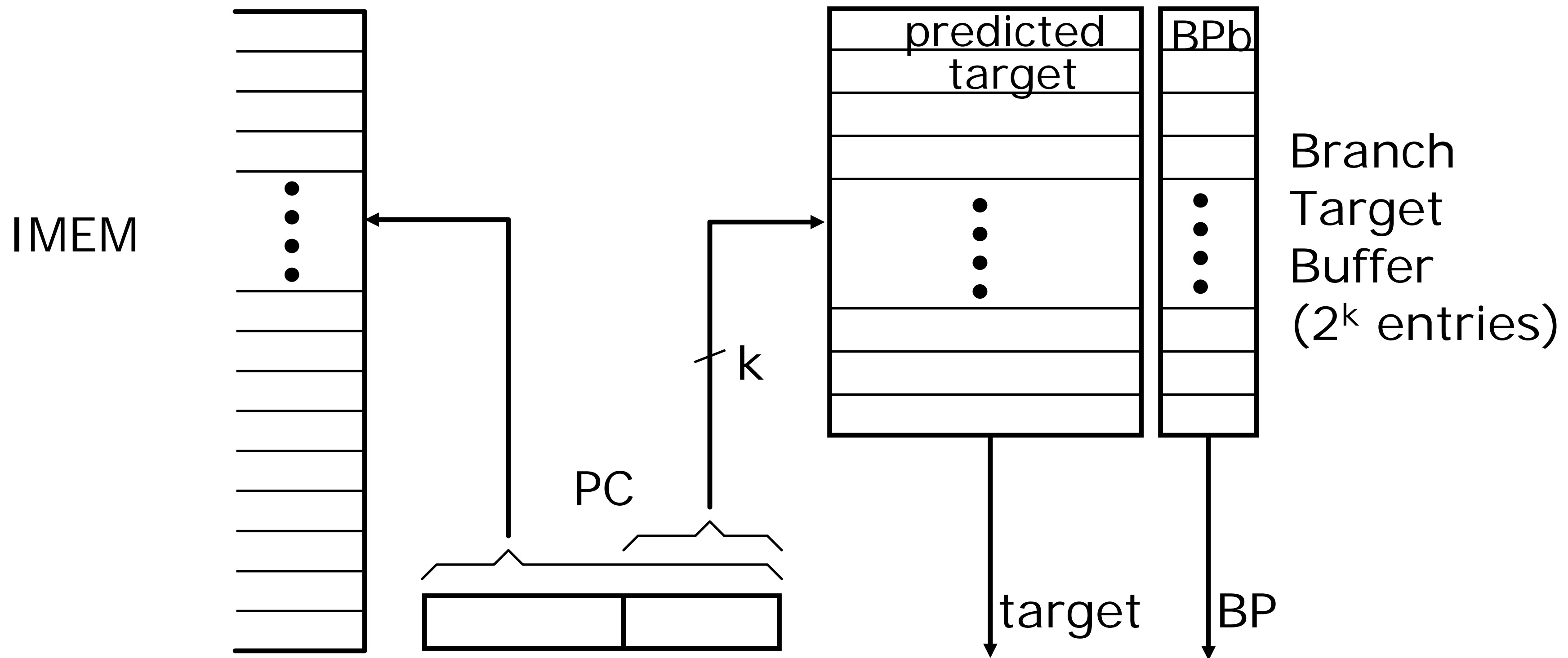
Limitations of BHTs

Cannot redirect fetch stream until after branch instruction is fetched and decoded, and target address determined



UltraSPARC-III fetch pipeline

Branch Target Buffer

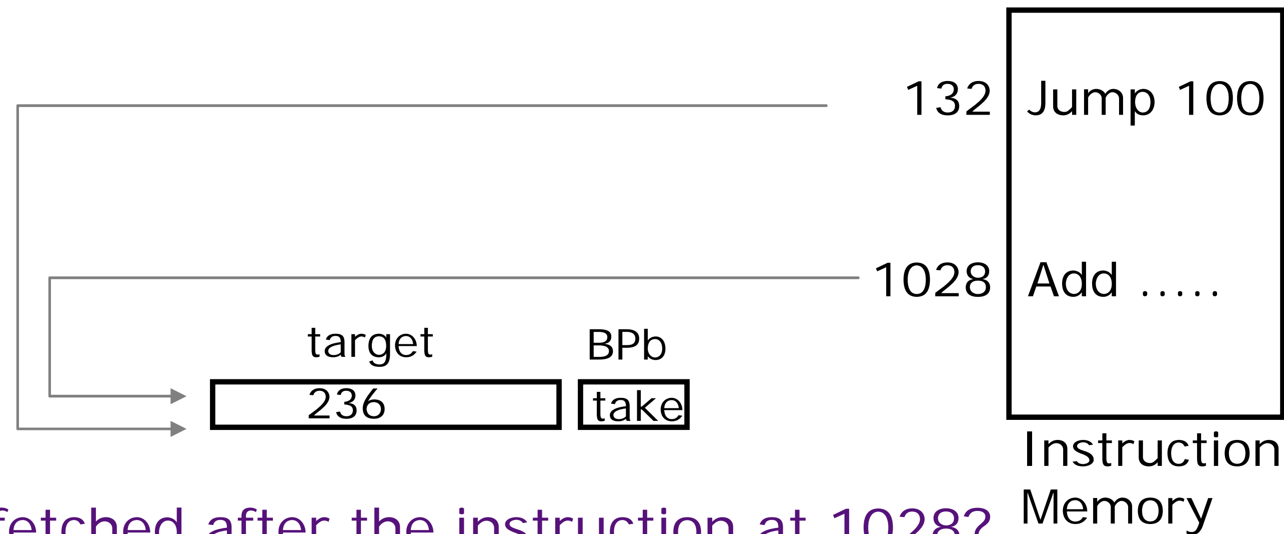


BP bits are stored with the predicted target address.

IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*
later: *check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb*

Address Collisions

Assume a 128-entry BTB



What will be fetched after the instruction at 1028?

BTB prediction = 236

Correct target = 1032

⇒ *kill* PC=236 and *fetch* PC=1032

Is this a common occurrence?
Can we avoid these bubbles?

BTB should be for Control Transfer instructions only

BTB contains useful information for branch and jump instructions only

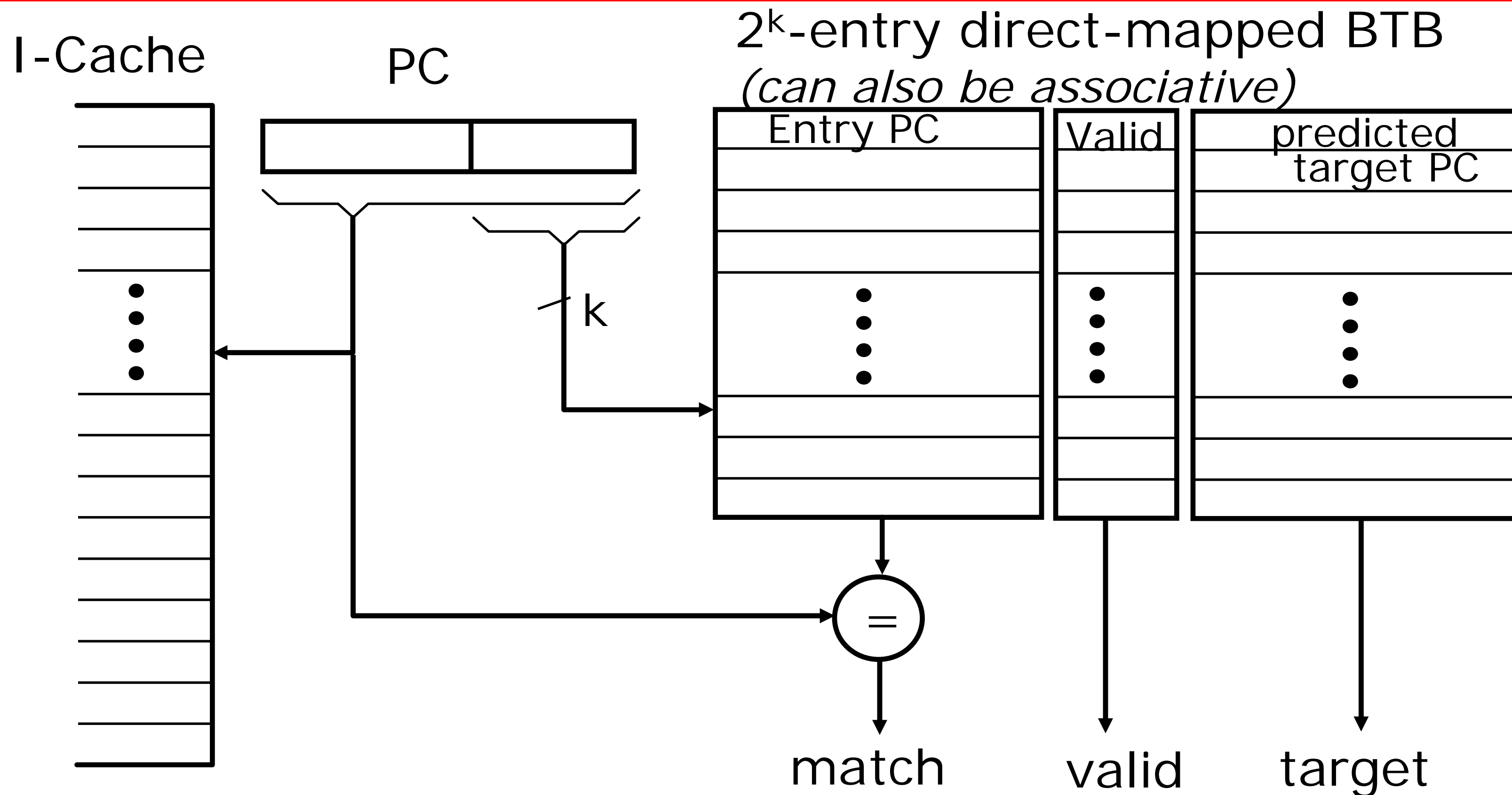
⇒ it should not be updated for other instructions

For all other instructions the next PC is (PC)+4 !

How to achieve this effect without decoding the instruction?

CSAIL

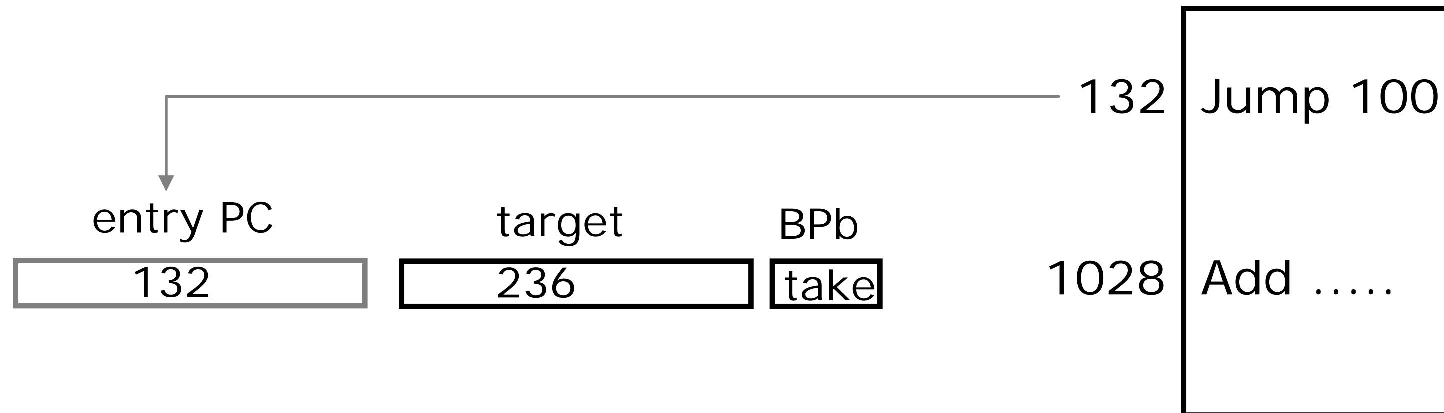
Branch Target Buffer (BTB)



- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

CSAIL

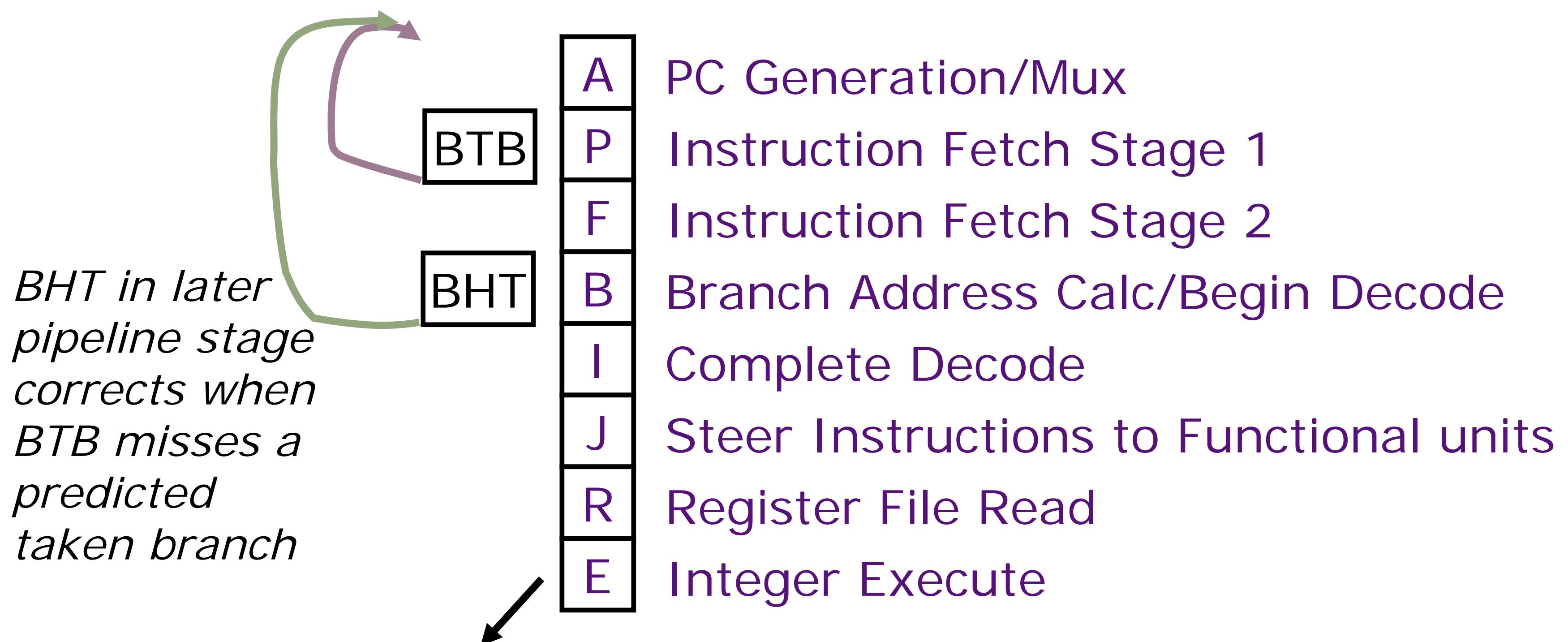
Consulting BTB Before Decoding



- The match for PC=1028 fails and 1028+4 is fetched
  ⇒ *eliminates false predictions after ALU instructions*
- BTB contains entries only for control transfer instructions
  ⇒ *more room to store branch targets*

Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate



BTB/BHT only updated after branch resolves in E stage

Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)

  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)

  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)

  BTB works well if usually return to the same place

  ⇒ *Often one function called from many different call sites!*

How well does BTB work for each of these cases?

Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

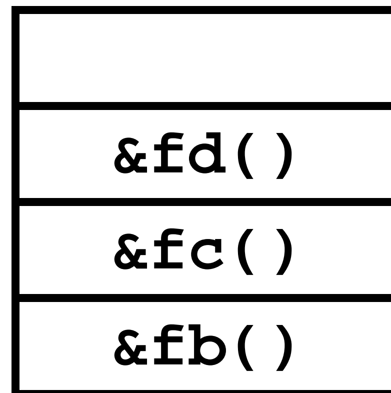
```
fa() { fb(); }

fb() { fc(); }

fc() { fd(); }
```

Push call address when function call executed

Pop return address when subroutine return decoded



k entries (typically k=8-16)

Outline

- Control transfer penalty
- Branch prediction schemes
- Branch misprediction recovery schemes ←

Five-minute break to stretch your legs

CSAIL

Mispredict Recovery

In-order execution machines:

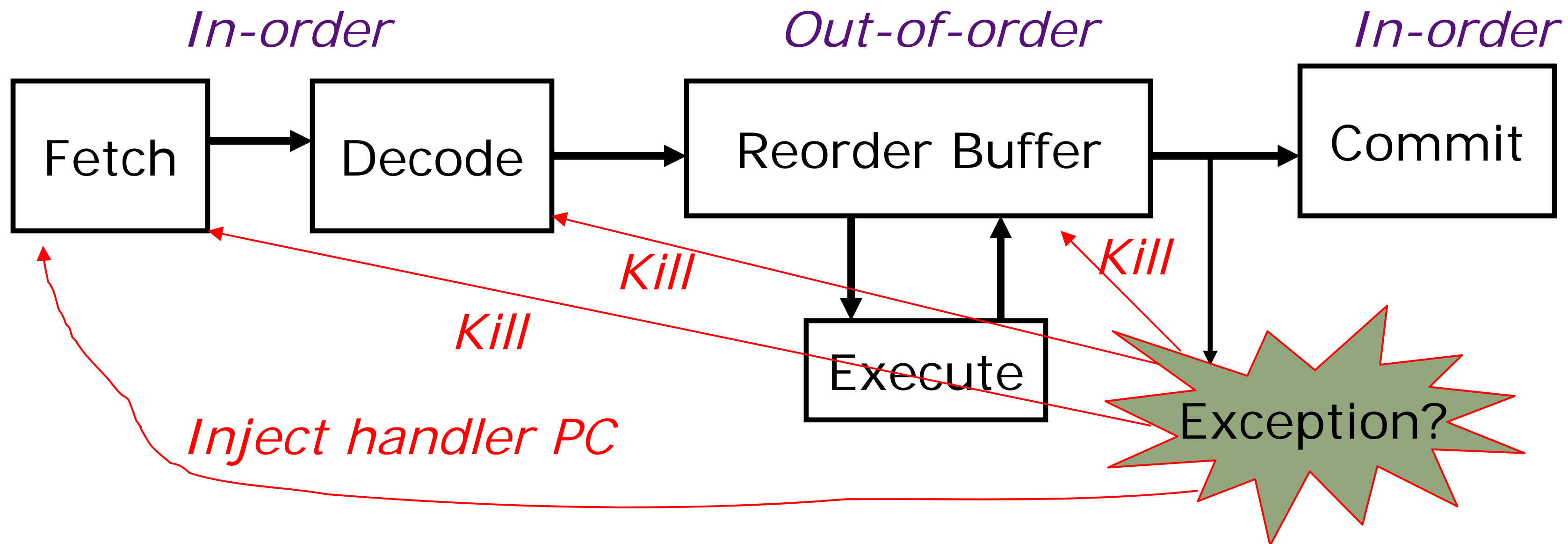
- Assume no instruction issued after branch can write-back before branch resolves
- Kill all instructions in pipeline behind mispredicted branch

Out-of-order execution?

