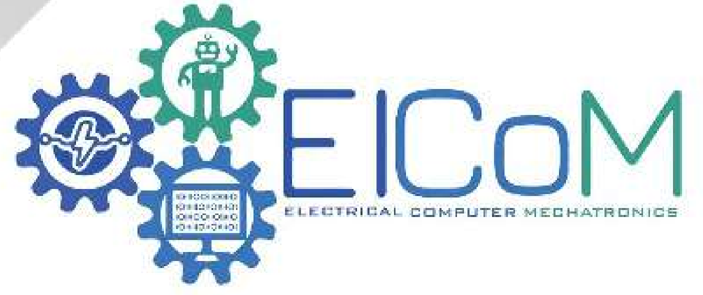


تنظيم حاسوب

للطالبة المبدعة
فيحاء الحديدي

إرادة - ثقة - تغيير

COD Ch. 1
Computer Abstractions and Technology

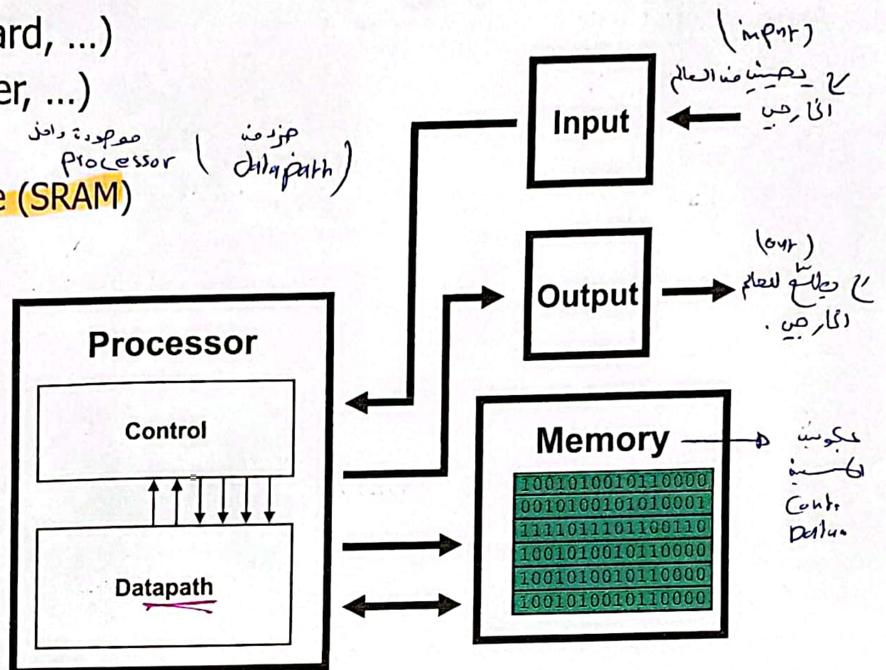
Introduction

- Rapidly changing field:
  - vacuum tube -> transistor -> IC -> VLSI
  - doubling every 1.5 years:
    - memory capacity
    - processor speed (due to advances in technology and hardware organization)
  - cute example: if Boeing had kept up with IBM we could *fly from Bangkok to HCM City in 10 minutes for 5 baht (2000 dong)* !!
- Things we'll be learning:
  - how computers work, what's a good design, what's not
  - how to make them – *yes, we will actually build working computers*!!
  - issues affecting modern processors (e.g., caches, pipelines)

CamScanner المسوحة ضوئيا بـ

The Five Classic Components of a Computer

- Input (mouse, keyboard, ...)
- Output (display, printer, ...)
- Memory
  - main (DRAM), cache (SRAM)
  - secondary (disk, CD, DVD, ...)
- Datapath } Processor
- Control } (CPU)



Our Primary Focus

- The processor (CPU)...
  - datapath
  - control
- ...implemented using millions of transistors
- ...impossible to understand by looking at individual transistors
- we need...

الممسوحة ضوئيا بـ CamScanner

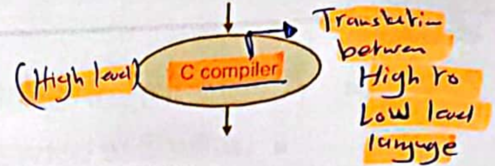
Abstraction

- Delving into the depths reveals more information, but…
- An abstraction omits "unneeded" detail, helps us cope with complexity

- *From the figure on the right, how does abstraction help the programmer and how does she avoid too much detail?*

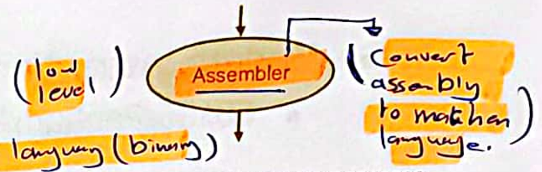
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

```
swap:
   muli $2, $5,4
   add  $2, $4,$2
   lw   $15, 0($2)
   lw   $16, 4($2)
   sw   $16, 0($2)
   sw   $15, 4($2)
   jr   $31
```

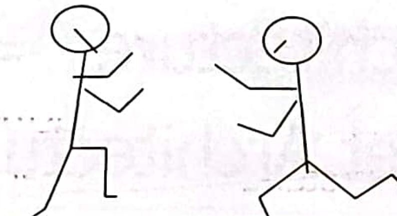


Binary machine language program (for MIPS)

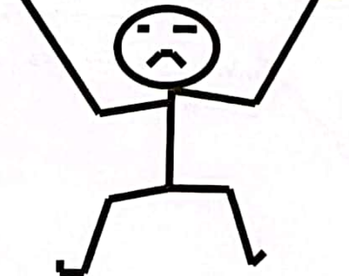
```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

The Instruction Set: a Critical Interface

software



instruction set



hardware

Instruction Set Architecture

- A very important abstraction:
  - *interface* between hardware and low-level software
  - *standardizes* instructions, machine language bit patterns, etc.
  - advantage: *allows different implementations of the same architecture*
  - disadvantage: *sometimes prevents adding new innovations*

- Modern instruction set architectures:
  - 80x86/Pentium/K6, PowerPC, DEC Alpha, MIPS, SPARC, HP

What is Computer Architecture? Easy Answer

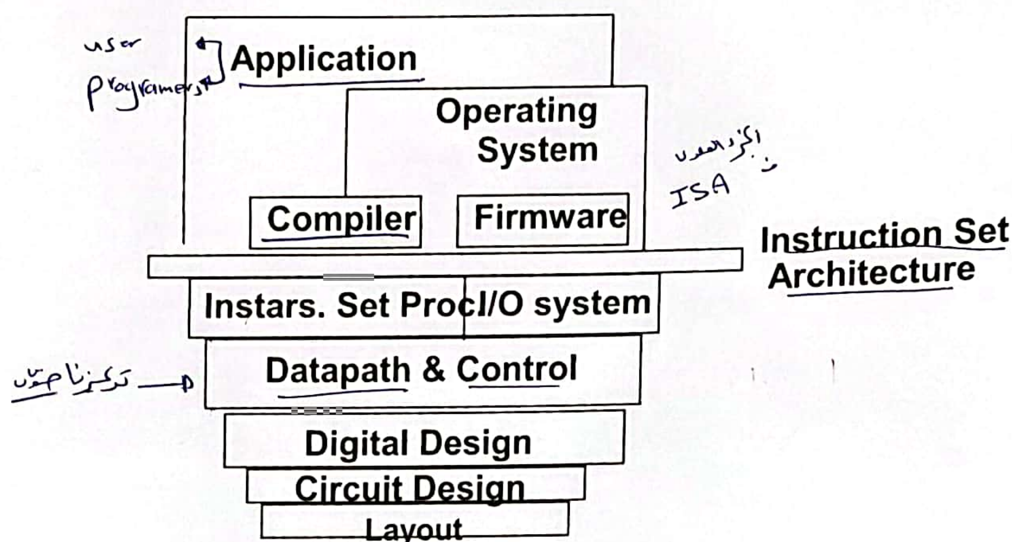
Computer Architecture =

Instruction Set Architecture +

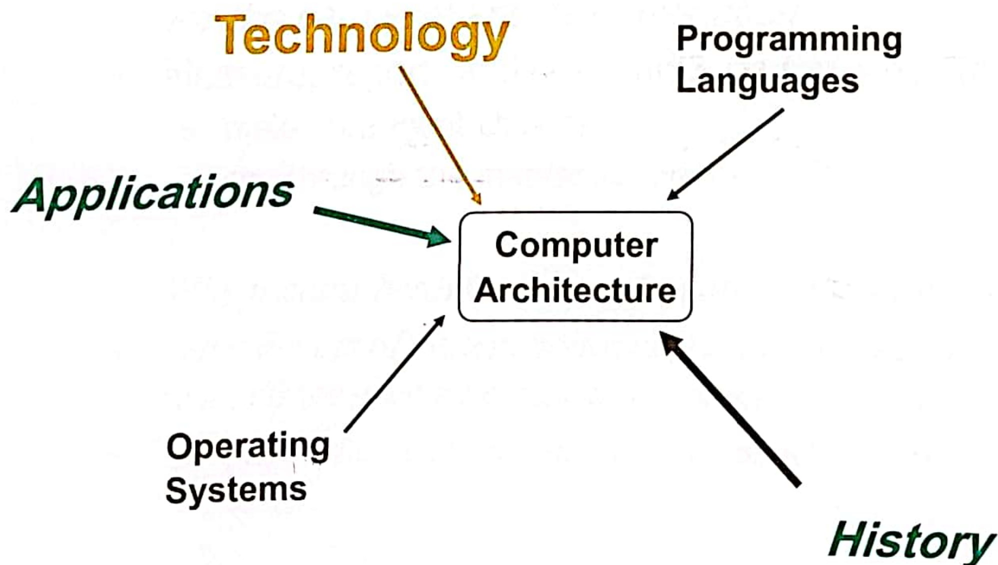
Machine Organization

CamScanner الممسوحة ضوئيا بـ

What is Computer Architecture? Better (More Detailed) Answer



Forces on Computer Architecture



CamScanner المسوحة ضوئيا بـ

COD Ch. 2
The Role of Performance

Performance

- *Performance is the key to understanding underlying motivation for the hardware and its organization*
- Measure, report, and summarize performance to enable users to
  - make intelligent choices
  - see through the marketing hype!

- *Why is some hardware better than others for different programs?*
- *What factors of system performance are hardware related?*
  (e.g., do we need a new machine, or a new operating system?)
- *How does the machine's instruction set affect performance?*

CamScanner الممسوحة ضوئيا بـ

What do we measure? Define performance….

Airplane	Passengers	Range (mi)	Speed (mph)
Boeing 737-100	101	630	598
Boeing 747	470	4150	610
BAC/Sud Concorde	132	4000	1350
Douglas DC-8-50	146	8720	544

- How much faster is the Concorde compared to the 747?
- How much bigger is the Boeing 747 than the Douglas DC-8?

- *So which of these airplanes has the best performance?!*

Computer Performance: TIME, TIME, TIME!!!

- *Response Time* (*elapsed time, latency*):
  - how long does it take for *my* job to run?
  - how long does it take to execute (start to finish) *my* job?
  - how long must *I* wait for the database query?

  Individual user concerns…

- *Throughput*:
  - how *many* jobs can the machine run at once?
  - what is the *average* execution rate?
  - how *much* work is getting done?

  Systems manager concerns…

- *If we upgrade a machine with a new processor what do we increase?*
- *If we add a new machine to the lab what do we increase?*

الممسوحة ضوئيا بـ CamScanner

Execution Time

- *Elapsed Time*
  - counts everything (*disk and memory accesses, waiting for I/O, running other programs, etc.*) from start to finish
  - a useful number, but often not good for comparison purposes

  elapsed time = CPU time + wait time (I/O, other programs, etc.)

- *CPU time*
  - doesn't count waiting for I/O or time spent running other programs
  - can be divided into *user CPU time* and *system CPU time* (OS calls)

  CPU time = user CPU time + system CPU time

  ⇒ elapsed time = user CPU time + system CPU time + wait time

- Our focus: *user CPU time* (*CPU execution time* or, simply, *execution time*)
  - time spent executing the lines of code that are *in our program*

Definition of Performance

- For some program running on machine X:

$$\text{Performance}_X = 1 / \text{Execution time}_X$$

- *X is n times faster than Y* means: $\Rightarrow \frac{\text{Pref}_x}{\text{Pref}_y} = n$

$$\text{Performance}_X / \text{Performance}_Y = n$$

E : Execution

$$\frac{E_y}{E_x} = n$$

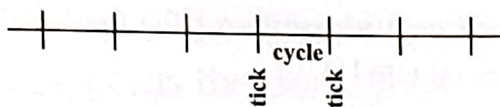
CamScanner الممسوحة ضوئيا بـ

Clock Cycles

- Instead of reporting execution time in seconds, we often use *cycles*. In modern computers hardware events progress cycle by cycle: in other words, each event, e.g., multiplication, addition, etc., is a sequence of cycles

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

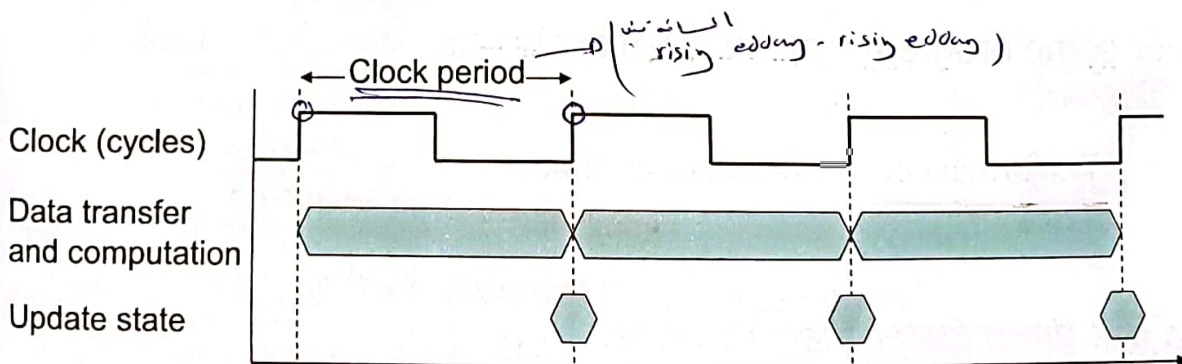
- *Clock ticks* indicate start and end of cycles:



- *cycle time* = time between ticks = seconds per cycle
- *clock rate* (*frequency*) = cycles per second (1 Hz. = 1 cycle/sec, 1 MHz. = 10^6 cycles/sec)
- *Example*: A 200 Mhz clock has a cycle time $\frac{1}{200 \times 10^6} \times 10^9$ = 5 nanoseconds

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



- Clock period: duration of a clock cycle
  - e.g., 250ps = 0.25ns = 250×10^{-12}s
- Clock frequency (rate): cycles per second
  - e.g., 4.0GHz = 4000MHz = 4.0×10^9Hz

الممسوحة ضوئيا بـ CamScanner

Performance Equation I

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

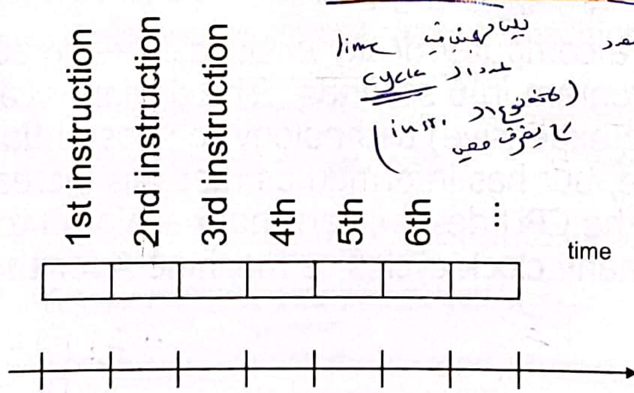
equivalently

$$\text{CPU execution time for a program} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

- So, to improve performance one can either:
  - reduce the number of cycles for a program, or
  - reduce the clock cycle time, or, equivalently,
  - increase the clock rate

How many cycles are required for a program?

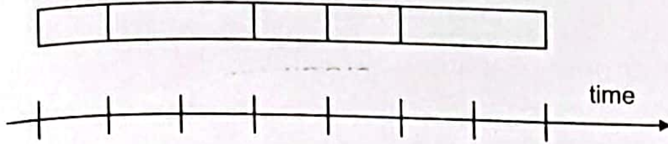
- Could assume that # of cycles = # of instructions



- *This assumption is incorrect!* Because:
  - Different instructions take different amounts of time (cycles)
  - Why...?

CamScanner الممسوحة ضوئيا بـ

How many cycles are required for a program?



Multiplication takes more time than addition

Floating point operations take longer than integer ones

Accessing memory takes more time than accessing registers

Important point: changing the cycle time often changes the number of cycles required for various instructions because it means changing the hardware design. More later...

Example

- Our favorite program runs in 10 seconds on computer A, which has a 400Mhz. clock.
- We are trying to help a computer designer build a new machine B, that will run this program in 6 seconds. The designer can use new (or perhaps more expensive) technology to substantially increase the clock rate, but has informed us that this increase will affect the rest of the CPU design, causing machine B to require 1.2 times as many clock cycles as machine A for the same program.
- *What clock rate should we tell the designer to target?*

CPU Time Example

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6s}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10s \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = 4\text{GHz}$$

Terminology

- A given program will require:
  - some number of instructions (machine instructions)
  - some number of cycles
  - some number of seconds
- We have a vocabulary that relates these quantities:
  - *cycle time* (seconds per cycle)
  - *clock rate* (cycles per second)
  - (average) *CPI* (cycles per instruction)
    - a floating point intensive application might have a higher average CPI
  - *MIPS* (millions of instructions per second)
    - this would be higher for a program using simple instructions

CamScanner الممسوحة ضوئيا بـ

Performance Measure

- *Performance is determined by execution time*

- Do any of these other variables equal performance?
  - # of cycles to execute program?
  - # of instructions in program?
  - # of cycles per second?
  - average # of cycles per instruction? CPI
  - average # of instructions per second?

- *Common pitfall* : thinking one of the variables is indicative of performance when it really isn't

Performance Equation II

CPU execution time for a program = Instruction count for a program × average CPI × Clock cycle time

- *Derive the above equation from Performance Equation I*

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

الممسوحة ضوئيا بـ CamScanner

CPI Example I

- Suppose we have two implementations of the same instruction set architecture (ISA). For some program:
  - machine A has a clock cycle time of 10 ns. and a CPI of 2.0
  - machine B has a clock cycle time of 20 ns. and a CPI of 1.2
- *Which machine is faster for this program, and by how much?*
- *If two machines have the same ISA, which of our quantities (e.g., clock rate, CPI, execution time, # of instructions, MIPS) will always be identical?*

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$ ← A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$

$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$ ← …by this much

المسموحة ضوئيا بـ CamScanner

CPI in More Detail

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)$$

Relative frequency

CPI Example II

- A compiler designer is trying to decide between two code sequences for a particular machine.
- Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require 1, 2 and 3 cycles (respectively).
- The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C
  The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C.
- *Which sequence will be faster? How much? What is the CPI for each sequence?*

CamScanner الممسوحة ضوئيا بـ

CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	**2**	**1**	**2**
IC in sequence 2	**4**	**1**	**1**

$$CPI = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n}\left(CPI_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}}\right)$$

Sequence 1:

IC = 5

CPI = 1 * 2/5 + 2 * 1/5 + 3 * 2/5

= 2.0

Sequence 2:

IC = 6

CPI = 1*4/6 + 2*1/6+3*1/6

= 1.5

Pitfall: MIPS as a Performance Metric

- MIPS: Millions of Instructions Per Second
  - Doesn't account for
    - Differences in ISAs between computers
    - Differences in complexity between instructions

$$MIPS = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} = \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times CPI}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{CPI \times 10^6}$$

- CPI varies between programs on a given CPU

المسموحة ضوئيا بـ CamScanner

MIPS Example

- Two different compilers are being tested for a 500 MHz. machine with three different classes of instructions: Class A, Class B, and Class C, which require 1, 2 and 3 cycles (respectively). Both compilers are used to produce code for a large piece of software.
- Compiler 1 generates code with 5 billion Class A instructions, 1 billion Class B instructions, and 1 billion Class C instructions.
- Compiler 2 generates code with 10 billion Class A instructions, 1 billion Class B instructions, and 1 billion Class C instructions.
- *Which sequence will be faster according to MIPS?*
- *Which sequence will be faster according to execution time?*

Benchmarks

- Performance best determined by running a real application
  - use programs typical of expected workload
  - or, typical of expected class of applications
    e.g., compilers/editors, scientific applications, graphics, etc.
- Small benchmarks
  - nice for architects and designers
  - easy to standardize
  - can be abused!
- Benchmark suites
  - Perfect Club: set of application codes
  - Livermore Loops: 24 loop kernels
  - Linpack: linear algebra package
  - SPEC: mix of code from industry organization

CamScanner الممسوحة ضوئيا بـ

SPEC (System Performance Evaluation Corporation)

- Sponsored by industry but independent and self-managed – trusted by code developers and machine vendors
- Clear guides for testing, see www.spec.org
- Regular updates (benchmarks are dropped and new ones added periodically according to relevance)
- Specialized benchmarks for particular classes of applications
- Can still be abused..., by selective optimization!

SPEC History

- First Round: SPEC CPU89
  - 10 programs yielding a single number
- Second Round: SPEC CPU92
  - SPEC CINT92 (6 integer programs) and SPEC CFP92 (14 floating point programs)
  - compiler flags can be set differently for different programs
- Third Round: SPEC CPU95
  - new set of programs: SPEC CINT95 (8 integer programs) and SPEC CFP95 (10 floating point)
  - single flag setting for all programs
- Fourth Round: SPEC CPU2000
  - new set of programs: SPEC CINT2000 (12 integer programs) and SPEC CFP2000 (14 floating point)
  - single flag setting for all programs
  - programs in C, C++, Fortran 77, and Fortran 90

الممسوحة ضوئيا بـ CamScanner

CINT2000 (Integer component of SPEC CPU2000)

Program	Language	What It Is
164.gzip	C	Compression
175.vpr	C	FPGA Circuit Placement and Routing
176.gcc	C	C Programming Language Compiler
181.mcf	C	Combinatorial Optimization
186.crafty	C	Game Playing: Chess
197.parser	C	Word Processing
252.eon	C++	Computer Visualization
253.perlbmk	C	PERL Programming Language
254.gap	C	Group Theory, Interpreter
255.vortex	C	Object-oriented Database
256.bzip2	C	Compression
300.twolf	C	Place and Route Simulator

CFP2000 (Floating point component of SPEC CPU2000)

Program	Language	What It Is
168.wupwise	Fortran 77	Physics / Quantum Chromodynamics
171.swim	Fortran 77	Shallow Water Modeling
172.mgrid	Fortran 77	Multi-grid Solver: 3D Potential Field
173.applu	Fortran 77	Parabolic / Elliptic Differential Equations
177.mesa	C	3-D Graphics Library
178.galgel	Fortran 90	Computational Fluid Dynamics
179.art	C	Image Recognition / Neural Networks
183.equake	C	Seismic Wave Propagation Simulation
187.facerec	Fortran 90	Image Processing: Face Recognition
188.ammp	C	Computational Chemistry
189.lucas	Fortran 90	Number Theory / Primality Testing
191.fma3d	Fortran 90	Finite-element Crash Simulation
200.sixtrack	Fortran 77	High Energy Physics Accelerator Design
301.apsi	Fortran 77	Meteorology: Pollutant Distribution

CamScanner المسوحة ضوئيا بـ

SPEC CPU2000 reporting

- Refer SPEC website www.spec.org for documentation
- Single number result – geometric mean of normalized ratios for each code in the suite
- Report precise description of machine
- Report compiler flag setting

CFP2000 Result

Copyright ©1999-2001, Standard Performance Evaluation Corporation

Advanced Micro Devices

Gigabyte GA-7DX Motherboard, 1.4GHz Athlon Processor

SPECfp2000 = 458

SPECfp_base2000 = 426

SPEC license # 49 | Tested by: AMD Austin TX | Test date: May-2001 | Hardware Avail: Jun-2001 | Software Avail: Jan-2001

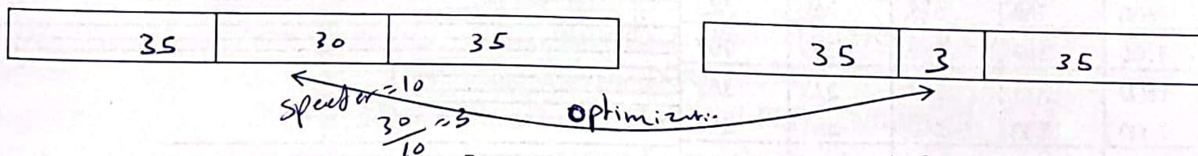
Benchmark	Reference Time	Base Runtime	Base Ratio	Runtime	Ratio
168.wupwise	1600	366	438	249	644
171.swim	3100	389	797	389	797
172.mgrid	1800	525	343	525	343
173.applu	2100	500	420	463	453
177.mesa	1400	252	556	225	621
178.galgel	2900	461	629	461	630
179.art	2600	725	359	701	371
183.equake	1300	403	323	337	386
187.facerec	1900	358	531	357	531
188.ammp	2200	591	372	585	376
189.lucas	2000	654	306	631	317
191.fma3d	2100	434	483	434	483
200.sixtrack	1100	366	301	311	354
301.apsi	2600	714	364	714	364

Specialized SPEC Benchmarks

- I/O
- Network
- Graphics
- Java
- Web server
- Transaction processing (databases)

Amdahl's Law

- Execution Time After Improvement = Execution Time Unaffected + (Execution Time Affected / Rate of Improvement)



Improved part of code

- *Example*:
  - Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time.
  - *How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?*
  - *How about making it 5 times faster?*
- Design Principle: *Make the common case fast*

المسوحة ضوئيا بـ CamScanner

Pitfall: Amdahl's Law

- The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
- Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{improved} = \frac{T_{affected}}{\text{improvement factor}} + T_{unaffected}$$

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

- Can't be done!

- Corollary: make the common case fast

Examples

- Suppose we enhance a machine making all floating-point instructions run five times faster. The execution time of some benchmark before the floating-point enhancement is 10 seconds.
  - *What will the speedup be if half of the 10 seconds is spent executing floating-point instructions?*
- We are looking for a benchmark to show off the new floating-point unit described above, and want the overall benchmark to show a speedup of 3. One benchmark we are considering runs for 100 seconds with the old floating-point hardware.
  - *How much of the execution time would floating-point instructions have to account for in this program in order to yield our desired speedup on this benchmark?*

CamScanner المسوحة ضوئيا بـ

Summary

- Performance is specific to a particular program
  - total execution time is a consistent summary of performance
- For a given architecture performance increases come from:
  - increases in clock rate (without adverse CPI affects)
  - improvements in processor organization that lower CPI
  - compiler enhancements that lower CPI and/or instruction count
- *Pitfall*: expecting improvement in one aspect of a machine's performance to affect the total performance
- You should not always believe everything you read! Read carefully! See newspaper articles, e.g., Exercise 2.37!!

Computer Organization

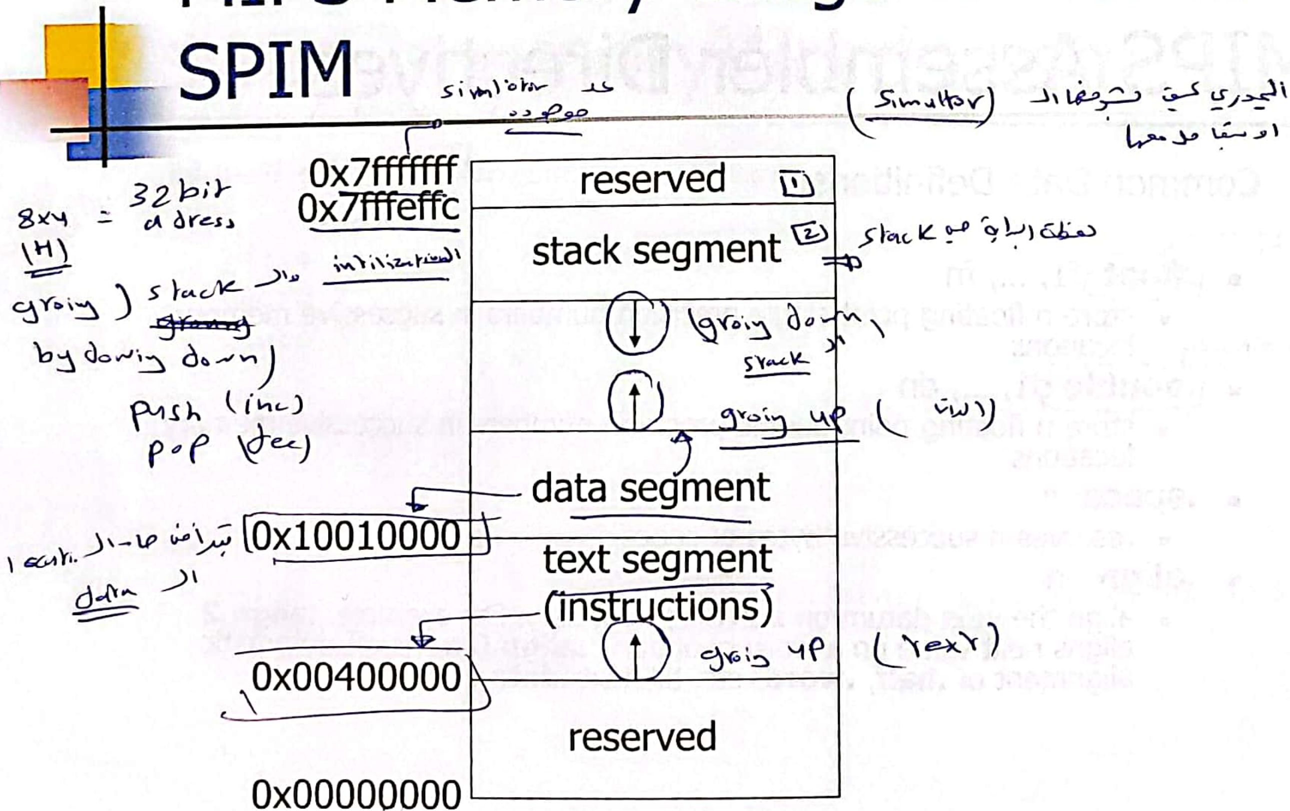
CamScanner المسوحة ضوئيا بـ

SPIM Example Program: add2numbersProg2.asm

```
# Program adds 10 and 20

        .text                       # text section
        .globl main                 # call main by SPIM

main:
        la $t0, value               # load address 'value' into $t0
        lw $t1, 0($t0)              # load word 0(value) into $t1
        lw $t2, 4($t0)              # load word 4(value) into $t2
        add $t3, $t1, $t2           # add two numbers into $t3
        sw $t3, 8($t0)              # store word $t3 into 8($t0)

        .data                       # data section
value:  .word 10, 20, 0             # load data integers. Default data
                                    # start address 0x10010000(= value)
```

Parse the machine code for these two instructions!

MIPS Memory Usage as viewed in SPIM



MIPS Assembler Directives

- Common Data Definitions:
  - **.word** w1, ..., wn
    - store n 32-bit quantities in successive memory words
  - **.half** h1, ..., hn
    - store n 16-bit quantities in successive memory halfwords
  - **.byte** b1, ..., bn
    - store n 8-bit quantities in successive memory bytes
  - **.ascii** str
    - store the string in memory but do not null-terminate it
      - strings are represented in double-quotes “str”
      - special characters, eg. \n, \t, follow C convention
  - **.asciiz** str
    - store the string in memory and null-terminate it

MIPS Assembler Directives

- Common Data Definitions:
  - **.float** f1, ..., fn
    - store n floating point single precision numbers in successive memory locations
  - **.double** d1, ..., dn
    - store n floating point double precision numbers in successive memory locations
  - **.space** n
    - reserves n successive bytes of space
  - **.align** n
    - align the next datum on a 2^n byte boundary. For example, **.align 2** aligns next value on a word boundary. **.align 0** turns off automatic alignment of **.half**, **.word**, etc. till next **.data** directive

CamScanner الممسوحة ضوئيا بـ

SPIM Example Program: storeWords.asm

```
## Program shows memory storage and access (big vs. little endian)
        .data
here:   .word 0xabc89725, 100
        .byte 0, 1, 2, 3
        .asciiz "Sample text"
there:  .space 6
        .byte 85
        .align 2
        .byte 32

        .text
        .globl main
main:
        la $t0, here
        lbu $t1, 0($t0)
        lbu $t2, 1($t0)
        lw  $t3, 0($t0)
        sw  $t3, 36($t0)
        sb  $t3, 41($t0)
```

SPIM's memory storage depends on the underlying machine: Intel 80x86 processors are **little-endian**!

Word placement in memory is exactly same in big or little endian – a copy is placed.

Byte placement in memory depends on if it is big or little endian. In big-endian bytes in a Word are counted from the byte 0 at the left (most significant) to byte 3 at the right (least significant); in little-endian it is the other way around.

Word access (lw, sw) is exactly same in big or little endian – it is a copy from register to a memory word or vice versa.

Byte access depends on if it is big or little endian, because bytes are counted 0 to 3 from left to right in big-endian and counted 0 to 3 from right to left in little-endian.

SPIM Example Program: swap2memoryWords.asm

```
## Program to swap two memory words

  .data            # load data
  .word 7
  .word 3

  .text
  .globl main

main:
  lui $s0, 0x1001 # load data area start address 0x10010000
  lw  $s1, 0($s0)
  lw  $s2, 4($s0)
  sw  $s2, 0($s0)
  sw  $s1, 4($s0)
```

CamScanner الممسوحة ضوئيا بـ

SPIM Example Program: branchJump.asm

```
## Nonsense program to show address calculations for
## branch and jump instructions

    .text                    # text section
    .globl main              # call main by SPIM

# Nonsense code
# Load in SPIM to see the address calculations
main:
     j label
     add $0, $0, $0
     beq $8, $9, label
     add $0, $0, $0
     add $0, $0, $0
     add $0, $0, $0
     add $0, $0, $0
label:
     add $0, $0, $0
```

SPIM Example Program: procCallsProg2.asm

```
## Procedure call to swap two array words

        .text
        .globl  main
main:

load para-   { la      $a0, array
meters for   { addi    $a1, $0, 0
swap
save return  { addi    $sp, $sp, -4
address $ra  { sw      $ra, 0($sp)
in stack
jump and     { jal     swap
link to swap
restore      { lw      $ra, 0($sp)
return       { addi    $sp, $sp, 4
address
jump to $ra  { jr      $ra

#       equivalent C code:
#       swap(int v[], int k)
#       {
#               int temp;
#               temp = v[k];
#               v[k] = v[k+1];
#               v[k+1] = temp;
#       }
# swap contents of elements $a1
# and $a1 + 1 of the array that
# starts at $a0
swap:   add     $t1, $a1, $a1
        add     $t1, $t1, $t1
        add     $t1, $a0, $t1
        lw      $t0, 0($t1)
        lw      $t2, 4($t1)
        sw      $t2, 0($t1)
        sw      $t0, 4($t1)
        jr      $ra

.data
array:  .word 5, 4, 3, 2, 1
```

المسوحة ضوئيا بـ CamScanner

MIPS: Software Conventions for Registers

No.	Name	Use
0	zero	constant 0
1	at	reserved for assembler
2	v0	results from callee
3	v1	returned to caller
4	a0	arguments to callee
5	a1	from caller: caller saves
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return Address (HW): caller saves

SPIM System Calls

- System Calls (syscall)
  - OS-like services
- Method
  - load system call code into register $v0 (see following table for codes)
  - load arguments into registers $a0, …, $a3
  - call system with SPIM instruction `syscall`
  - after call return value is in register $v0, or $f0 for floating point results

CamScanner الممسوحة ضوئيا بـ

SPIM System Call Codes

Service	Code (put in $v0)	Arguments	Result
print_int	1	$a0=integer	
print_float	2	$f12=float	
print_double	3	$f12=double	
print_string	4	$a0=addr. of string	
read_int	5		int in $v0
read_float	6		float in $f0
read_double	7		double in $f0
read_string	8	$a0=buffer, $a1=length	
sbrk	9	$a0=amount	addr in $v0
exit	10		

SPIM Example Program: systemCalls.asm

```
## Enter two integers in
## console window
## Sum is displayed
.text
.globl main

main:
   la $t0, value

   li $v0, 5
   syscall
   sw $v0, 0($t0)

   li $v0, 5
   syscall
   sw $v0, 4($t0)

   lw $t1, 0($t0)
   lw $t2, 4($t0)
   add $t3, $t1, $t2
   sw $t3, 8($t0)

   li $v0, 4
   la $a0, msg1
   syscall

   li $v0, 1
   move $a0, $t3
   syscall

   li  $v0, 10
   syscall

.data
value:  .word 0, 0, 0
msg1:   .asciiz "Sum = "
```

- `li $v0, 5` — system call code for read_int
- `sw $v0, 0($t0)` — result returned by call
- `li $v0, 4` — system call code for print_string
- `la $a0, msg1` — argument to print_string call
- `li $v0, 1` — system call code for print_int
- `move $a0, $t3` — argument to print_int call
- `li $v0, 10` — system call code for exit

CamScanner الممسوحة ضوئيا بـ

COD Ch. 3 Instructions: Language of the Machine

Instructions: Overview

- Language of the machine
- More primitive than higher level languages, e.g., no sophisticated control flow such as *while* or *for* loops
- Very restrictive
  - e.g., MIPS arithmetic instructions
- We'll be working with the MIPS instruction set architecture
  - inspired most architectures developed since the 80's
  - used by NEC, Nintendo, Silicon Graphics, Sony
  - the name is not related to *millions of instructions per second* !
  - it stands for **m***icrocomputer without* **i***nterlocked* **p***ipeline* **s***tages* !
- Design goals: *maximize performance* and *minimize cost* and *reduce design time*

الممسوحة ضوئيا بـ CamScanner

MIPS Arithmetic

- All MIPS arithmetic instructions have 3 operands
- Operand order is fixed (e.g., destination first)
- *Example*:

C code: A = B + C

MIPS code: `add $s0, $s1, $s2`

compiler's job to associate variables with registers

MIPS Arithmetic

- Design Principle 1: *simplicity favors regularity.*
  Translation: Regular instructions make for simple hardware!
- *Simpler hardware reduces design time and manufacturing cost.*
- Of course this complicates some things...

C code:

```
A = B + C + D;
E = F - A;
```

MIPS code (arithmetic):

```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

Allowing variable number of operands would simplify the assembly code but complicate the hardware.