- Multiple instructions following branch in program order can complete before branch resolves

CSAIL

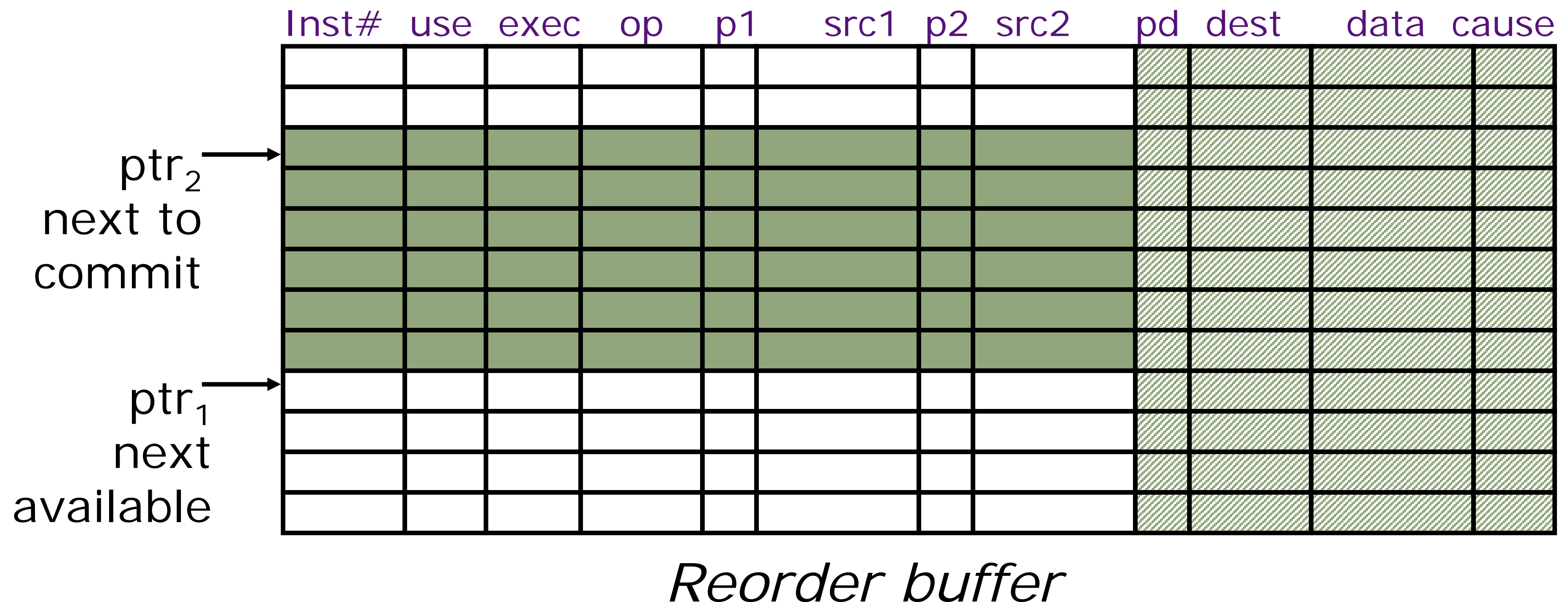
In-Order Commit for Precise Exceptions



- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ out-of-order completion)
- *Commit* (write-back to architectural state, i.e., regfile & memory, is in-order

Temporary storage needed in ROB to hold results before commit

Extensions for Precise Exceptions

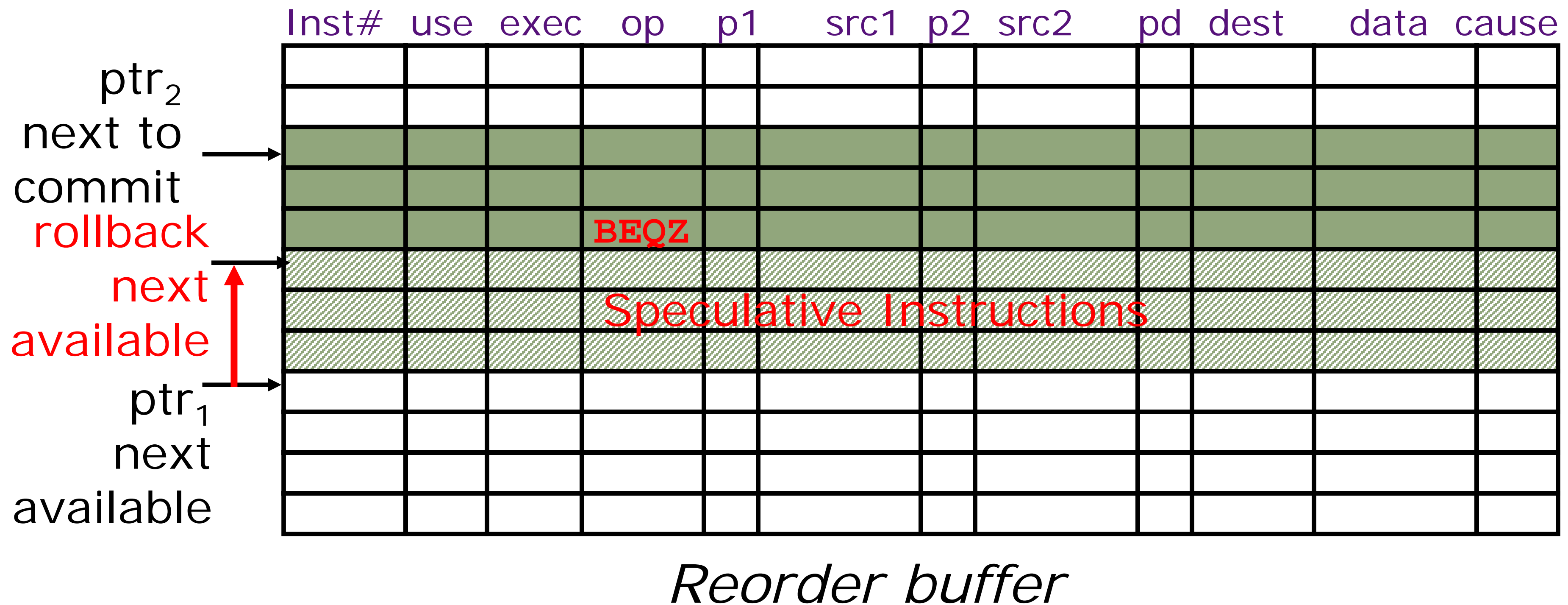


- add <pd, dest, data, cause> fields in the instruction template
- commit instructions to reg file and memory in program order ⇒ buffers can be maintained circularly
- on exception, clear reorder buffer by resetting $ptr_1 = ptr_2$

  (stores must wait for commit before updating memory)

CSAIL

Branch Misprediction Recovery

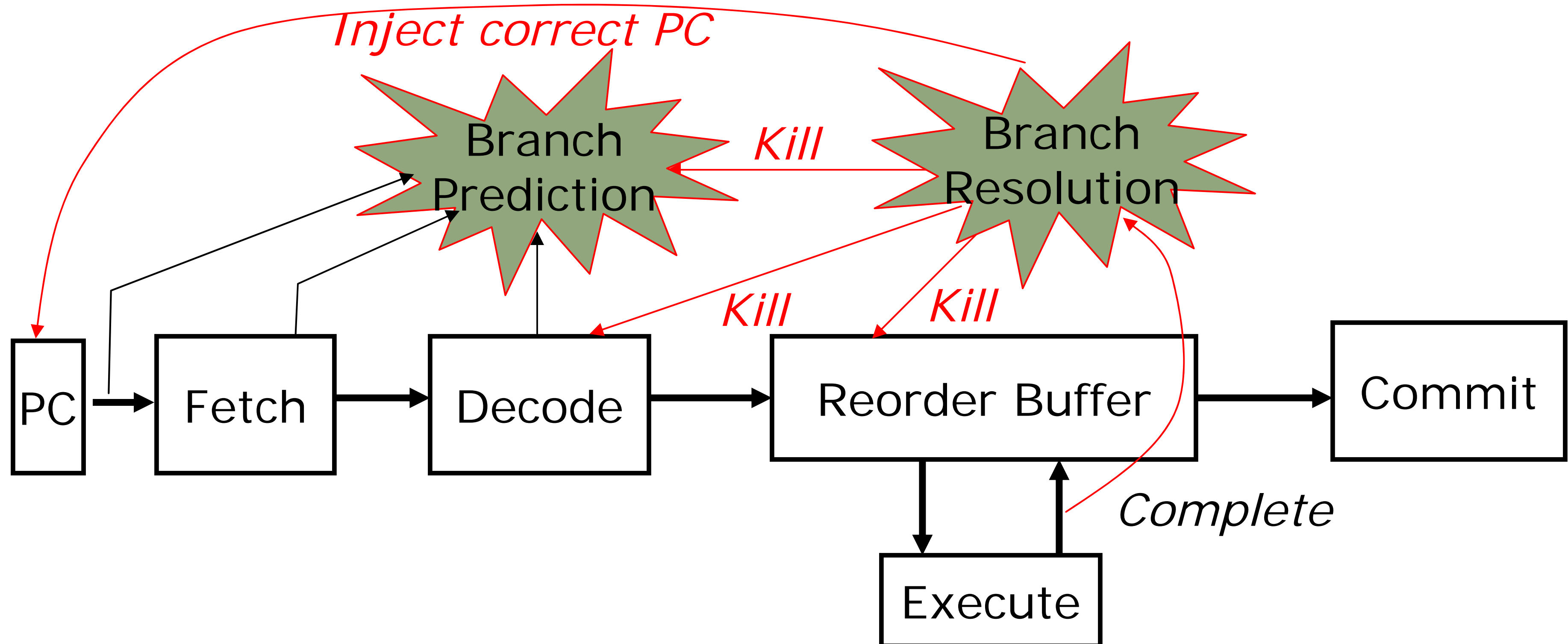


Reorder buffer

On mispredict

- Roll back "next available" pointer to just after branch
- Reset use bits
- Flush mis-speculated instructions from pipelines
- Restart fetch on correct branch path

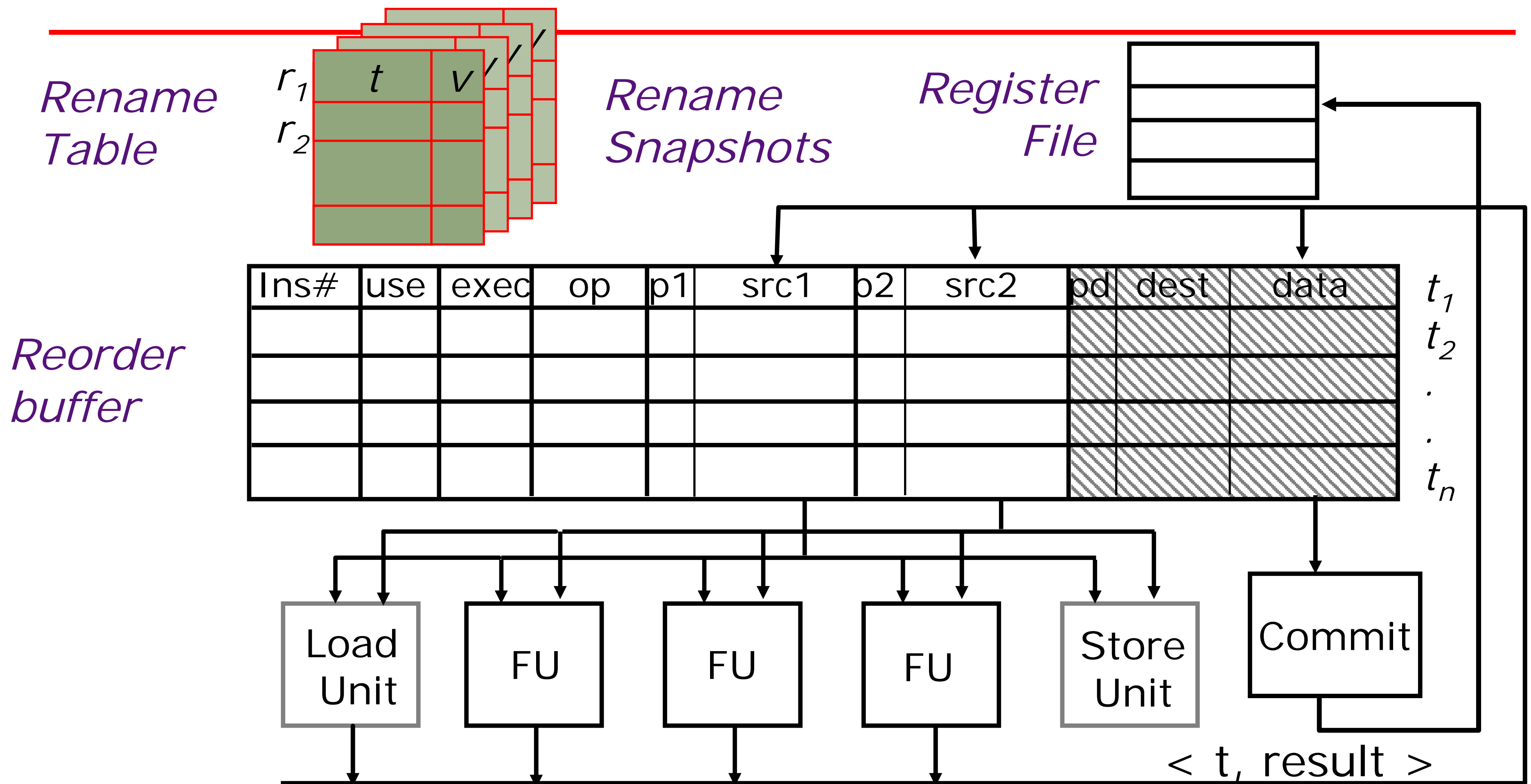
Branch Misprediction in Pipeline



- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch

CSAIL

Recovering Renaming Table



Take snapshot of register rename table at each predicted branch, recover earlier snapshot if branch mispredicted

CSAIL

Speculating Both Directions

An alternative to branch prediction is to execute both directions of a branch *speculatively*

- resource requirement is proportional to the number of concurrent speculative executions
- only half the resources engage in useful work when both directions of a branch are executed speculatively
- branch prediction takes less resources than speculative execution of both paths

With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction

CSAIL

Thank you !