- Performance penalty: high-level code translates to denser machine code.

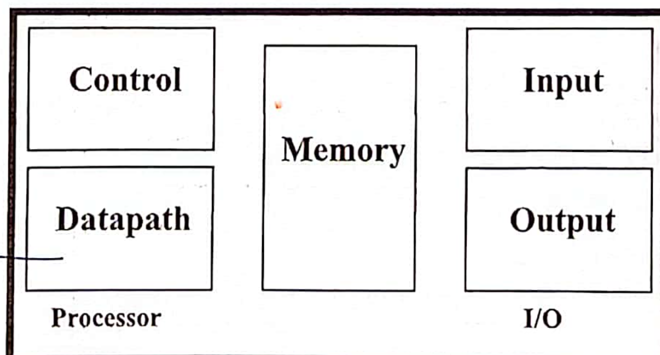
المسموحة ضوئيا بـ CamScanner

MIPS Arithmetic

- *Operands must be in registers* – only 32 registers provided (which require 5 bits to select one register). Reason for small number of registers:
- Design Principle 2: *smaller is faster. Why?*
  - *Electronic signals have to travel further on a physically larger chip increasing clock cycle time.*
  - *Smaller is also cheaper!*

Registers vs. Memory

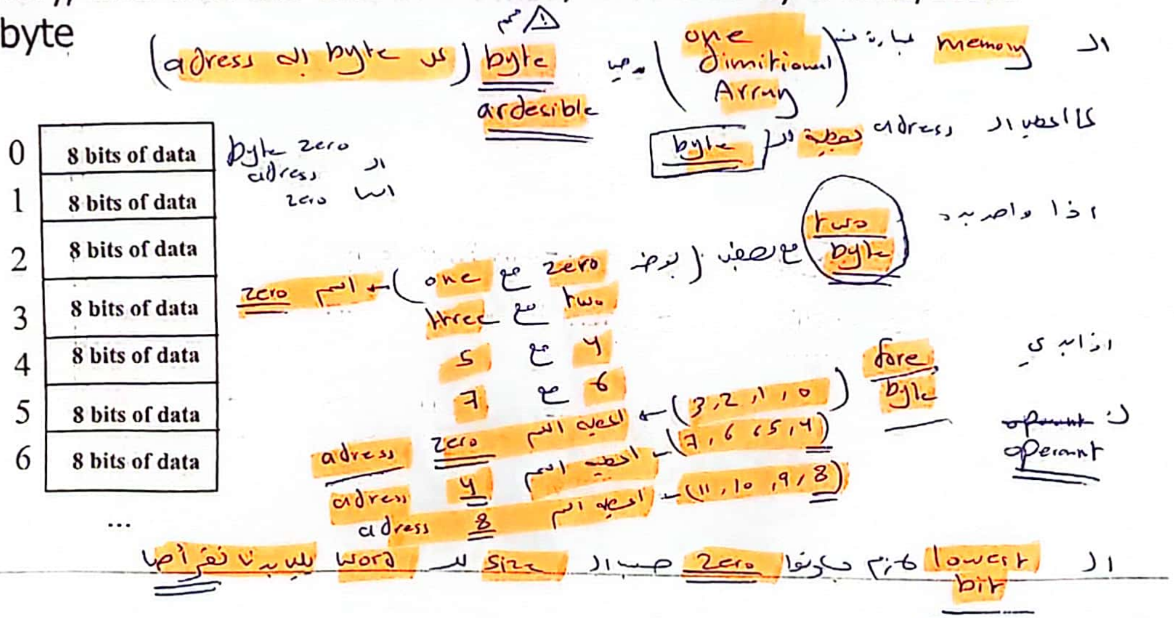
- Arithmetic instructions operands must be in registers
  - MIPS has 32 registers
- Compiler associates variables with registers
- What about programs with lots of variables (arrays, etc.)? Use *memory*, *load/store* operations to transfer data from memory to register – if not enough registers *spill registers* to memory
- *MIPS is a load/store architecture*



الممسوحة ضوئيا بـ CamScanner

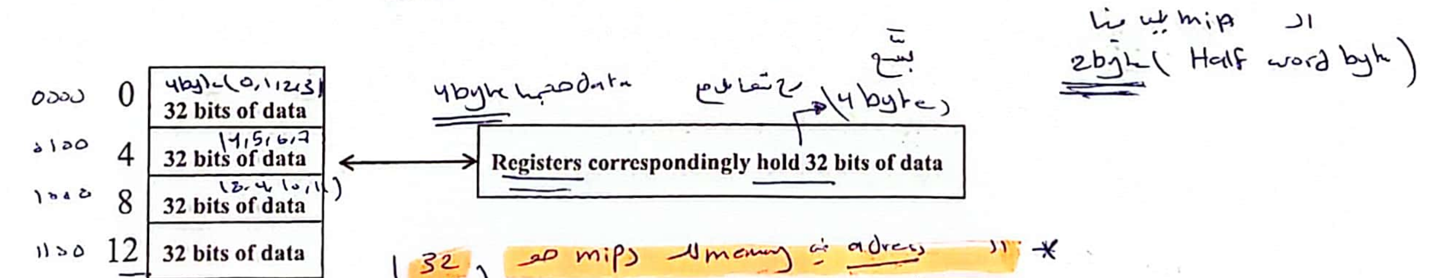
Memory Organization

- Viewed as a large single-dimension array with access by *address*
- A memory address is an *index* into the memory array
- *Byte addressing* means that the index points to a byte of memory, and that the unit of memory accessed by a load/store is a byte

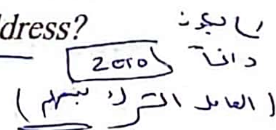


Memory Organization

- Bytes are load/store units, but most data items use larger *words*
- For MIPS, a word is 32 bits or 4 bytes.



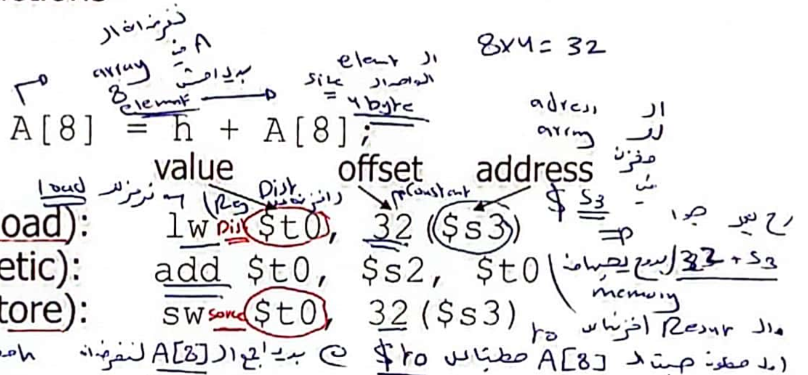
- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
  - i.e., words are *aligned*
  - *what are the least 2 significant bits of a word address?*



الممسوحة ضوئيا بـ CamScanner

Load/Store Instructions

- *Load* and *store* instructions
- *Example*:



C code: `A[8] = h + A[8];`

value offset address

MIPS code (load): `lw $t0, 32($s3)`
(arithmetic): `add $t0, $s2, $t0`
(store): `sw $t0, 32($s3)`

- Load word has destination first, store has destination last
- Remember MIPS arithmetic operands are registers, not memory locations
  - therefore, words must first be moved from memory to registers using loads before they can be operated on; then result can be stored back to memory

Memory Operand Example 2

- C code:

  `A[12] = h + A[8];`

  - h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32

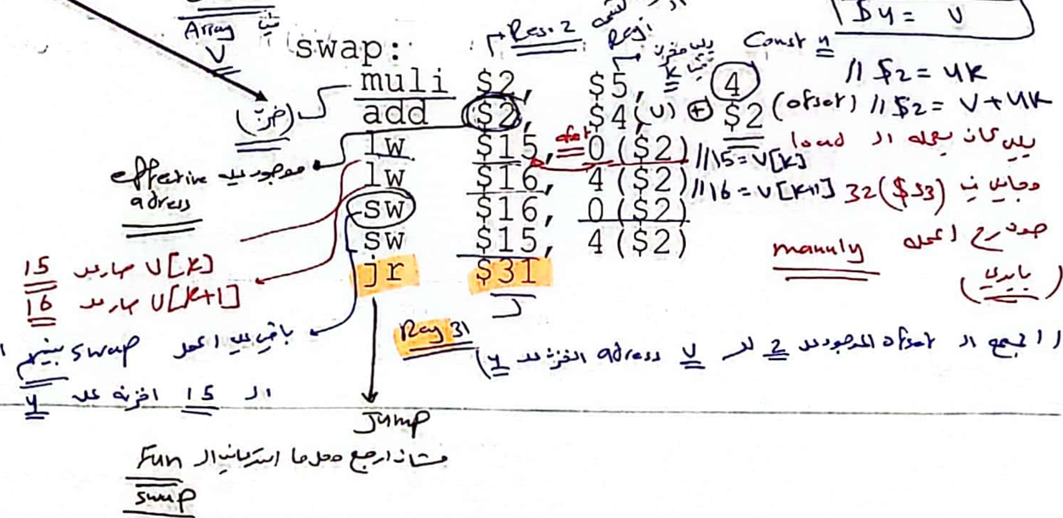
```
lw   $t0, 32($s3)     # load word
add  $t0, $s2, $t0
sw   $t0, 48($s3)     # store word
```

الممسوحة ضوئيا بـ CamScanner

A MIPS Example

- *Can we figure out the assembly code?*

```
swap(int v[], int k);
{ int temp;
  temp   =  v[k];
  v[k]   =  v[k+1];
  v[k+1] =  temp;
}
```



So far we've learned:

- MIPS
  - loading words but addressing bytes
  - arithmetic on registers only
- Instruction / Meaning

Instruction	Meaning
add $s1, $s2, $s3	$s1 = $s2 + $s3
sub $s1, $s2, $s3	$s1 = $s2 – $s3
lw $s1, 100($s2)	$s1 = Memory[$s2+100]
sw $s1, 100($s2)	Memory[$s2+100]= $s1

المسوحة ضوئيا بـ CamScanner

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - *Example*: `add $t0, $s1, $s2`
  - registers are numbered, e.g., `$t0` is `8`, `$s1` is `17`, `$s2` is `18`
- Instruction Format **R-type** ("R" for aRithmetic):

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct
opcode – operation	first register source operand	second register source operand	register destin- ation operand	shift amount	function field – selects variant of operation
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Language

- Consider the load-word and store-word instructions,
  - what would the regularity principle have us do?
    - we would have only 5 or 6 bits to determine the offset from a base register - too little…
- Design Principle 3: *Good design demands a compromise*
- Introduce a new type of instruction format
  - **I-type** ("I" for Immediate) for data transfer instructions
  - *Example*: `lw $t0, 1002($s2)`

100011	10010	01000	0000001111101010
6 bits	5 bits	5 bits	16 bits
op	rs	rt	16 bit offset

CamScanner المسوحة ضوئيا بـ

Stored Program Concept

- *Instructions are bit sequences*, just like data
- Programs are stored in memory
  - to be read or written just like data

Processor	Memory

memory for data, programs, compilers, editors, etc.

- Fetch & Execute Cycle
  - instructions are *fetched* and put into a special register
  - bits in the register *control* the *subsequent actions* (= *execution*)
  - fetch the next instruction and *repeat*

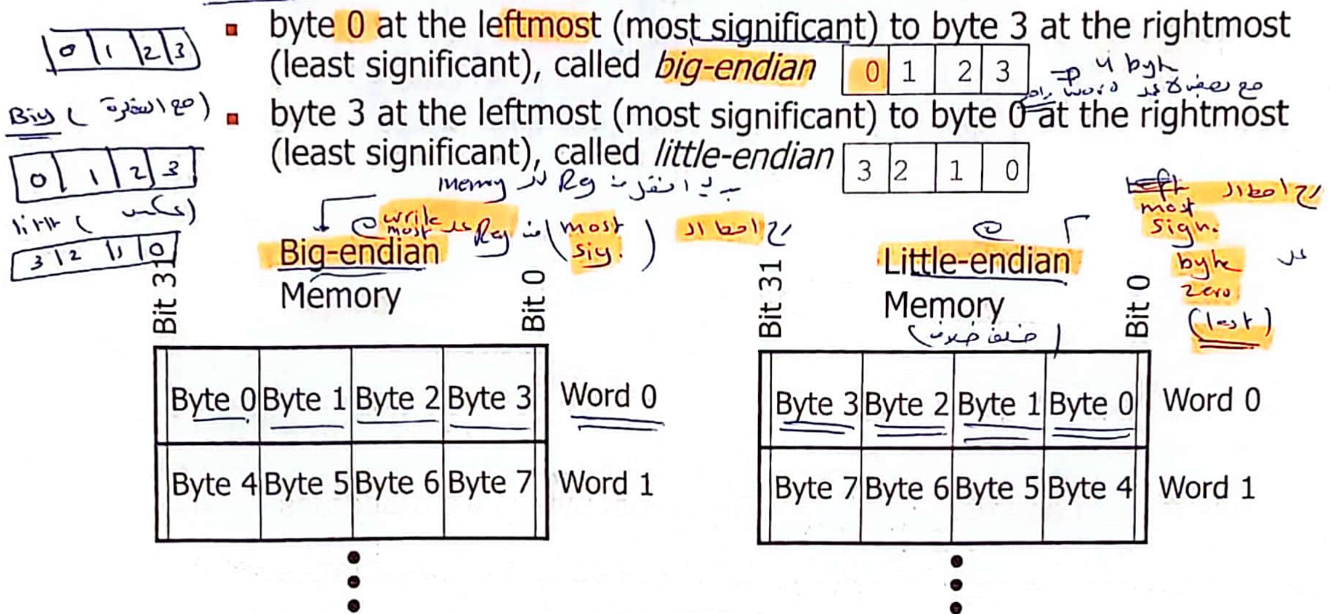
SPIM – the MIPS simulator

- SPIM (MIPS spelt backwards!) is a MIPS simulator that
  - *reads* MIPS assembly language files and *translates* to machine language
  - *executes* the machine language instructions
  - shows contents of *registers* and *memory*
  - works as a *debugger* (supports *break-points* and *single-stepping*)
  - provides basic *OS-like services*, like simple I/O
- SPIM is freely available on-line
- *An important part of our course is to actually write MIPS assembly code and run using SPIM – the only way to learn assembly (or any programming language) is to write lots and lots of code!!!*
- Refer to our material, including slides, on SPIM

CamScanner الممسوحة ضوئيا بـ

Memory Organization: Big/Little Endian Byte Order

- Bytes in a word can be numbered in two ways:
  - byte 0 at the leftmost (most significant) to byte 3 at the rightmost (least significant), called *big-endian* 0 1 2 3
  - byte 3 at the leftmost (most significant) to byte 0 at the rightmost (least significant), called *little-endian* 3 2 1 0

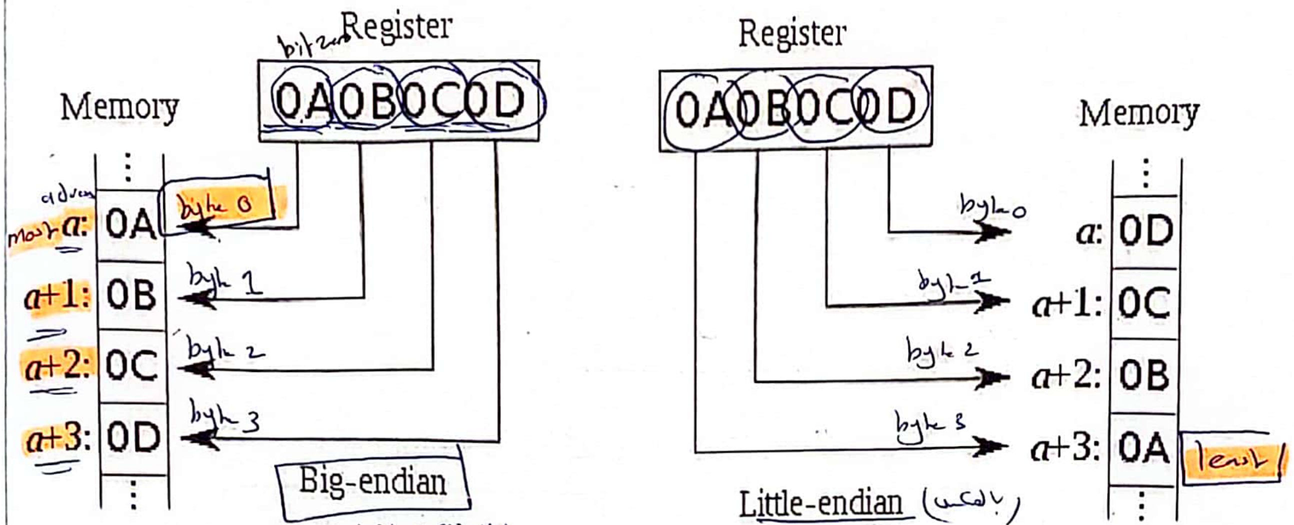


Memory Organization: Big/Little Endian Byte Order

- SPIM's memory storage depends on that of the underlying machine
  - *Intel 80x86 processors are **little-endian***
  - because SPIM *always shows* words from left to right a "mental adjustment" has to be made for little-endian memory as in Intel PCs in our labs: start at right of first word go left, start at right of next word go left, …!
- *Word placement* in memory (from `.data` area of code) or *word access* (lw, sw) is the same in big or little endian
- *Byte placement* and *byte access* (lb, lbu, sb) depend on big or little endian because of the different numbering of bytes within a word
- *Character placement* in memory (from `.data` area of code) depend on big or little endian because it is equivalent to byte placement after ASCII encoding
- Run `storeWords.asm` from SPIM examples!!

CamScanner الممسوحة ضوئيا بـ

Byte Addresses



Big Endian: Leftmost byte is word address

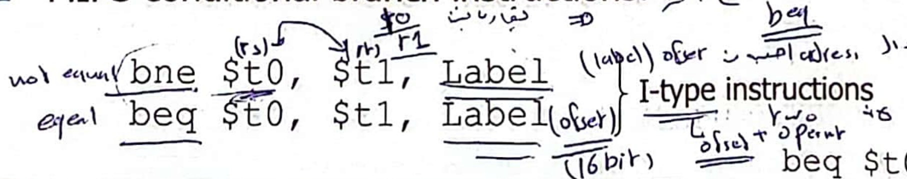
LS Byte has biggest address in the word

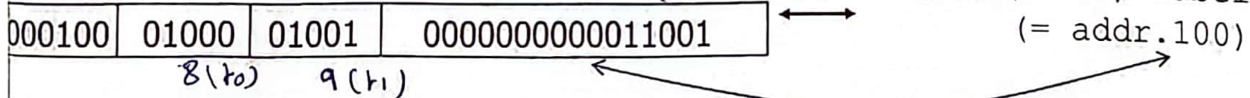
Little Endian: Rightmost byte is word address

LS Byte has little address in the word.

Control: Conditional Branch

- Decision making instructions
  - alter the control flow,
    - i.e., change the next instruction to be executed
- MIPS conditional branch instructions:





beq $t0, $t1, Label (= addr.100)

- *Example*: if (i==j) h = i + j;

```
    bne $s0, $s1, Label
    add $s3, $s0, $s1
Label:
```

word-relative addressing: 25 words = 100 bytes; also *PC-relative* (more...)

المسوحة ضوئيا بـ CamScanner

Addresses in Branch

- Instructions:

```
bne $t4,$t5,Label    Next instruction is at Label if $t4 != $t5
beq $t4,$t5,Label    Next instruction is at Label if $t4 = $t5
```

- Format:

I	op	rs	rt	16 bit offset

- 16 bits is too small a reach in a 2^{32} address space
- Solution: specify a register (as for `lw` and `sw`) and add it to offset
  - use PC (= program counter), called *PC-relative* addressing, based on
  - *principle of locality*: most branches are to instructions near current instruction (e.g., loops and *if* statements)

Addresses in Branch

- Further extend reach of branch by observing all MIPS instructions are a word (= 4 bytes), therefore *word-relative* addressing:
- MIPS branch destination address = (PC + 4) + (4 * offset)

  Because hardware typically increments PC early in execute cycle to point to next instruction

- so offset = (branch destination address – PC – 4)/4
- *but SPIM does* offset = (branch destination address – PC)/4

CamScanner الممسوحة ضوئيا بـ

Control: Unconditional Branch (Jump)

- MIPS unconditional branch instructions:

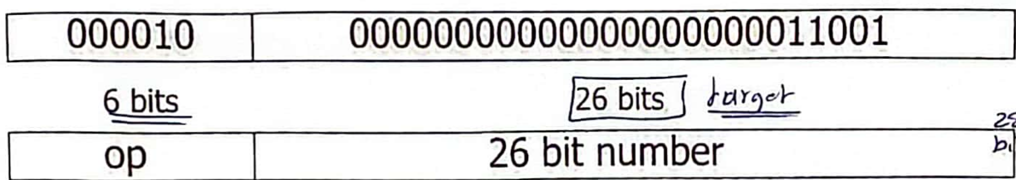
```
j Label
```

- *Example*:

```
if (i!=j)          beq $s4, $s5, Lab1
    h=i+j;         add $s3, $s4, $s5
else               j Lab2
    h=i-j;         Lab1: sub $s3, $s4, $s5
                   Lab2: ...
```

- **J-type** ("J" for Jump) instruction format
  - *Example*: `j Label # addr. Label = 100`

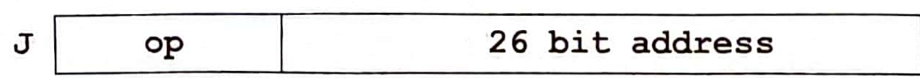
word-relative addressing: 25 words = 100 bytes



000010	00000000000000000000011001
6 bits	26 bits
op	26 bit number

Addresses in Jump

- Word-relative addressing also for jump instructions



J	op	26 bit address

- MIPS jump `j` instruction replaces *lower* 28 bits of the PC with `A00` where `A` is the 26 bit address; it *never changes* upper 4 bits
  - *Example*: if `PC = 1011X` (where X = 28 bits), it is replaced with `1011A00`
  - there are 16(=2^4) partitions of the 2^{32} size address space, each partition of size 256 MB (=2^{28}), *such that*, in each partition the upper 4 bits of the address is same.
  - if a program crosses an address partition, then a `j` that reaches a different partition has to be replaced by `jr` with a full 32-bit address first loaded into the jump register
  - therefore, OS should always try to load a program inside a single partition

CamScanner الممسوحة ضوئيا بـ

Constants

- Small constants are used quite frequently (50% of operands)

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

- Solutions? Will these work?
  - create hard-wired registers (like $zero) for constants like 1
  - put program constants in memory and load them as required

- MIPS Instructions:

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

- *How to make this work?*

The Constant Zero

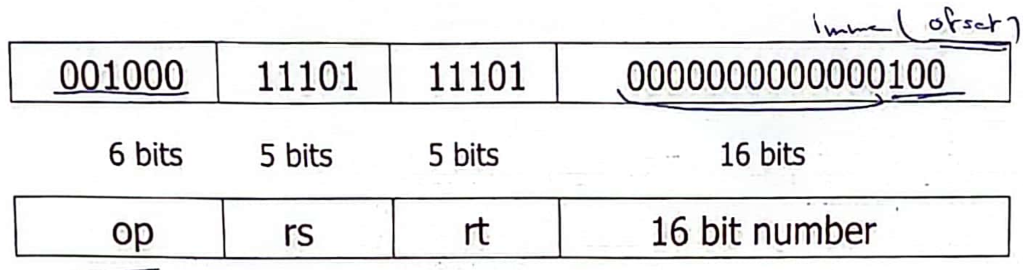
- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```


CamScanner الممسوحة ضوئيا بـ

Immediate Operands

- Make operand part of instruction itself!
- Design Principle 4: *Make the common case fast*
- *Example*: `addi $sp, $sp, 4 # $sp = $sp + 4`



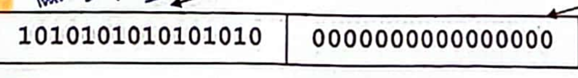
001000	11101	11101	0000000000000100
6 bits	5 bits	5 bits	16 bits
op	rs	rt	16 bit number

How about larger constants?

- First we need to load a 32 bit constant into a register
- Must use two instructions for this: first new *load upper immediate* instruction for upper 16 bits

```
lui $t0, 1010101010101010
```

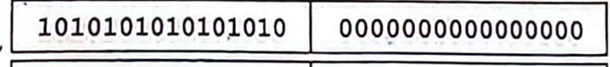
filled with zeros



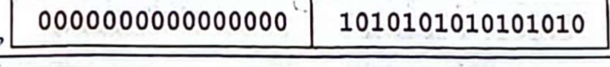
1010101010101010	0000000000000000

- Then get lower 16 bits in place:

```
ori $t0, $t0, 1010101010101010
```

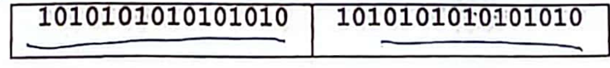


1010101010101010	0000000000000000



0000000000000000	1010101010101010

ori



1010101010101010	1010101010101010

- Now the constant is in place, use register-register arithmetic

CamScanner المسوحة ضوئيا بـ

So far

- Instruction / Format / Meaning

Instruction	Format	Meaning
`add $s1,$s2,$s3`	R	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	R	`$s1 = $s2 - $s3`
`lw $s1,100($s2)`	I	`$s1 = Memory[$s2+100]`
`sw $s1,100($s2)`	I	`Memory[$s2+100] = $s1`
`bne $s4,$s5,Lab1`	I	`Next instr. is at Lab1 if $s4 != $s5`
`beq $s4,$s5,Lab2`	I	`Next instr. is at Lab2 if $s4 = $s5`
`j Lab3`	J	`Next instr. is at Lab3`

- Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

Useful for extracting and inserting groups of bits in a word

CamScanner المسوحة ضوئيا بـ

Control Flow

- We have: beq, bne. What about *branch-if-less-than*?
- New instruction:

```
slt $t0, $s1, $s2      <-->      if   $s1 < $s2 then
                                      $t0 = 1
                                 else
                                      $t0 = 0
```

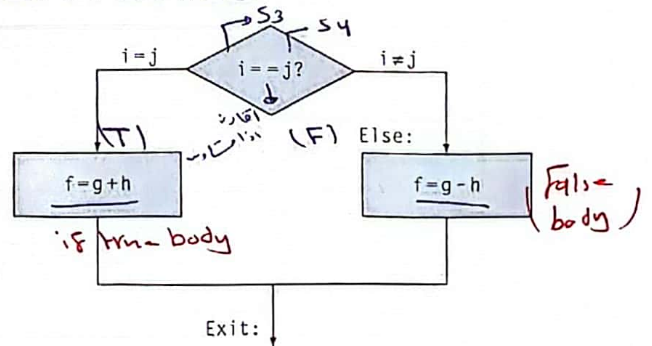
- Can use this instruction to build `blt $s1, $s2, Label`
  - *how?* We generate more than one instruction – *pseudo-instruction*
  - can now build general control structures
- The assembler needs a register to manufacture instructions from pseudo-instructions
- There is a *convention* (not mandatory) for use of registers

Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, ... in $s0, $s1, ...