Chapter 6

Parallel Processors from Client to Cloud

Introduction

- Goal: connecting multiple computers to get higher performance
  - Multiprocessors
  - Scalability, availability, power efficiency
- Task-level (process-level) parallelism
  - High throughput for independent jobs
- Parallel processing program
  - Single program run on multiple processors
- Multicore microprocessors
  - Chips with multiple processors (cores)

Hardware and Software

- Hardware
  - Serial: e.g., Pentium 4
  - Parallel: e.g., quad-core Xeon e5345
- Software
  - Sequential: e.g., matrix multiplication
  - Concurrent: e.g., operating system
- Sequential/concurrent software can run on serial/parallel hardware
  - Challenge: making effective use of parallel hardware

Parallel Programming

- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!
- Difficulties
  - Partitioning(load-balancing,scheduling)
  - Coordination(syncronization)
  - Communications overhead

Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
  - $T_{new} = T_{parallelizable}/100 + T_{sequential}$
  - $$\text{Speedup} = \frac{1}{(1 - F_{parallelizable}) + F_{parallelizable}/100} = 90$$
  - Solving: $F_{parallelizable} = 0.999 = 99.9\%$
- Need sequential part to be 0.1% of original time

Scaling Example

- Workload: sum of 10 scalars, and 10 × 10 matrix sum. Matrix sum is parallelizable
  - Find Speed up from 10 to 100 processors
- Single processor: Time = $(10 + 100) \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - Speedup = 110/20 = 5.5 (55% of potential speedup **10x(ideal**))
- 100 processors
  - Time = $10 \times t_{add} + 100/100 \times t_{add} = 11 \times t_{add}$
  - Speedup = 110/11 = 10 (10% of potential)
- Assumes load can be balanced across processors

Scaling Example (cont)

- What if matrix size is 100 × 100?
- Single processor: Time = $(10 + 10000) \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + 10000/10 \times t_{add} = 1010 \times t_{add}$
  - Speedup = 10010/1010 = 9.9 (99% of potential)
- 100 processors
  - Time = $10 \times t_{add} + 10000/100 \times t_{add} = 110 \times t_{add}$
  - Speedup = 10010/110 = 91 (91% of potential)
- Assuming load balanced

Strong vs Weak Scaling

- Strong scaling: problem size fixed
  - As in example
- Weak scaling: problem size proportional to number of processors
  - 10 processors, 10 × 10 matrix
    - Time = $20 \times t_{add}$
  - 100 processors, 32 × 32 matrix
    - Time = $10 \times t_{add} + 1000/100 \times t_{add} = 20 \times t_{add}$
  - Constant performance in this example

Instruction and Data Streams

- Flynn's taxonomy

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

Example: DAXPY (Y = a × X + Y)

- Conventional LEGv8 code:

```
       LDURD D0,[X28,a]    //load scalar a
       ADDI  X0,X19,512    //upper bound of what to load
loop:  LDURD D2,[X19,#0]   //load x(i)
       FMULD D2,D2,D0      //a x x(i)
       LDURD D4,[X20,#0]   //load y(i)
       FADDD D4,D4,D2      //a x x(i) + y(i)
       STURD D4,[X20,#0]   //store into y(i)
       ADDI X19,X19,#8     //increment index to x
       ADDI X20,X20,#8     //increment index to y
       CMPB X0,X19         //compute bound
       B.NE loop           //check if done
```

Staring address of X.Y are x19,x20

Vector LEGv8 code:

```
       LDURD D0,[X28,a]    //load scalar a
       LDURDV V1,[X19,#0] //load vector x
       FMULDVS V2,V1,D0    //vector-scalar multiply
       LDURDV V3,[X20,#0] //load vector y
       FADDDV V4,V2,V3     //add y to product
       STURDV V4,[X20,#0] //store the result
```

Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to LEGv8
  - 32 × 64-element registers (64-bit elements)
  - Vector instructions
    - `lv`, `sv`: load/store vector
    - `addv.d`: add vectors of double
    - `addvs.d`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Vector vs. Scalar

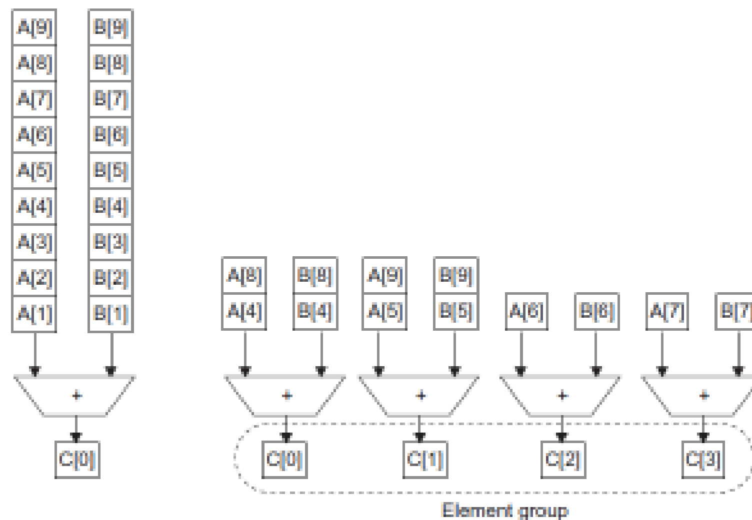
- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology

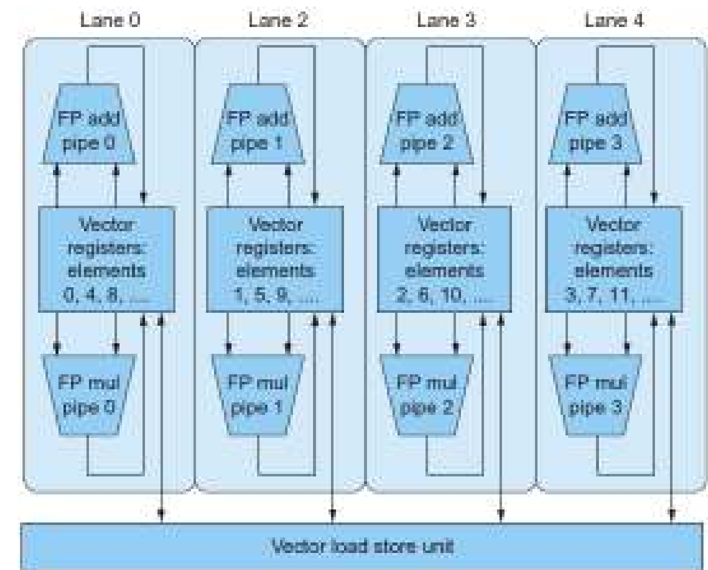
SIMD

- Operate elementwise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

Vector vs. Multimedia Extensions

- Vector instructions have a variable vector width, multimedia extensions have a fixed width
- Vector instructions support strided access, multimedia extensions do not
- Vector units can be combination of pipelined and arrayed functional units:





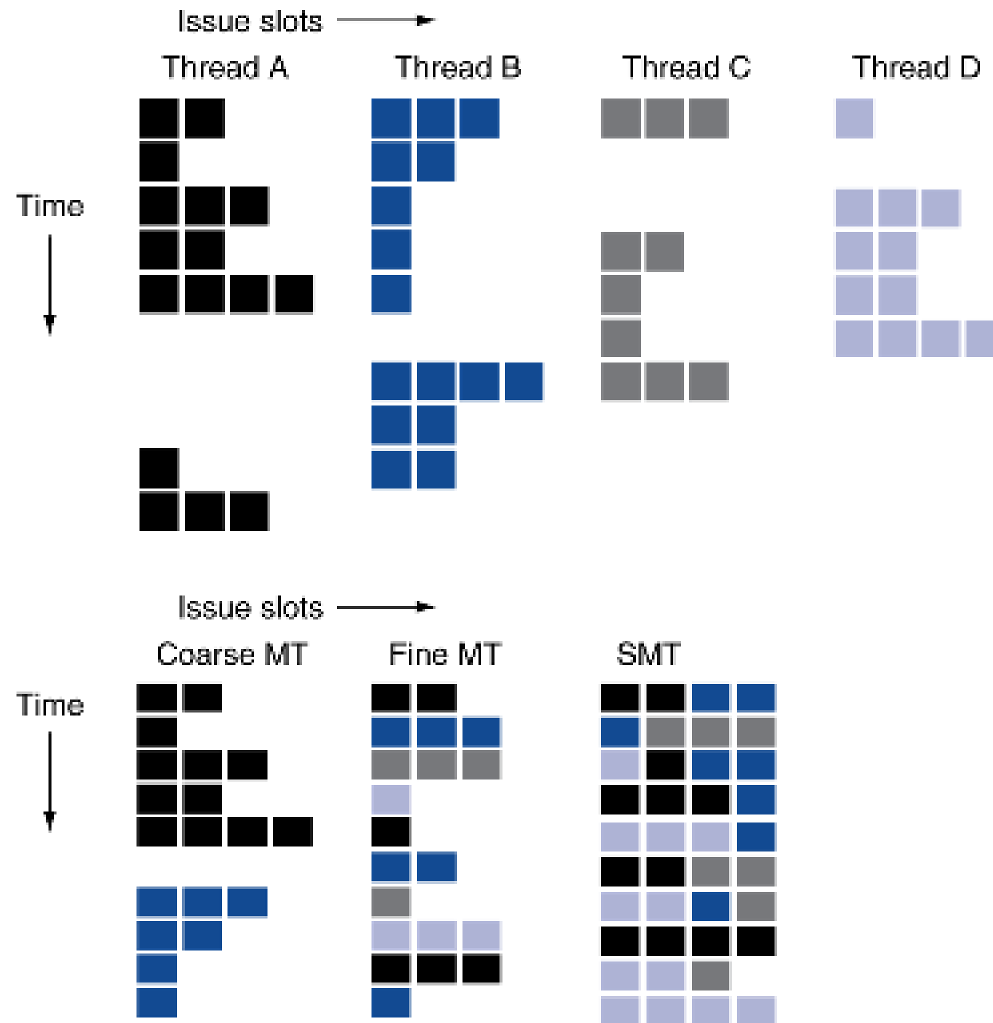
Multithreading

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

Multithreading Example

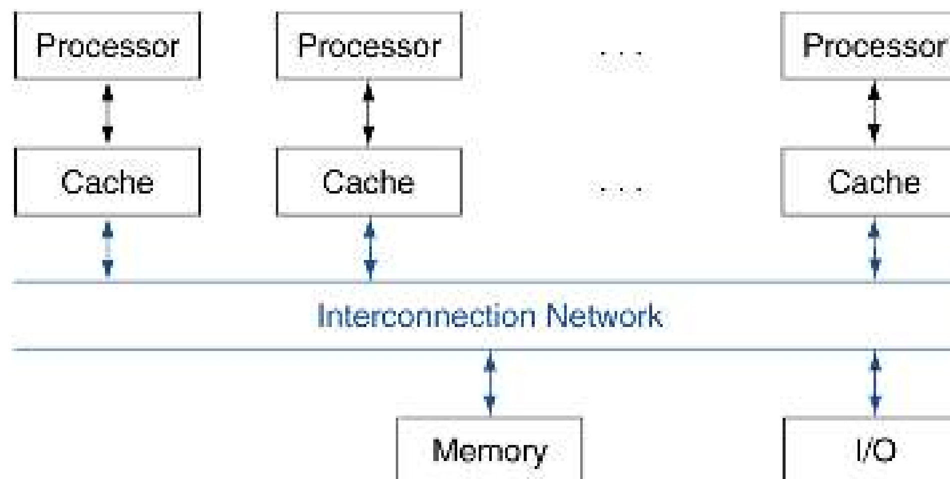


Future of Multithreading

- Will it survive? In what form?
- Power considerations $\Rightarrow$ simplified microarchitectures
  - Simpler forms of multithreading
- Tolerating cache-miss latency
  - Thread switch may be most effective
- Multiple simple cores might share resources more effectively

Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks
  - Memory access time
    - UMA (uniform) vs. NUMA (nonuniform)



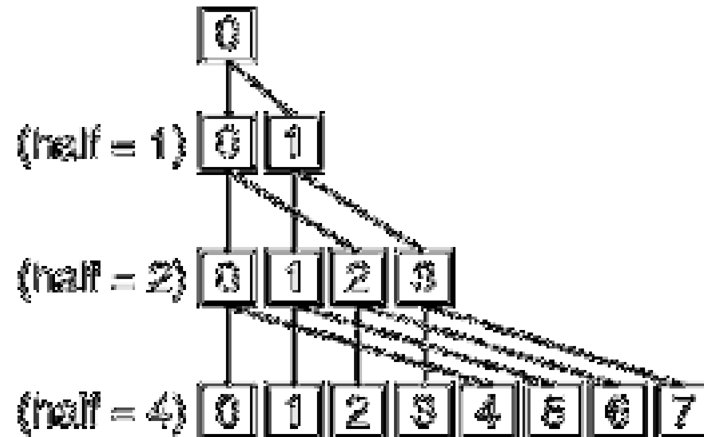
Example: Sum Reduction

- Sum 100,000 numbers on 100 processor UMA
  - Each processor has ID: $0 \leq Pn \leq 99$
  - Partition 1000 numbers per processor
  - Initial summation on each processor

```
sum[Pn] = 0;
  for (i = 1000*Pn;
       i < 1000*(Pn+1); i = i + 1)
     sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, …
  - Need to synchronize between reduction steps

Example: Sum Reduction




```
half = 100;
repeat
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

History of GPUs

- Early video cards
  - Frame buffer memory with address generation for video output
- 3D graphics processing
  - Originally high-end computers (e.g., SGI)
  - Moore's Law $\Rightarrow$ lower cost, higher density
  - 3D graphics cards for PCs and game consoles
- Graphics Processing Units
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, rasterization

Graphics in the System

Intel CPU
Front Side Bus
x16 PCI-Express Link
GPU
display
North Bridge
DDR2 Memory
GPU Memory
x4 PCI-Express Link derivative
128-bit 667 MT/s
South Bridge

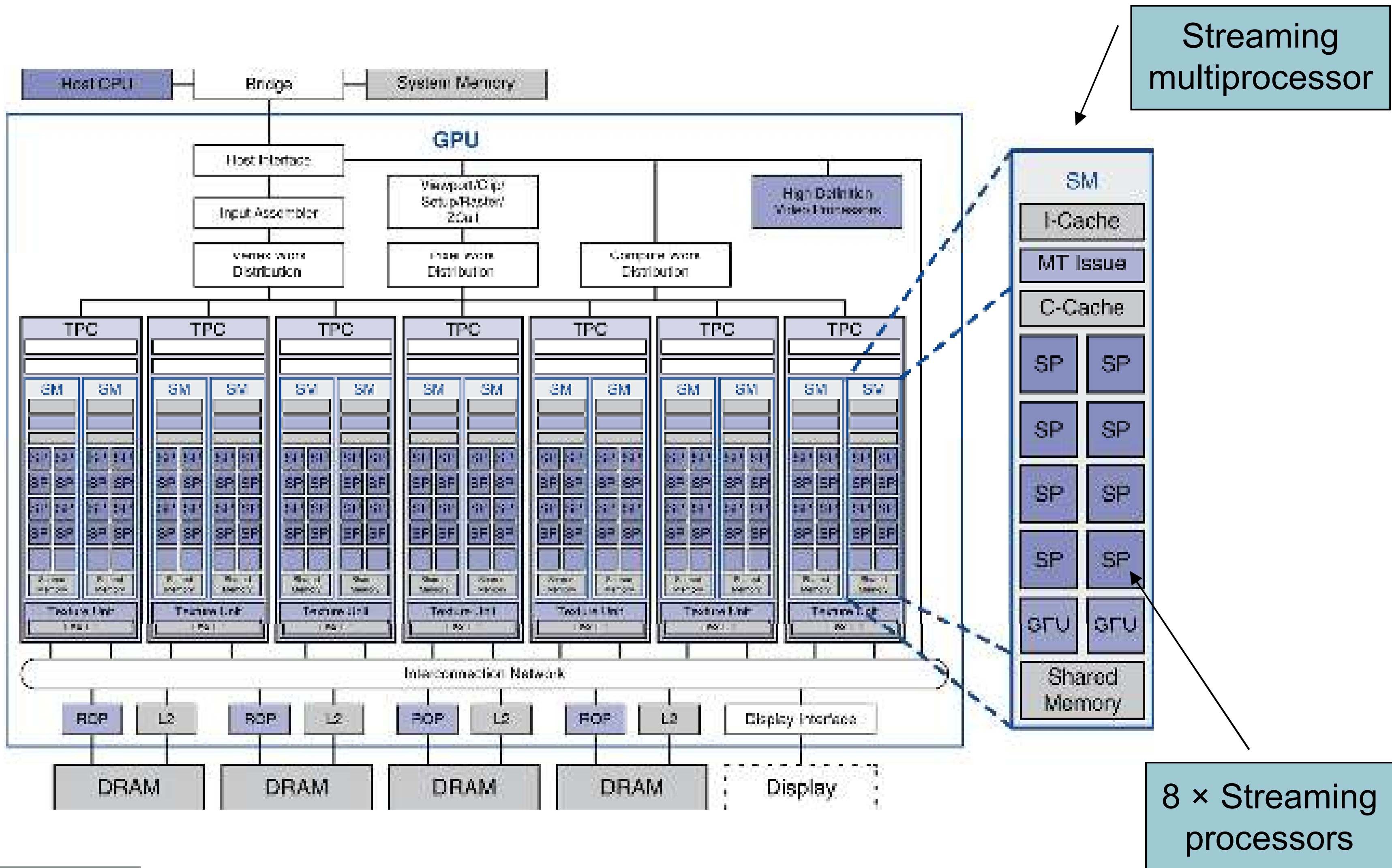
CPU
Front Side Bus
North Bridge
Memory
PCI Bus
South Bridge
VGA Controller
Framebuffer Memory
LAN
UART
VGA Display

AMD CPU
CPU core
internal bus
North Bridge
128-bit 667 MT/s
DDR2 Memory
x16 PCI-Express Link
HyperTransport 1.03
GPU
display
Chipset
GPU Memory

GPU Architectures

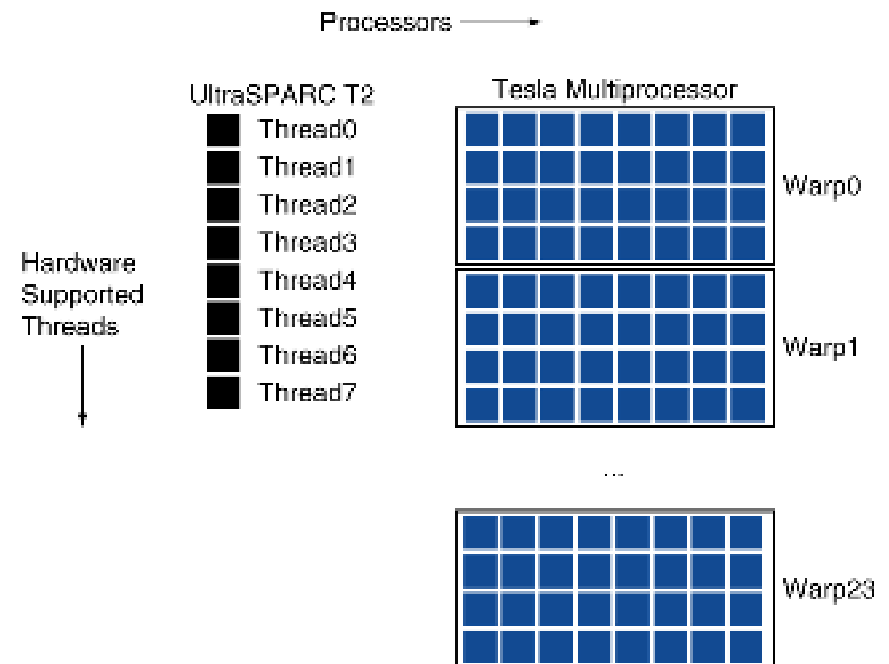
- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

Example: NVIDIA Tesla



Example: NVIDIA Tesla

- Streaming Processors
  - Single-precision FP and integer units
  - Each SP is fine-grained multithreaded
- Warp: group of 32 threads
  - Executed in parallel, SIMD style
    - 8 SPs × 4 clock cycles
  - Hardware contexts for 24 warps
    - Registers, PCs, …