- Compiled MIPS code:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: ...
```

Assembler calculates addresses

المسوحة ضوئيا بـ CamScanner

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - i in $s3, k in $s5, address of save in $s6
- Compiled MIPS code:

```
Loop: sll   $t1, $s3, 2      ;; t1= s3*4
      add   $t1, $t1, $s6    ;; addr of save[i]
      lw    $t0, 0($t1)      ;; t0 = save[i]
      bne   $t0, $s5, Exit   ;; Comp
      addi  $s3, $s3, 1      ;;; i+1
      j     Loop
Exit: ...
```


Policy-of-Use Convention for Registers

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

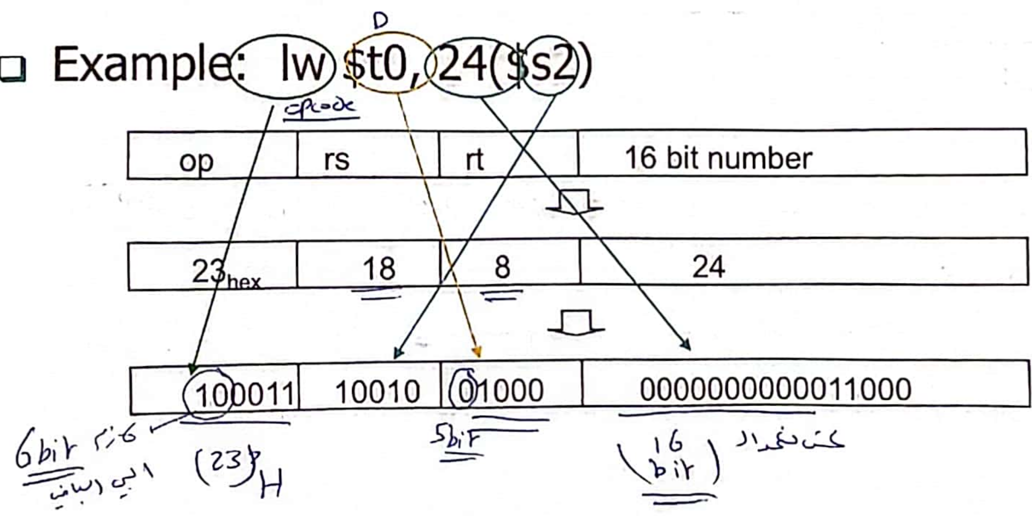
Register 1, called $at, is reserved for the assembler; registers 26-27, called $k0 and $k1 are reserved for the operating system.

Assembly Language vs. Machine Language

- Assembly provides convenient *symbolic representation*
  - much easier than writing down numbers
  - regular rules: e.g., destination first
- Machine language is the *underlying reality*
  - e.g., destination is no longer first
- Assembly can provide *pseudo-instructions*
  - e.g., `move $t0, $t1` exists only in assembly
  - would be implemented using `add $t0, $t1, $zero`
- When considering performance you should count actual number of machine instructions that will execute

Machine Language - Load Instruction

- Consider the load-word and store-word instr's
  - What would the regularity principle have us do?
    - But . . . Good design demands compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions (previous format was R-type for register)
- Example: lw $t0, 24($s2)

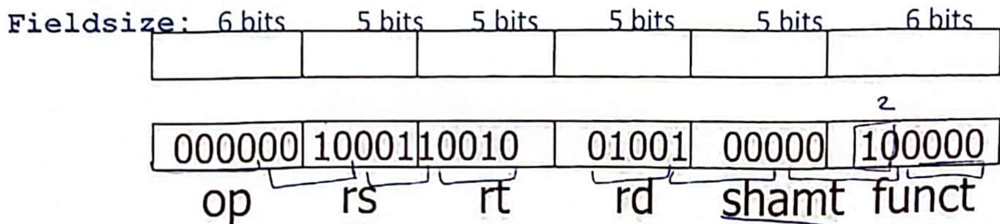


CamScanner الممسوحة ضوئيا بـ

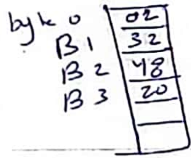
Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: add $t1, $s1, $s2
  - registers have numbers, $t1=9, $s1=17, $s2=18

- Instruction Format:



(023248 20)



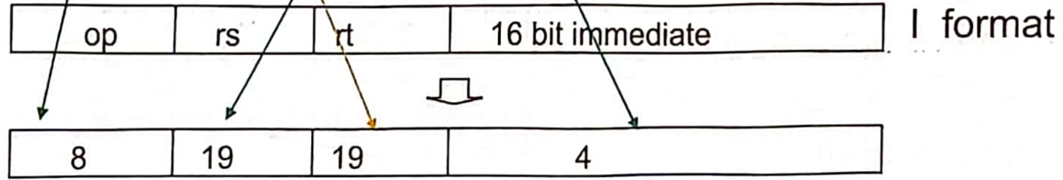
- *Can you guess what the field names stand for?*

(0x02324820)

Machine Language – Immediate Instructions

- What instruction format is used for the addi ?
  addi $s3, $s3, 4 #$s3 = $s3 + 4

- Machine format:



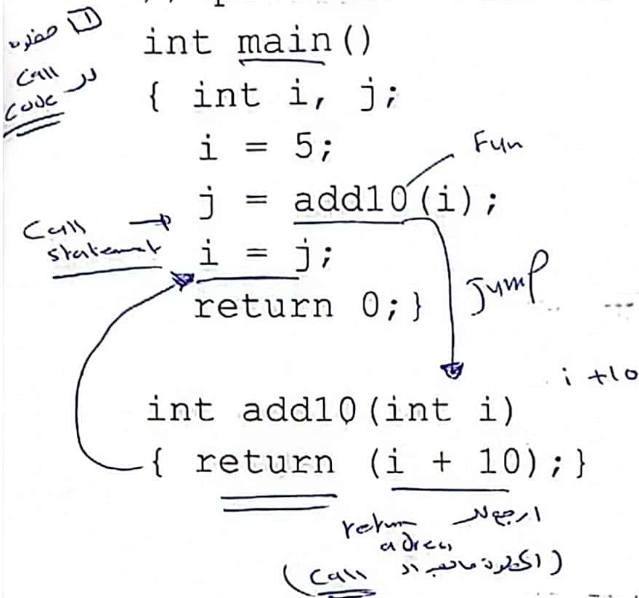
- The constant is kept inside the instruction itself!
  - So must use the I format – Immediate format
  - Limits immediate values to the range +215–1 to -215

$(2^{15}-1)$ to -2^{15}

CamScanner الممسوحة ضوئيا بـ

Procedures

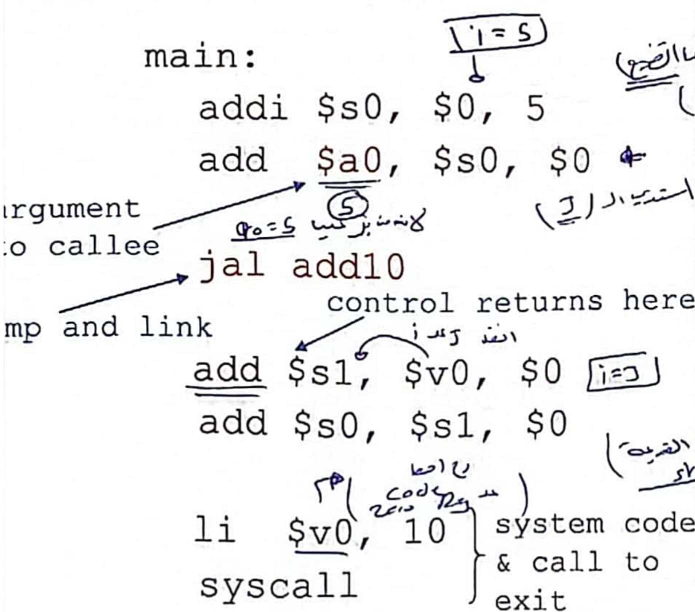
- *Example C code:*



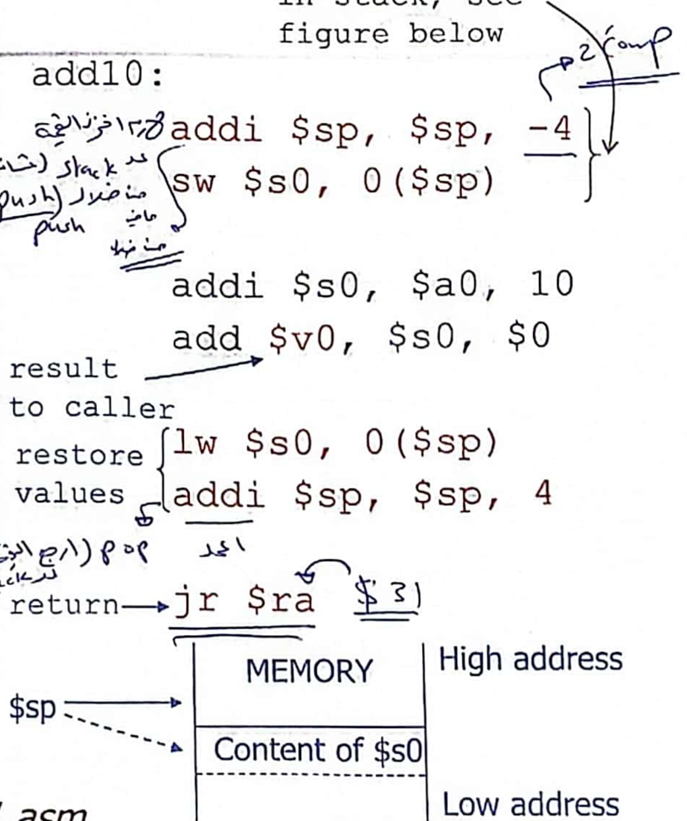
```
// procedure adds 10 to input parameter
int main()
{ int i, j;
  i = 5;
  j = add10(i);
  i = j;
  return 0;}

int add10(int i)
{ return (i + 10);}
```

Procedures

- *Translated MIPS assembly*
- *Note more efficient use of registers possible!*



```
    .text
    .globl main

main:
    addi $s0, $0, 5
    add  $a0, $s0, $0        argument to callee
    jal  add10               jump and link
                             control returns here
    add  $s1, $v0, $0
    add  $s0, $s1, $0

    li   $v0, 10             system code
    syscall                  & call to exit
```



```
                   save register
                   in stack, see
                   figure below
add10:
    addi $sp, $sp, -4
    sw   $s0, 0($sp)

    addi $s0, $a0, 10
    add  $v0, $s0, $0        result to caller

    lw   $s0, 0($sp)         restore values
    addi $sp, $sp, 4

    jr   $ra                 return
```


Run this code with PCSpim: procCallsProg1.asm

الممسوحة ضوئيا بـ CamScanner

MIPS: Software Conventions for Registers

#	Name	Usage
0	zero	constant 0
1	at	reserved for assembler
2	v0	results from callee
3	v1	returned to caller
4	a0	arguments to callee
5	a1	from caller: caller saves
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return Address (HW): caller saves

Procedures (recursive)

- *Example C code* – recursive factorial subroutine:

```
int main()
{ int i;
  i = 4;
  j = fact(i);
  return 0;}

int fact(int n)
{ if (n < 1) return (1);
  else return ( n*fact(n-1) );}
```

CamScanner الممسوحة ضوئيا بـ

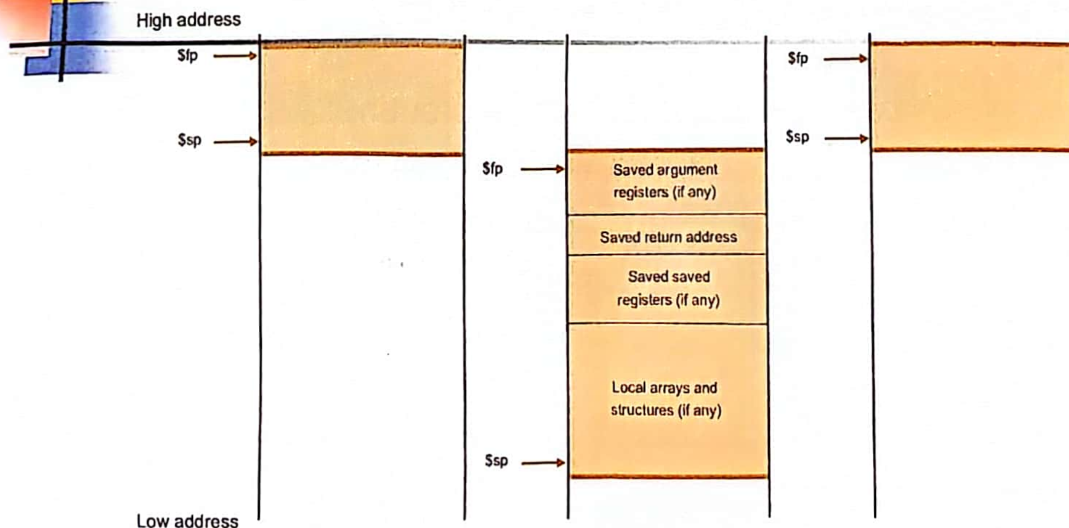
Procedures (recursive)

- *Translated MIPS assembly*:

```
        .text
        .globl    main

main:
                addi $a0, $0, 4
control         jal fact
returns       { nop
from fact

print value   { move $a0, $v0
returned by     li $v0, 1
fact            syscall

exit          { li  $v0, 10
                syscall

fact:
save return   { addi $sp, $sp, -8
address and     sw $ra, 4($sp)
argument in     sw $a0, 0($sp)
stack

branch to     { slti $t0, $a0, 1
L1 if n>=1      beq $t0, $0, L1
                nop

return 1      { addi $v0, $0, 1
if n < 1        addi $sp, $sp, 8
                jr $ra
L1:
if n>=1 call  { addi $a0, $a0, -1
fact recursively jal fact
with argument   nop
n-1

restore return { lw $a0, 0($sp)
address, argument, lw $ra, 4($sp)
and stack pointer  addi $sp, $sp, 8

return
n*fact(n-1)   { mul $v0, $a0, $v0

return control { jr $ra
```

in this code with PCSpim: factorialRecursive.asm

Using a Frame Pointer



Variables that are local to a procedure but do not fit into registers (e.g., local arrays, structures, etc.) are also stored in the stack. This area of the stack is the *frame*. The *frame pointer* $fp points to the top of the frame and the stack pointer to the bottom. The frame pointer does not change during procedure execution, unlike the stack pointer, so it is a stable base register from which to compute offsets to local variables.
Use of the frame pointer is *optional*. If there are no local variables to store in the stack it is not efficient to use a frame pointer.

الممسوحة ضوئيا بـ CamScanner

Using a Frame Pointer

- *Example*: procCallsProg1Modified.asm

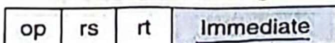
  This program shows code where it may be better to use $fp
  - Because the stack size is changing, the offset of variables stored in the stack w.r.t. the stack pointer $sp changes as well. However, the offset w.r.t. $fp would remain constant.
  - Why would this be better?

    The compiler, when generating assembly, typically maintains a table of program variables and their locations. If these locations are offsets w.r.t $sp, then every entry must be updated every time the stack size changes!
- *Exercise*:

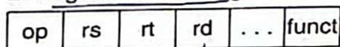
  Modify procCallsProg1Modified.asm *to use a frame pointer*
  - Observe that SPIM names register 30 as s8 rather than fp. Of course, you can use it as fp, but make sure to initialize it with the same value as sp, i.e., *7fffeffc*.

MIPS Addressing Modes

1. Immediate addressing

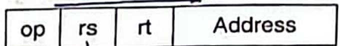


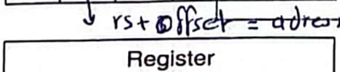
2. Register addressing



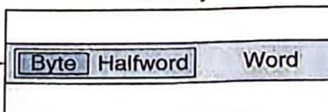
Registers

3. Base addressing

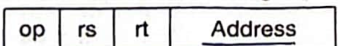


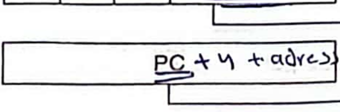


Memory

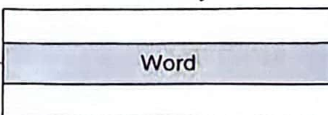


4. PC-relative addressing

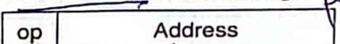


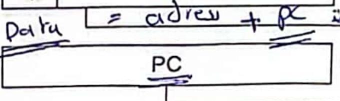


Memory

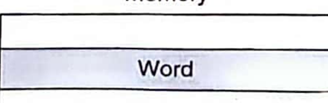


5. Pseudodirect addressing





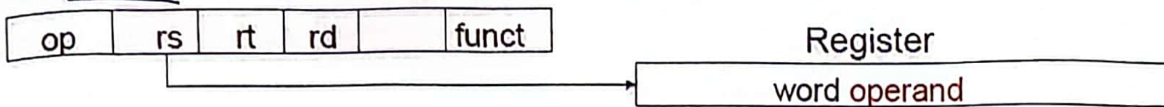
Memory



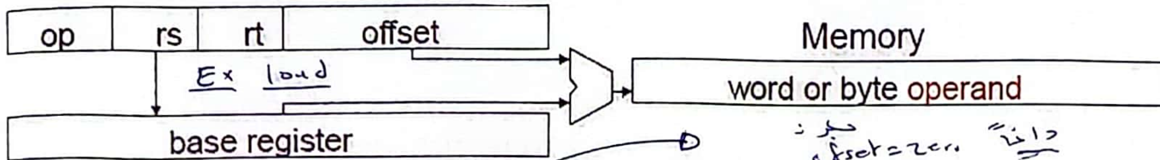
CamScanner الممسوحة ضوئيا بـ

Review of MIPS Operand Addressing Modes

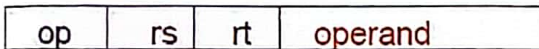
- Register addressing – operand is in a register



- Base (displacement) addressing – operand is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction



  - Register relative (indirect) with 0($a0)



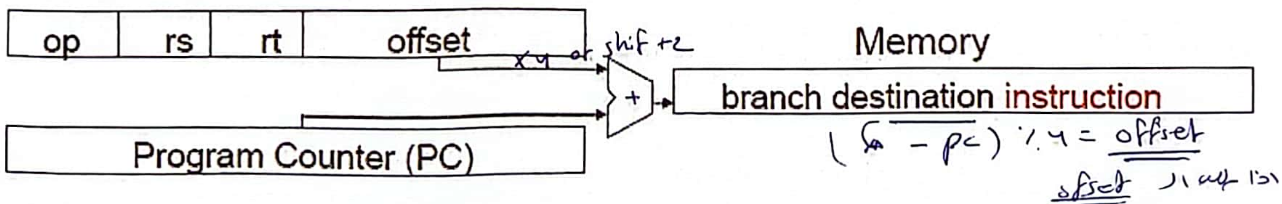
  - Pseudo-direct with addr($zero)

- Immediate addressing – operand is a 16-bit constant contained within the instruction

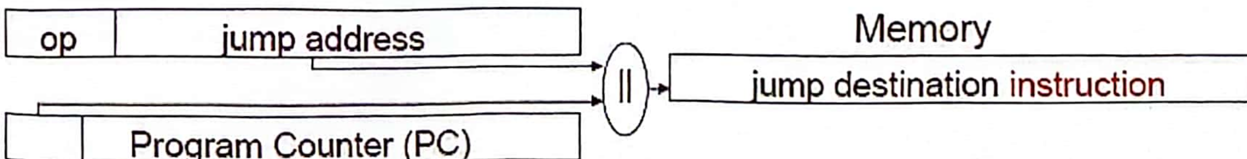
op	rs	rt	operand

Review of MIPS Instruction Addressing Modes

- PC-relative addressing –instruction address is the sum of the PC and a 16-bit constant contained within the instruction



- Pseudo-direct addressing – instruction address is the 26-bit constant contained within the instruction concatenated with the upper 4 bits of the PC



CamScanner المسوحة ضوئيا بـ

Overview of MIPS

- Simple instructions – all 32 bits wide
- Very structured – no unnecessary baggage
- Only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

- Rely on compiler to achieve performance
  - *what are the compiler's goals?*
- Help compiler where we can

Summarize MIPS:

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^16	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as *R(educed)ISC vs. C(omplex)ISC*
  - virtually all new instruction sets since 1982 have been RISC
- We'll look at PowerPC and 80x86

PowerPC Special Instructions

- Indexed addressing
  - *Example*: `lw $t1,$a0+$s3 #$t1=Memory[$a0+$s3]`
  - what do we have to do in MIPS?

```
add  $t0, $a0, $s3
lw   $t1, 0($t0)
```

- Update addressing
  - update a register as part of load (for marching through arrays)
  - *Example*: `lwu $t0,4($s3) #$t0=Memory[$s3+4];$s3=$s3+4`
  - what do we have to do in MIPS?

```
lw   $t0, 4($s3)
addi $s3, $s3, 4
```

- Others:
  - load multiple words/store multiple words
  - a special counter register to improve loop performance:

```
bc   Loop, ctrl != 0   # decrement counter, if not 0 goto loop
```

  - MIPS:

```
addi $t0, $t0, -1
bne  $t0, $zero, Loop
```

الممسوحة ضوئيا بـ CamScanner

A dominant architecture: 80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added

"this history illustrates the impact of the "golden handcuffs" of compatibility"

"adding new features as someone might add clothing to a packed bag"

A dominant architecture: 80x86

- Complexity
  - instructions from 1 to 17 bytes long
  - one operand *must* act as both a source and destination
  - one operand *may* come from memory
  - several complex addressing modes
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

"an architecture that is difficult to explain and impossible to love"

"what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective"

CamScanner الممسوحة ضوئيا بـ

Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!

Computer Organization

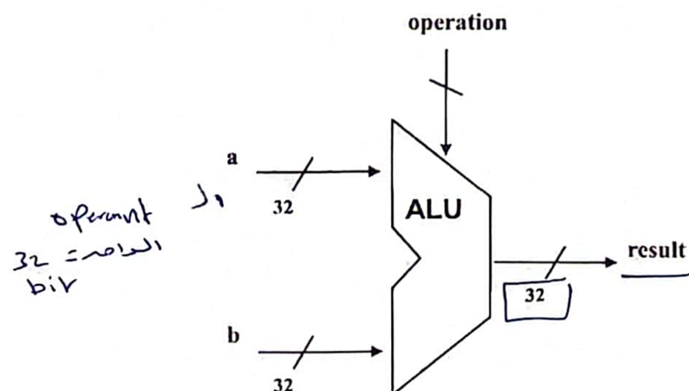
Slide Sources: Patterson & Hennessy COD book website (copyright Morgan Kaufmann) adapted and supplemented

CamScanner المسوحة ضوئيا بـ

COD Ch. 4
Arithmetic for Computers

Arithmetic

- Where we've been:
  - performance
  - abstractions
    - *instruction set architecture*
    - *assembly language* and *machine language*
- What's up ahead:
  - *implementing* the architecture



CamScanner الممسوحة ضوئيا بـ

Numbers

- Bits are just bits (no inherent meaning)
  - conventions define relationship between bits and numbers
- Binary integers (base 2)
  - 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  - decimal: 0, ..., 2^n-1 n bits
- Of course it gets more complicated:
  - bit strings are *finite*, but
    - for some *fractions* and *real* numbers, finitely many bits is not enough, so
    - *overflow* & *approximation* errors: e.g., represent 1/3 as binary!
  - *negative* integers
- How do we represent negative integers?
  - which bit patterns will represent which integers?

Possible Representations

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = 0	000 = 0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = 0	111 = -1
ambiguous zero	ambiguous zero	unequal no. of negatives and positives; unique zero

- Issues:
  - *balance* – equal number of negatives and positives
  - *ambiguous zero* – whether more than one zero representation
  - ease of arithmetic operations
- *Which representation is best? Can we get both balance and non-ambiguous zero?*

CamScanner الممسوحة ضوئيا بـ

Representation Formulae

- Two's complement:

$$x_n x_{n-1} \ldots x_0 = x_n * -2^n + x_{n-1} * 2^{n-1} + \ldots + x_0 * 2^0$$

or

$x_n X' = x_n * -2^n + X'$ (writing rightmost n bits $x_{n-1} \ldots x_0$ as X')

$$= \begin{cases} X', & \text{if } x_n = 0 \\ -2^n + X', & \text{if } x_n = 1 \end{cases}$$

- One's complement:

$$x_n X' = \begin{cases} X', & \text{if } x_n = 0 \\ -2^n + 1 + X', & \text{if } x_n = 1 \end{cases}$$

MIPS – 2's complement

- 32 bit signed numbers:

Binary		Decimal	
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$	=	0_{ten}	
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$	=	$+\ 1_{ten}$	
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$	=	$+\ 2_{ten}$	
...			
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$	=	$+\ 2,147,483,646_{ten}$	
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$	=	$+\ 2,147,483,647_{ten}$	maxint
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$	=	$-\ 2,147,483,648_{ten}$	minint
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$	=	$-\ 2,147,483,647_{ten}$	
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$	=	$-\ 2,147,483,646_{ten}$	
...			
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two}$	=	$-\ 3_{ten}$	
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$	=	$-\ 2_{ten}$	
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$	=	$-\ 1_{ten}$	

Negative integers are exactly those that have leftmost bit 1

المسوحة ضوئيا بـ CamScanner

Two's Complement Operations

- Negation Shortcut: To *negate* any two's complement integer (except for minint) *invert* all bits and *add 1*
  - note that *negate* and *invert* are different operations!
  - *why does this work? Remember we don't know how to add in 2's complement yet! Later…!*
- Sign Extension Shortcut: To convert an n-bit integer into an integer with more than n bits – i.e., to make a narrow integer fill a wider word – *replicate the most significant bit* (*msb*) of the original number to fill the new bits to its left
  - *Example*: 4-bit 8-bit