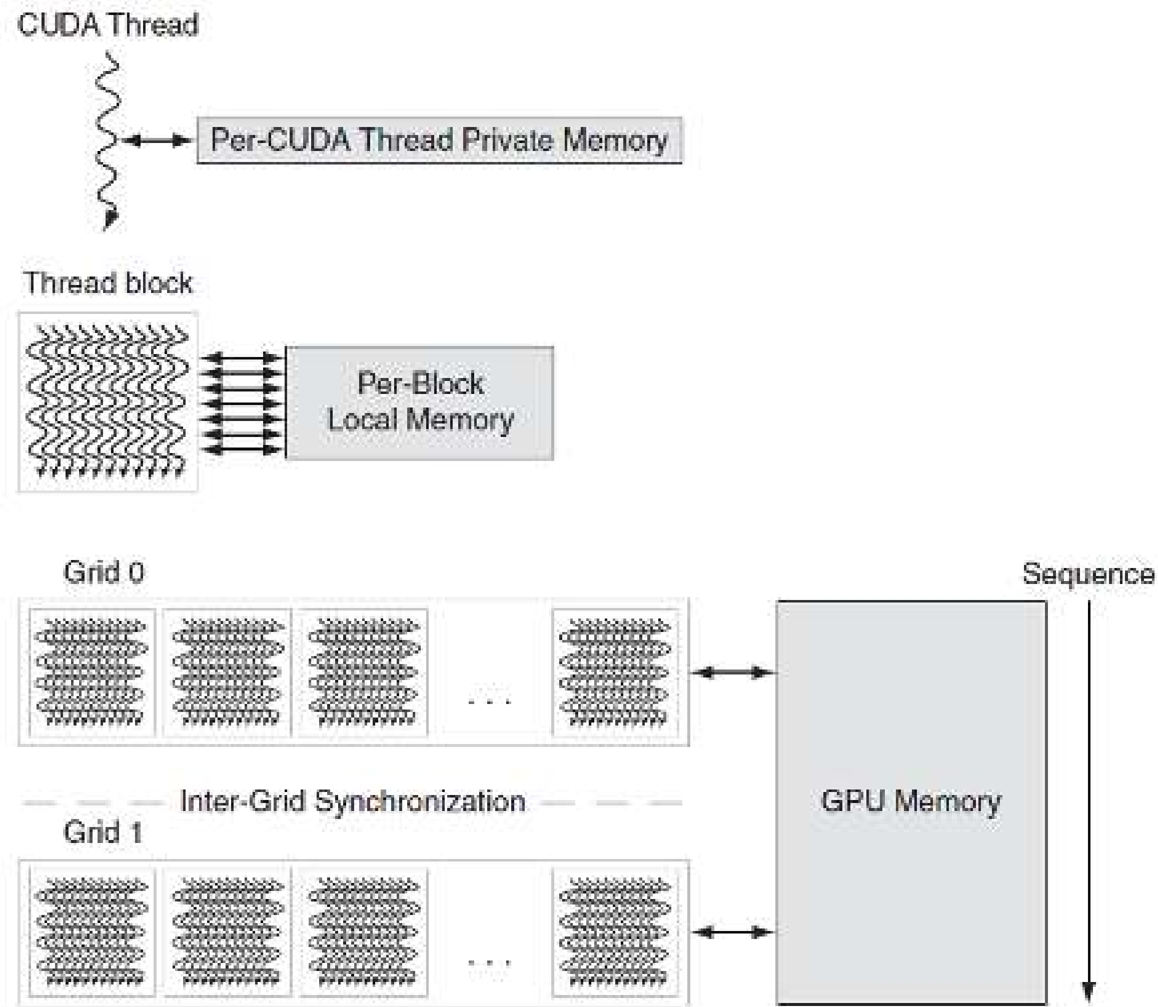


Classifying GPUs

- Don't fit nicely into SIMD/MIMD model
  - Conditional execution in a thread allows an illusion of MIMD
    - But with performance degredation
    - Need to write general purpose code with care

	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	**Tesla Multiprocessor**

GPU Memory Structures



Putting GPUs into Perspective

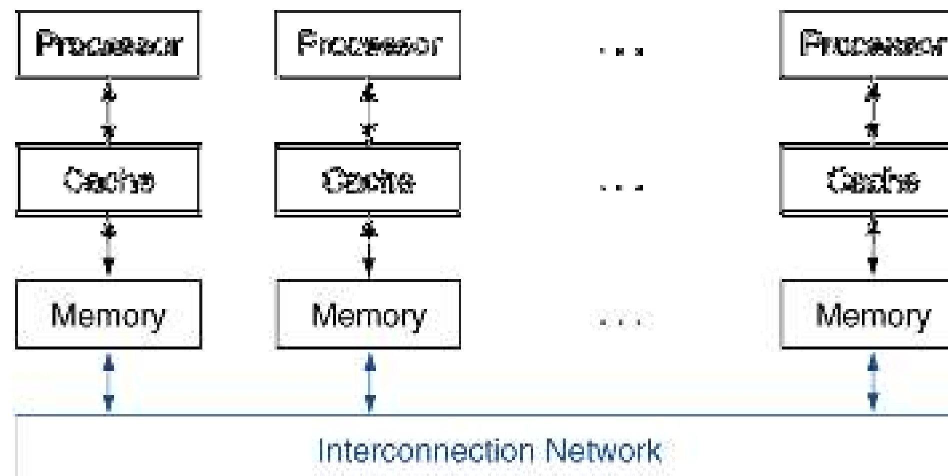
Feature	Multicore with SIMD	GPU
SIMD processors	4 to 8	8 to 16
SIMD lanes/processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Typical ratio of single precision to double-precision performance	2:1	2:1
Largest cache size	8 MB	0.75 MB
Size of memory address	64-bit	64-bit
Size of main memory	8 GB to 256 GB	4 GB to 6 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Integrated scalar processor/SIMD processor	Yes	No
Cache coherent	Yes	No

Guide to GPU Terms

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

Message Passing

- Each processor has private physical address space
- Hardware sends/receives messages between processors



Loosely Coupled Clusters

- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- High availability, scalable, affordable
- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP

Sum Reduction (Again)

- Sum 100,000 on 100 processors
- First distribute 100 numbers to each
  - The do partial sums

```
sum = 0;
for (i = 0; i<1000; i = i + 1)
  sum = sum + AN[i];
```

- Reduction
  - Half the processors send, other half receive and add
  - The quarter send, quarter receive and add, …

Sum Reduction (Again)

- Given send() and receive() operations

```
limit = 100; half = 100;/* 100 processors */
repeat
  half = (half+1)/2; /* send vs. receive
                        dividing line */
  if (Pn >= half && Pn < limit)
    send(Pn - half, sum);
  if (Pn < (limit/2))
    sum = sum + receive();
  limit = half; /* upper limit of senders */
until (half == 1); /* exit with final sum */
```

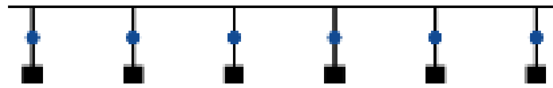
  - Send/receive also provide synchronization
  - Assumes send/receive take similar time to addition

Grid Computing

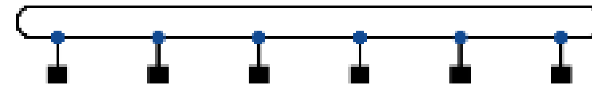
- Separate computers interconnected by long-haul networks
  - E.g., Internet connections
  - Work units farmed out, results sent back
- Can make use of idle time on PCs
  - E.g., SETI@home, World Community Grid

Interconnection Networks

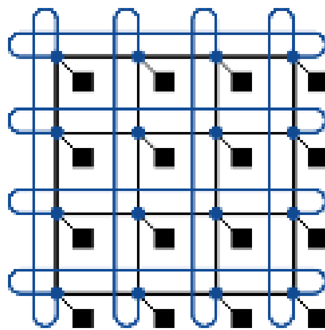
- Network topologies
  - Arrangements of processors, switches, and links



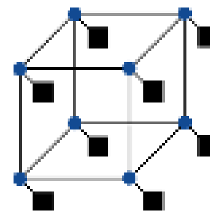
Bus



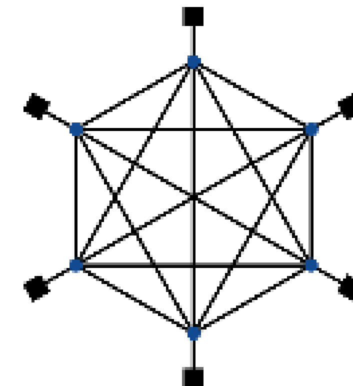
Ring



2D Mesh

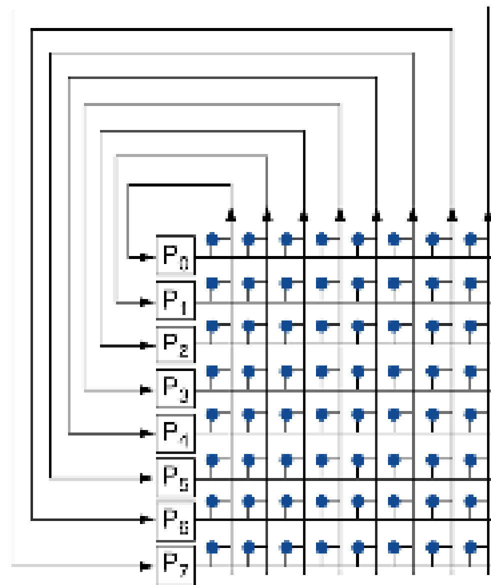


N-cube (N = 3)

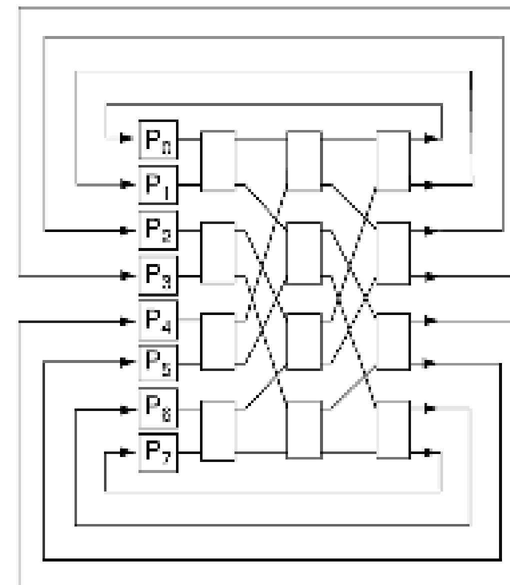


Fully connected

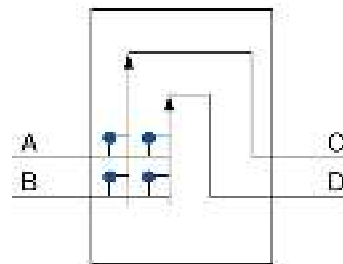
Multistage Networks



a. Crossbar



b. Omega network



c. Omega network switch box

Network Characteristics

- Performance
  - Latency per message (unloaded network)
  - Throughput
    - Link bandwidth
    - Total network bandwidth
    - Bisection bandwidth
  - Congestion delays (depending on traffic)
- Cost
- Power
- Routability in silicon

Parallel Benchmarks

- Linpack: matrix linear algebra
- SPECrate: parallel run of SPEC CPU programs
  - Job-level parallelism
- SPLASH: Stanford Parallel Applications for Shared Memory
  - Mix of kernels and applications, strong scaling
- NAS (NASA Advanced Supercomputing) suite
  - computational fluid dynamics kernels
- PARSEC (Princeton Application Repository for Shared Memory Computers) suite
  - Multithreaded applications using Pthreads and OpenMP

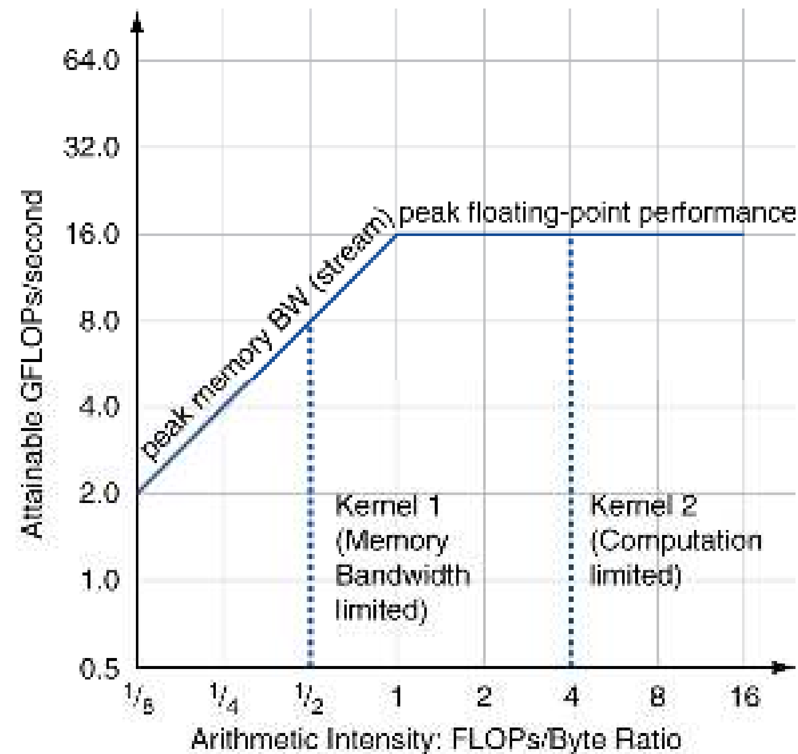
Code or Applications?

- Traditional benchmarks
  - Fixed code and data sets
- Parallel programming is evolving
  - Should algorithms, programming languages, and tools be part of the system?
  - Compare systems, provided they implement a given application
  - E.g., Linpack, Berkeley Design Patterns
- Would foster innovation in approaches to parallelism

Modeling Performance

- Assume performance metric of interest is achievable GFLOPs/sec
  - Measured using computational kernels from Berkeley Design Patterns
- Arithmetic intensity of a kernel
  - FLOPs per byte of memory accessed
- For a given computer, determine
  - Peak GFLOPS (from data sheet)
  - Peak memory bytes/sec (using Stream benchmark)

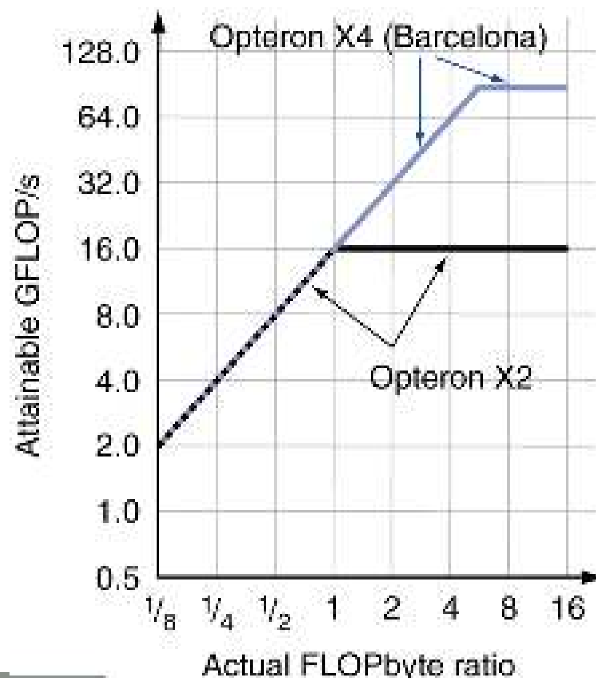
Roofline Diagram



Attainable GPLOPs/sec
= Max (Peak Memory BW × Arithmetic Intensity, Peak FP Performance)

Comparing Systems

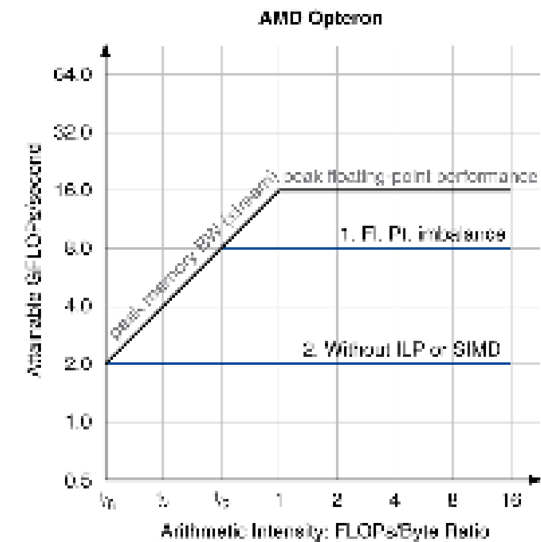
- Example: Opteron X2 vs. Opteron X4
  - 2-core vs. 4-core, 2× FP performance/core, 2.2GHz vs. 2.3GHz
  - Same memory system

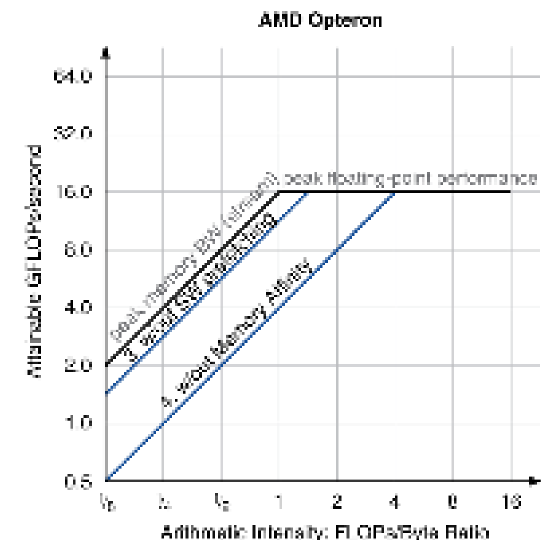


- To get higher performance on X4 than X2
  - Need high arithmetic intensity
  - Or working set must fit in X4's 2MB L-3 cache