```
0010   =   0000 0010
1010   =   1111 1010
```

  - *why is this correct? Prove!*

MIPS Notes

- `lb` vs. `lbu`
  - signed load sign extends to fill 24 left bits
  - unsigned load fills left bits with 0's
- `slt` & `slti`
  - compare signed numbers
- `sltu` & `sltiu`
  - compare unsigned numbers, i.e., treat both operands as non-negative

CamScanner الممسوحة ضوئيا بـ

Two's Complement Addition

- Perform add just as in junior school (carry/borrow 1s)
  - Examples (4-bits):

0101	0110	1011	1001	1111
0001	0101	0111	1010	1110

Do these sums **now**!! Remember all registers are 4-bit including result register!

So you have to **throw away** the carry-out from the msb!!

- Have to beware of *overflow* : if the *fixed* number of bits (4, 8, 16, 32, etc.) in a register *cannot represent the result* of the operation
  - *terminology alert*: overflow *does not mean* there was a carry-out from the msb that we lost (though it sounds like that!) – it means simply that the result in the fixed-sized register is incorrect
    - as can be seen from the above examples there are cases when the result is correct even after losing the carry-out from the msb

Two's Complement Addition: Verifying Carry/Borrow method

- Two (n+1)-bit integers: $X = x_n X'$, $Y = y_n Y'$

Carry/borrow add X + Y	$0 \le X' + Y' < 2^n$ (*no CarryIn to last bit*)	$2^n \le X' + Y' < 2^{n+1} - 1$ (*CarryIn to last bit*)
$x_n = 0,\ y_n = 0$	ok	not ok(overflow!)
$x_n = 1,\ y_n = 0$	ok	ok
$x_n = 0,\ y_n = 1$	ok	ok
$x_n = 1,\ y_n = 1$	not ok(overflow!)	ok

- *Prove the cases above!*
- Prove if there is *one more bit* (total n+2 then) available for the result then there is no problem with overflow in add!

CamScanner الممسوحة ضوئيا بـ

Two's Complement Operations

- *Now verify the negation shortcut!*
  - consider $X + (\overline{X} + 1) = (X + \overline{X}) + 1$:
    associative law – but what if there is overflow in one of the adds on either side, i.e., the result is wrong…!
  - think *minint* !
  - *Examples*:
    - $-0101 = 1010 + 1 = 1011$
    - $-1100 = 0011 + 1 = 0100$
    - $-1000 \neq 0111 + 1 = 1000$

Detecting Overflow

- *No overflow* when adding a positive and a negative number
- *No overflow* when subtracting numbers with the same sign
- *Overflow occurs* when the result has "wrong" sign (*verify*!):

Operation	Operand A	Operand B	Result Indicating Overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A – B	≥ 0	< 0	< 0
A – B	< 0	≥ 0	≥ 0

- Consider the operations A + B, and A – B
  - *can overflow occur if B is 0 ?*
  - *can overflow occur if A is 0 ?*

CamScanner المسوحة ضوئيا بـ

Effects of Overflow

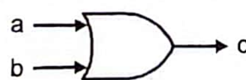
- If an *exception* (interrupt) occurs
  - control jumps to predefined address for exception
  - interrupted address is saved for possible resumption
- Details based on software system/language
  - SPIM: see the EPC and Cause registers
- Don't always want to cause exception on overflow
  - `add, addi, sub` *cause exceptions* on overflow
  - `addu, addiu, subu` *do not cause exceptions* on overflow

Review: Basic Hardware

1. AND gate (c = a . b)

a	b	c = a . b
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate (c = a + b)

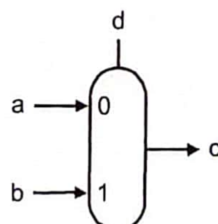


a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter (c = a)

a	c = $\bar{a}$
0	1
1	0

4. Multiplexor (if d = = 0, c = a; else c = b)



d	c
0	a
1	b

المسوحة ضوئيا بـ CamScanner

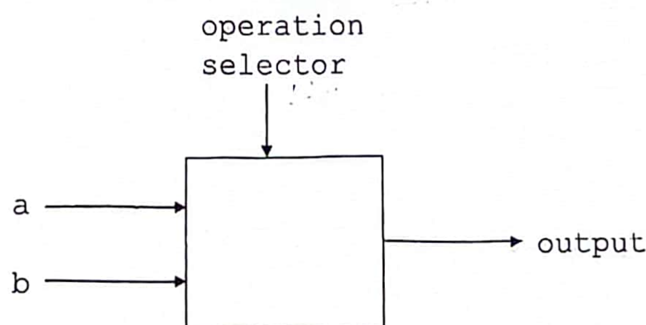
Review: Boolean Algebra & Gates

- *Problem*: Consider logic functions with three inputs: A, B, C.
  - output D is true if at least one input is true
  - output E is true if exactly two inputs are true
  - output F is true only if all three inputs are true

- *Show the truth table for these three functions*
- *Show the Boolean equations for these three functions*
- *Show an implementation consisting of inverters, AND, and OR gates.*

A Simple Multi-Function Logic Unit

- To warm up let's build a logic unit to support the `and` and `or` instructions for MIPS (32-bit registers)
  - we'll just build a 1-bit unit and use 32 of them

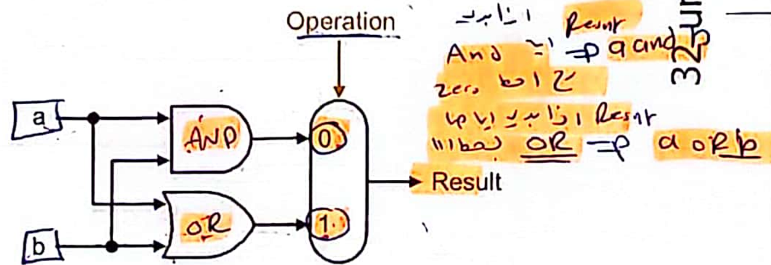


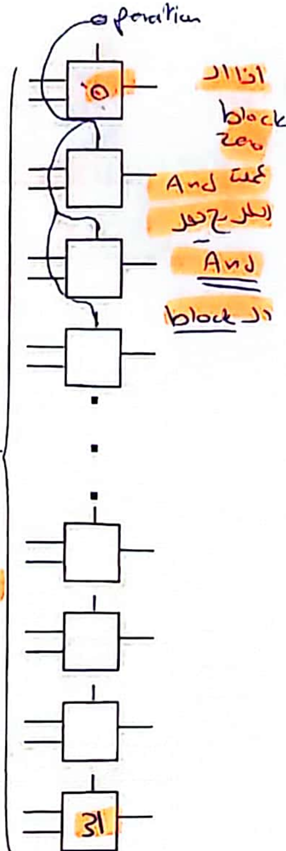
- Possible implementation using a *multiplexor* :

CamScanner الممسوحة ضوئيا بـ

Implementation with a Multiplexor

- Selects one of the inputs to be the output based on a control input

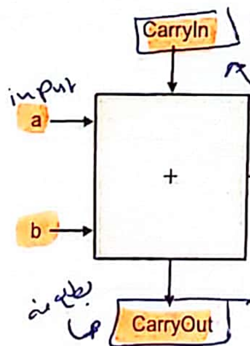




- Lets build our ALU using a MUX (multiplexor):

Implementations

- Not easy to decide the *best* way to implement something
  - do not want too many inputs to a single gate
  - do not want to have to go through too many gates (= levels)
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:



$$c_{out} = a.b + a.c_{in} + b.c_{in}$$

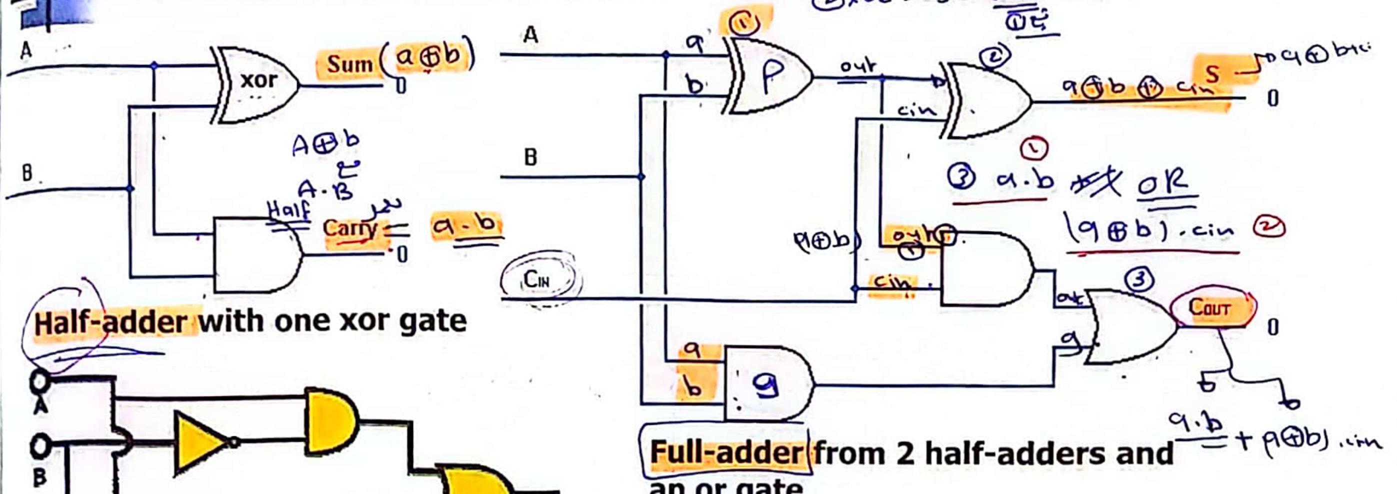
$$sum = a.\bar{b}.\bar{c}_{in} + \bar{a}.b.\bar{c}_{in} + \bar{a}.\bar{b}.c_{in} + a.b.c_{in}$$
$$= a \oplus b \oplus c_{in}$$

exclusive or (xor)

- *How could we build a 1-bit ALU for add, and, and or?*
- *How could we build a 32-bit ALU?*

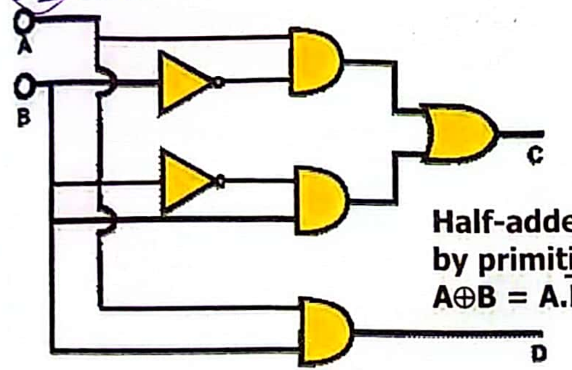
المسوحة ضوئيا بـ CamScanner

1-bit Adder Logic



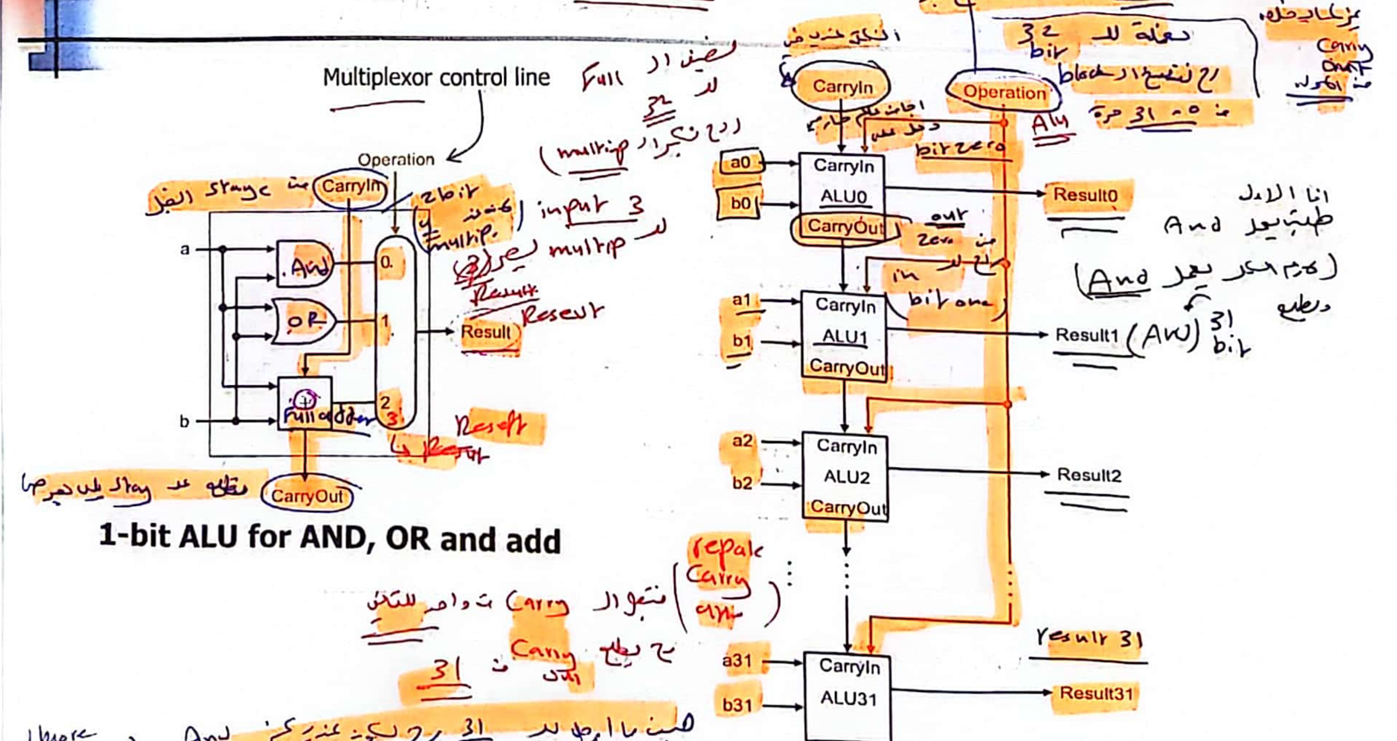
Half-adder with one xor gate

Full-adder from 2 half-adders and an or gate



Half-adder with the xor gate replaced by primitive gates using the equation $A \oplus B = A.\bar{B} + \bar{A}.B$

Building a 32-bit ALU



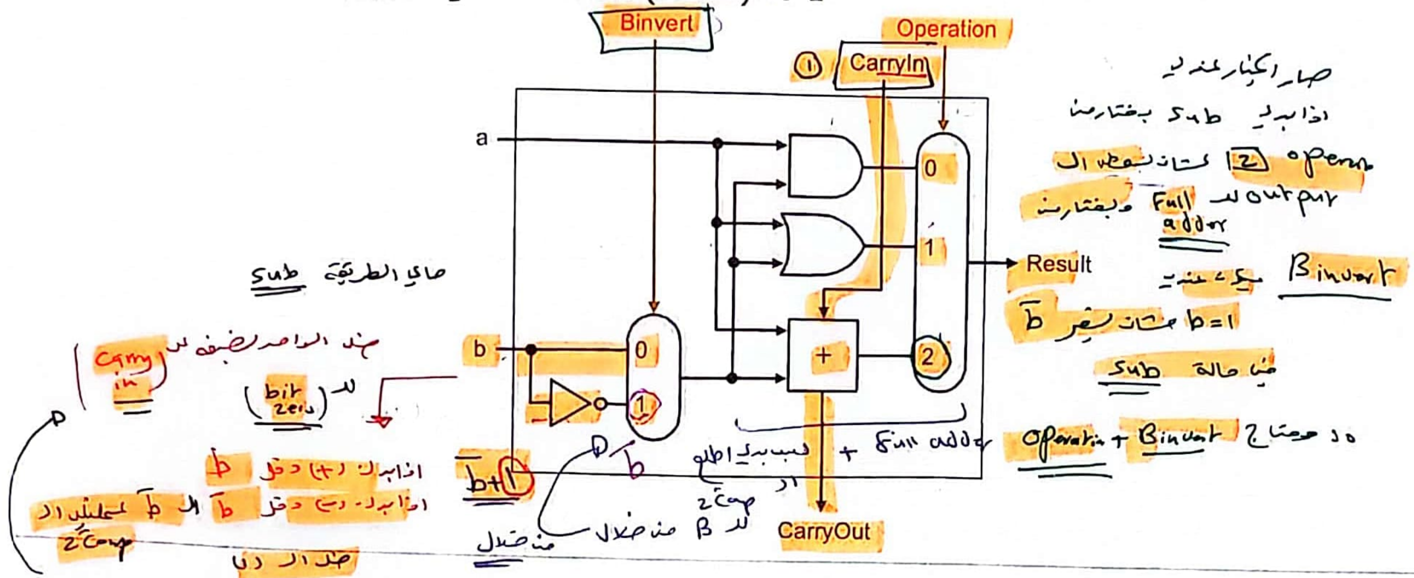
1-bit ALU for AND, OR and add

Ripple-Carry Logic for 32-bit ALU

الممسوحة ضوئيا بـ CamScanner

What about Subtraction (a – b) ?

- Two's complement approach: just negate b and add.
- How do we negate?
  - recall *negation shortcut* : invert each bit of b and set CarryIn to *least significant bit* (ALU0) to 1



Tailoring the ALU to MIPS: Test for Less-than and Equality

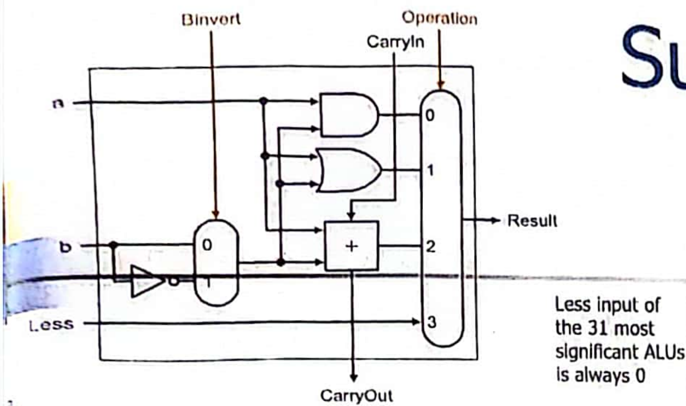
- Need to support the *set-on-less-than* instruction
  - e.g., `slt $t0, $t3, $t4`
  - remember: `slt` is an *R-type instruction* that produces 1 if rs < rt and 0 otherwise
  - idea is to use subtraction: rs < rt ⇔ rs – rt < 0. Recall msb of negative number is 1
  - two cases after subtraction rs – rt:
    - if no overflow then rs < rt ⇔ most significant bit of rs – rt = 1
    - if overflow then rs < rt ⇔ most significant bit of rs – rt = 0
  - why?
  - e.g., $5_{ten} - 6_{ten}$ = 0101 – 0110 = 0101 + 1010 = 1111 (ok!)
    $-7_{ten} - 6_{ten}$ = 1001 – 0110 = 1001 + 1010 = 0011 (overflow!)
  - therefore

    set bit = msb of rs – rt ⊕ overflow bit

    where *set bit*, which is output from ALU31, gives the result of `slt`
    - Fig. 4.17(lower) indicates set bit is the adder output – *not correct* !!
  - set bit is sent from ALU31 to ALU0 as the *Less* bit at ALU0; all other Less bits are hardwired 0; so Less is the 32-bit result of `slt`

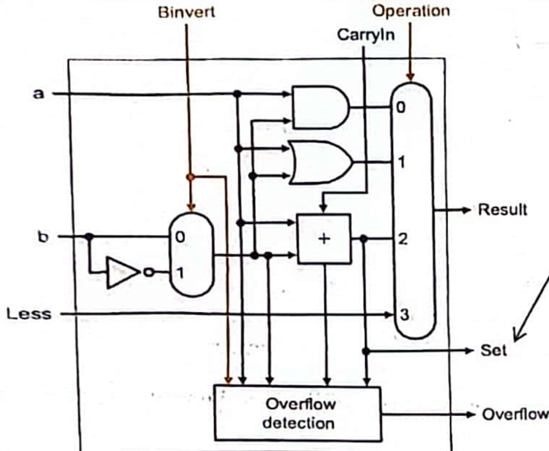
الممسوحة ضوئيا بـ CamScanner

Supporting slt



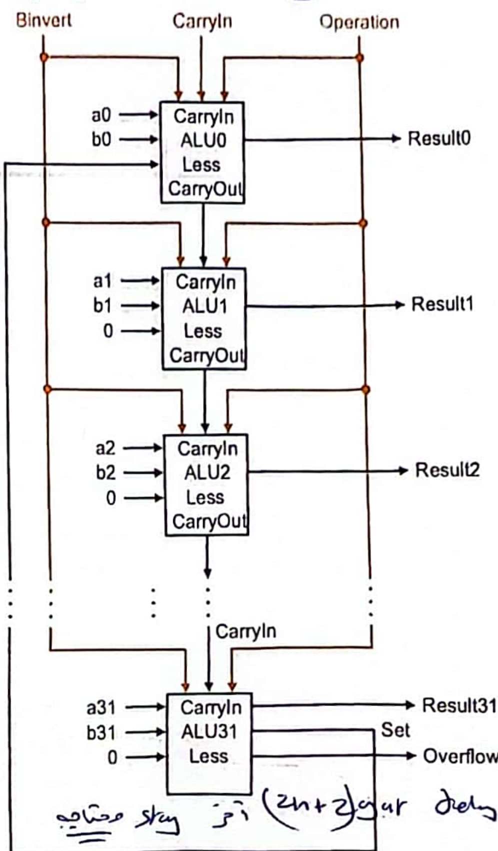
1- bit ALU for the 31 least significant bits

Extra set bit, to be routed to the Less input of the least significant 1-bit ALU, is computed from the most significant Result bit and the Overflow bit (it is *not* the output of the adder as the figure seems to indicate)



1-bit ALU for the most significant bit

Less input of the 31 most significant ALUs is always 0



32-bit ALU from 31 copies of ALU at top left and 1 copy of ALU at bottom left in the most significant position

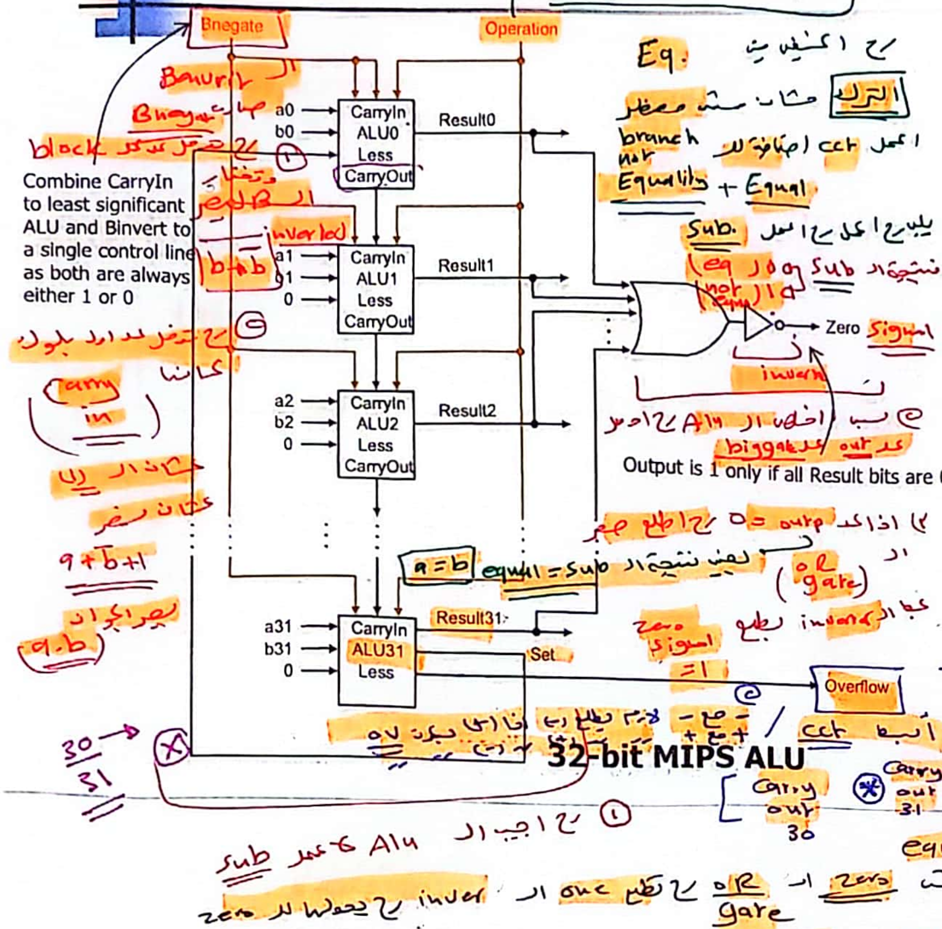
Tailoring the ALU to MIPS: Test for Less-than and Equality

- What about logic for the *overflow bit* ?
  - overflow bit = carry *in to* msb ⊕ carry *out of* msb
  - verify!
  - logic for overflow detection therefore can be put in to ALU31
- Need to support *test for equality*
  - e.g., `beq $t5, $t6, $t7`
  - use subtraction: rs - rt = 0 ⇔ rs = rt
  - *do we need to consider overflow?*

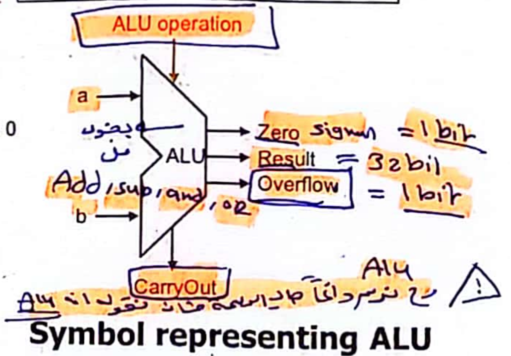
CamScanner الممسوحة ضوئيا بـ

Supporting Test for Equality

ALU control lines		
Bnegate	Operation	Function
0	00	and
0	01	or
0	10	add
1	10	sub
1	11	slt



32-bit MIPS ALU



Symbol representing ALU

Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all gates are always working
  - speed of a gate depends number of inputs (fan-in) to the gate
  - speed of a circuit depends on number of gates in series (particularly, on the *critical path* to the deepest level of logic)
- Speed of MIPS operations
  - clever changes to organization can improve performance (similar to using better algorithms in software)
  - we'll look at examples for addition, multiplication and division

المسوحة ضوئيا بـ CamScanner

Problem: Ripple-carry Adder is Slow

- *Is a 32-bit ALU as fast as a 1-bit ALU? Why?*
  - *n-bit ripple carry adder is approximately 2n + 2 gate delays*
- *Is there more than one way to do addition? Yes:*
  - one extreme: *ripple-carry* – carry ripples through 32 ALUs, slow!
  - other extreme: sum-of-products for each CarryIn bit – super fast!
  - CarryIn bits:

> Note: c_i is CarryIn bit *into i* th ALU; c_0 is the forced CarryIn into the least significant ALU

$$c_1 = b_0 . c_0 + a_0 . c_0 + a_0 . b_0$$

$$c_2 = b_1 . c_1 + a_1 . c_1 + a_1 . b_1$$
$$= a_1 . a_0 . b_0 + a_1 . a_0 . c_0 + a_1 . b_0 . c_0 \quad \text{(substituting for } c_1\text{)}$$
$$+ b_1 . a_0 . b_0 + b_1 . a_0 . c_0 + b_1 . b_0 . c_0 + a_1 . b_1$$

$$c_3 = b_2 . c_2 + a_2 . c_2 + a_2 . b_2$$
$$= \ldots = \text{sum of 15 4-term products} \ldots$$

- How fast? But not feasible for a 32-bit ALU! Why? Exponential complexity!!

Two-level Carry-lookahead Adder: First Level

- An approach between our two extremes
- Motivation:
  - if we didn't know the value of a carry-in, what could we do?
  - when would we always generate a carry? (generate) $g_i = a_i . b_i$
  - when would we propagate the carry? (propagate) $p_i = a_i + b_i$
- Express (carry-in equations in terms of generate/propagates)

$$c_1 = g_0 + p_0 \cdot c_0$$
$$c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$$
$$c_3 = g_2 + p_2 \cdot c_2 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0$$
$$c_4 = g_3 + p_3 \cdot c_3 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0$$
$$+ p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

- Feasible for 4-bit adders – with wider adders unacceptable complexity.
  - solution: build a *first level using 4-bit adders*, then a *second level on top*

CamScanner الممسوحة ضوئيا بـ

Two-level Carry-lookahead Adder: Second Level for a 16-bit adder

- Propagate signals for each of the four 4-bit adder blocks:

$$P_0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0$$
$$P_1 = p_7 \cdot p_6 \cdot p_5 \cdot p_4$$
$$P_2 = p_{11} \cdot p_{10} \cdot p_9 \cdot p_8$$
$$P_3 = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}$$

- Generate signals for each of the four 4-bit adder blocks:

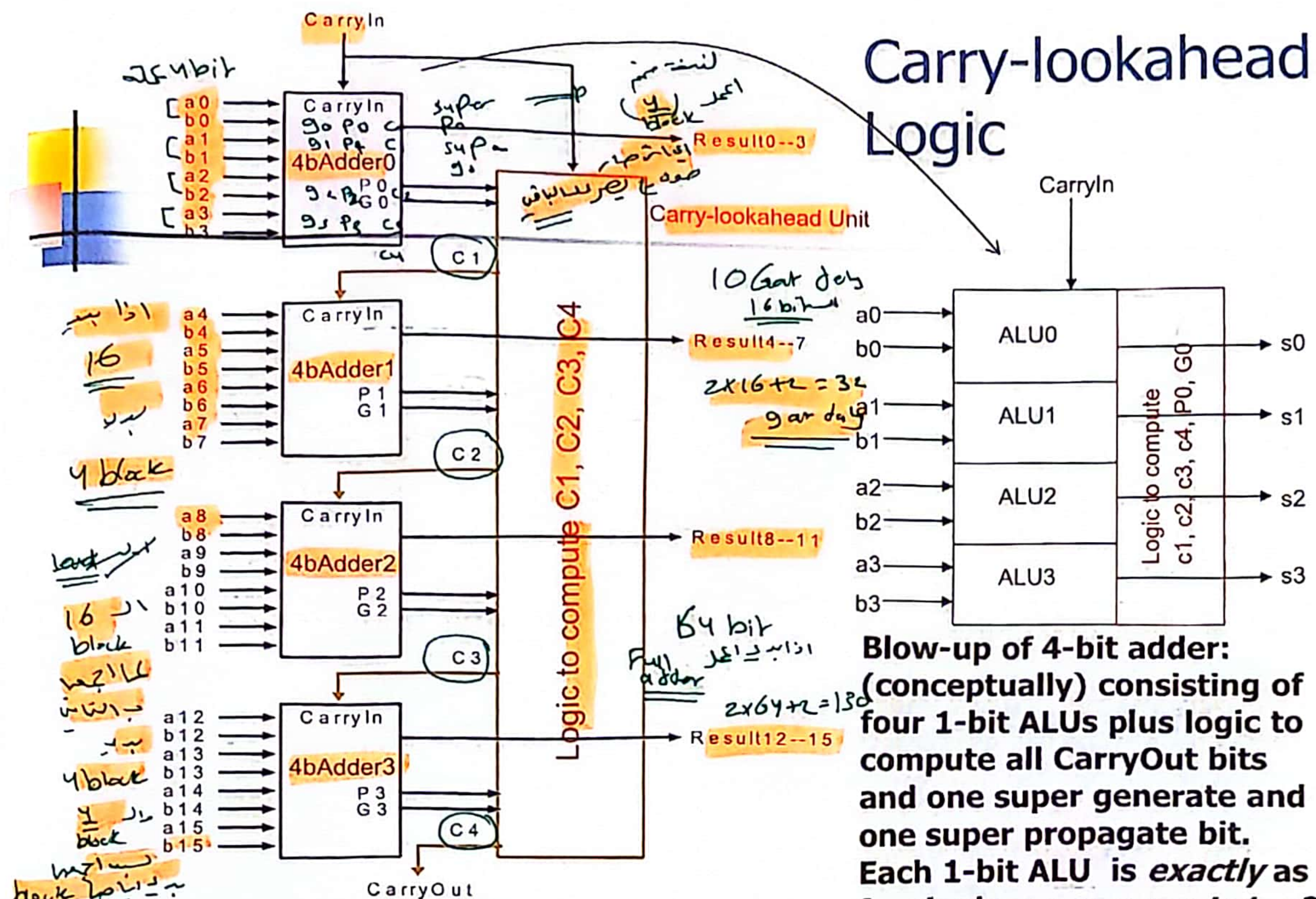
$$G_0 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0$$
$$G_1 = g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4$$
$$G_2 = g_{11} + p_{11} \cdot g_{10} + p_{11} \cdot p_{10} \cdot g_9 + p_{11} \cdot p_{10} \cdot p_9 \cdot g_8$$
$$G_3 = g_{15} + p_{15} \cdot g_{14} + p_{15} \cdot p_{14} \cdot g_{13} + p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12}$$

Two-level Carry-lookahead Adder: Second Level for a 16-bit adder

- CarryIn signals for each of the four 4-bit adder blocks (see earlier carry-in equations in terms of generate/propagates):

$$C_1 = G_0 + P_0 \cdot c_0$$
$$C_2 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot c_0$$
$$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot c_0$$
$$C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0$$

الممسوحة ضوئيا بـ CamScanner



16-bit carry-lookahead adder from four 4-bit adders and one carry-lookahead unit

Blow-up of 4-bit adder: (conceptually) consisting of four 1-bit ALUs plus logic to compute all CarryOut bits and one super generate and one super propagate bit. Each 1-bit ALU is *exactly* as for ripple-carry *except* c1, c2, c3 for ALUs 1, 2, 3 comes from the extra logic

Two-level Carry-lookahead Adder: Second Level for a 16-bit adder

- Two-level carry-lookahead logic *steps*:
  1. compute p_i's and g_i's at each 1-bit ALU
  2. compute P_i's and G_i's at each 4-bit adder unit
  3. compute C_i's in carry-lookahead unit
  4. compute c_i's at each 4-bit adder unit
  5. compute results (sum bits) at each 1-bit ALU
- *E.g.*, add using carry-lookahead logic:
  - 0001 1010 0011 0011
  - 1110 0101 1110 1011
- *Compare times for ripple-carry vs. carry-lookahead for a 16-bit adder assuming unit delay at each gate*

CamScanner المسوحة ضوئيا بـ

6707 – 6677 = 30

A	0001	1010	0011	0011
B	1110	0101	1110	1011
gi	0000	0000	0010	0011
pi	1111	1111	1111	1011
Pi	1	1	1	0
Gi	0	0	1	0
Ci	1	1	1	0
Sum	0000	0000	0001	1110

Multiply

- Grade school shift-add method:

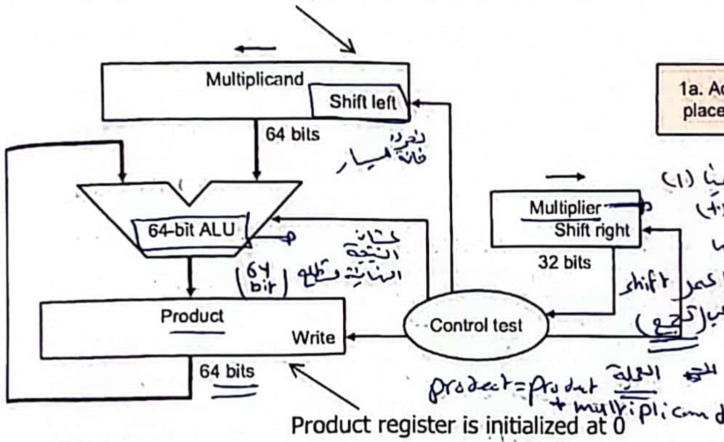
Multiplicand	1000
Multiplier	x 1001
	1000
	0000
	0000
	1000
Product	01001000

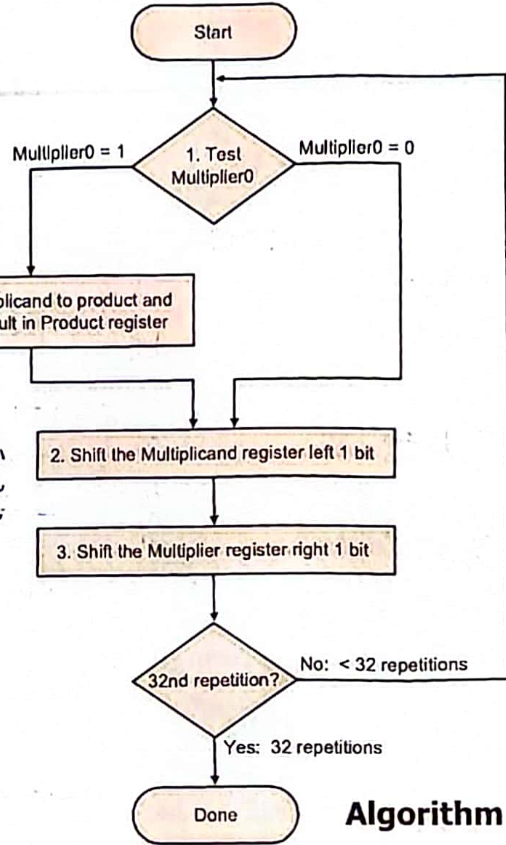
- m bits x n bits = m+n bit product
- Binary makes it easy:
  - multiplier bit 1 => copy multiplicand (1 x multiplicand)
  - multiplier bit 0 => place 0 (0 x multiplicand)
- 3 versions of multiply hardware & algorithm:

الممسوحة ضوئيا بـ CamScanner

Shift-add Multiplier Version 1

32-bit multiplicand starts at right half of multiplicand register

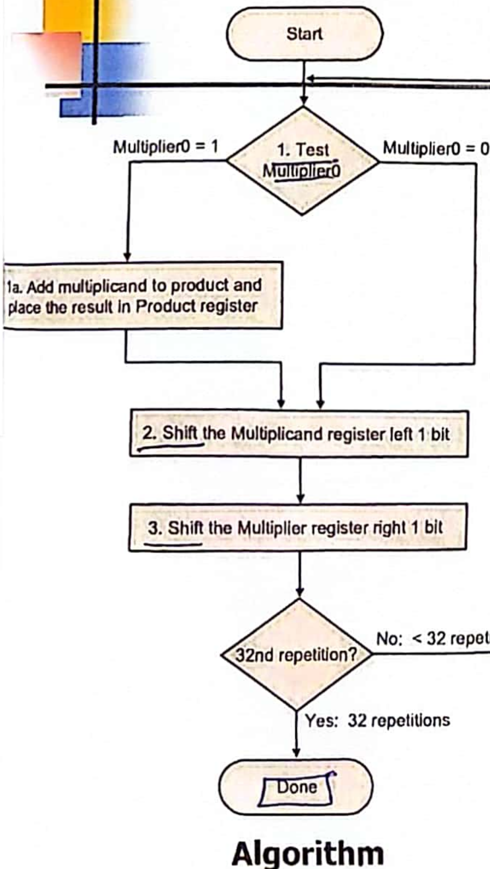




Algorithm

Multiplicand register, product register, ALU are 64-bit wide; multiplier register is 32-bit wide

Shift-add Multiplier Version1



Algorithm

Example: 0010 * 0011:

Itera -tion	Step	Multiplier	Multiplicand	Product
0	initial values	0011	0000 0010	0000 0000
1	1a => 1	0011	0000 0010	0000 0010
	2	0011	0000 0100	0000 0010
	3	0001	0000 0100	0000 0010
2	..			

CamScanner الممسوحة ضوئيا بـ

Step	Multiplier	Multiplicand	Product
	M'ier: 0011	M'and: 0000 0010	P: 0000 0000
1a. 1=>P=P+Mcand	M'ier: 0011	Mcand: 0000 0010	P: 0000 0010
2. Shl Mcand	M'ier: 0011	Mcand: 0000 0100	P: 0000 0010
3. Shr M'ier	M'ier: 0001	Mcand: 0000 0100	P: 0000 0010
1a. 1=>P=P+Mcand	M'ier: 0001	Mcand: 0000 0100	P: 0000 0110
2. Shl Mcand	M'ier: 0001	Mcand: 0000 1000	P: 0000 0110
3. Shr M'ier	M'ier: 0000	Mcand: 0000 1000	P: 0000 0110
1. 0=>nop	M'ier: 0000	Mcand: 0000 1000	P: 0000 0110
2. Shl Mcand	M'ier: 0000	Mcand: 0001 0000	P: 0000 0110
3. Shr M'ier	M'ier: 0000	Mcand: 0001 0000	P: 0000 0110
1. 0=>nop	M'ier: 0000	Mcand: 0001 0000	P: 0000 0110
2. Shl Mcand	M'ier: 0000	Mcand: 0010 0000	P: 0000 0110
3. Shr M'ier	M'ier: 0000	Mcand: 0010 0000	P: 0000 0110

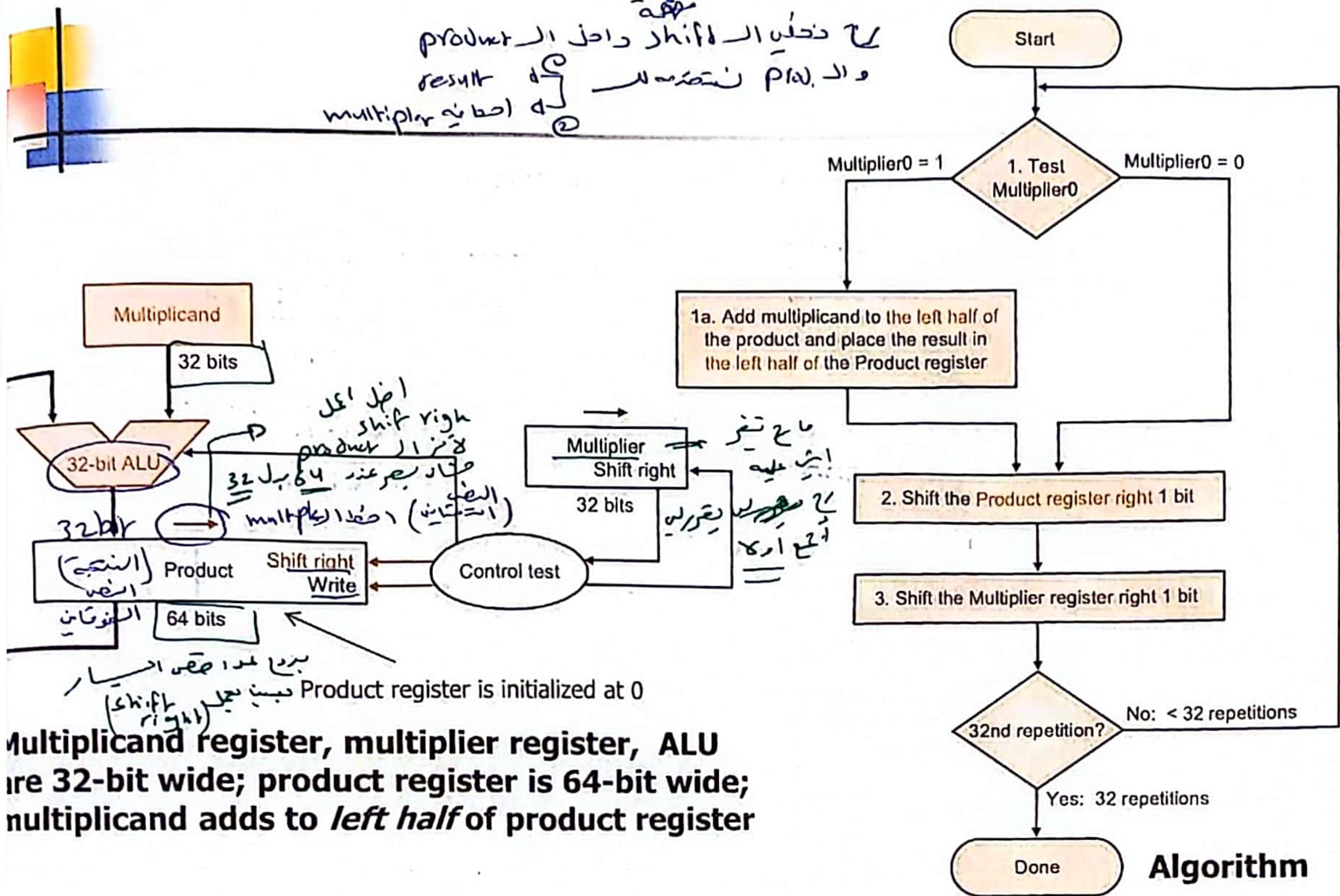
Observations on Multiply Version 1

- 1 step per clock cycle ⇒ nearly 100 clock cycles to multiply two 32-bit numbers
- Half the bits in the multiplicand register always 0
  ⇒ 64-bit adder is wasted
- 0's inserted to right as multiplicand is shifted left
  ⇒ least significant bits of product never change once formed

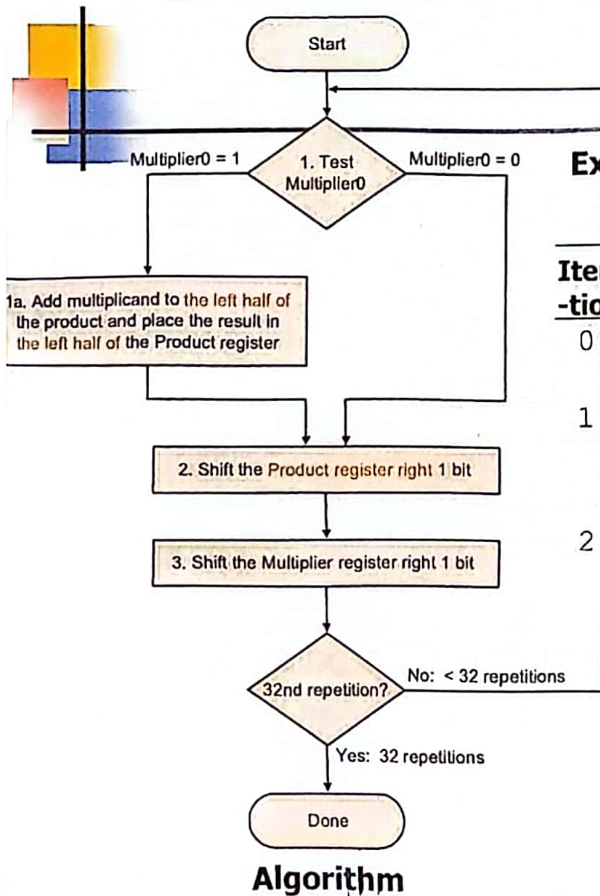
- Intuition: instead of shifting multiplicand to left, shift product to right...

CamScanner الممسوحة ضوئيا بـ

Shift-add Multiplier Version 2



Shift-add Multiplier Version 2



Example: 0010 * 0011:

Itera-tion	Step	Multiplier	Multiplicand	Product
0	init values	0011	0010	0000 0000
1	1a	0011	0010	0010 0000
	2	0011	0010	0001 0000
	3	0001	0010	0001 0000
2	...			

المسوحة ضوئيا بـ CamScanner

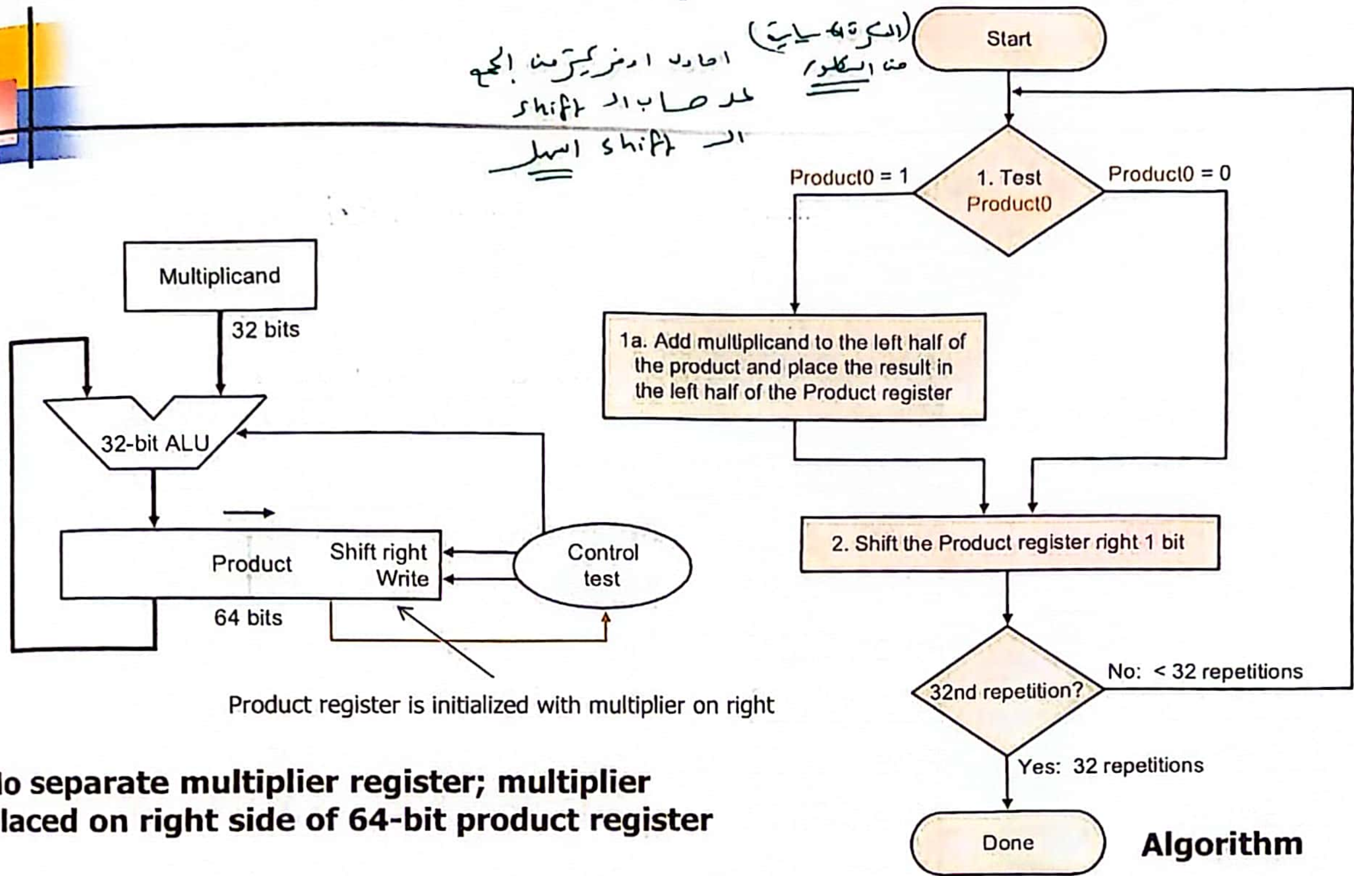
Step	Multiplier	Multiplicand	Product
	M'ier: 0011	Mcand: 0010	P: 0000 0000
1a. 1=>P=P+Mcand	M'ier: 0011	Mcand: 0010	P: 0010 0000
2. Shr P	M'ier: 0011	Mcand: 0010	P: 0001 0000
3. Shr M'ier	M'ier: 0001	Mcand: 0010	P: 0001 0000
1a. 1=>P=P+Mcand	M'ier: 0001	Mcand: 0010	P: 0011 0000
2. Shr P	M'ier: 0001	Mcand: 0010	P: 0001 1000
3. Shr M'ier	M'ier: 0000	Mcand: 0010	P: 0001 1000
1. 0=>nop	M'ier: 0000	Mcand: 0010	P: 0001 1000
2. Shr P	M'ier: 0000	Mcand: 0010	P: 0000 1100
3. Shr M'ier	M'ier: 0000	Mcand: 0010	P: 0000 1100
1. 0=>nop	M'ier: 0000	Mcand: 0010	P: 0000 1100
2. Shr P	M'ier: 0000	Mcand: 0010	P: 0000 0110
3. Shr M'ier	M'ier: 0000	Mcand: 0010	P: 0000 0110

Observations on Multiply Version 2

- Each step the product register wastes space that exactly matches the current size of the multiplier
- Intuition: combine multiplier register and product register…

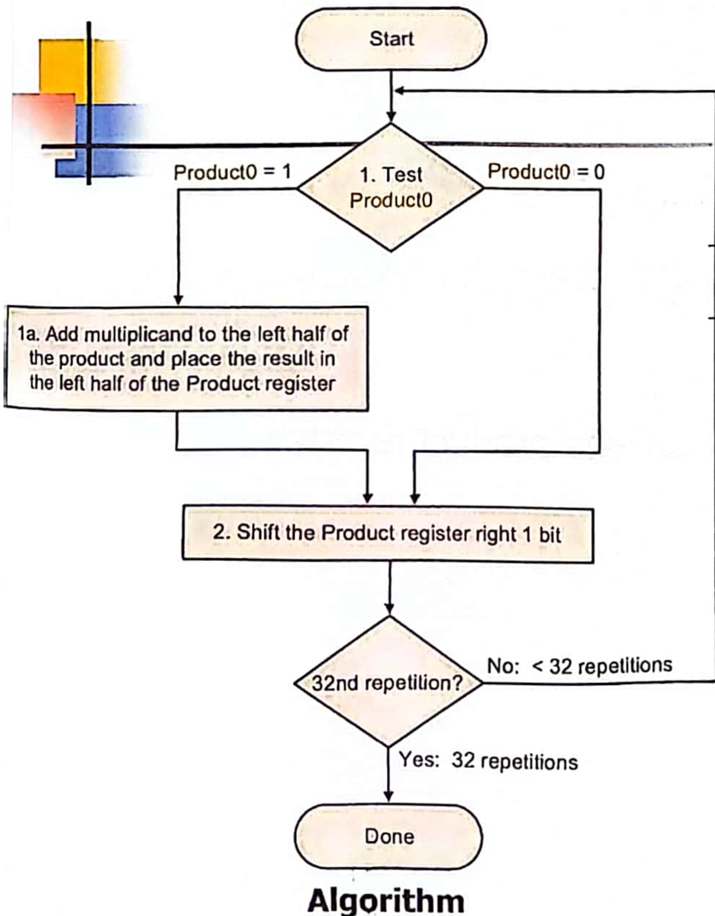
CamScanner الممسوحة ضوئيا بـ

Shift-add Multiplier Version 3



No separate multiplier register; multiplier placed on right side of 64-bit product register

Shift-add Multiplier Version 3



Example: 0010 * 0011:

Itera-tion	Step	Multiplicand	Product
0	init values	0010	0000 0011
1	1a	0010	0010 0011
	2	0010	0001 0001
2	…		

CamScanner المسوحة ضوئيا بـ

Observations on Multiply Version 3

- 2 steps per bit because multiplier & product combined
- What about *signed* multiplication?
  - easiest solution is to make both positive and remember whether to negate product when done, i.e., leave out the sign bit, run for 31 steps, then negate if multiplier and multiplicand have opposite signs

- Booth's Algorithm is an elegant way to multiply signed numbers using same hardware – it also often quicker...

Motivating Booth's algorithm

- Example 0010 * 0110. Traditional:

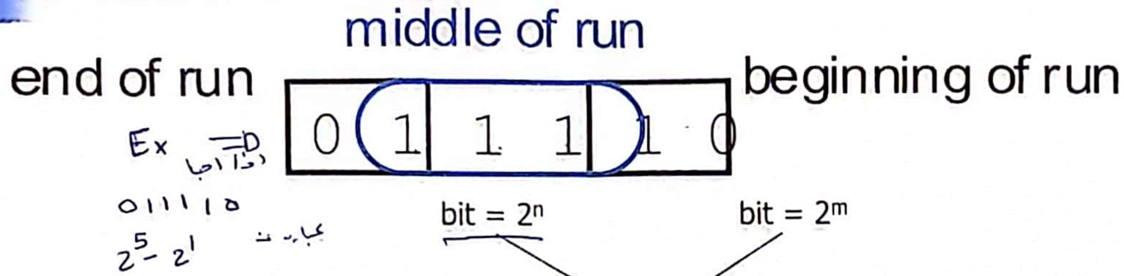
```
      0010
   x  0110
      ----
      0000    shift (0 in multiplier)
     0010     add (1 in multiplier)
    0010      add (1 in multiplier)
   0000       shift (0 in multiplier)
  --------
  00001100
```

- Same example. But observe there are two successive 1's in multiplier $0110 = 2^2 + 2^1 = 2^3 - 2^1$, so can replace successive 1's by subtract and then add:

```
      0010
      0110
      ----
      0000    shift (0 in multiplier)
    -0010     sub  (first 1 in multiplier)
    0000      shift (middle of string of 1's)
   0010       add (previous step had last 1)
  --------
  00001100
```

CamScanner المسوحة ضوئيا بـ

Motivating Booth's Algorithm



- Math idea: string of 1's ...011...10... has successive 1's

  value the sum $2^n + 2^{n-1} + \ldots + 2^m = 2^{n+1} - 2^m$

- Replace a string of 1s in multiplier with an *initial subtract when we first see a one* and then later *add after the last one*
  - *What if the string of 1's started from the left of the (2's complement) number, e.g., 11110001 – would the formula above have to be modified?!*

Booth from Multiply Version 3

- *Modify Step 1* of the algorithm Multiply Version 3 to consider *2 bits* of the multiplier: *the current bit and the bit to the right* (i.e., the current bit of the previous step). Instead of two outcomes, now there are four:

Case	Current Bit	Bit to the Right	Explanation	Example	Op
1a	0	0	Middle of run of 0s	0001111000	none
1b	0	1	End of run of 1s	0001111000	add
1c	1	0	Begins run of 1s	0001111000	sub
1d	1	1	Middle of run of 1s	0001111000	none

- *Modify Step 2* of Multiply Version 3 to *sign extend* when the product is shifted right (*arithmetic right shift*, rather than *logical right shift*) because the product is a signed number
- *Now draw the flowchart for Booth's algorithm* !
- Multiply Version 3 and Booth share the same hardware, *except* Booth requires one extra flipflop to remember the bit to the right of the current bit in the product register – which is the bit pushed out by the preceding right shift

CamScanner المسوحة ضوئيا بـ

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt / multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd / mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

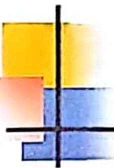
MIPS Notes

- MIPS provides two 32-bit registers `Hi` and `Lo` to hold a 64-bit product
- `mult`, `multu` (unsigned) put the product of two 32-bit register operands into `Hi` and `Lo`: overflow is ignored by MIPS but can be detected by programmer by examining contents of `Hi`
- `mflo`, `mfhi` moves content of `Hi` or `Lo` to a general-purpose register
- Pseudo-instructions `mul` (without overflow), `mulo` (with overflow), `mulou` (unsigned with overflow) take three 32-bit register operands, putting the product of two registers into the third

CamScanner المسوحة ضوئيا بـ

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt` / `divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result



MIPS Notes

- `div` (signed), `divu` (unsigned), with two 32-bit register operands, divide the contents of the operands and put remainder in `Hi` register and quotient in `Lo`; overflow is ignored in both cases
- pseudo-instructions `div` (signed with overflow), `divu` (unsigned without overflow) with three 32-bit register operands puts quotients of two registers into third

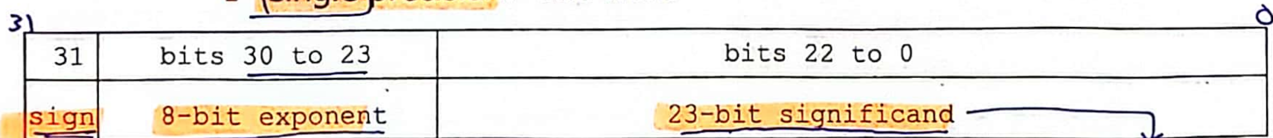
CamScanner الممسوحة ضوئيا بـ

Floating Point

- We need a way to represent
  - numbers with fractions, e.g., `3.1416`
  - very small numbers (in absolute value), e.g., `.00000000023`
  - very large numbers (in absolute value) , e.g., $-3.15576 * 10^{46}$
- Representation:
  - *scientific*: sign, exponent, significand form:
    - $(-1)^{sign} * \text{significand} * 2^{exponent}$. E.g., $101.001101 * 2^{111001}$ ← binary point
  - more bits for *significand* gives more accuracy
  - more bits for *exponent* increases range
  - if $1 \le$ significand $< 10_{two}(=2_{ten})$ then number is *normalized*, **except for** number 0 which is normalized to significand 0
    - E.g., $-101.001101 * 2^{111001} = -1.01001101 * 2^{111011}$ (normalized)

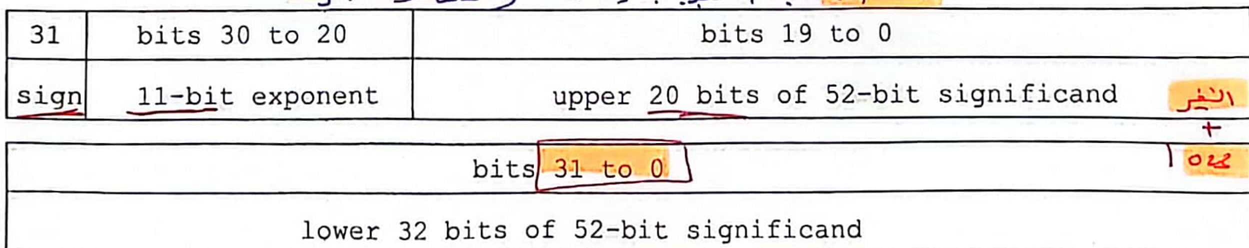
IEEE 754 Floating-point Standard

- IEEE 754 floating point standard:
  - single precision: one word



31	bits 30 to 23	bits 22 to 0
sign	8-bit exponent	23-bit significand

  - double precision: two words



31	bits 30 to 20	bits 19 to 0
sign	11-bit exponent	upper 20 bits of 52-bit significand

bits 31 to 0
lower 32 bits of 52-bit significand

CamScanner المسوحة ضوئيا بـ

IEEE 754 Floating-point Standard

- Sign bit is 0 for positive numbers, 1 for negative numbers
- Number is assumed normalized and leading 1 bit of significand left of binary point (for non-zero numbers) is *assumed* and not shown
  - e.g., significand 1.1001... is represented as 1001...,
  - **exception** is number 0 which is represented as all 0s (see next slide)
  - for other numbers:
    value = $(-1)^{sign} * (1 + \text{significand}) * 2^{\text{exponent value}}$
- Exponent is *biased* to make sorting easier
  - all 0s is smallest exponent, all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - therefore, for non-0 numbers:
    value = $(-1)^{sign} * (1 + \text{significand}) * 2^{(\text{exponent} - \text{bias})}$ (exponent − bias: equals exponent value)

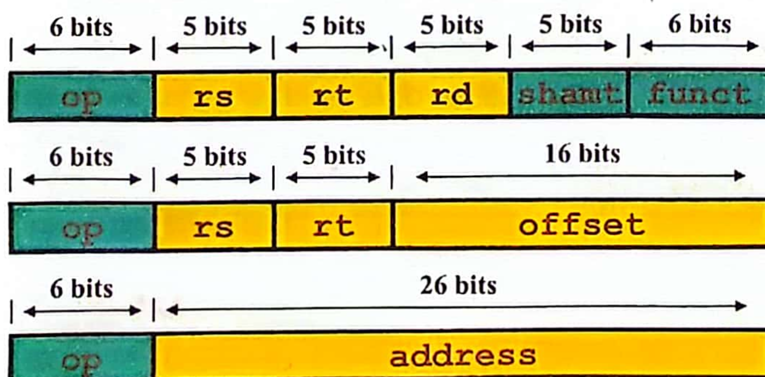
IEEE 754 Floating-point Standard

- Special treatment of 0:
  - if exponent is all 0 and significand is all 0, then the value is 0 (sign bit may be 0 or 1)
  - if exponent is all 0 and significand is *not* all 0, then the value is $(-1)^{sign} * (1 + \text{significand}) * 2^{-127}$
    - therefore, all 0s is taken to be 0 and not 2^{-127} (as would be for a non-zero normalized number); similarly, 1 followed by all 0's is taken to be 0 and not -2^{-127}
- *Example*: Represent -0.75_{ten} in IEEE 754 single precision
  - decimal: $-0.75 = -3/4 = -3/2^2$
  - binary: $-11/100 = -.11 = -1.1 \times 2^{-1}$
  - IEEE single precision floating point exponent = bias + exponent value $= 127 + (-1) = 126_{ten} = 01111110_{two}$
  - IEEE single precision: 10111111010000000000000000000000
    (sign | exponent | significand)

COD Ch. 5
The Processor: Datapath and Control

Implementing MIPS

- We're ready to look at an implementation of the MIPS instruction set
- Simplified to contain only
  - arithmetic-logic instructions: `add`, `sub`, `and`, `or`, `slt`
  - memory-reference instructions: `lw`, `sw`
  - control-flow instructions: `beq`, `j`



R-Format

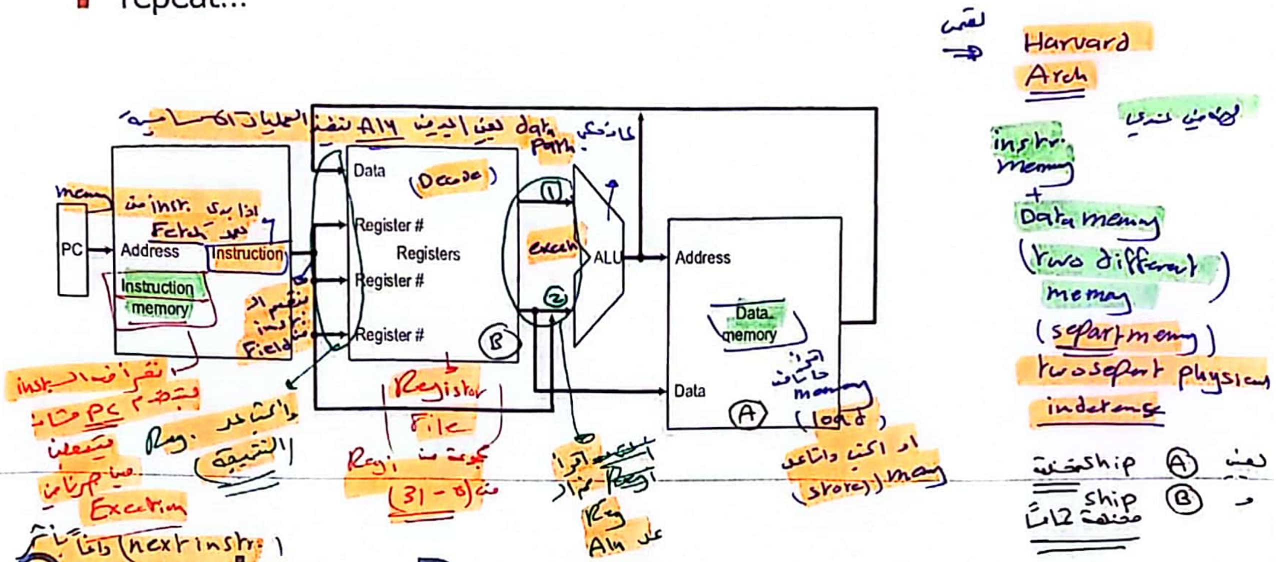
I-Format

J-Format

CamScanner المسوحة ضوئيا بـ

Implementing MIPS: the Fetch/Execute Cycle

- High-level abstract view of *fetch/execute* implementation
  - use the program counter (PC) to read instruction address
  - *fetch* the instruction from memory and increment PC
  - use fields of the instruction to select registers to read
  - *execute* depending on the instruction
  - repeat...



Overview: Processor Implementation Styles

- Single Cycle
  - perform each instruction in 1 clock cycle
  - clock cycle must be long enough for slowest instruction; therefore,
  - disadvantage: only as fast as slowest instruction
- Multi-Cycle
  - break fetch/execute cycle into multiple steps
  - perform 1 step in each clock cycle
  - advantage: each instruction uses only as many cycles as it needs
- Pipelined
  - execute each instruction in multiple steps
  - perform 1 step / instruction in each clock cycle
  - process multiple instructions in parallel – assembly line

الممسوحة ضوئيا بـ CamScanner

Functional Elements

- Two types of functional elements in the hardware:
  - elements that *operate on* data (called *combinational elements*)
  - elements that *contain* data (called *state* or *sequential elements*)

Combinational Elements

- Works as an *input* ⇒ *output function*, e.g., ALU
- Combinational logic *reads input data from one register and writes output data to another*, or same, register
  - *read/write happens in a single cycle* – combinational element *cannot store data* from one cycle to a future one

Combinational logic hardware units

المسموحة ضوئيا بـ CamScanner

State Elements

- State elements contain *data* in internal storage, e.g., *registers* and *memory*
- All state elements together *define* the *state of the machine*
  - *What does this mean? Think of shutting down and starting up again…*
- *Flipflops* and *latches* are 1-bit state elements, equivalently, they are *1-bit memories*
- The *output*(s) of a flipflop or latch *always* depends on the bit value stored, i.e., its state, and can be called *1/0* or *high/low* or *true/false*
- The *input* to a flipflop or latch can change its state depending on whether it is clocked or not…

AvgCPI = Total cycle / Total inst

Ex) 8 inst + 7 stall = 15 cycle + 2 before loop

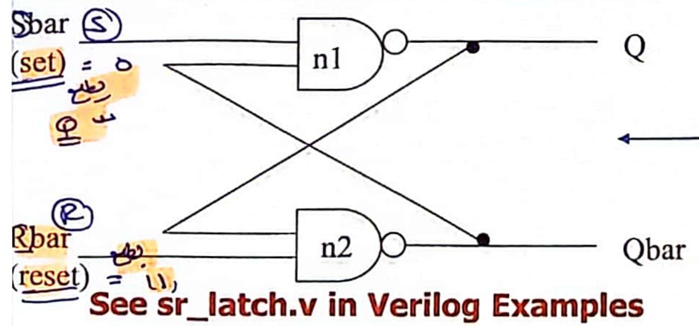
Total cycle = (Iter) × # of cycle + [2]

Total inst. Ex = 8 × [Iter] + [2]

Set, Reset latch

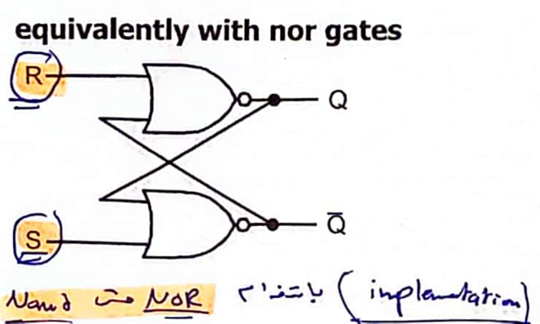
Set-Reset (SR-) latch (unclocked)

Think of Sbar as $\overline{S}$, the inverse of *set* (which sets Q to 1), and Rbar as $\overline{R}$, the inverse of *reset*.



See sr_latch.v in Verilog Examples

equivalently with nor gates



Nand → NOR (implementation)

A set-reset latch made from two cross-coupled *nand* gates is a basic memory unit.

When both Sbar and Rbar are 1, then either *one of the following two* states is *stable*:

a) Q = 1 & Qbar = 0
b) Q = 0 & Qbar = 1

and the latch will *continue* in the current stable state.

If Sbar changes to 0 (while Rbar remains at 1), then the latch is forced to the *exactly one* possible stable state (a). If Rbar changes to 0 (while Sbar remains at 1), the latch is forced to the *exactly one* possible stable state (b).

So, the latch *remembers* which of Sbar or Rbar was last 0 *during* the time they are both 1.

When both Sbar and Rbar are 0 the *exactly one* stable state is Q = Qbar = 1. However, if after that both Sbar and Rbar return to 1, the latch must then *jump non-deterministically* to one of stable states (a) or (b), which is undesirable behavior.

CamScanner الممسوحة ضوئيا بـ

Synchronous Logic: Clocked Latches and Flipflops

- Clocks are used in *synchronous* logic to determine *when* a state element is to be updated
  - in *level-triggered* clocking methodology either the state changes only when the clock is high or only when it is low (technology-dependent)

Falling edge

Clock period

Rising edge

  - in *edge-triggered* clocking methodology either the *rising edge* or *falling edge* is active (depending on technology) – i.e., states change only on rising edges or only on falling edge

- Latches are level-triggered
- Flipflops are edge-triggered

Clocked SR-latch

- State can change only when clock is *high*
- *Potential problem* : both inputs `Sbar = 0` & `Rbar = 0` will cause non-deterministic behavior

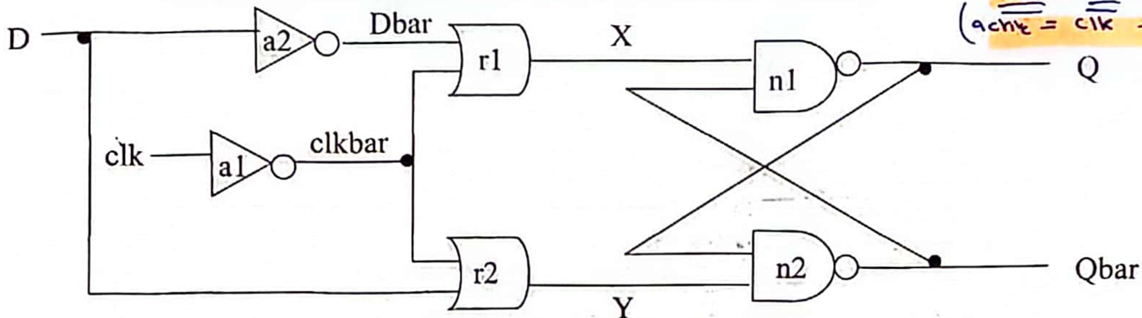
Sbar, clk, a, clkbar, Rbar, r1, r2, X, Y, n1, n2, Q, Qbar

See clockedSr_latch.v in Verilog Examples

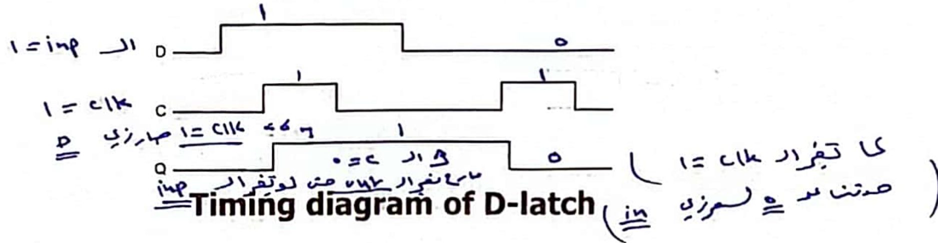
CamScanner المسوحة ضوئيا بـ

Clocked D-latch

- State can change only when clock is *high*
- Only *single* data input (compare SR-latch)
- *No problem* with non-deterministic behavior



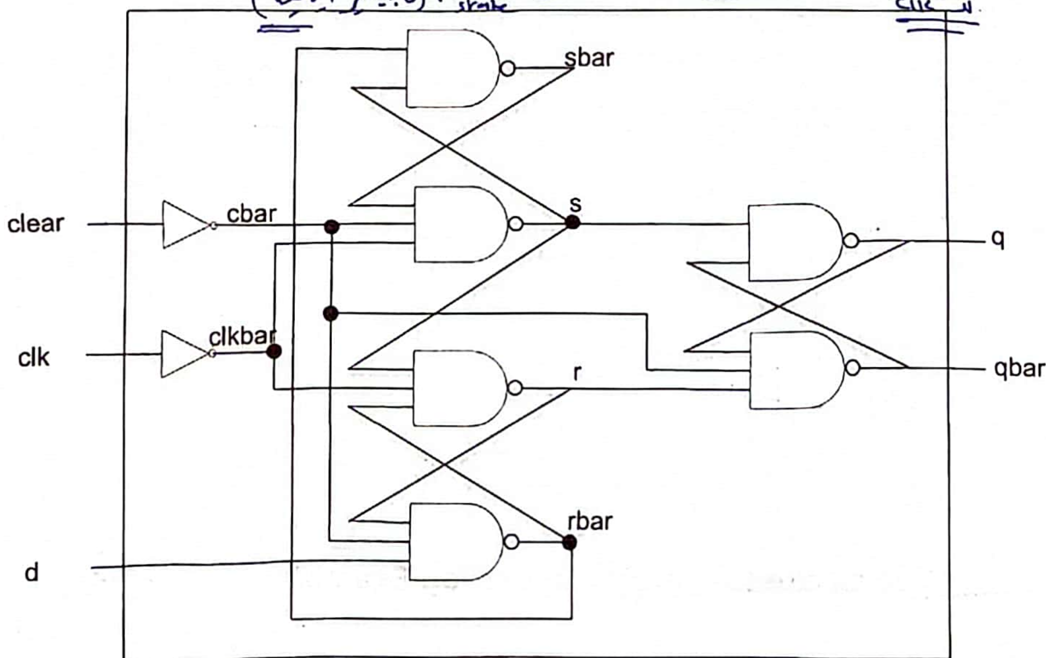
See clockedD_latch.v in Verilog Examples



Timing diagram of D-latch

Clocked D-flipflop

- *Negative edge-triggered*
- Made from *three SR-latches*



See edge_dffGates.v in Verilog Examples

CamScanner المسوحة ضوئيا بـ

State Elements on the Datapath: Register File

- Registers are implemented with arrays of D-flipflops

Register file with two read ports and one write port

State Elements on the Datapath: Register File

- Port implementation:

Read ports are implemented with a pair of multiplexors – 5 bit multiplexors for 32 registers

Write port is implemented using a decoder – 5-to-32 decoder for 32 registers. Clock is relevant to write as register state may change only at clock edge

CamScanner الممسوحة ضوئيا بـ

Verilog

- All components that we have discussed – and shall discuss – can be fabricated using Verilog
- Refer to our Verilog slides and examples

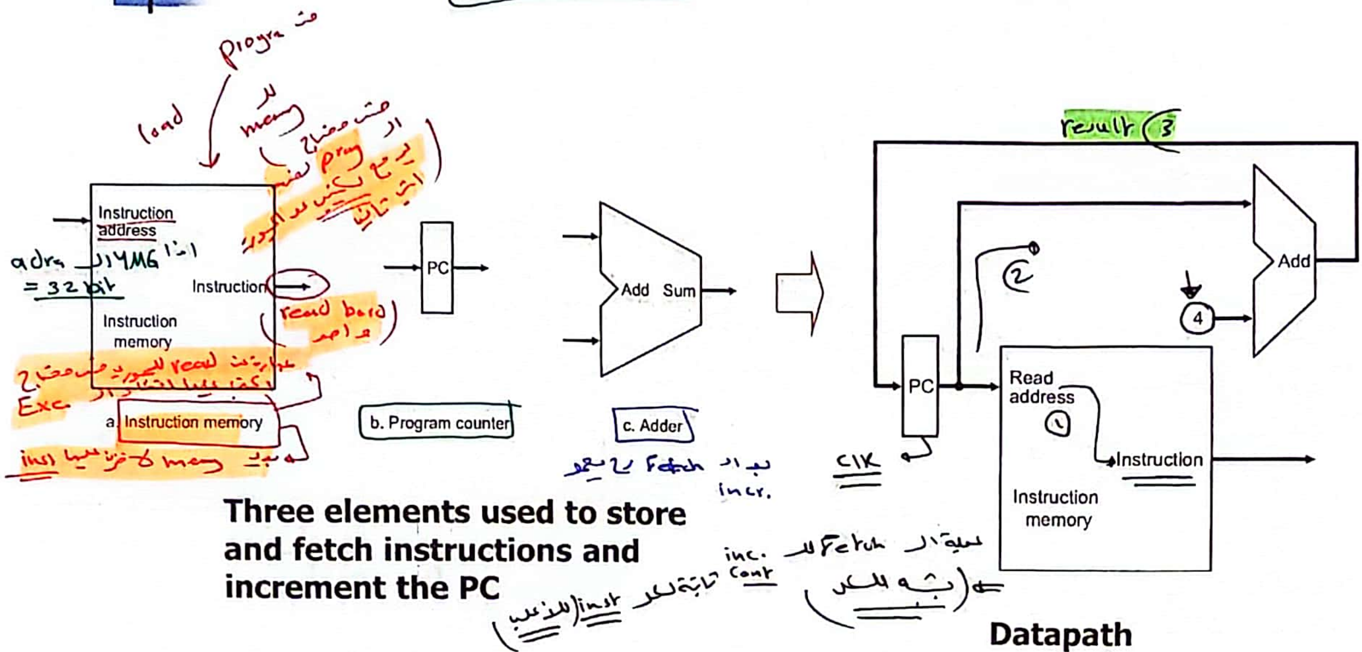
Single-cycle Implementation of MIPS

- Our first implementation of MIPS will use a *single* long clock cycle for every instruction
- Every instruction begins on one up (or, down) clock edge and ends on the next up (or, down) clock edge
- This approach is *not practical* as it is much slower than a *multicycle* implementation where different instruction classes can take different numbers of cycles
  - in a single-cycle implementation every instruction must take the same amount of time as the slowest instruction
  - in a multicycle implementation this problem is avoided by allowing quicker instructions to use fewer cycles
- Even though the single-cycle approach is not practical it is simple and useful to understand first

- *Note* : we shall implement `jump` at the very end

CamScanner الممسوحة ضوئيا بـ

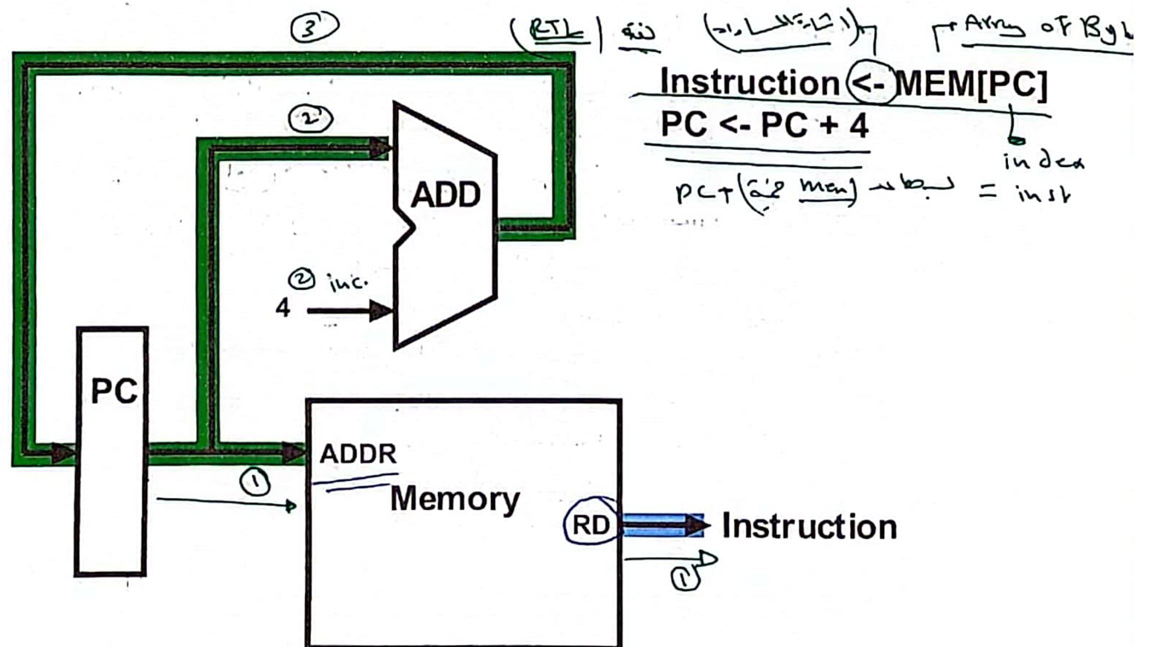
Datapath: Instruction Store/Fetch & PC Increment



Three elements used to store and fetch instructions and increment the PC

Datapath

Animating the Datapath

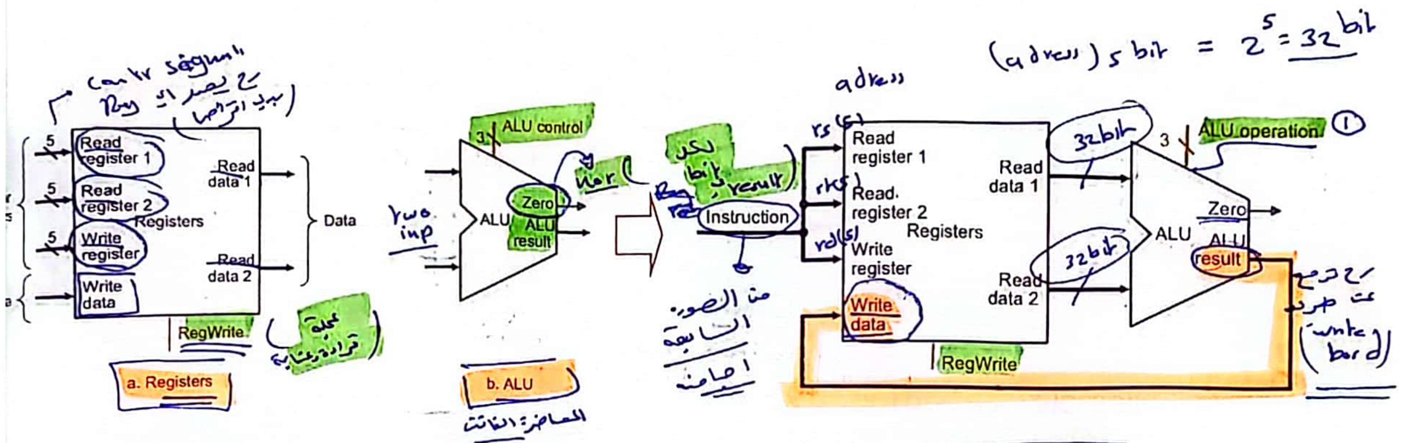


Instruction <- MEM[PC]

PC <- PC + 4

المسوحة ضوئيا بـ CamScanner

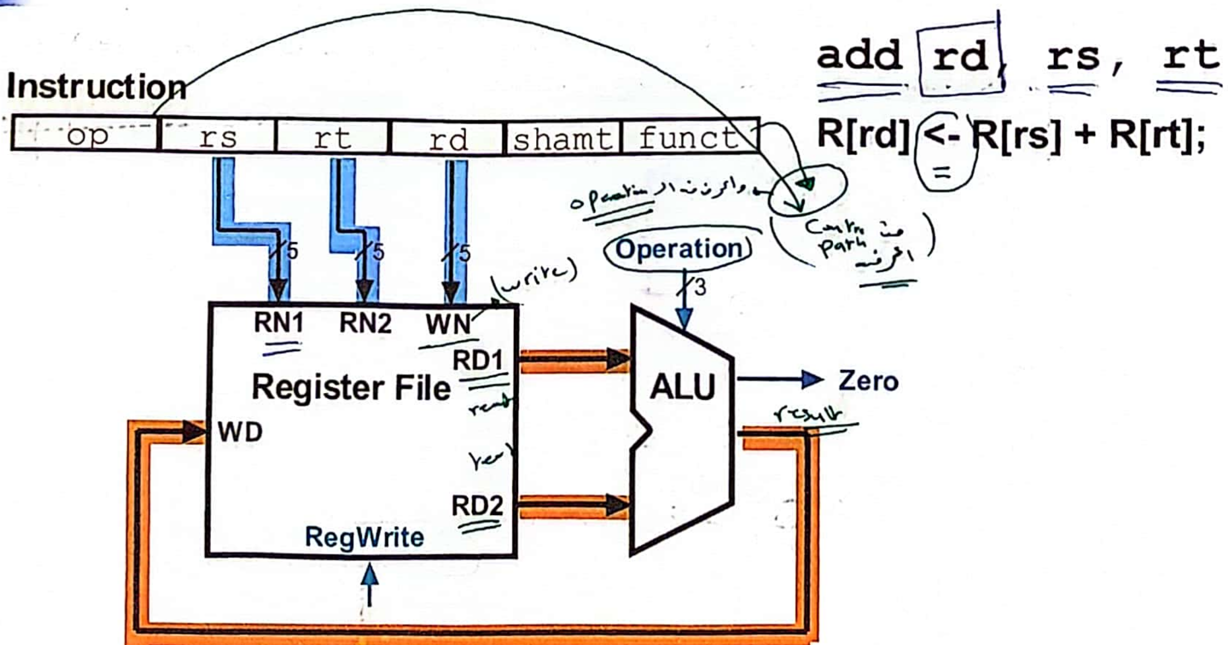
Datapath: R-Type Instruction



Two elements used to implement R-type instructions

Datapath

Animating the Datapath

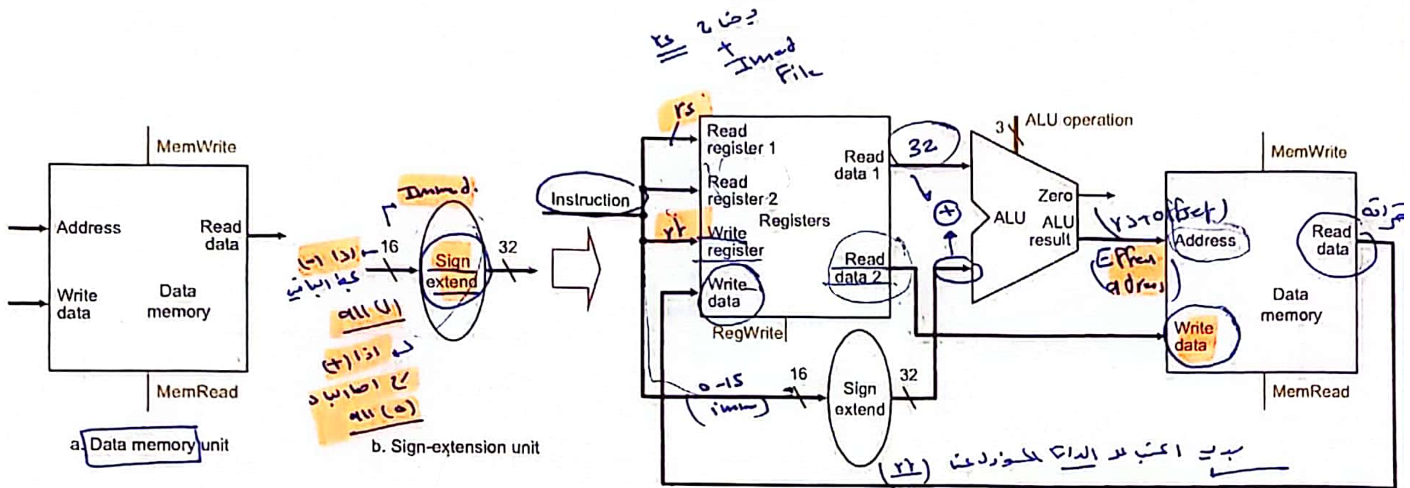


add rd, rs, rt

R[rd] <- R[rs] + R[rt];

الممسوحة ضوئيا بـ CamScanner

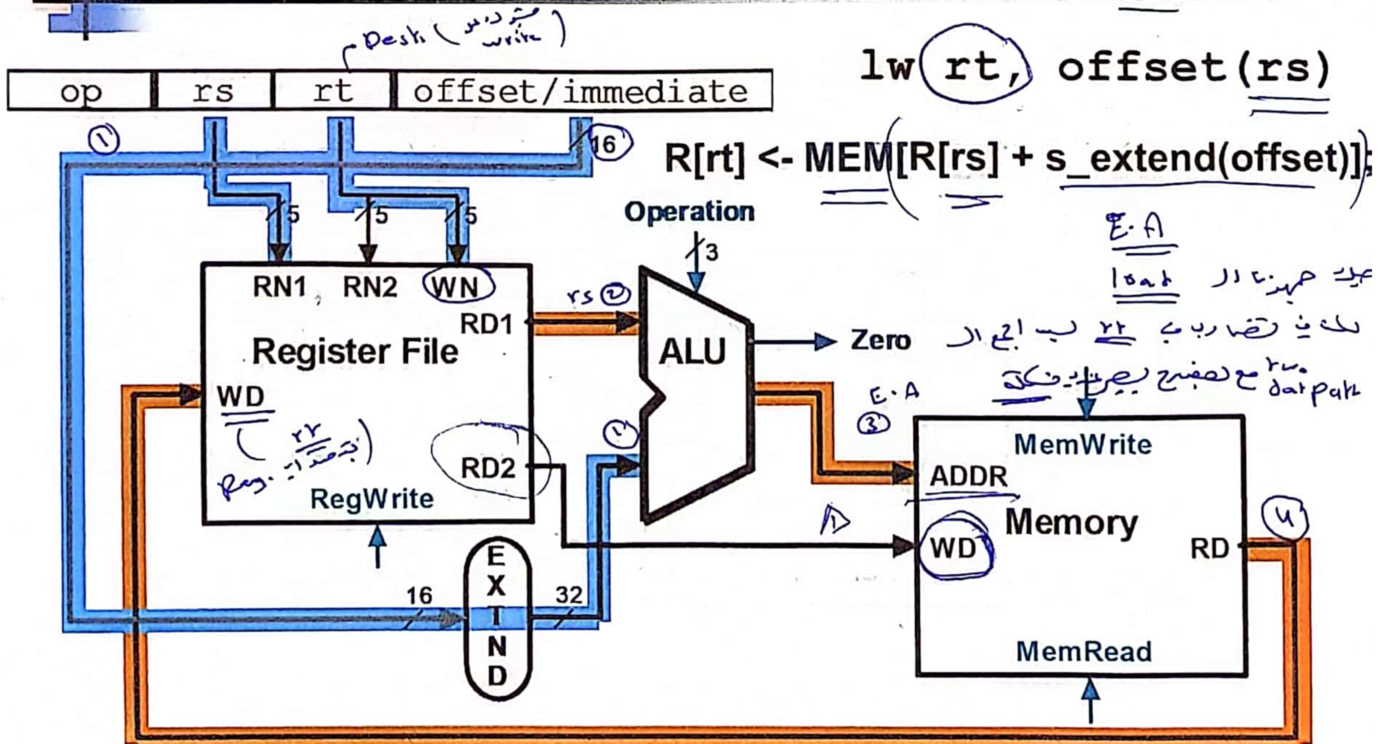
Datapath: Load/Store Instruction



Two additional elements used To implement load/stores

Datapath

Animating the Datapath



CamScanner المسوحة ضوئيا بـ

Animating the Datapath

sw rt, offset(rs)

MEM[R[rs] + sign_extend(offset)] <- R[rt]

Datapath: Branch Instruction

Datapath

CamScanner الممسوحة ضوئيا بـ

Animating the Datapath

beq rs, rt, offset

if (R[rs] == R[rt]) then
PC <- PC+4 + s_extend(offset<<2

MIPS Datapath I: Single-Cycle

Combining the datapaths for R-type instructions and load/stores using two multiplexors

CamScanner المسوحة ضوئيا بـ

Animating the Datapath: R-type Instruction

`add rd,rs,rt`

Instruction

32 16

RN1 RN2 WN

Register File

RD1

WD

RD2

RegWrite

Operation

3

ALU

Zero

MUX

ALUSrc

16 EXTND 32

MemWrite

ADDR

Data Memory

RD

WD

MemRead

MemtoReg

MUX

Animating the Datapath: Load Instruction

`lw rt,offset(rs)`

Instruction

32 16

5 5 5

RN1 RN2 WN

Register File

RD1

WD

RD2

RegWrite

Operation

3

ALU

Zero

MUX

ALUSrc

16 EXTND 32

MemWrite

ADDR

Data Memory

RD

WD

MemRead

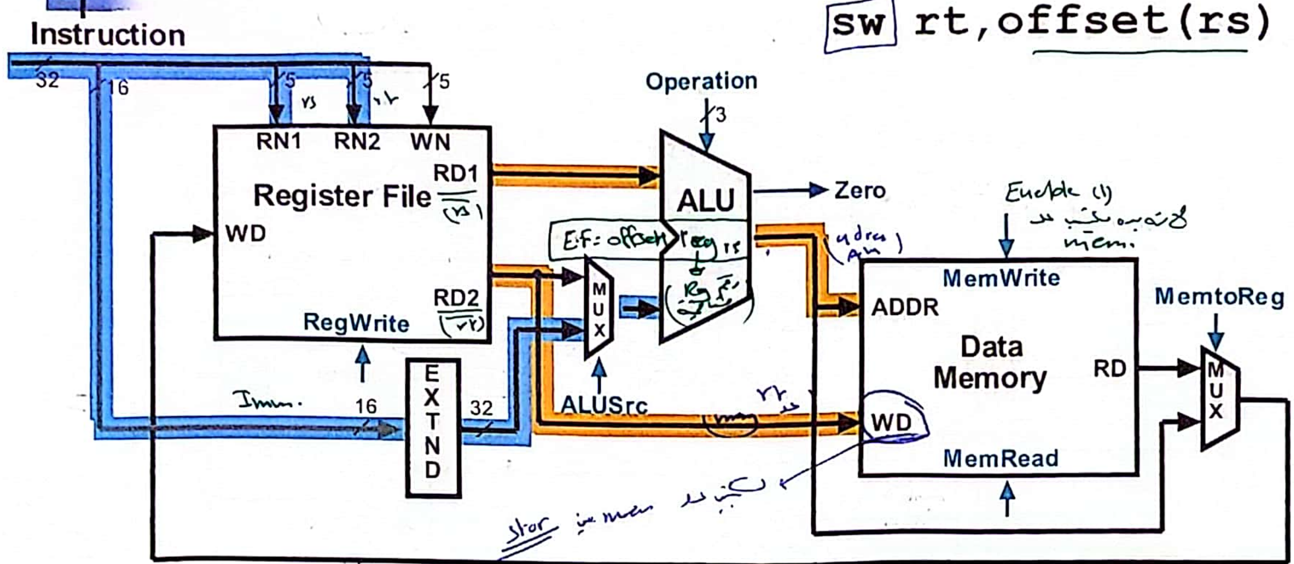
MemtoReg

MUX

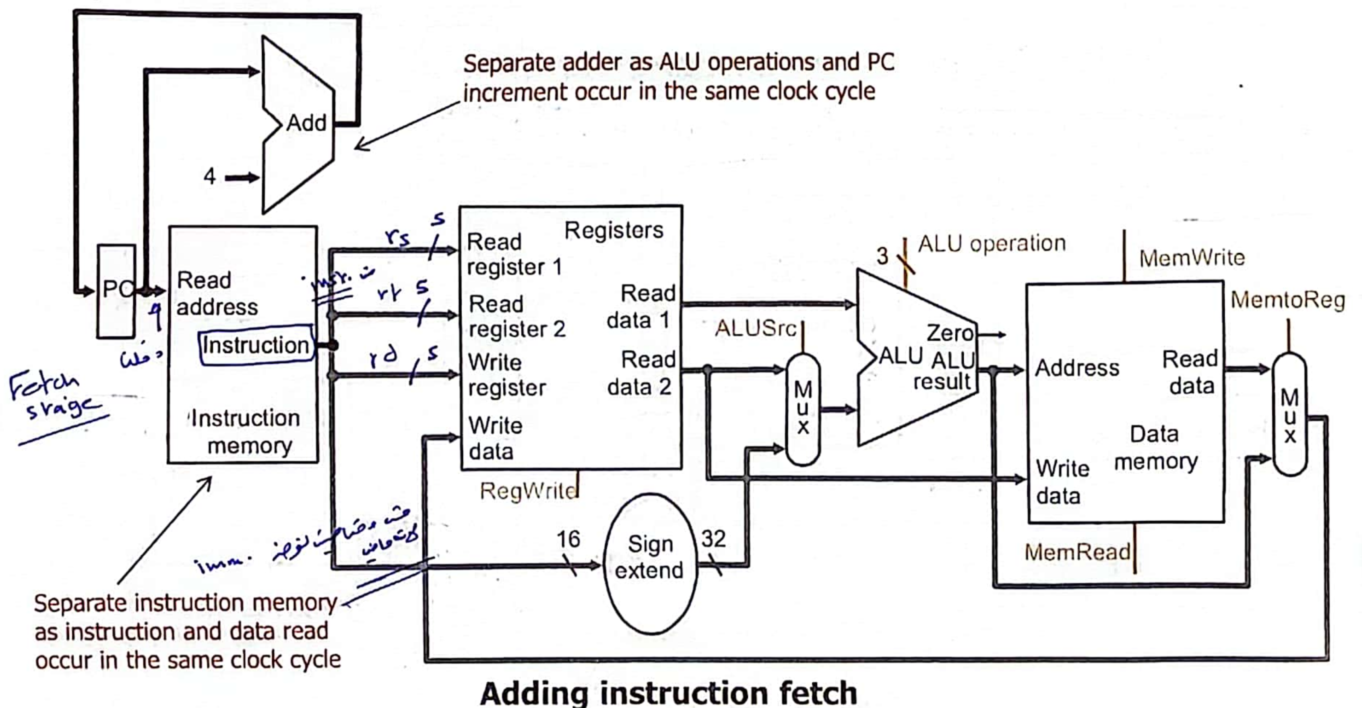
الممسوحة ضوئيا بـ CamScanner

Animating the Datapath: Store Instruction

sw rt,offset(rs)



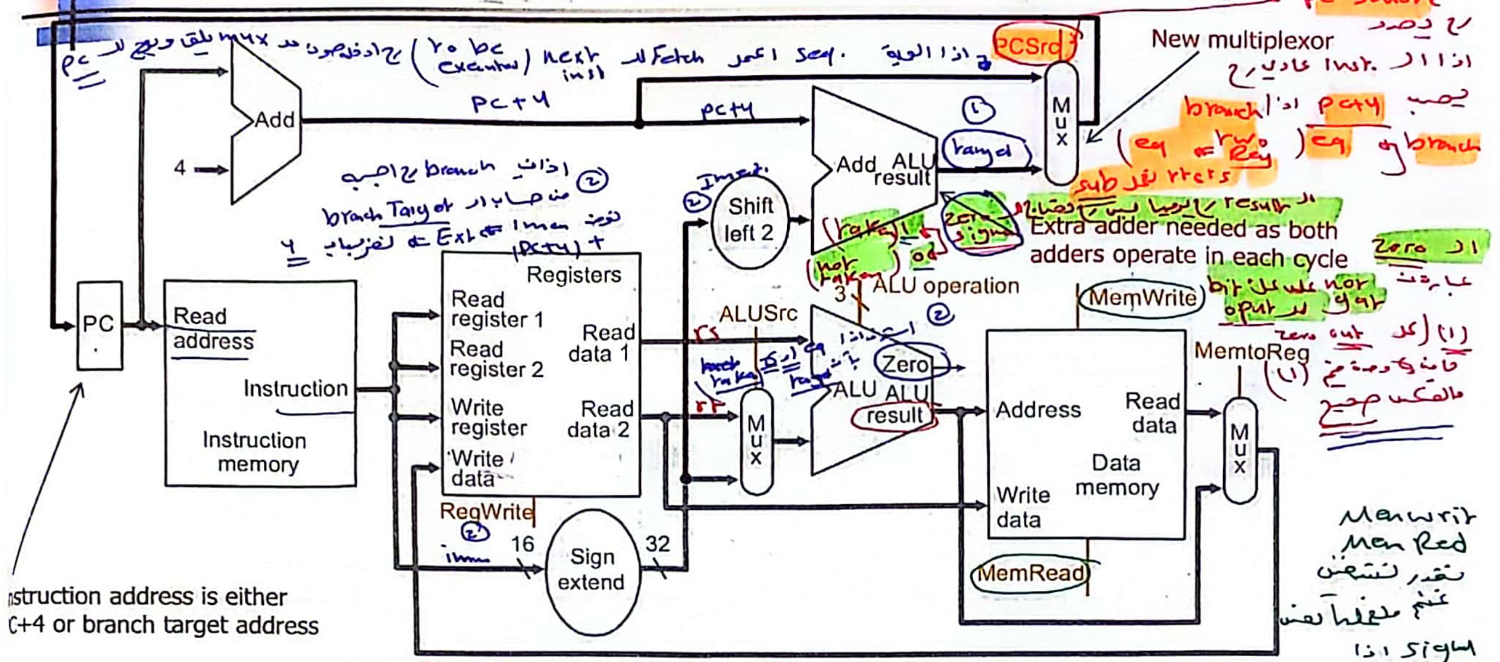
MIPS Datapath II: Single-Cycle



Adding instruction fetch

CamScanner الممسوحة ضوئيا بـ

MIPS Datapath III: Single-Cycle

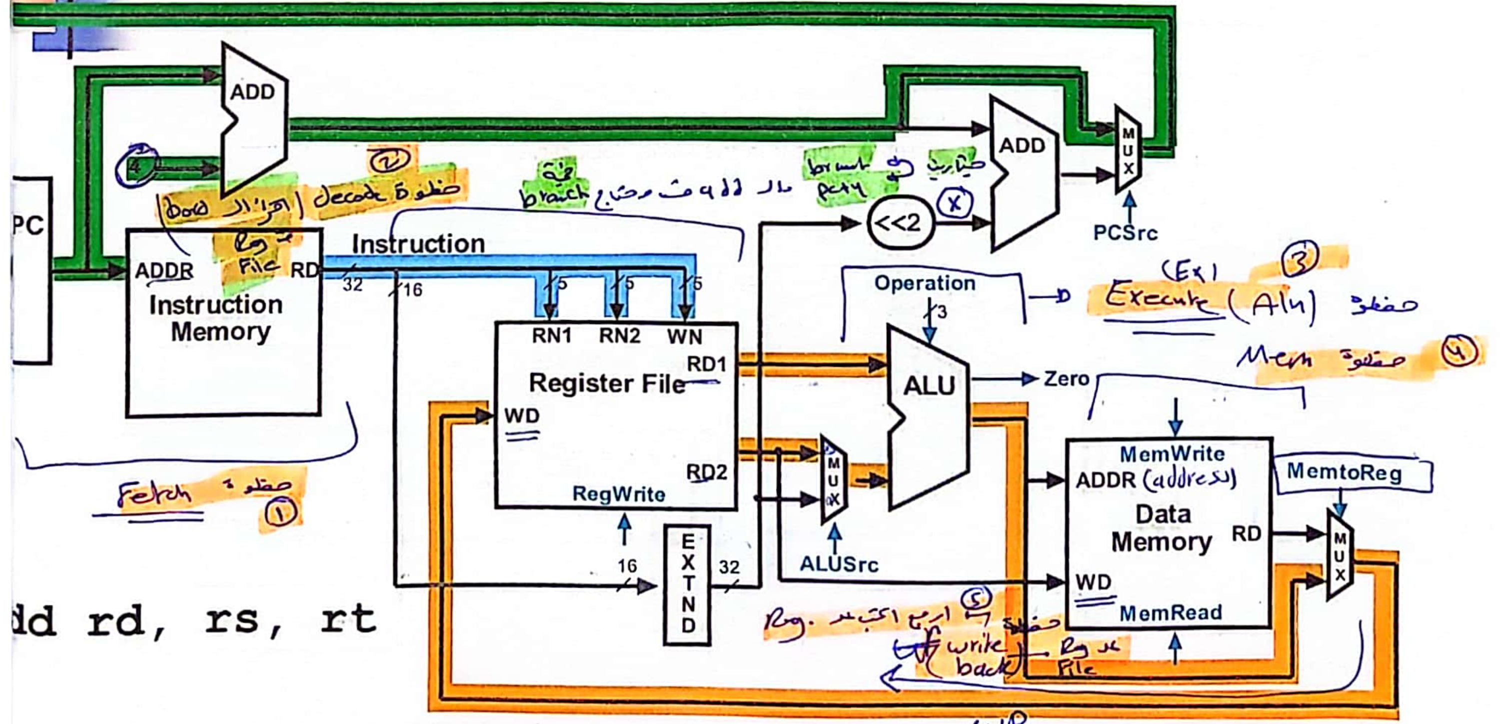


Adding branch capability and another multiplexor

Important note: in a single-cycle implementation data cannot be stored during an instruction – it only moves through combinational logic

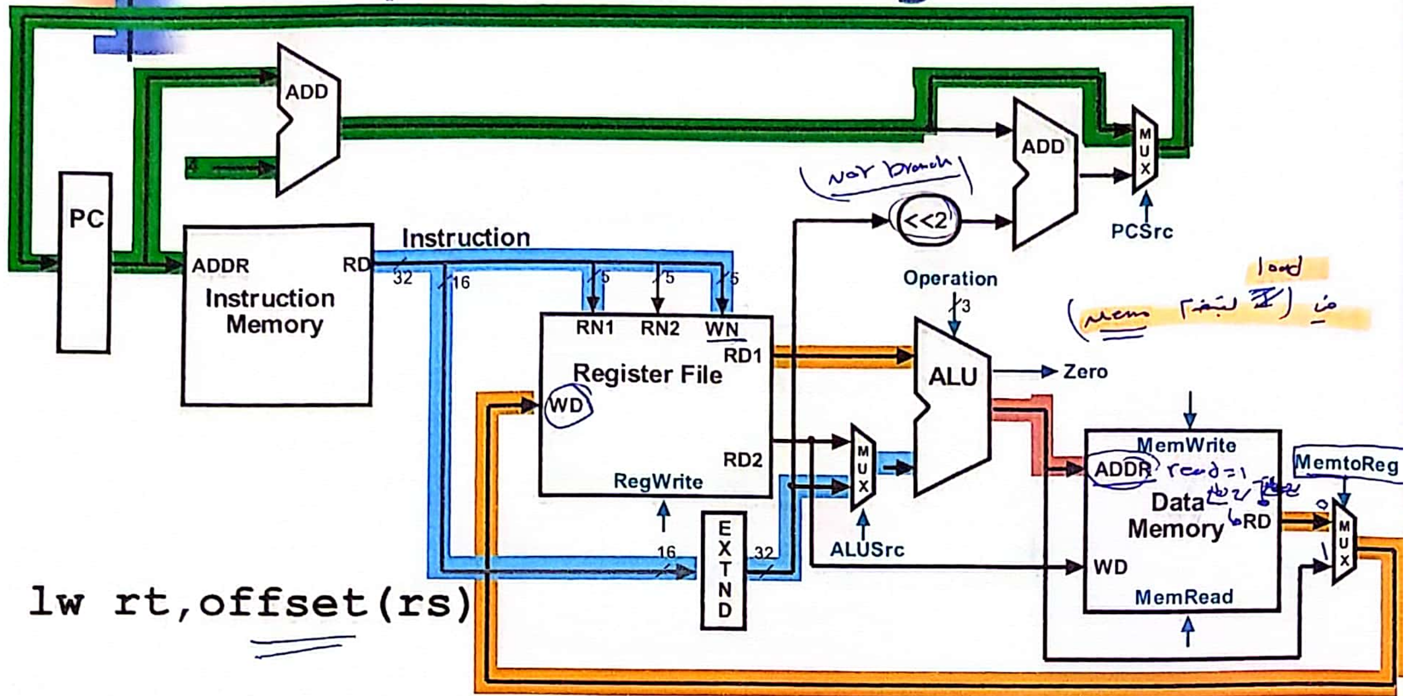
Question: is the MemRead signal really needed?! Think of RegWrite...!

Datapath Executing add



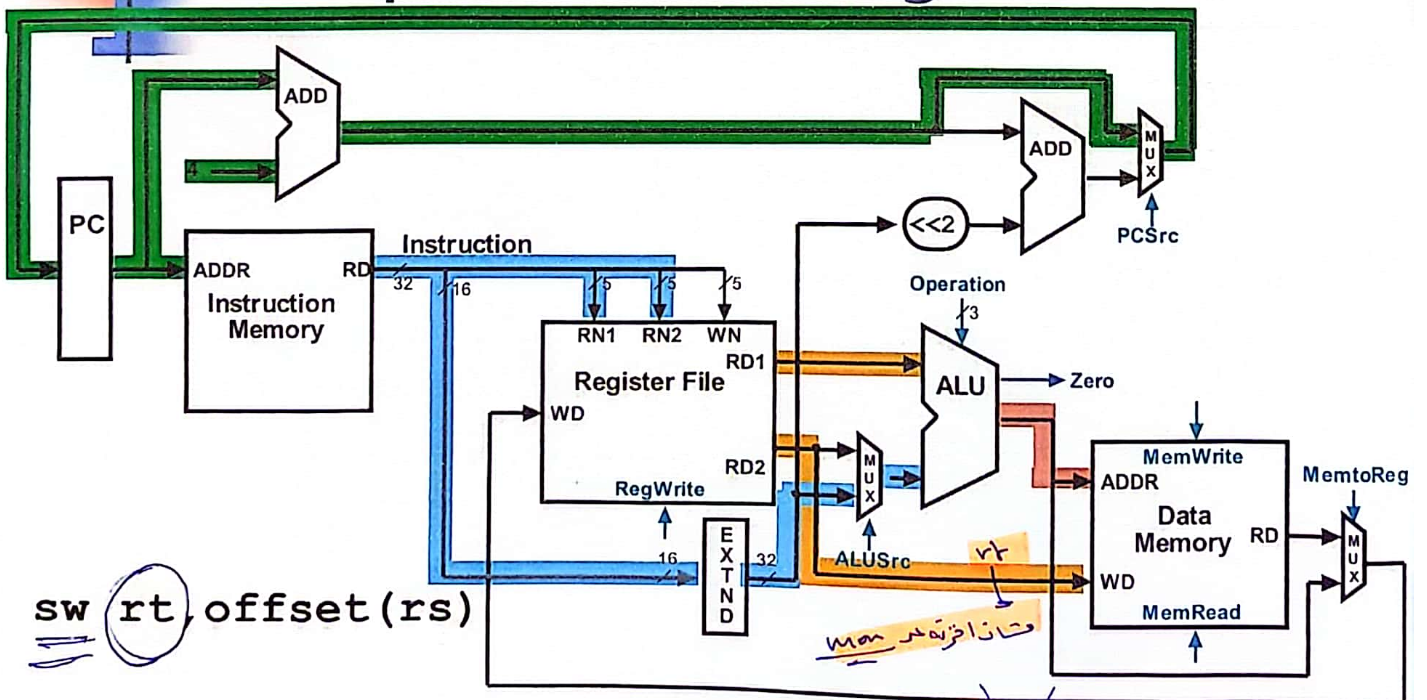
الممسوحة ضوئيا بـ CamScanner

Datapath Executing lw



lw rt,offset(rs)

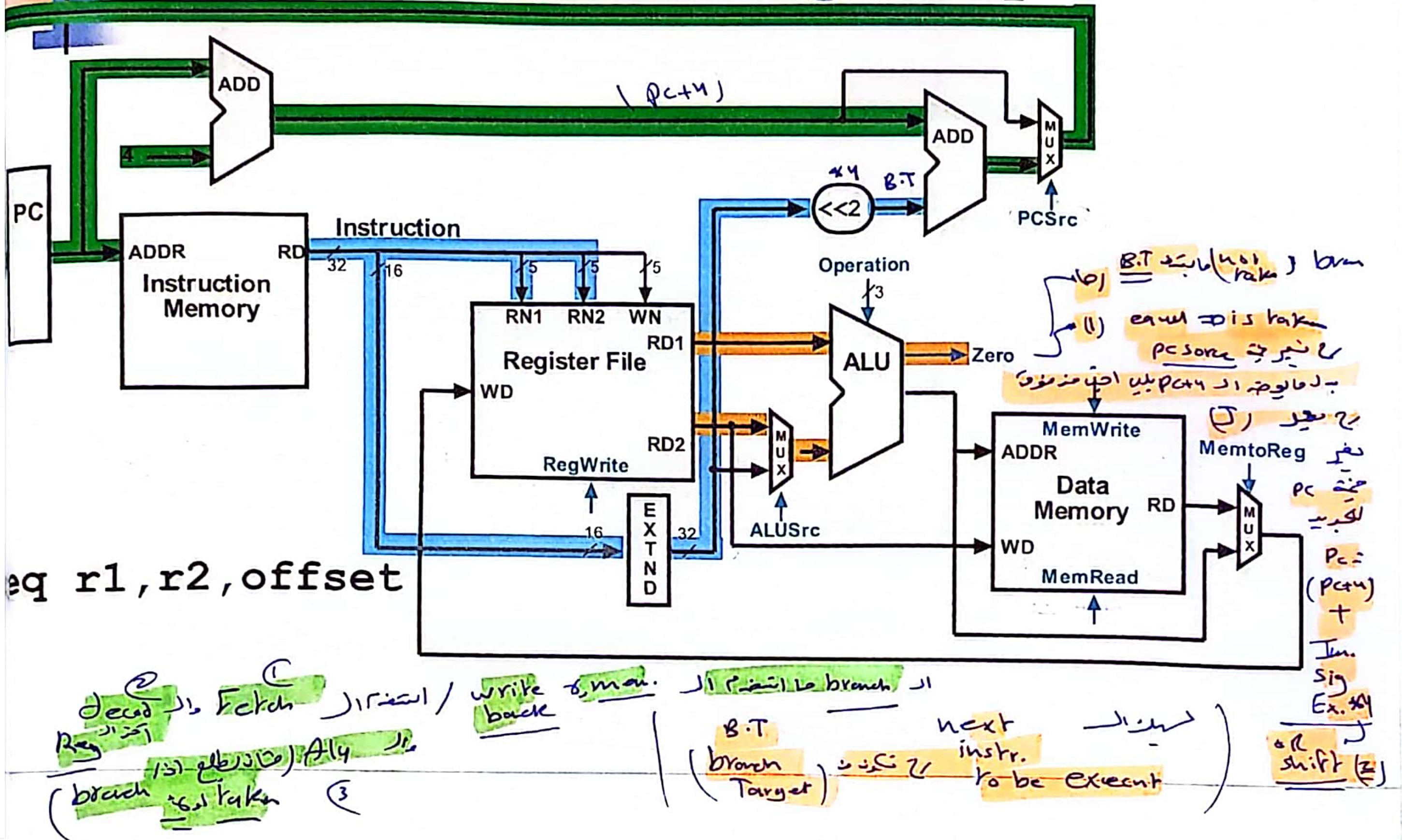
Datapath Executing sw



sw rt,offset(rs)

الممسوحة ضوئيا بـ CamScanner

Datapath Executing beq



Control

- Control unit takes input from
  - the instruction opcode bits
- Control unit generates
  - ALU control input
  - write enable (possibly, read enable also) signals for each storage element
  - selector controls for each multiplexor

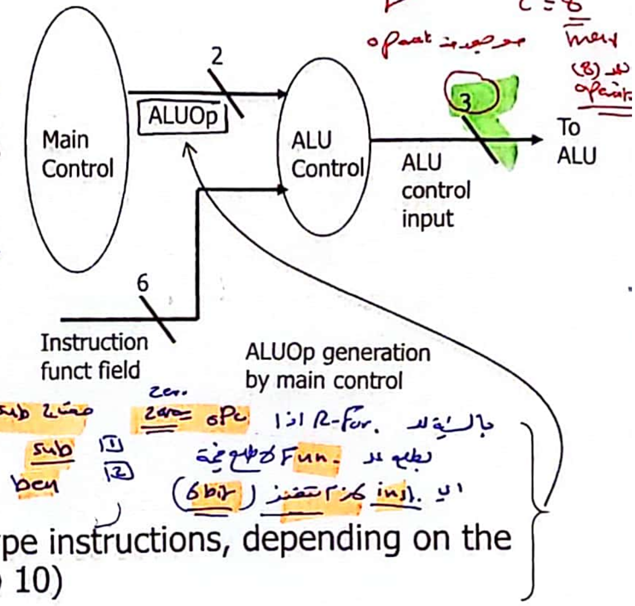
CamScanner المسوحة ضوئيا بـ

ALU Control

- Plan to control ALU: main control sends a 2-bit ALUOp control field to the ALU control. Based on ALUOp and funct field of instruction the ALU control generates the 3-bit ALU control field

Recall from Ch. 4

ALU control field	Function
000	and
001	or
010	add
110	sub
111	slt



- ALU must perform
  - *add* for load/stores (ALUOp 00)
  - *sub* for branches (ALUOp 01)
  - one of *and*, *or*, *add*, *sub*, *slt* for R-type instructions, depending on the instruction's 6-bit funct field (ALUOp 10)

Setting ALU Control Bits

Instruction opcode	AluOp	Instruction operation	Funct Field	Desired ALU action	ALU control input
LW	00	load word	xxxxxx	add	010
SW	00	store word	xxxxxx	add	010
Branch eq	01	branch eq	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less	101010	set on less	111

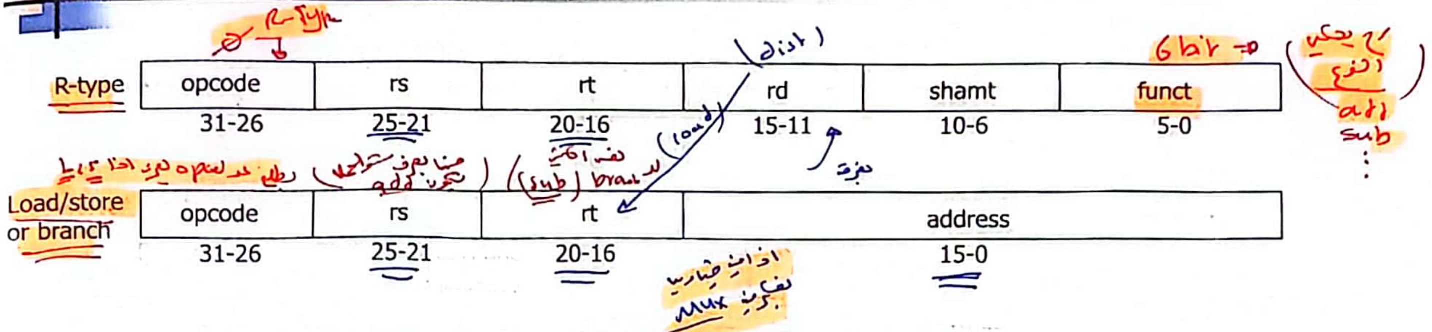
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0*	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Truth table for ALU control bits

*Typo in text Fig. 5.15: if it is X then there is potential conflict between line 2 and lines 3-7!

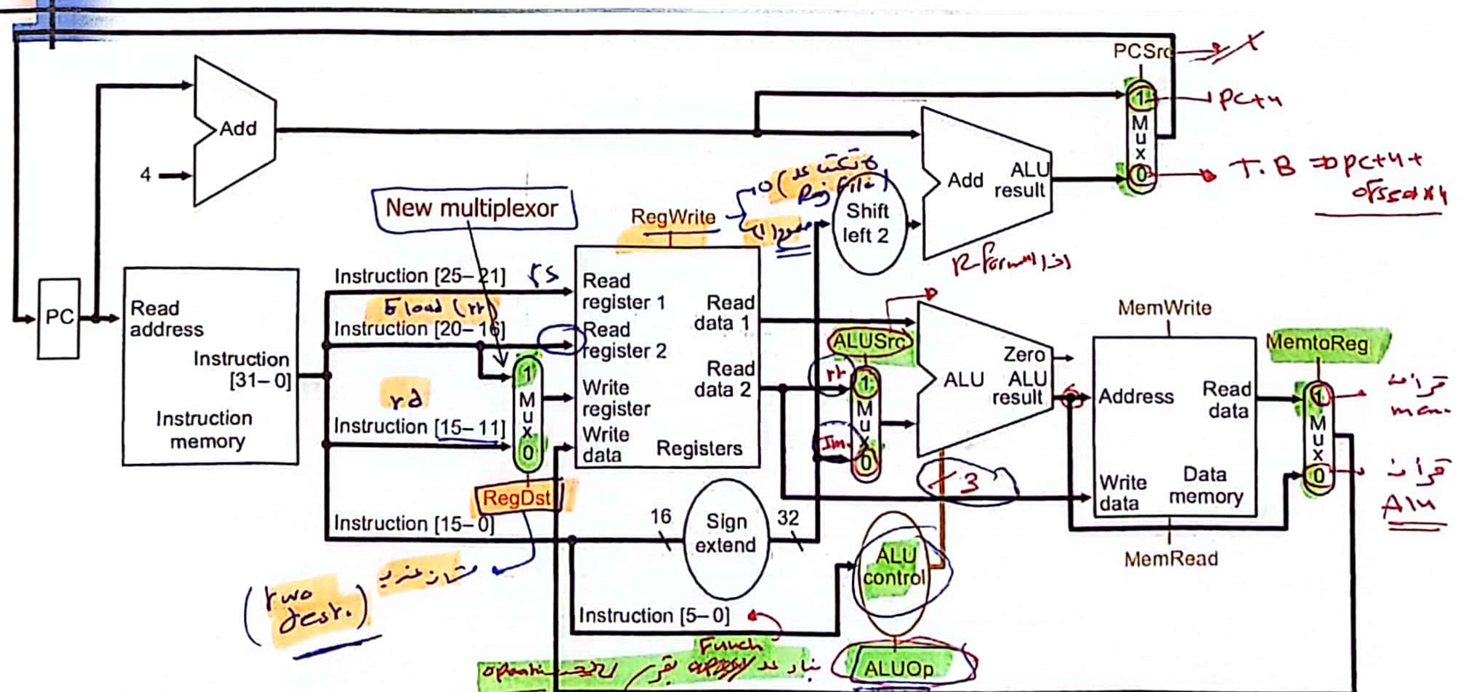
CamScanner الممسوحة ضوئيا بـ

Designing the Main Control



- Observations about MIPS instruction format
  - opcode is always in bits 31-26
  - two registers to be read are always rs (bits 25-21) and rt (bits 20-16)
  - base register for load/stores is always rs (bits 25-21)
  - 16-bit offset for branch equal and load/store is always bits 15-0
  - destination register for loads is in bits 20-16 (rt) while for R-type instructions it is in bits 15-11 (rd) (*will require multiplexor to select*)

Datapath with Control I



Adding control to the MIPS Datapath III (and a new multiplexor to select field to specify destination register): what are the functions of the 9 control signals?

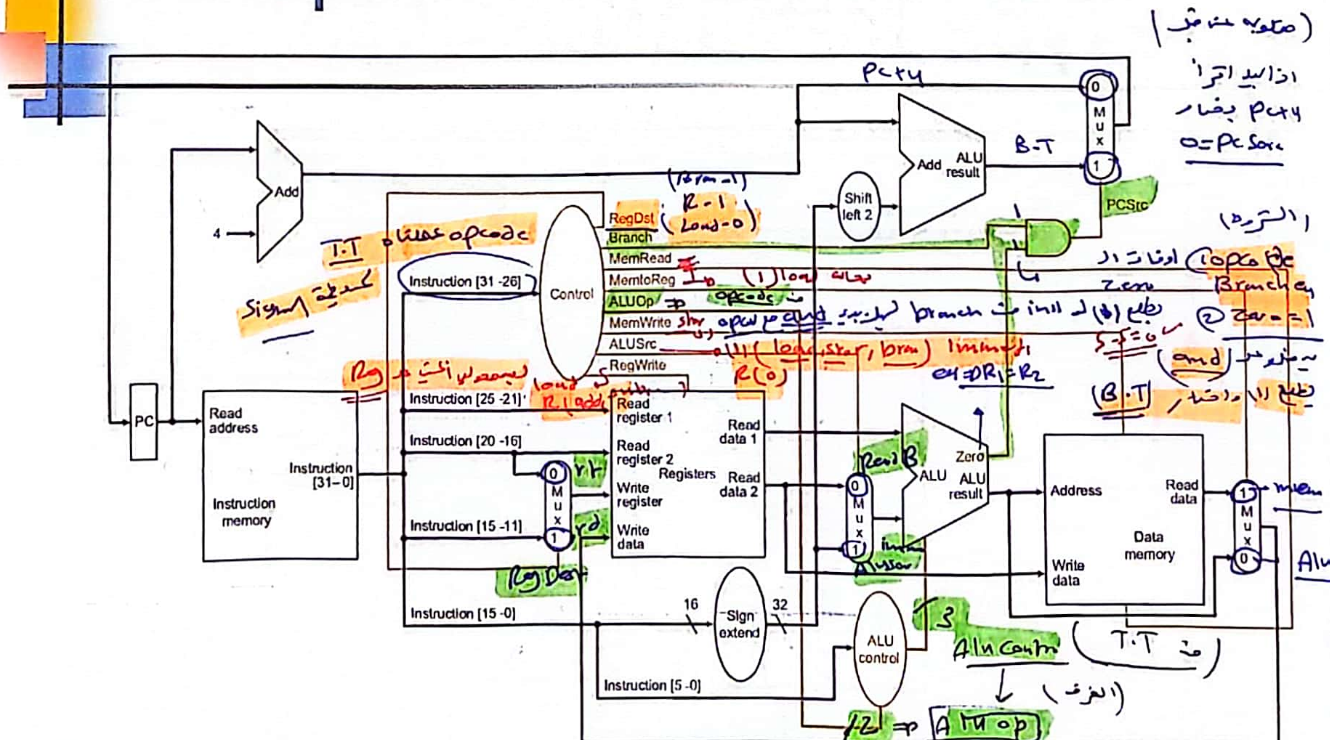
CamScanner الممسوحة ضوئيا بـ

Control Signals

Signal Name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20-16)	The register destination number for the Write register comes from the rd field (bits 15-11)
RegWrite	None	The register on the Write register input is written with the value on the Write data input
AlLUSrc	The second ALU operand comes from the second register file output (Read data 2)	The second ALU operand is the sign-extended, lower 16 bits of the instruction
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4	The PC is replaced by the output of the adder that computes the branch target
MemRead	None	Data memory contents designated by the address input are put on the first Read data output
MemWrite	None	Data memory contents designated by the address input are replaced by the value of the Write data input
MemtoReg	The value fed to the register Write data input comes from the ALU	The value fed to the register Write data input comes from the data memory