Optimizing Performance

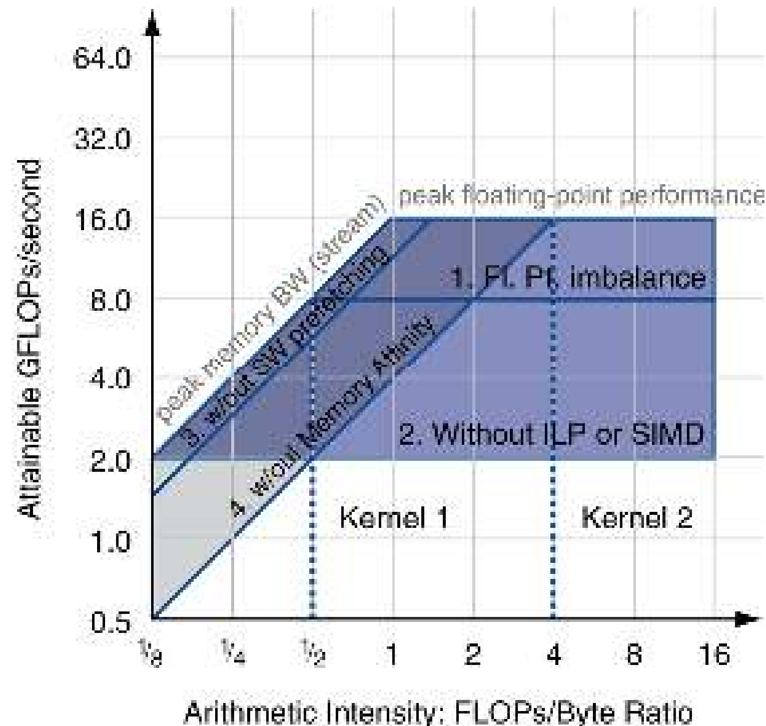
- Optimize FP performance
  - Balance adds & multiplies
  - Improve superscalar ILP and use of SIMD instructions
- Optimize memory usage
  - Software prefetch
    - Avoid load stalls
  - Memory affinity
    - Avoid non-local data accesses





Optimizing Performance

- Choice of optimization depends on arithmetic intensity of code

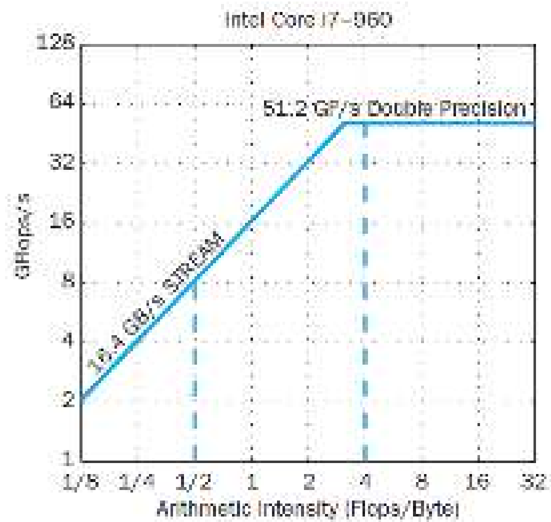


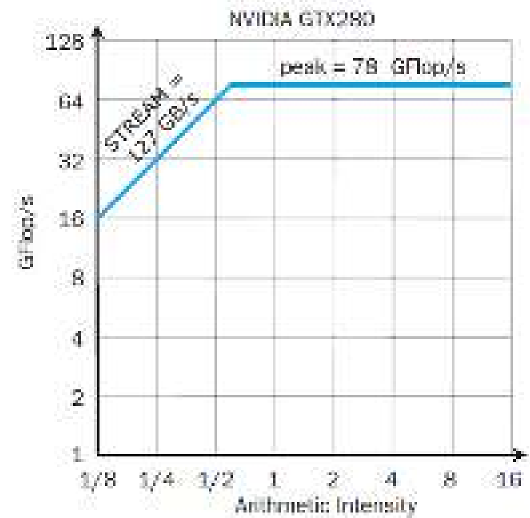
- Arithmetic intensity is not always fixed
  - May scale with problem size
  - Caching reduces memory accesses
    - Increases arithmetic intensity

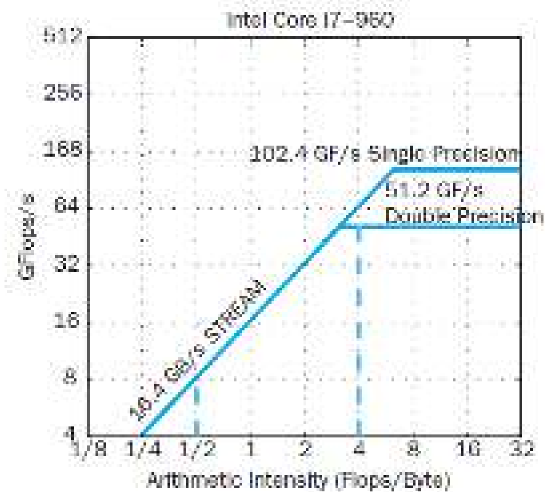
i7-960 vs. NVIDIA Tesla 280/480

	Core i7-960	GTX 280	GTX 480	Ratio 280/i7	Ratio 480/i7
Number of processing elements (cores or SMs)	4	30	15	7.5	3.8
Clock frequency (GHz)	3.2	1.3	1.4	0.41	0.44
Die size	263	576	520	2.2	2.0
Technology	Intel 45 nm	TCMS 65 nm	TCMS 40 nm	1.6	1.0
Power (chip, not module)	130	130	167	1.0	1.3
Transistors	700 M	1400 M	3100 M	2.0	4.4
Memory brandwith (GBytes/sec)	32	141	177	4.4	5.5
Single frecision SIMD width	4	8	32	2.0	8.0
Dobule precision SIMD with	2	1	16	0.5	8.0
Peak Single frecision scalar FLOPS (GFLOP/sec)	26	117	63	4.6	2.5
Peak Single frecision s SIMD FLOPS (GFLOP/Sec)	102	311 to 933	515 to 1344	3.0-9.1	6.6-13.1
(SP 1 add or multiply)	N.A.	(311)	(515)	(3.0)	(6.6)
(SP 1 instruction fused)	N.A	(622)	(1344)	(6.1)	(13.1)
(face SP dual issue fused)	N.A	(933)	N.A	(9.1)	–
Peal double frecision SIMD FLOPS (GFLOP/sec)	51	78	515	1.5	10.1

Rooflines







Benchmarks

Kernel	Units	Core I7-960	GTX 280	GTX 280/ I7-960
SGEMM	GFLOP/sec	94	364	3.9
MC	Billion paths/sec	0.8	1.4	1.8
Conv	Million pixels/sec	1250	3500	2.8
FFT	GFLOP/sec	71.4	213	3.0
SAXPY	GBytes/sec	16.8	88.8	5.3
LBM	Million lookups/sec	85	426	5.0
Solv	Frames/sec	103	52	0.5
SpMV	GFLOP/sec	4.9	9.1	1.9
GJK	Frames/sec	67	1020	15.2
Sort	Million elements/sec	250	198	0.8
RC	Frames/sec	5	8.1	1.6
Search	Million queries/sec	50	90	1.8
Hist	Million pixels/sec	1517	2583	1.7
Bilat	Million pixels/sec	83	475	5.7

Performance Summary

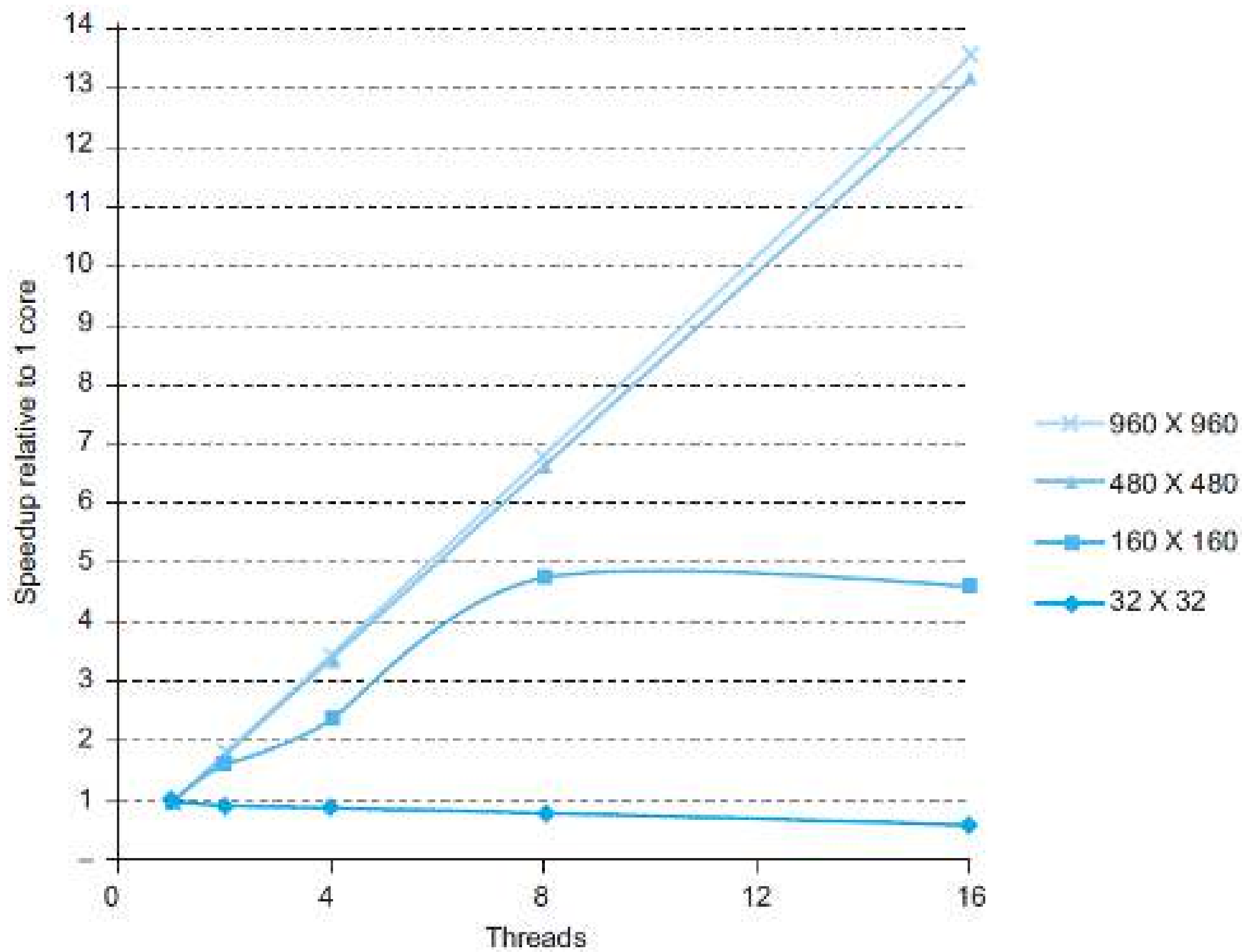
- GPU (480) has 4.4 X the memory bandwidth
  - Benefits memory bound kernels
- GPU has 13.1 X the single precision throughout, 2.5 X the double precision throughput
  - Benefits FP compute bound kernels
- CPU cache prevents some kernels from becoming memory bound when they otherwise would on GPU
- GPUs offer scatter-gather, which assists with kernels with strided data
- Lack of synchronization and memory consistency support on GPU limits performance for some kernels

Multi-threading DGEMM

- Use OpenMP:

```
void dgemm (int n, double* A, double* B, double* C)
{
#pragma omp parallel for
 for ( int sj = 0; sj < n; sj += BLOCKSIZE )
  for ( int si = 0; si < n; si += BLOCKSIZE )
   for ( int sk = 0; sk < n; sk += BLOCKSIZE )
    do_block(n, si, sj, sk, A, B, C);
}
```

Multithreaded DGEMM



Multithreaded DGEMM

32x32 160x160 480x480 960x960

200

150

100

50

-

GFLOPS

14 13 12 12

12 20 22 23

11 31 43 44

11 61 85 87

8 60 169 174

1 2 4 8 16

Threads

Fallacies

- Amdahl's Law doesn't apply to parallel computers
  - Since we can achieve linear speedup
  - But only on applications with weak scaling
- Peak performance tracks observed performance
  - Marketers like this approach!
  - But compare Xeon with others in example
  - Need to be aware of bottlenecks

Pitfalls

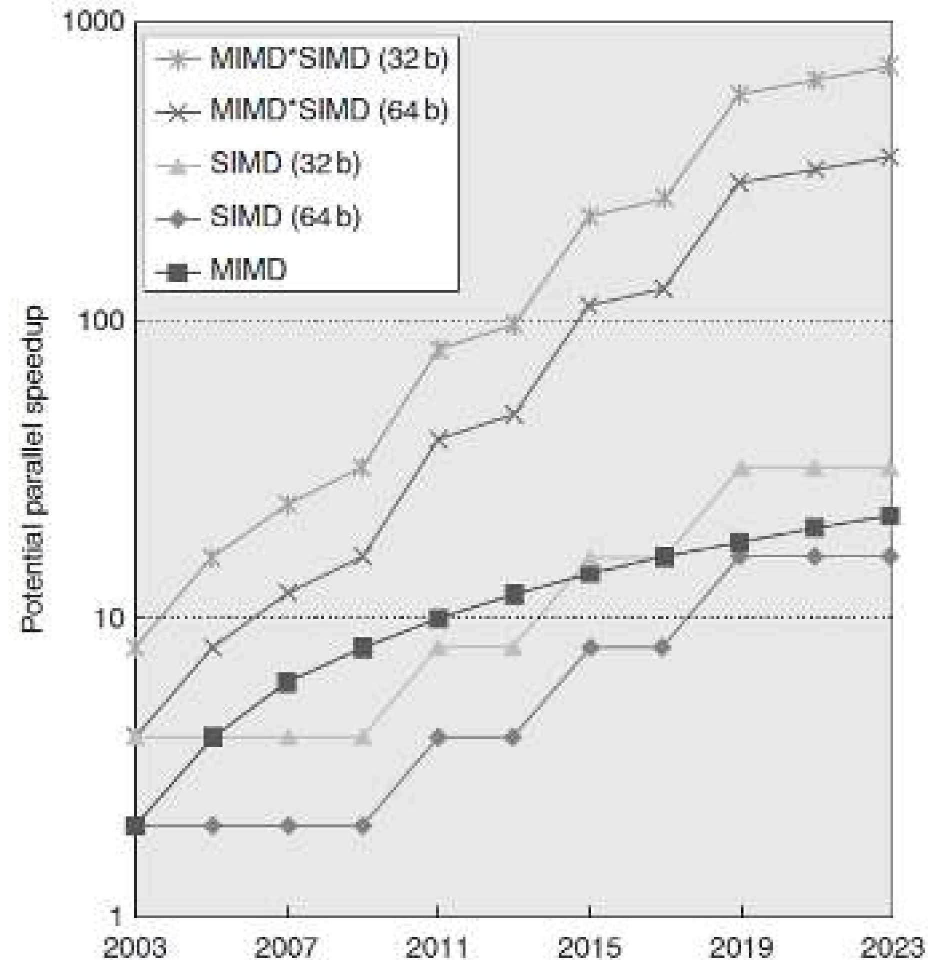
- Not developing the software to take account of a multiprocessor architecture
  - Example: using a single lock for a shared composite resource
    - Serializes accesses, even if they could be done in parallel
    - Use finer-granularity locking

Concluding Remarks

- Goal: higher performance by using multiple processors
- Difficulties
  - Developing parallel software
  - Devising appropriate architectures
- SaaS importance is growing and clusters are a good match
- Performance per dollar and performance per Joule drive both mobile and WSC

Concluding Remarks (con't)

- SIMD and vector operations match multimedia applications and are easy to program

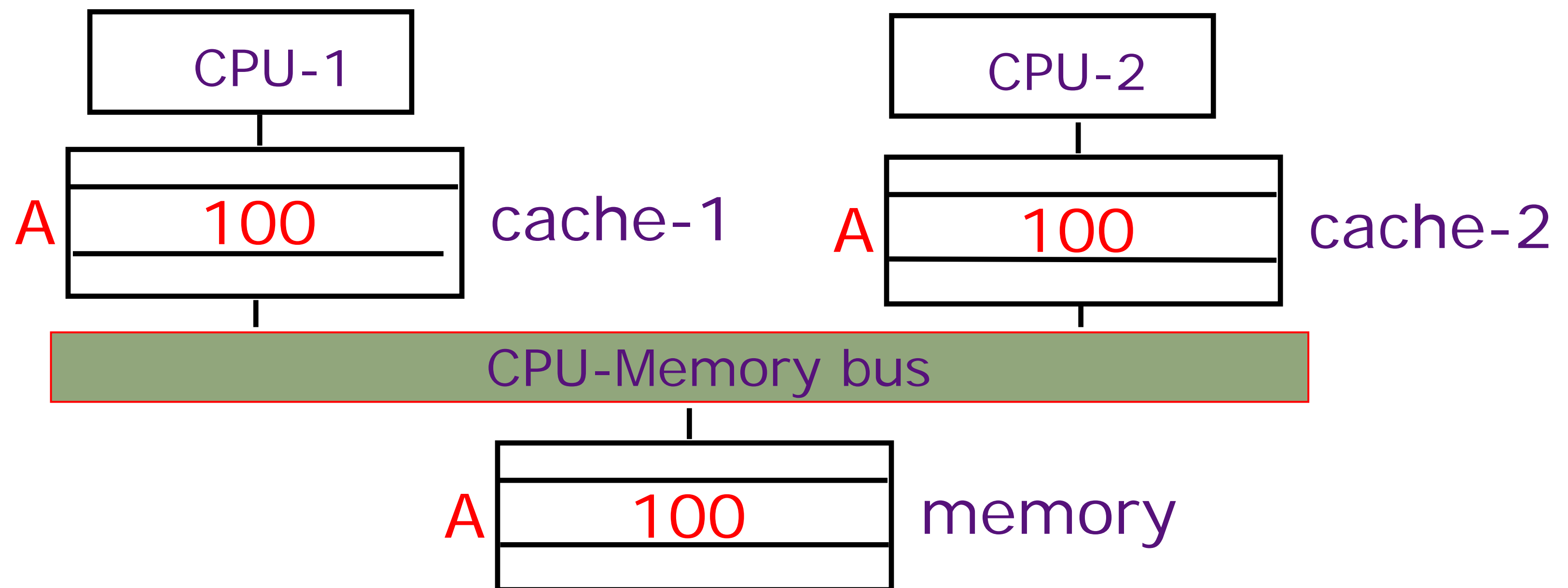


Sequential Consistency and Cache Coherence Protocols

Arvind
Computer Science and Artificial Intelligence Lab
M.I.T.

Based on the material prepared by Arvind and Krste Asanovic

Memory Consistency in SMPs



Suppose CPU-1 updates A to 200.

write-back: memory and cache-2 have stale values

write-through: cache-2 has a stale value

Do these stale values matter?

What is the view of shared memory for programming?

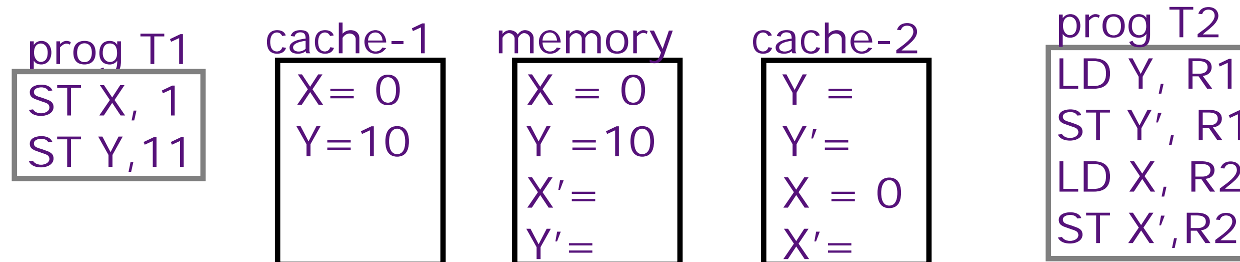
Write-back Caches & SC

	prog T1	cache-1	memory	cache-2	prog T2

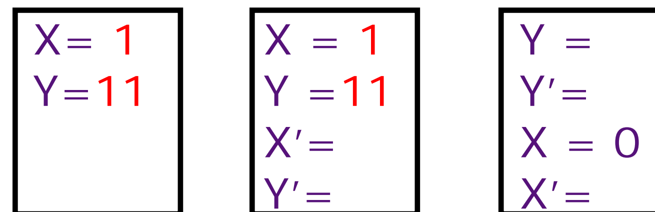
• T1 is executed	ST X, 1
	ST Y,11
		X= 1
		Y=11
			X = 0
			Y =10
			X′=
			Y′=
				Y =
				Y′=
				X =
				X′=
					LD Y, R1
					ST Y′, R1
					LD X, R2
					ST X′,R2

• cache-1 writes back Y		X= 1 Y=11	X = 0 Y =11 X′= Y′=	Y = Y′= X = X′=	
• T2 executed		X= 1 Y=11	X = 0 Y =11 X′= Y′=	Y = 11 Y′= 11 X = 0 X′= 0	
• cache-1 writes back X		X= 1 Y=11	X = 1 Y =11 X′= Y′=	Y = 11 Y′= 11 X = 0 X′= 0	
• cache-2 writes back X′ & Y′		X= 1 Y=11	X = 1 Y =11 X′= 0 Y′=11	Y =11 Y′=11 X = 0 X′= 0	incoherent

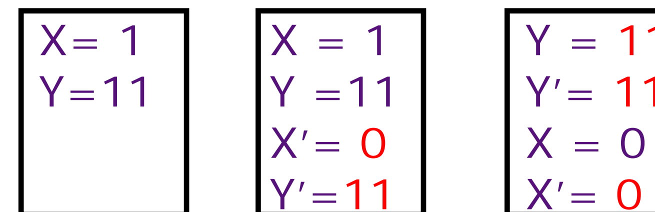
Write-through Caches & SC



- T1 executed



- T2 executed



Write-through caches don't preserve sequential consistency either

CSAIL

Maintaining Sequential Consistency

SC is sufficient for correct producer-consumer and mutual exclusion code (e.g., Dekker)

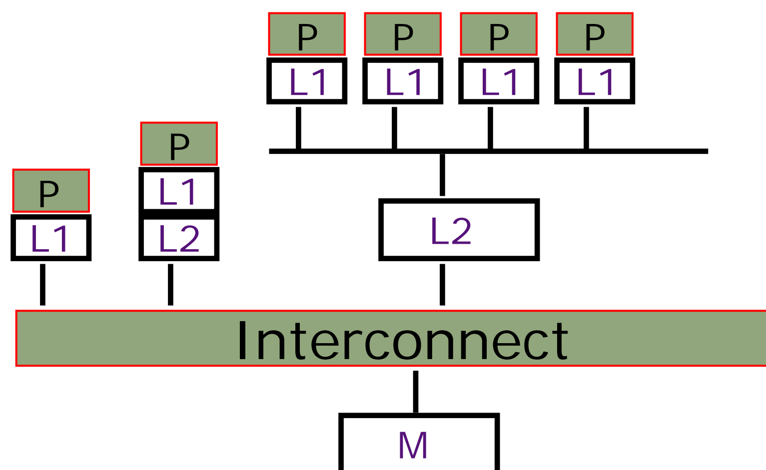
Multiple copies of a location in various caches can cause SC to break down.

Hardware support is required such that

- only one processor at a time has write permission for a location
- no processor can load a stale copy of the location after a write

$\Rightarrow$ *cache coherence protocols*

A System with Multiple Caches



- Modern systems often have hierarchical caches
- Each cache has exactly one parent but can have zero or more children
- Only a parent and its children can communicate directly
- *Inclusion property* is maintained between a parent and its children, i.e.,

$$a \in L_i \Rightarrow a \in L_{i+1}$$

CSAIL

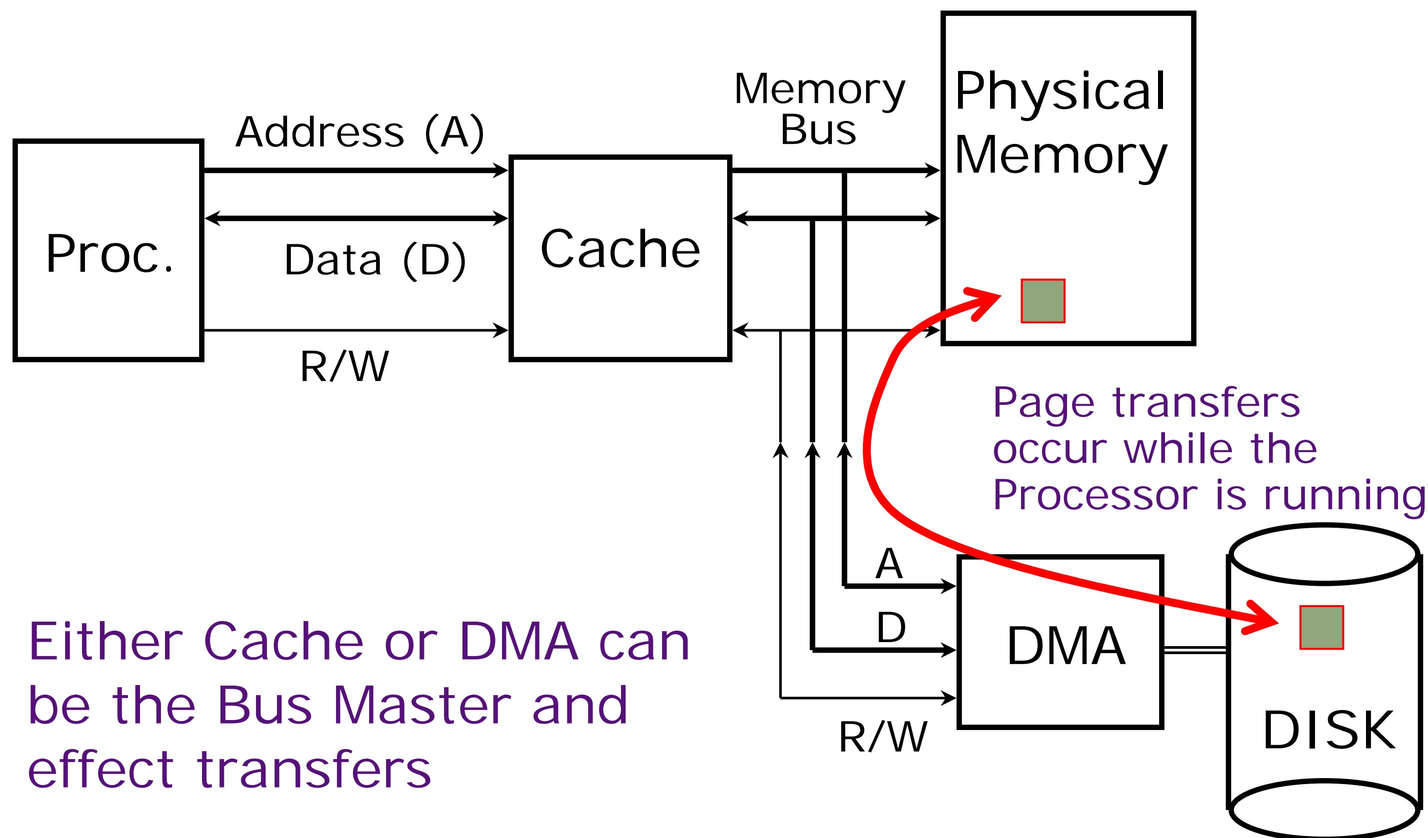
Cache Coherence Protocols for SC

write request:
the address is *invalidated* (*updated)* in all other caches *before* (*after*) the write is performed

read request:
if a dirty copy is found in some cache, a write-back is performed before the memory is read

We will focus on Invalidation protocols as opposed to Update protocols

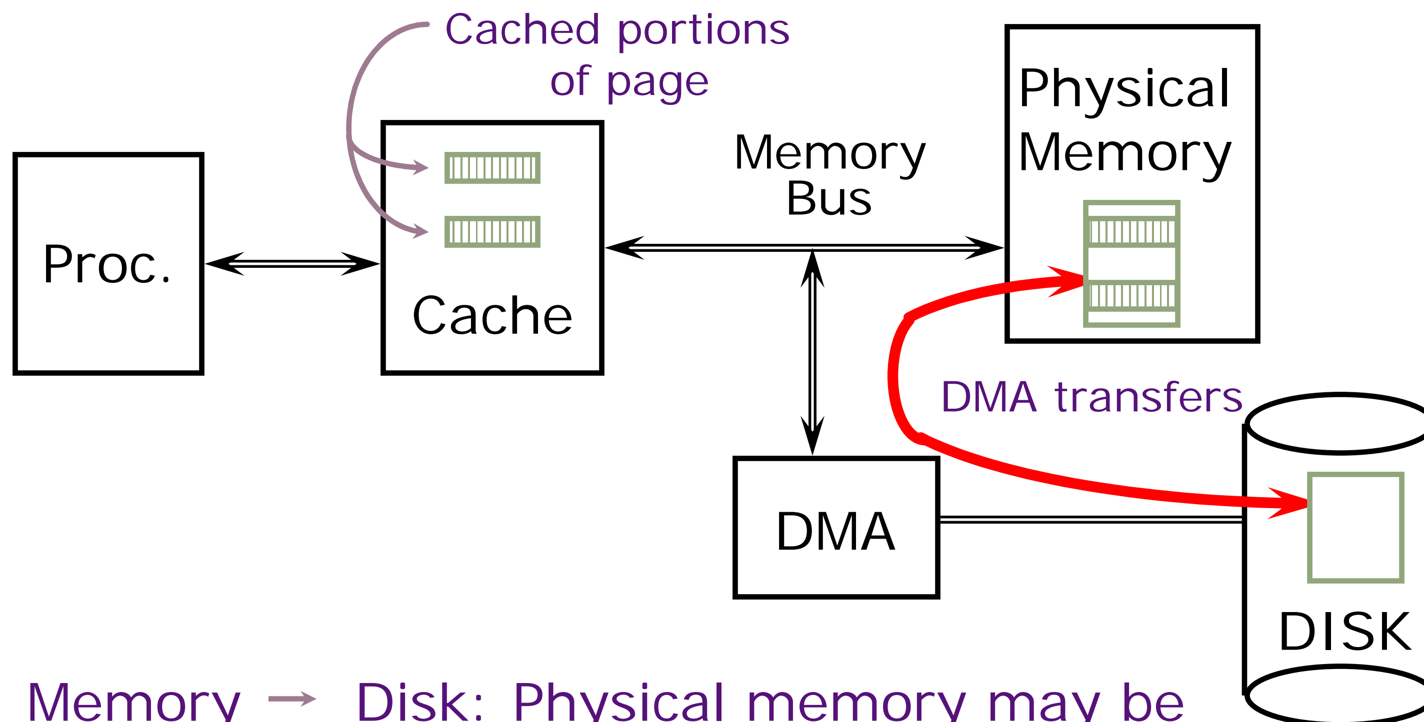
Warmup: Parallel I/O



DMA stands for Direct Memory Access

CSAIL

Problems with Parallel I/O

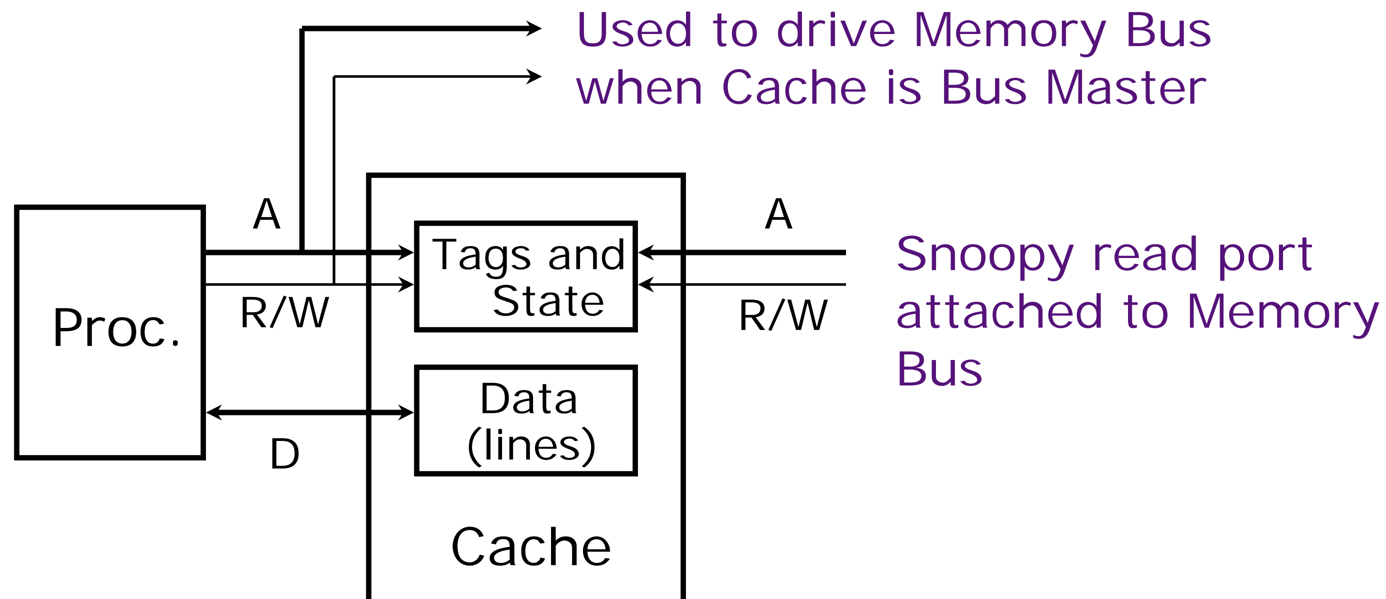


Memory → Disk: Physical memory may be stale if Cache copy is dirty

Disk → Memory: Cache may have data corresponding to the memory

Snoopy Cache *Goodman 1983*

- Idea: Have cache watch (or snoop upon) DMA transfers, and then "do the right thing"
- Snoopy cache tags are dual-ported



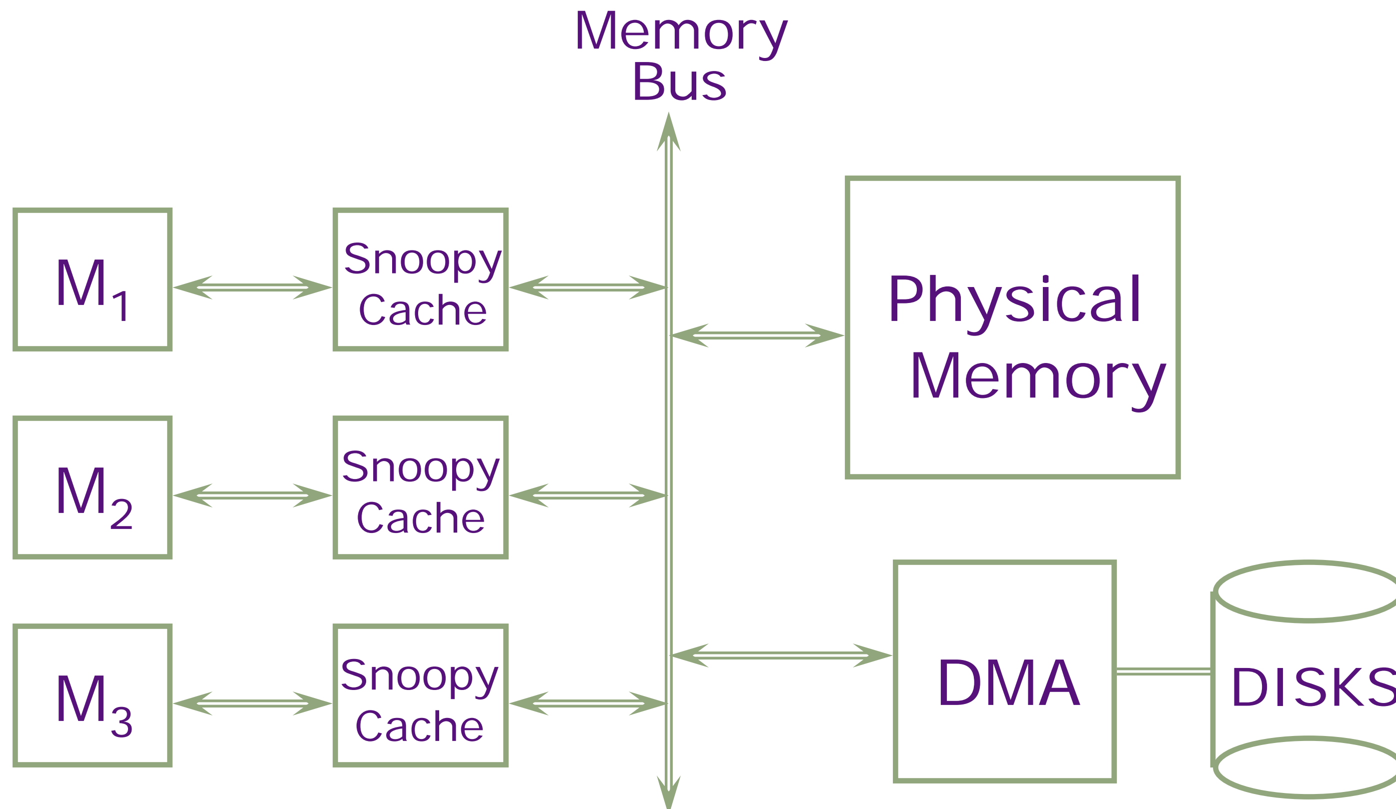
CSAIL

Snoopy Cache Actions

Observed Bus Cycle	Cache State	Cache Action
	Address not cached	No action
Read Cycle	Cached, unmodified	No action
Memory → Disk	Cached, modified	Cache intervenes
	Address not cached	No action
Write Cycle	Cached, unmodified	Cache purges its copy
Disk → Memory	Cached, modified	???