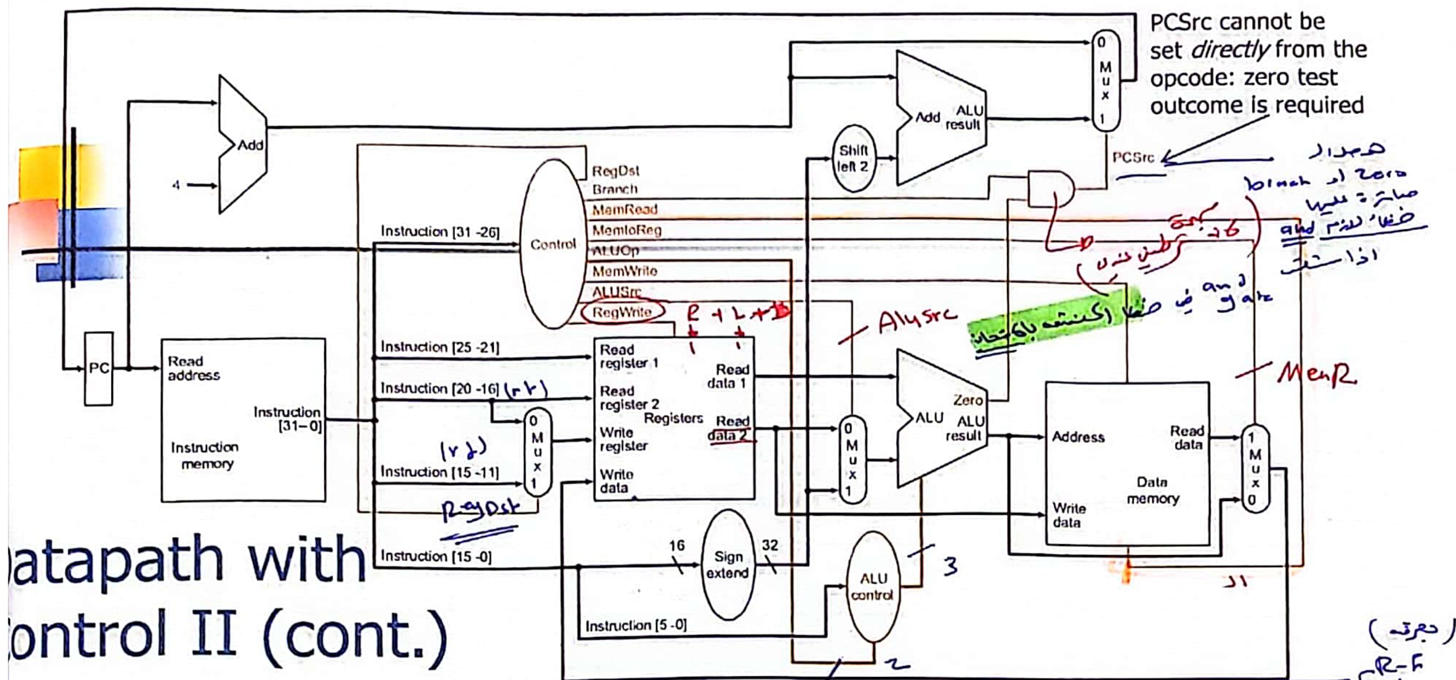
Effects of the seven control signals

Datapath with Control II



MIPS datapath with the control unit: input to control is the 6-bit instruction opcode field, output is seven 1-bit signals and the 2-bit ALUOp signal

CamScanner الممسوحة ضوئيا بـ

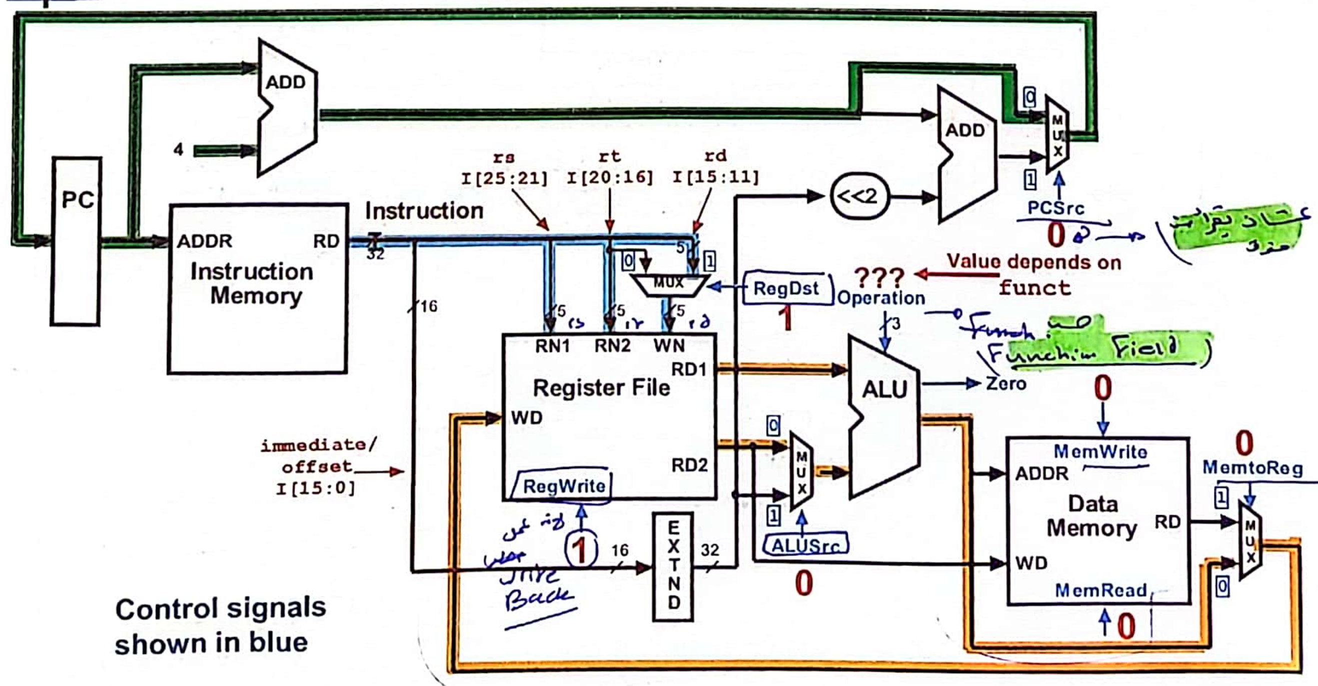


Datapath with Control II (cont.)

Determining control signals for the MIPS datapath based on instruction opcode

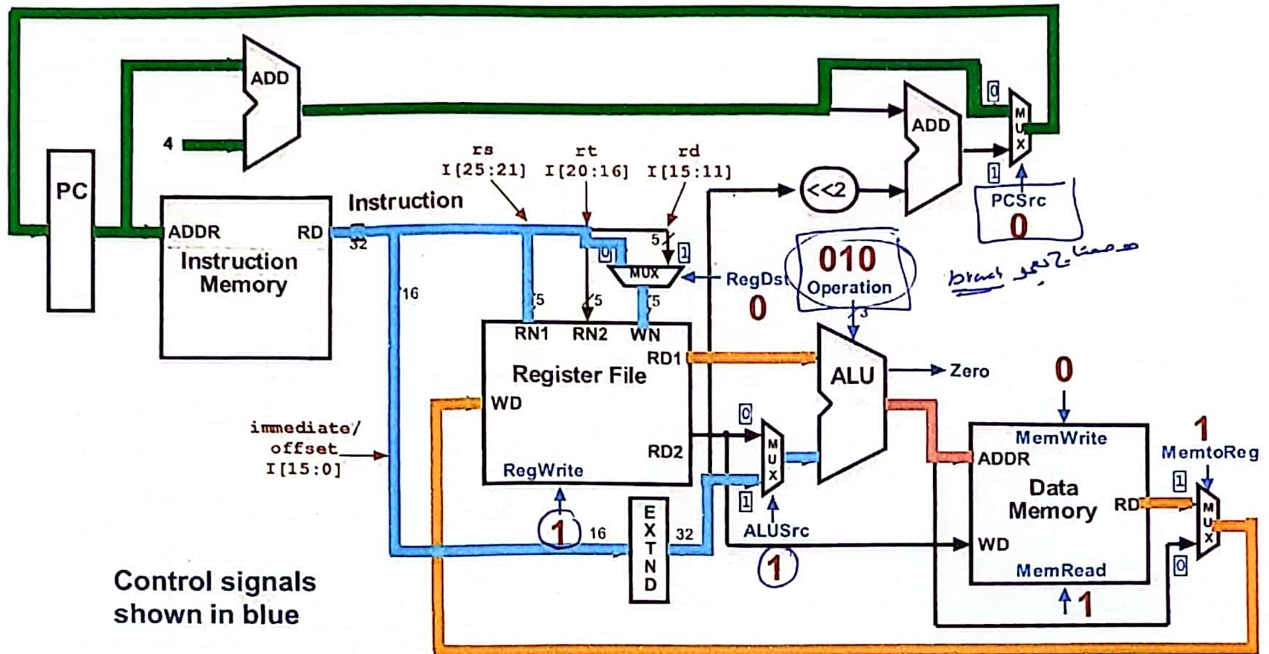
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Control Signals: R-Type Instruction

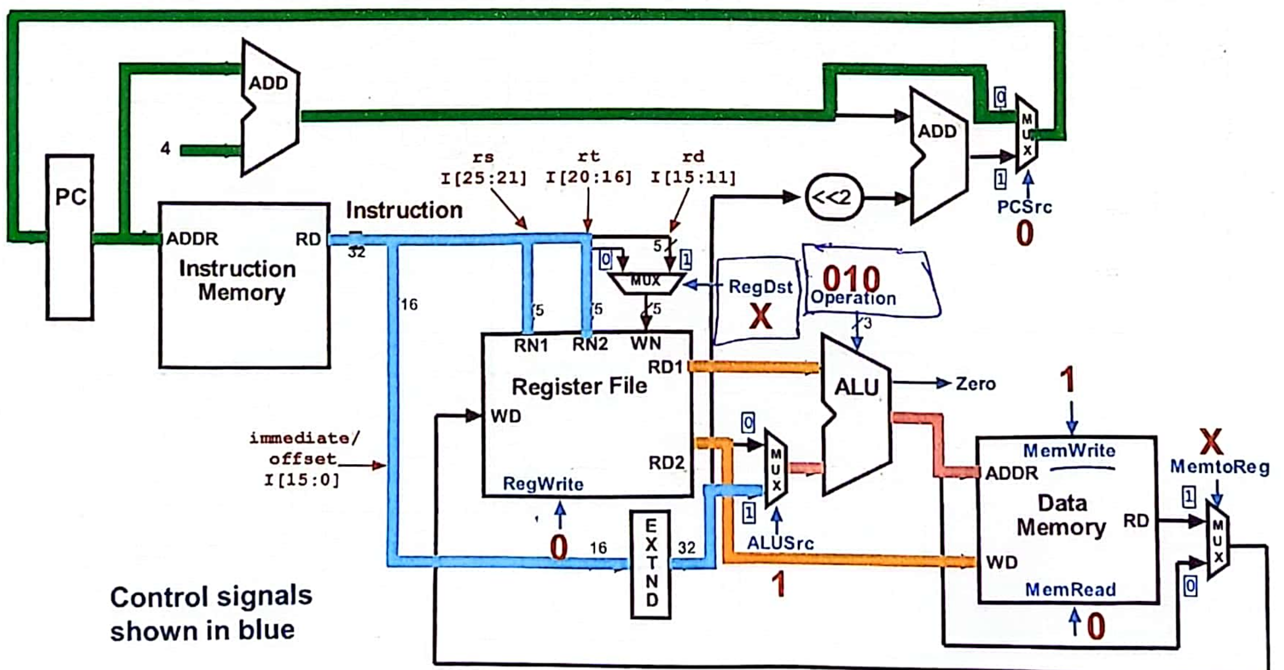


الممسوحة ضوئيا بـ CamScanner

Control Signals: lw Instruction



Control Signals: sw Instruction



CamScanner المسوحة ضوئيا بـ

Control Signals: beq Instruction

PC
ADD
4
ADD
<<2
rs I[25:21]
rt I[20:16]
rd I[15:11]
Instruction
ADDR RD 32
Instruction Memory
16
0 MUX 1
RegDst X
5 5 5
RN1 RN2 WN
RD1
Register File
WD
RD2
RegWrite
0
immediate/ offset I[15:0]
16 EXTND 32
110 Operation
3
ALU
Zero
0 MUX 1
ALUSrc
0
0 MUX 1
PCSrc
1 if Zero=1
0
MemWrite
ADDR
Data Memory
RD
WD
MemRead
0
X
MemtoReg
1 MUX 0

Control signals shown in blue

Datapath with Control III

Jump	opcode	address
	31-26	25-0

Composing jump target address

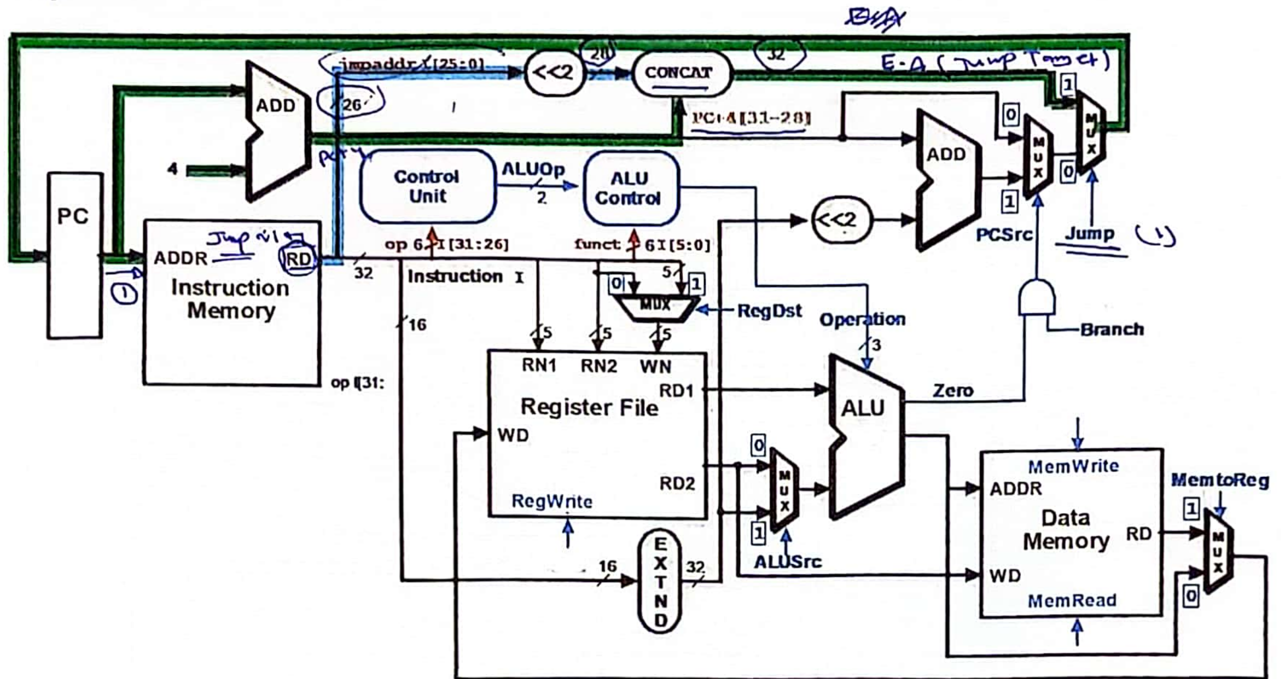
New multiplexor with additional control bit Jump

Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [31–26]
Control
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Shift left 2
Add ALU result
0 Mux 1
1 Mux 0
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
0 Mux 1
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
0 Mux 1
ALU
Zero
ALU result
Address
Read data
Write data
Data memory
1 Mux 0
Instruction [15–0]
16
Sign extend
32
ALU control
Instruction [5–0]

MIPS datapath extended to jumps: control unit generates new Jump control bit

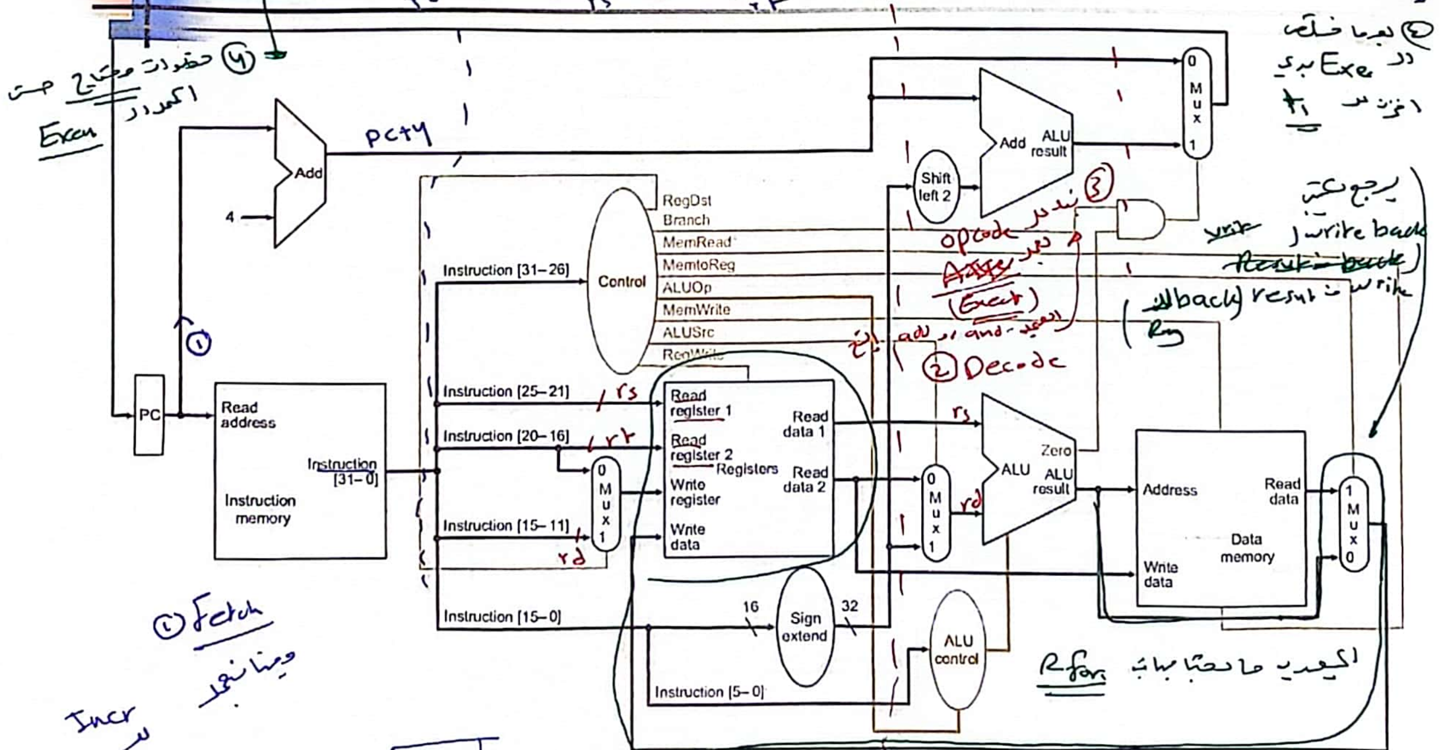
CamScanner الممسوحة ضوئيا بـ

Datapath Executing j



R-type Instruction: Step 1

add $t1, $t2, $t3 (active = bold)



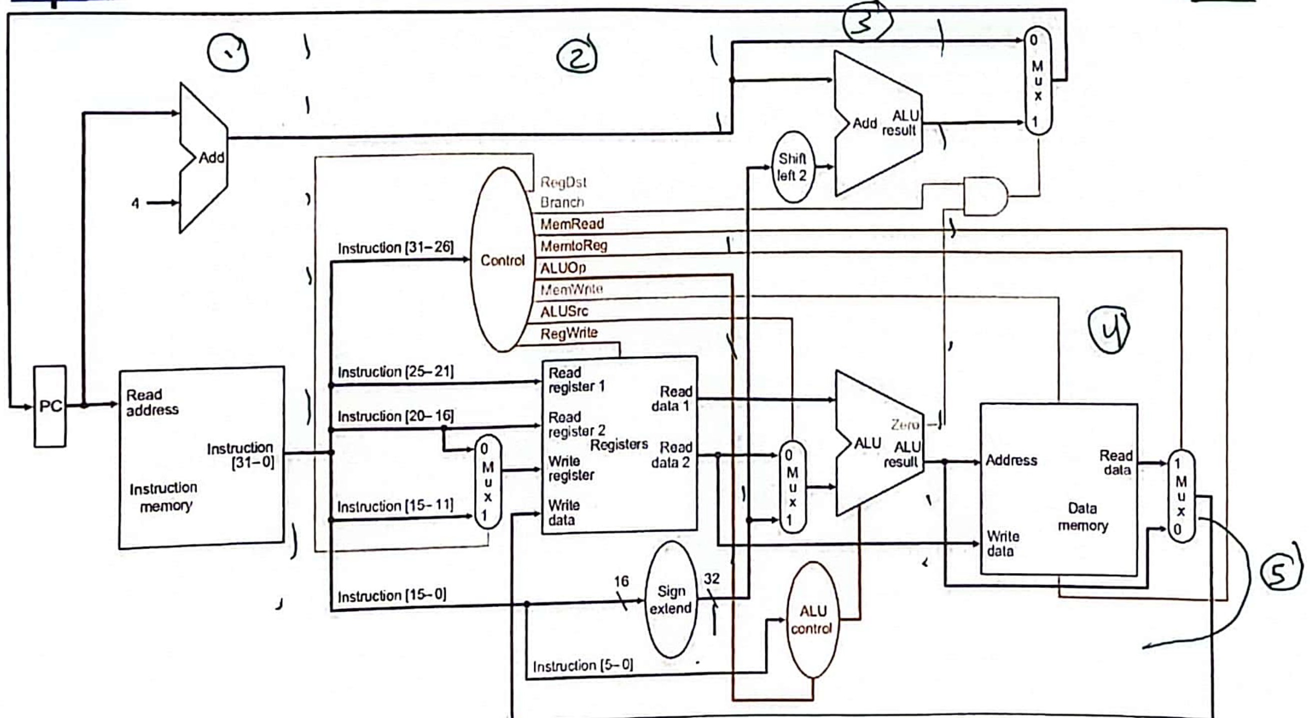
Fetch instruction and increment PC count

الممسوحة ضوئيا بـ CamScanner

Load Instruction Steps
lw $t1, offset($t2)

1. Fetch instruction and increment PC
2. Read base register from the register file: the base register ($t2) is given by bits 25-21 of the instruction
3. ALU computes sum of value read from the register file and the sign-extended lower 16 bits (offset) of the instruction
4. The sum from the ALU is used as the address for the data memory
5. The data from the memory unit is written into the register file: the destination register ($t1) is given by bits 20-16 of the instruction

Load Instruction
lw $t1, offset($t2)

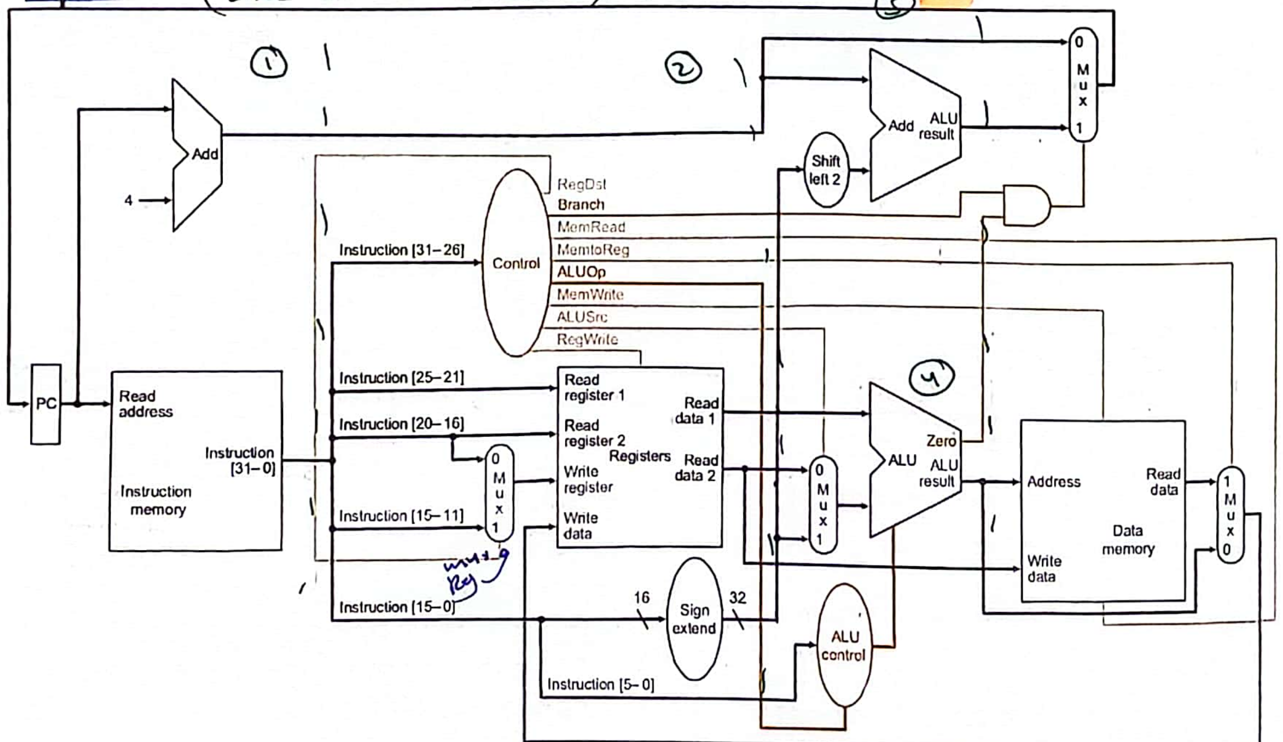


CamScanner الممسوحة ضوئيا بـ

Branch Instruction Steps beq $t1, $t2, offset

1. Fetch instruction and increment PC
2. Read two register ($t1 and $t2) from the register file
3. ALU performs a subtract on the data values from the register file; the value of PC+4 is added to the sign-extended lower 16 bits (offset) of the instruction shifted left by two to give the branch target address
4. The Zero result from the ALU is used to decide which adder result (from step 1 or 3) to store in the PC

Branch Instruction beq $t1, $t2, offset

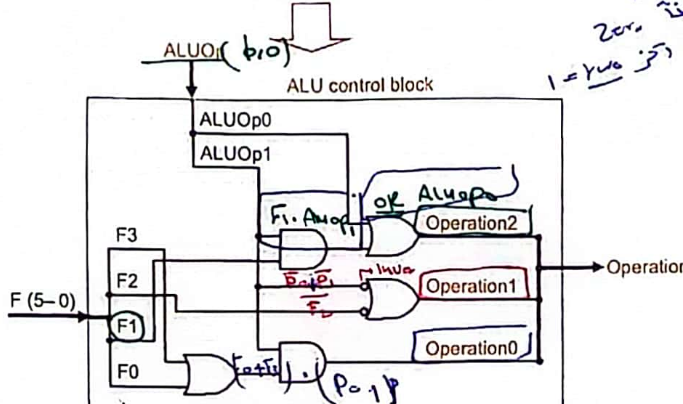


mplementation: ALU Control Block

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0*	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Truth table for ALU control bits

*Typo in text Fig. 5.15: if it is X then there is potential conflict between line 2 and lines 3-7!

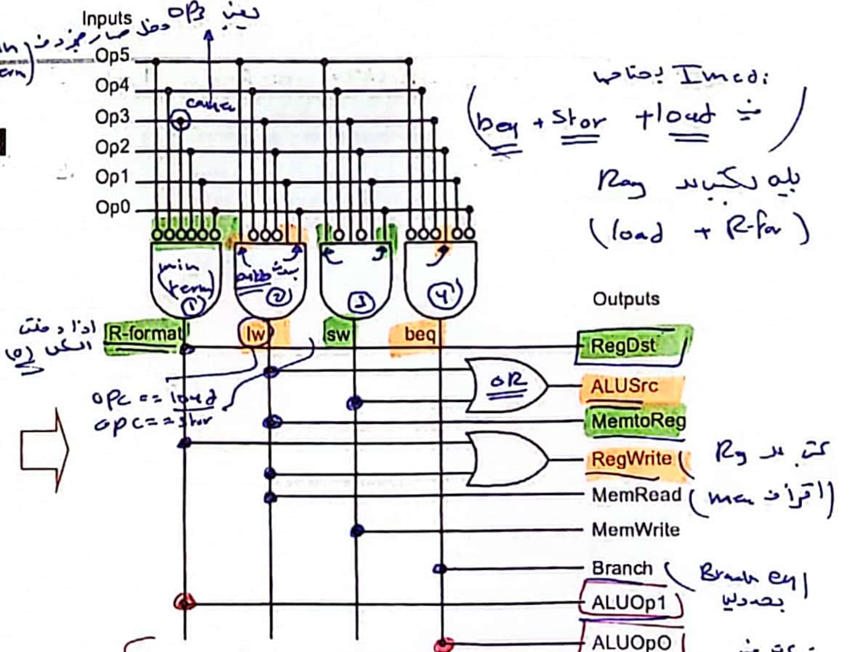


ALU control logic

Implementation: Main Control Block

Signal name	R-format	lw	sw	beq
Op5	0	1	1	0
Op4	0	0	0	0
Op3	0	0	1	0
Op2	0	0	0	1
Op1	0	1	1	0
Op0	0	1	1	0
RegDst	1	0	x	x
ALUSrc	0	1	1	0
MemtoReg	0	1	x	x
RegWrite	1	1	0	0
MemRead	0	1	0	0
MemWrite	0	0	1	0
Branch	0	0	0	1
ALUOp1	1	0	0	0
ALUOP2	0	0	0	1

Truth table for main control signals



Main control PLA (programmable logic array): principle underlying PLAs is that any logical expression can be written as a sum-of-products

المسوحة ضوئيا بـ CamScanner

Single-Cycle Design Problems

- Assuming fixed-period clock every instruction datapath uses one clock cycle implies:
  - CPI = 1
  - cycle time determined by length of the longest instruction path (load)
    - but several instructions could run in a shorter clock cycle: *waste of time*
    - consider if we have more complicated instructions like floating point!
  - resources used more than once in the same cycle need to be duplicated
    - *waste of hardware and chip area*

Example: Fixed-period clock vs. variable-period clock in a single-cycle implementation

- Consider a machine with an additional floating point unit. Assume functional unit delays as follows
  - *memory*: 2 ns., *ALU* and *adders*: 2 ns., *FPU add*: 8 ns., *FPU multiply*: 16 ns., *register file access* (read or write): 1 ns.
  - *multiplexors, control unit, PC accesses, sign extension, wires*: no delay
- Assume instruction mix as follows
  - all loads take same time and comprise 31%
  - all stores take same time and comprise 21%
  - R-format instructions comprise 27%
  - branches comprise 5%
  - jumps comprise 2%
  - FP adds and subtracts take the same time and totally comprise 7%
  - FP multiplys and divides take the same time and totally comprise 7%
- *Compare the performance of (a) a single-cycle implementation using a fixed-period clock with (b) one using a variable-period clock where each instruction executes in one clock cycle that is only as long as it needs to be (not really practical but pretend it's possible!)*

CamScanner الممسوحة ضوئيا بـ

Solution

Instruction class	Instr. mem.	Register read	ALU oper.	Data mem.	Register write	FPU add/ sub	FPU mul/ div	Total time ns.
Load word	2	1	2	2	1			8
Store word	2	1	2	2				7
R-format	2	1	2	0	1			6
Branch	2	1	2					5
Jump	2							2
FP mul/div	2	1			1		16	20
FP add/sub	2	1			1	8		12

- Clock period for fixed-period clock = longest instruction time = 20 ns.
- Average clock period for variable-period clock = 8 × 31% + 7 × 21% + 6 × 27% + 5 × 5% + 2 × 2% + 20 × 7% + 12 × 7% = 7.0 ns.
- Therefore, $performance_{var\text{-}period} / performance_{fixed\text{-}period}$ = 20/7 = 2.9

Fixing the problem with single-cycle designs

- One solution: a variable-period clock with different cycle times for each instruction class
  - *unfeasible*, as implementing a variable-speed clock is technically difficult
- Another solution:
  - use a smaller cycle time…
  - …have different instructions take different numbers of cycles by breaking instructions into steps and fitting each step into one cycle
  - *feasible: multicyle approach*!

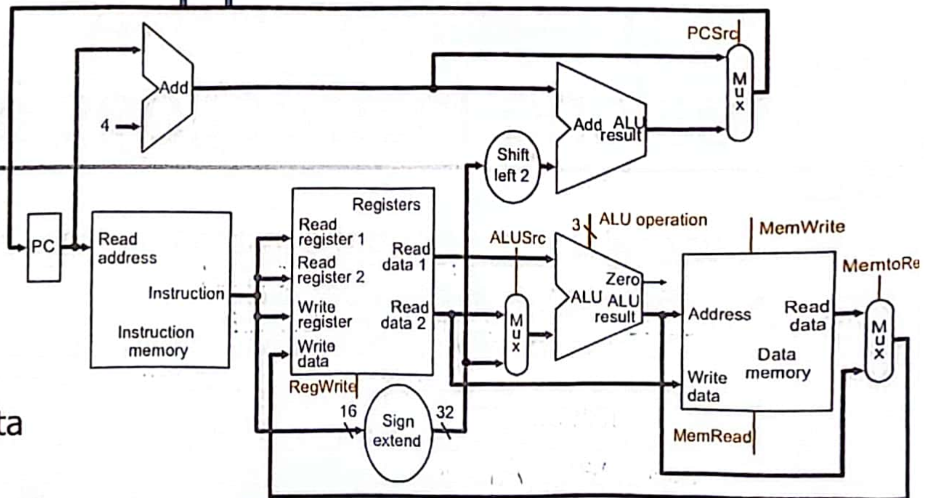
CamScanner المسموحة ضوئيا بـ

Multicycle Approach

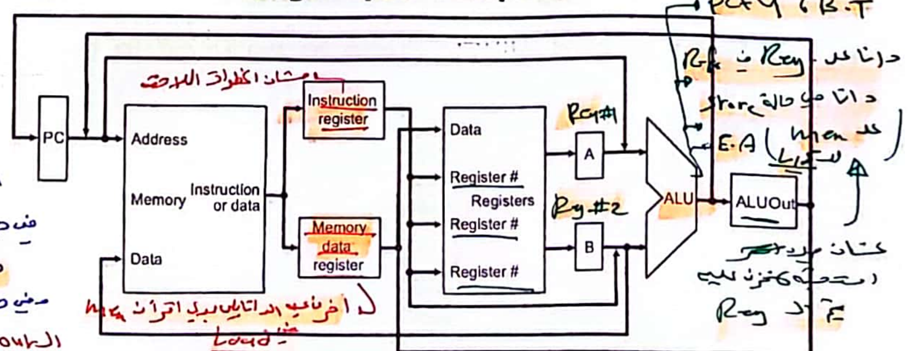
- Break up the instructions into *steps*
  - each step takes one clock cycle
  - balance the amount of work to be done in each step/cycle so that they are about equal
  - restrict each cycle to use at most once each major functional unit so that such units do not have to be replicated
  - functional units can be shared between different cycles within one instruction
- Between steps/cycles
  - At the end of one cycle store data to be used in *later cycles of the same* instruction
    - need to introduce additional *internal* (programmer-invisible) registers for this purpose
  - Data to be used in *later instructions* are stored in programmer-visible state elements: the register file, PC, memory

Multicycle Approach

- Note particularities of multicyle vs. single-diagrams
  - single memory for data and instructions
  - single ALU, no extra adders
  - extra registers to hold data between clock cycles



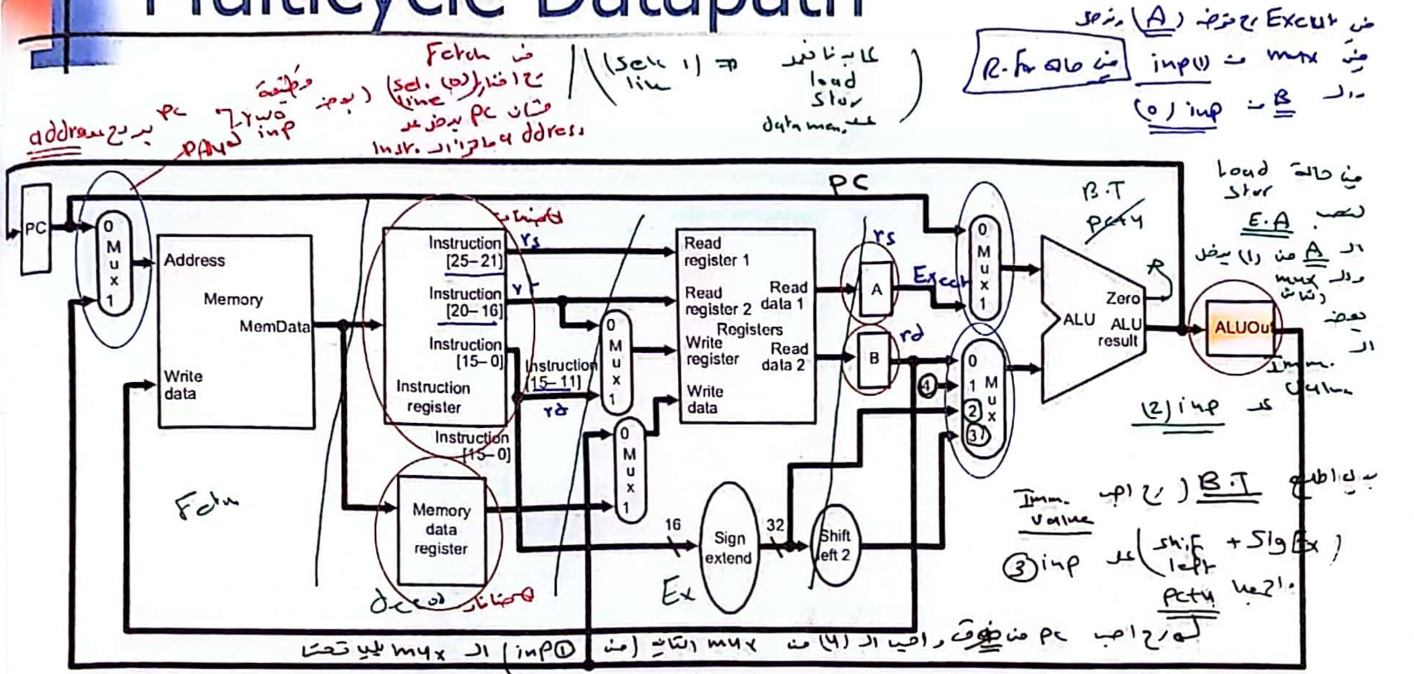
Single-cycle datapath



Multicycle datapath (high-level view)

المسوحة ضوئيا بـ CamScanner

Multicycle Datapath



Basic multicycle MIPS datapath handles R-type instructions and load/stores: new internal register in red ovals, new multiplexors in blue ovals

Breaking instructions into steps

- Our goal is to break up the instructions into *steps* so that
  - each step takes one clock cycle
  - the amount of work to be done in each step/cycle is about equal
  - each cycle uses at most once each major functional unit so that such units do not have to be replicated
  - functional units can be shared between different cycles within one instruction
- Data at end of one cycle to be used in next *must be stored* !!

الممسوحة ضوئيا بـ CamScanner

Breaking instructions into steps

- We break instructions into the following *potential* execution steps – not all instructions require all the steps – each step takes one clock cycle
  1. Instruction fetch and PC increment (**IF**)
  2. Instruction decode and register fetch (**ID**)
  3. Execution, memory address computation, or branch completion (**EX**)
  4. Memory access or R-type instruction completion (**MEM**)
  5. Memory read completion (**WB**)
- Each MIPS instruction takes from 3 – 5 cycles (steps)

Step 1: Instruction Fetch & PC Increment (IF)

- Use PC to get instruction and put it in the instruction register. Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using *RTL* (*Register-Transfer Language*):