CSAIL

Shared Memory Multiprocessor



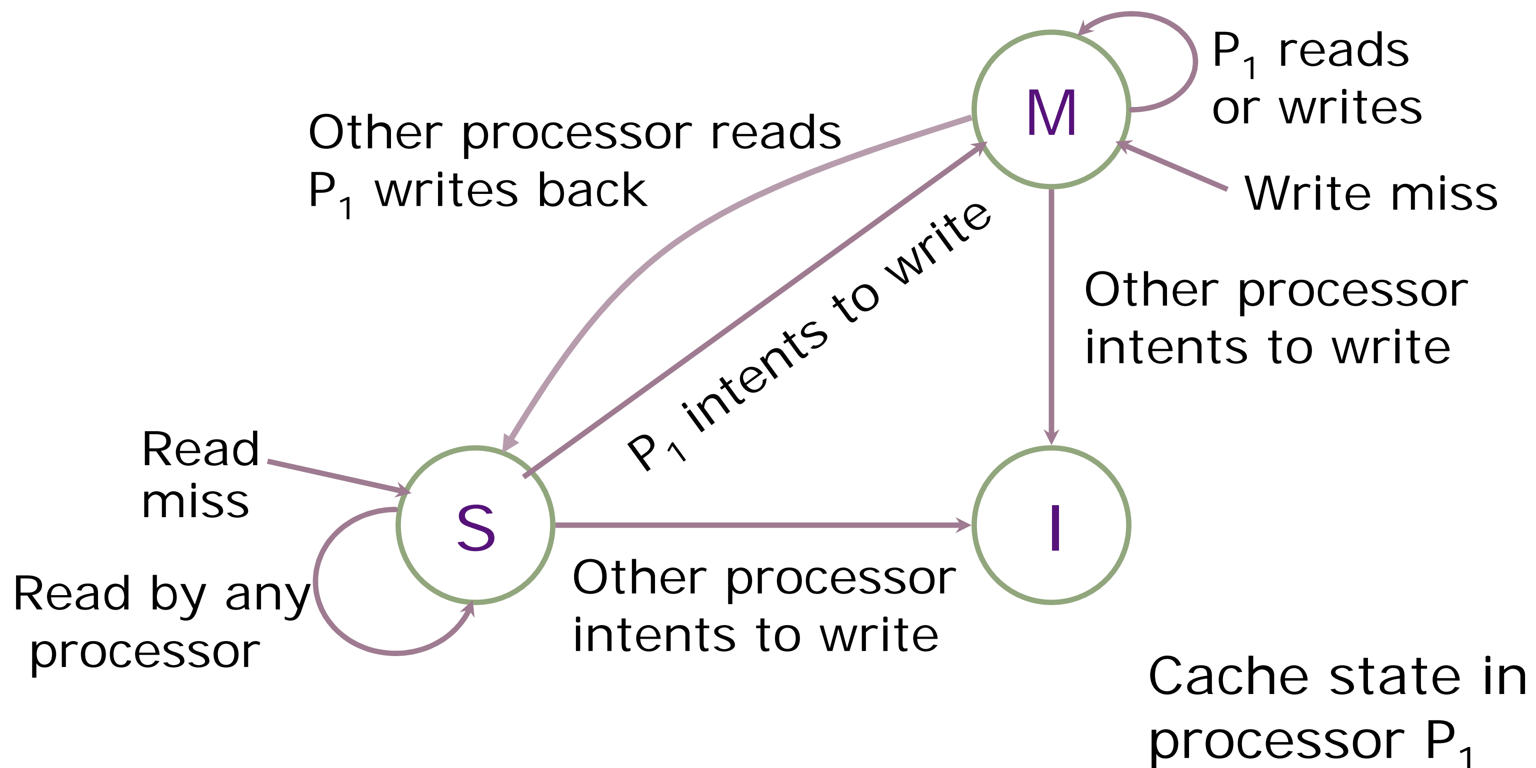
Use snoopy mechanism to keep all processors' view of memory coherent

CSAIL

Cache State Transition Diagram
The MSI protocol

Each cache line has a tag

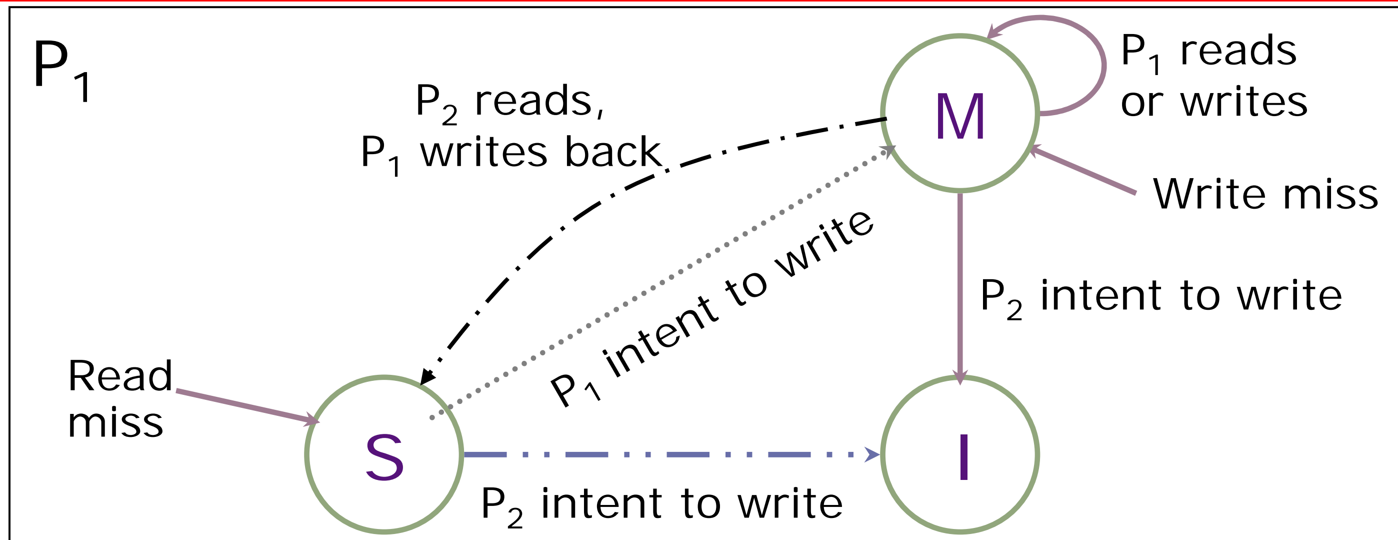
M: Modified
S: Shared
I: Invalid

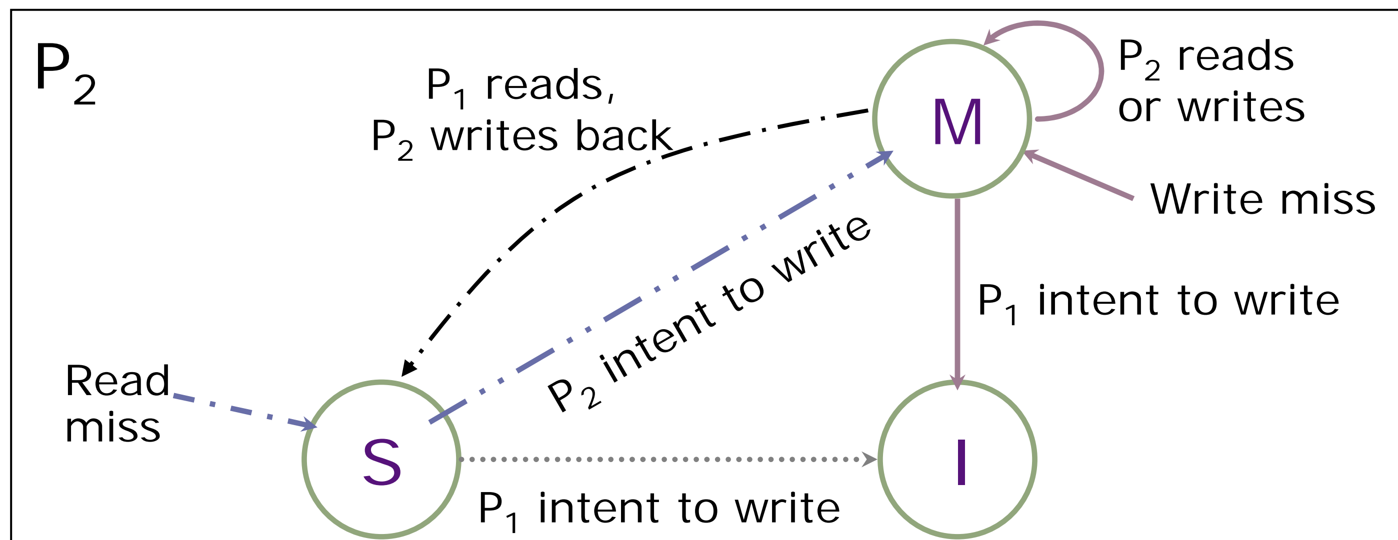


Cache state in processor P_1

2 Processor Example

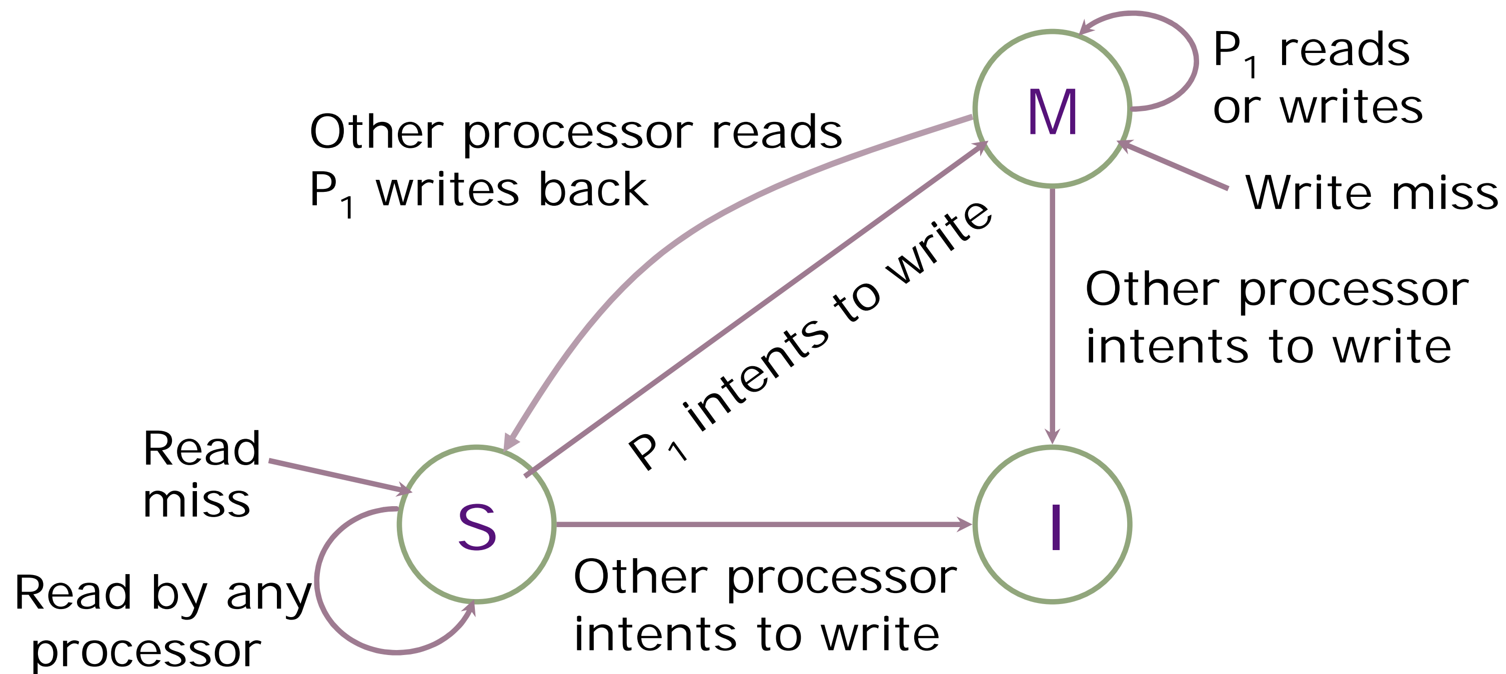
- P_1 reads
- P_1 writes
- P_2 reads
- P_2 writes
- P_1 reads
- P_1 writes
- P_2 writes
- P_1 writes





CSAIL

Observation



- If a line is in the M state then no other cache can have a copy of the line!
  - Memory stays coherent, multiple differing copies cannot exist

CSAIL

MESI: An Enhanced MSI protocol

Each cache line has a tag

M: Modified Exclusive
E: Exclusive, unmodified
S: Shared
I: Invalid

M
E
S
I

P_1 write or read

P_1 write

P_1 read

Write miss

Other processor reads
P_1 writes back

P_1 intent to write

Other processor intent to write

Read miss, shared

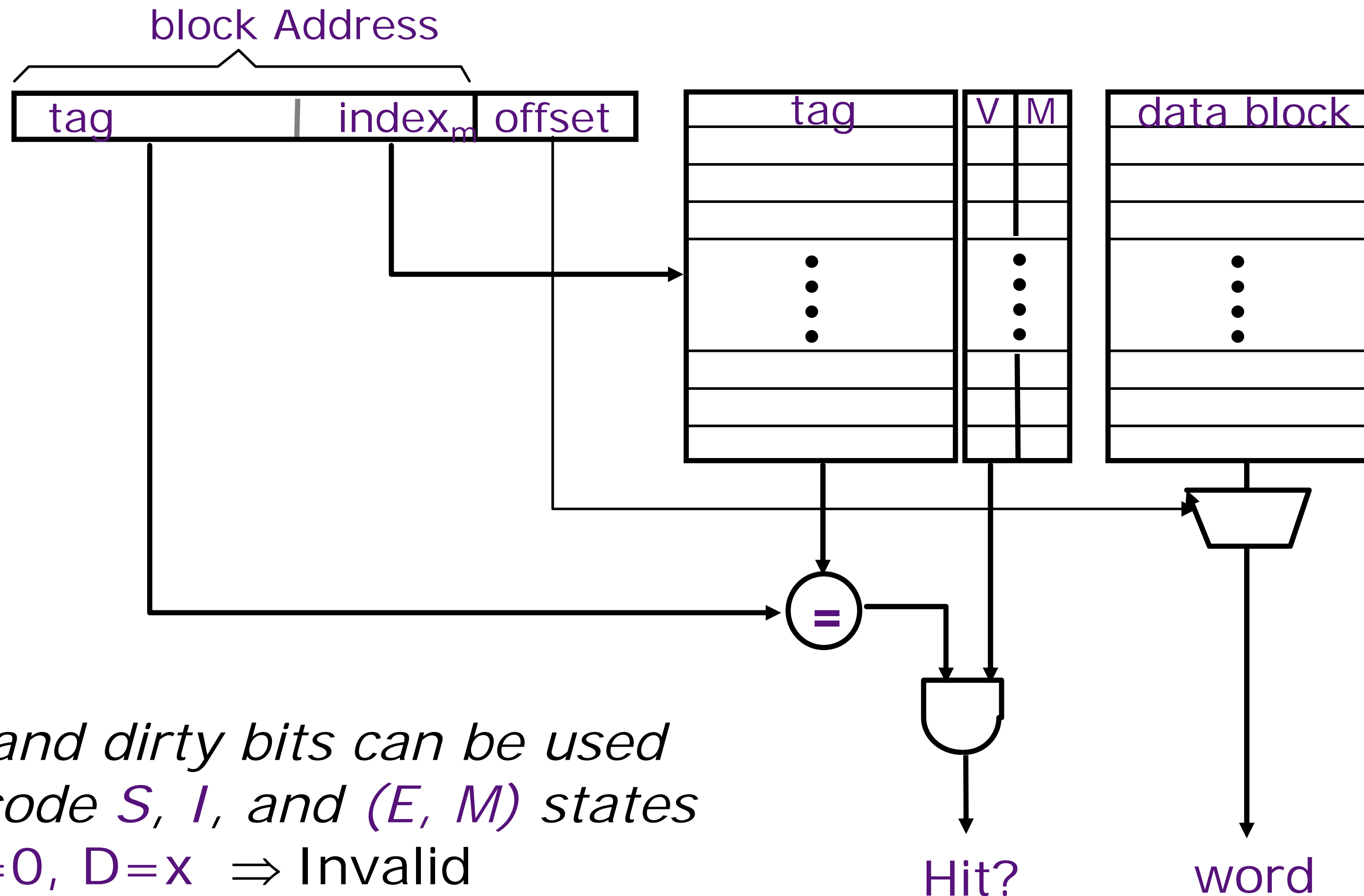
Read by any processor

Other processor intent to write

Cache state in processor P_1

Five-minute break to stretch your legs

Cache Coherence State Encoding



Valid and dirty bits can be used to encode S, I, and (E, M) states

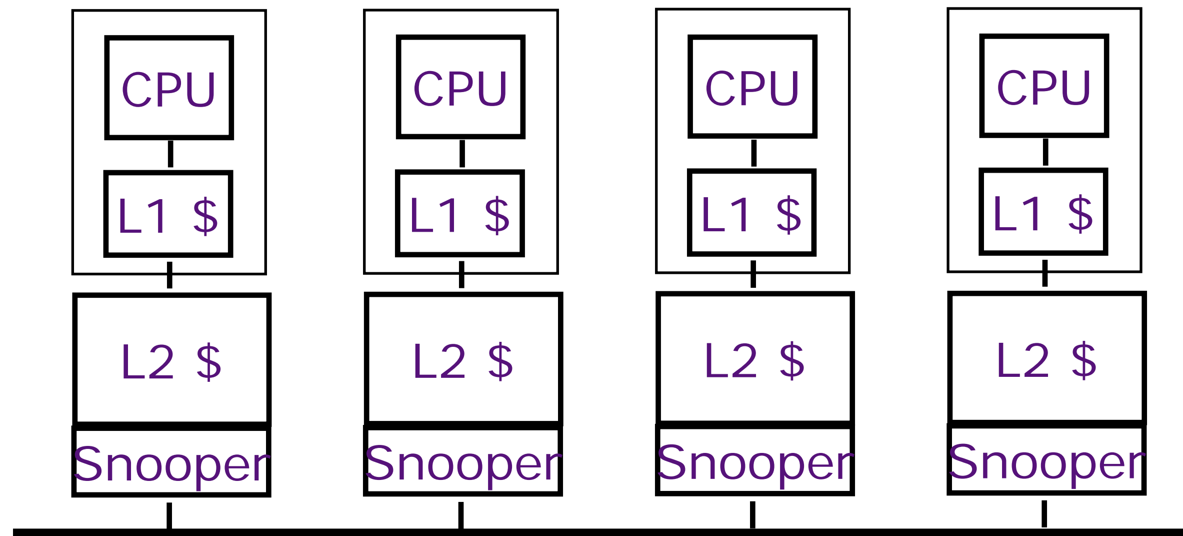
V=0, D=x ⇒ Invalid

V=1, D=0 ⇒ Shared *(not dirty)*

V=1, D=1 ⇒ Exclusive *(dirty)*

CSAIL

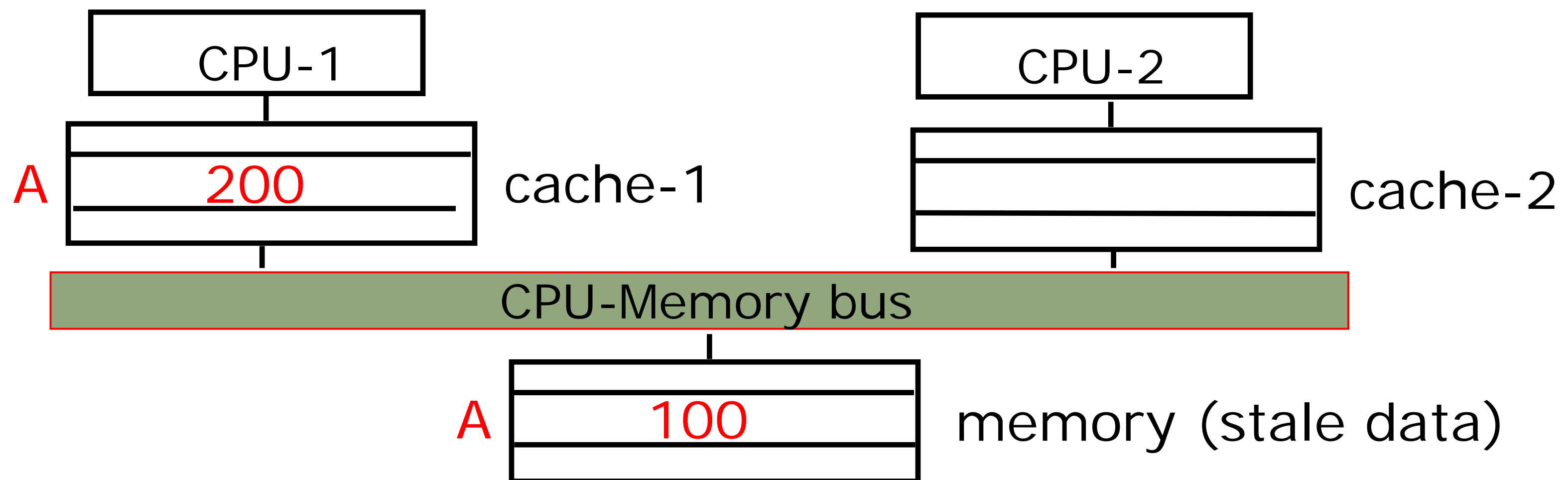
2-Level Caches



- Processors often have two-level caches
  - Small L1 on chip, large L2 off chip
- *Inclusion property:* entries in L1 must be in L2
  invalidation in L2 ⇒ invalidation in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

What problem could occur?

Intervention



When a read-miss for A occurs in cache-2, a read request for A is placed on the bus

- Cache-1 needs to supply & change its state to shared
- The memory may respond to the request also!

Does memory know it has stale data?

Cache-1 needs to intervene through memory controller to supply correct data to cache-2

CSAIL

False Sharing

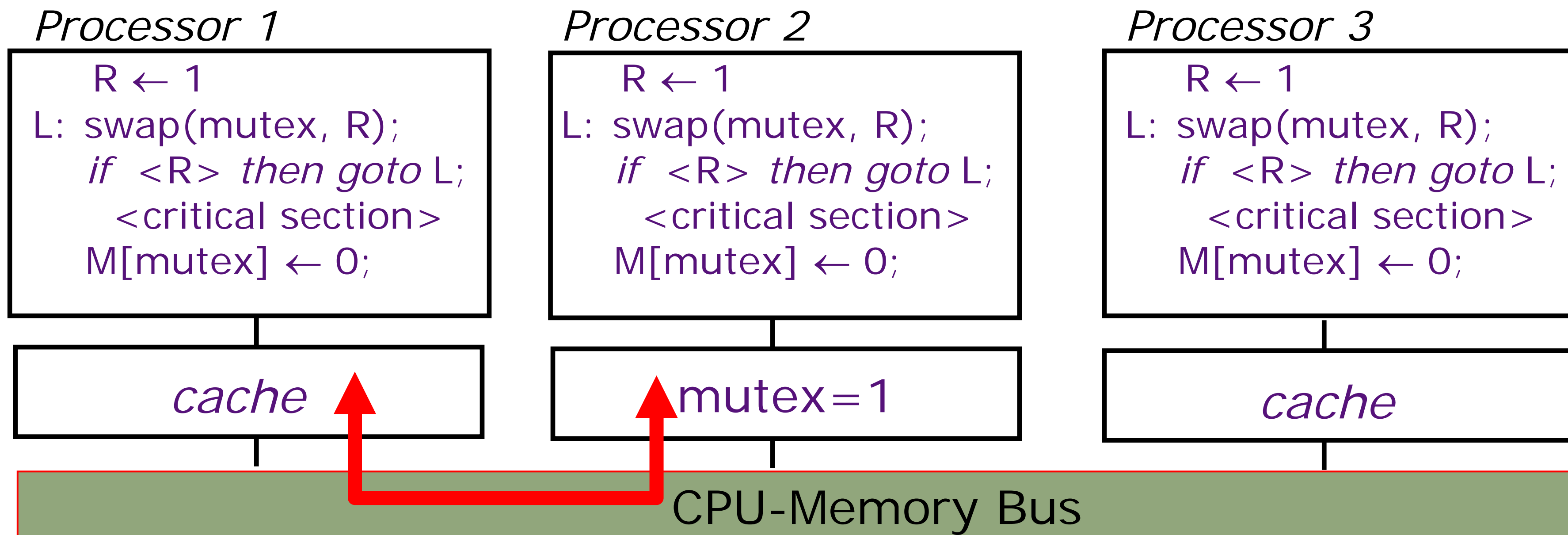
A cache block contains more than one word

Cache-coherence is done at the block-level and not word-level

Suppose M_1 writes $word_i$ and M_2 writes $word_k$ and both words have the same block address.

What can happen?

Synchronization and Caches:
Performance Issues



Cache-coherence protocols will cause mutex to *ping-pong* between P1's and P2's caches.

Ping-ponging can be reduced by first reading the mutex location *(non-atomically)* and executing a swap only if it is found to be zero.

CSAIL

Performance Related to Bus occupancy

In general, a *read-modify-write* instruction requires two memory (bus) operations without intervening memory operations by other processors

In a multiprocessor setting, bus needs to be locked for the entire duration of the atomic read and write operation

$\Rightarrow$ expensive for simple buses

$\Rightarrow$ *very expensive* for split-transaction buses

modern processors use

load-reserve

store-conditional

CSAIL

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve(R, a):
    <flag, adr> ← <1, a>;
    R ← M[a];
```

```
Store-conditional(a, R):
    if <flag, adr> == <1, a>
    then  cancel other procs'
          reservation on a;
          M[a] ← <R>;
          status ← succeed;
    else  status ← fail;
```

If the snooper sees a store transaction to the address in the reserve register, the reserve bit is set to 0

- Several processors may reserve 'a' simultaneously
- These instructions are like ordinary loads and stores with respect to the bus traffic

Performance:
Load-reserve & Store-conditional

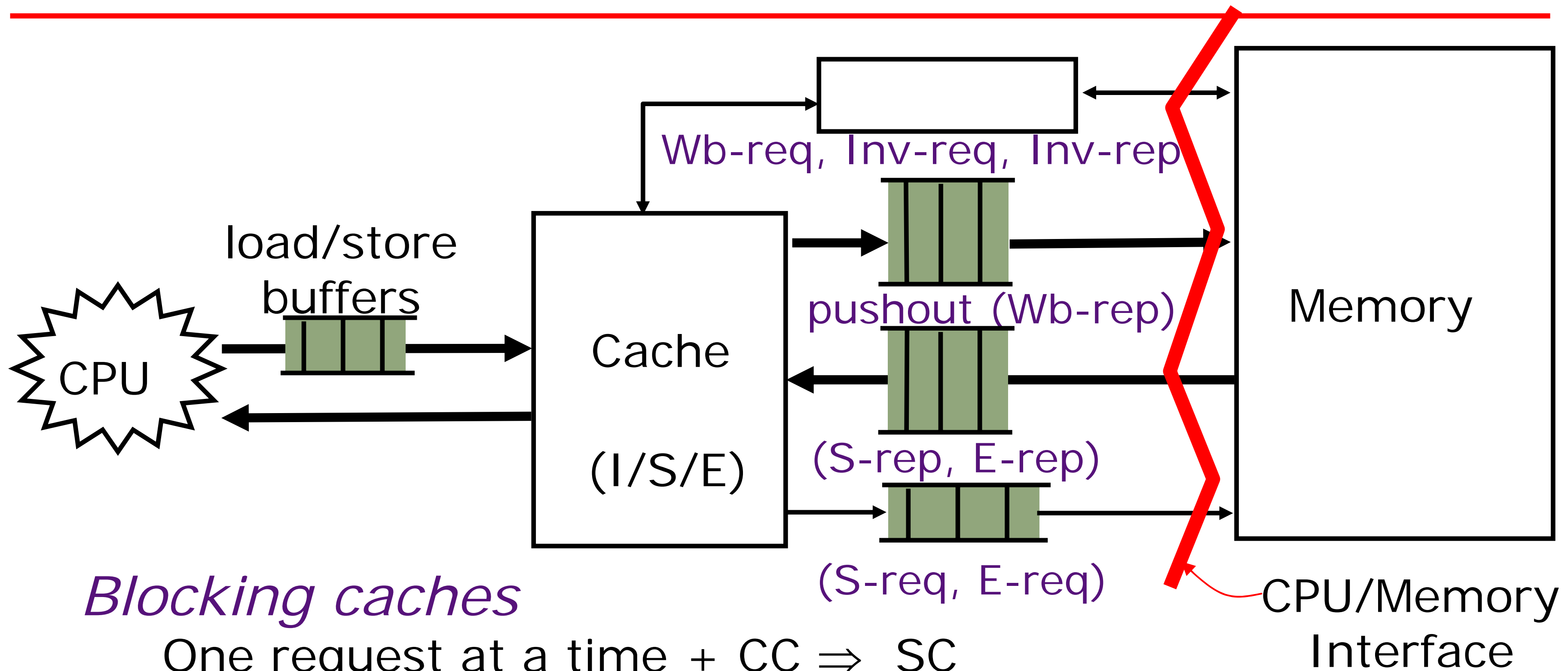
The total number of memory (bus) transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

- *increases bus utilization* (and reduces processor stall time), especially in split-transaction buses

- *reduces cache ping-pong effect* because processors trying to acquire a semaphore do not have to perform a store each time

CSAIL

Out-of-Order Loads/Stores & CC



Blocking caches

One request at a time + CC $\Rightarrow$ SC

Non-blocking caches

Multiple requests (different addresses) concurrently + CC $\Rightarrow$ Relaxed memory models

CC ensures that all processors observe the same order of loads and stores to an address

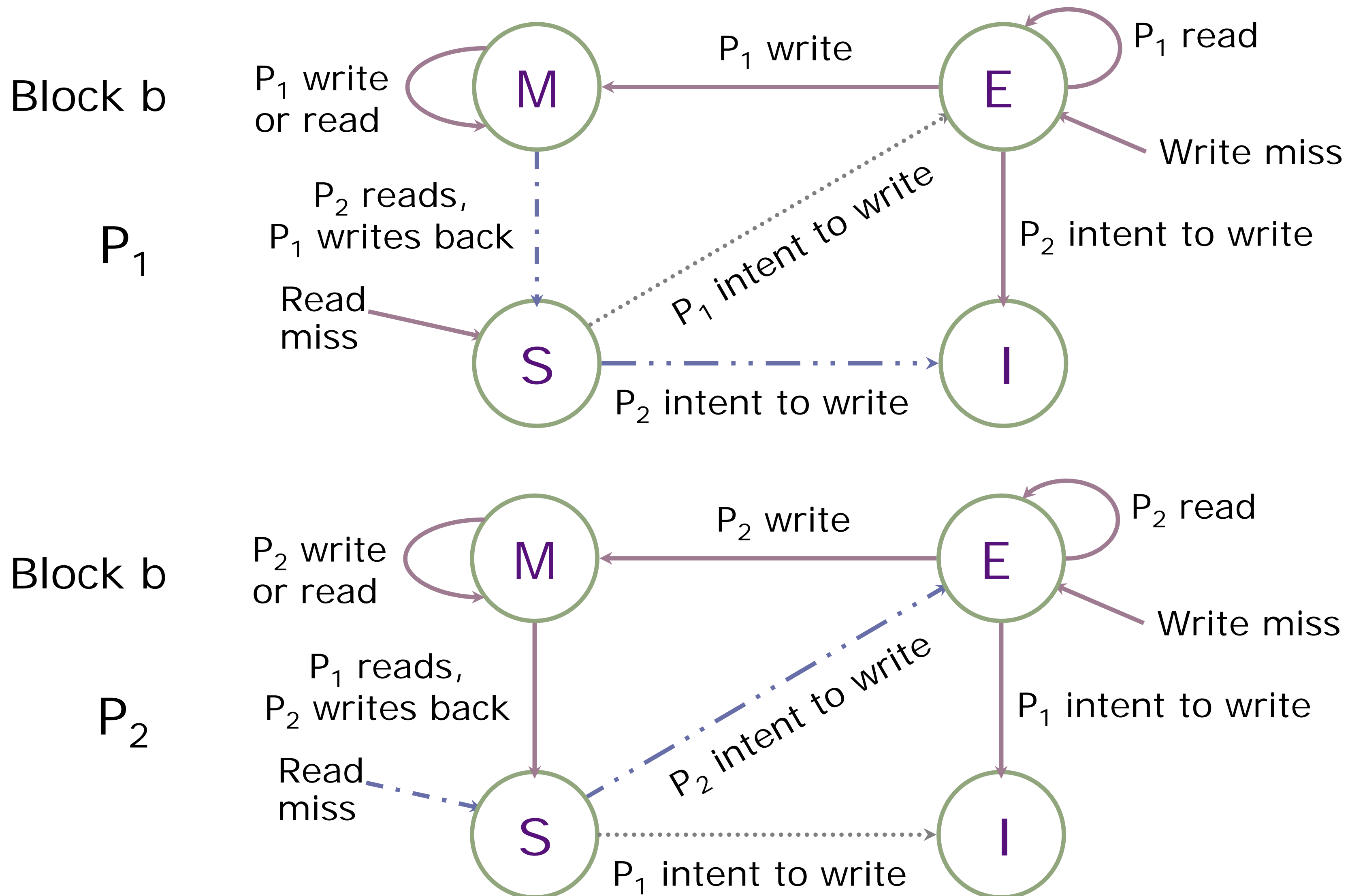
next time

Designing a Cache Coherence Protocol

CSAIL

Thank you !

2 Processor Example



CSAIL