```
IR = Memory[PC];
PC = PC + 4;
```

الممسوحة ضوئيا بـ CamScanner

Step 2: Instruction Decode and Register Fetch (ID)

- Read registers rs and rt in case we need them.
  Compute the branch address in case the instruction is a branch.
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

Step 3: Execution, Address Computation or Branch Completion (EX)

- ALU performs one of four functions *depending* on instruction type
  - memory reference:

```
ALUOut = A + sign-extend(IR[15-0]);
```

  - R-type:

```
ALUOut = A op B;
```

  - branch (instruction *completes*):

```
if (A==B) PC = ALUOut;
```

  - jump (instruction *completes*):

```
PC = PC[31-28] || (IR(25-0) << 2)
```

CamScanner المسوحة ضوئيا بـ

Step 4: Memory access or R-type Instruction Completion (MEM)

- Again *depending* on instruction type:
- Loads and stores access memory
  - load

    `MDR = Memory[ALUOut];`
  - store (instruction *completes*)

    `Memory[ALUOut] = B;`

- R-type (instructions *completes*)

  `Reg[IR[15-11]] = ALUOut;`

Step 5: Memory Read Completion (WB)

- Again *depending* on instruction type:
- Load writes back (instruction *completes*)

  `Reg[IR[20-16]]= MDR;`

Important: There is no reason from a datapath (or control) point of view that Step 5 cannot be eliminated by performing

`Reg[IR[20-16]]= Memory[ALUOut];`

for loads in Step 4. This would eliminate the MDR as well.

The reason this is not done is that, to keep steps balanced in length, the design restriction is to allow each step to contain *at most* one ALU operation, or one register access, or one memory access.

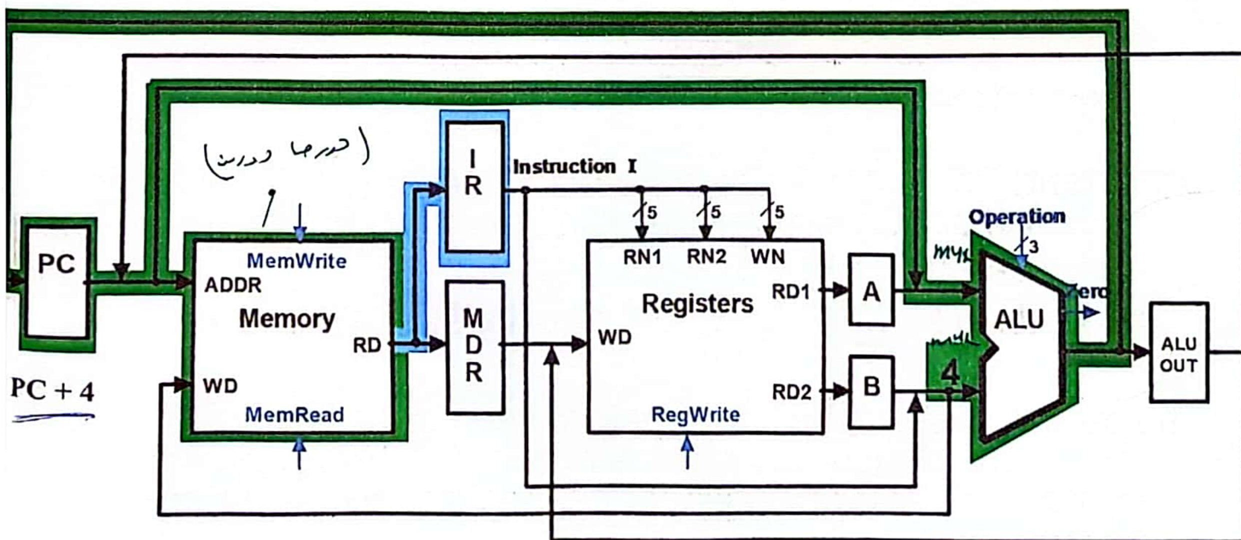
CamScanner الممسوحة ضوئيا بـ

Summary of Instruction Execution

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

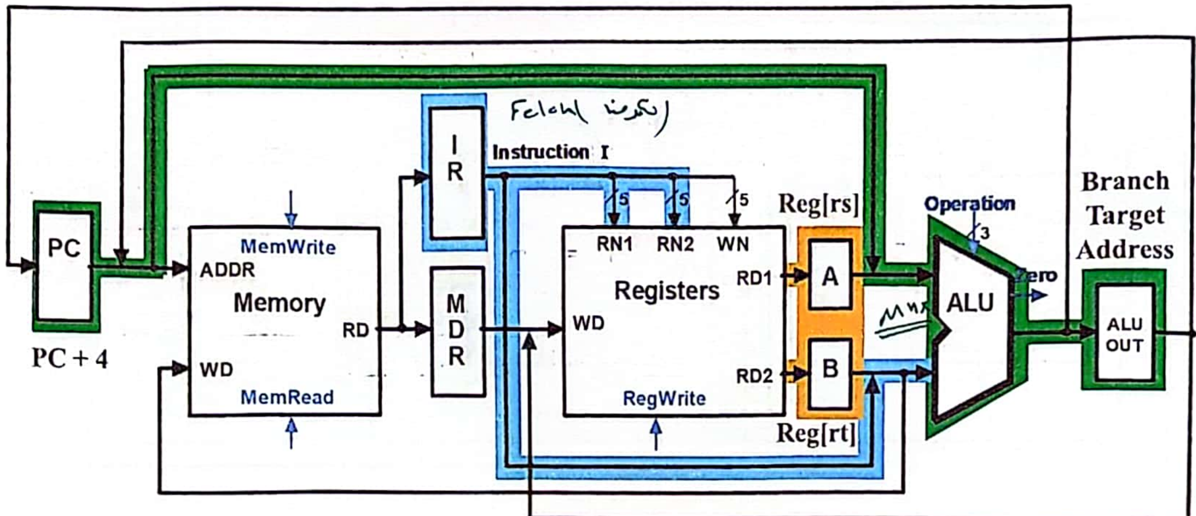
Multicycle Execution Step (1): Instruction Fetch

```
IR = Memory[PC];
PC = PC + 4;
```



Multicycle Execution Step (2): Instruction Decode & Register Fetch

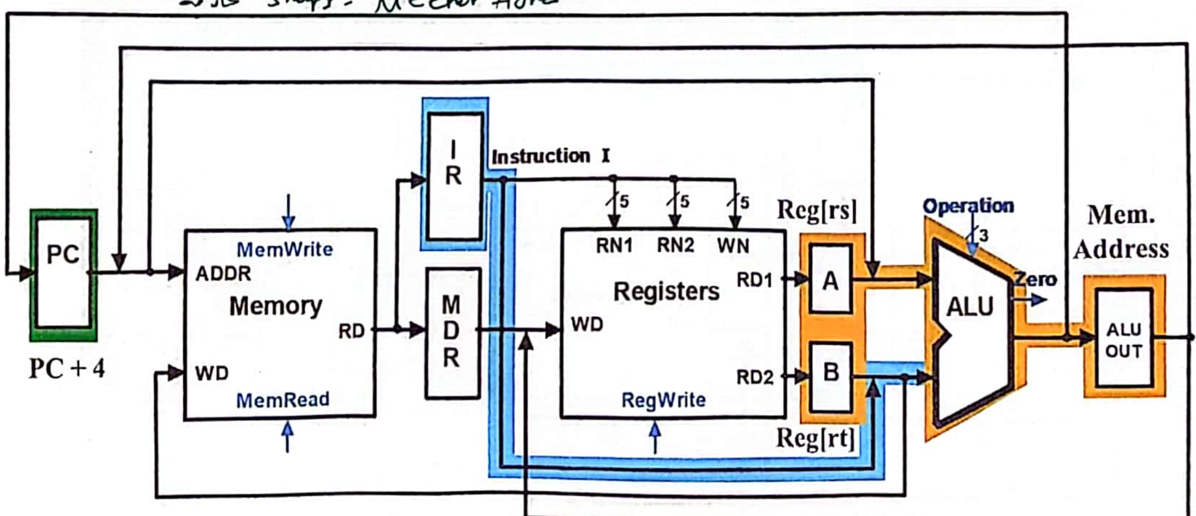
```
A = Reg[IR[25-21]];            (A = Reg[rs])
B = Reg[IR[20-15]];            (B = Reg[rt])
ALUOut = (PC + sign-extend(IR[15-0]) << 2)
```



Multicycle Execution Step (3): Memory Reference Instructions

```
ALUOut = A + sign-extend(IR[15-0]);
```

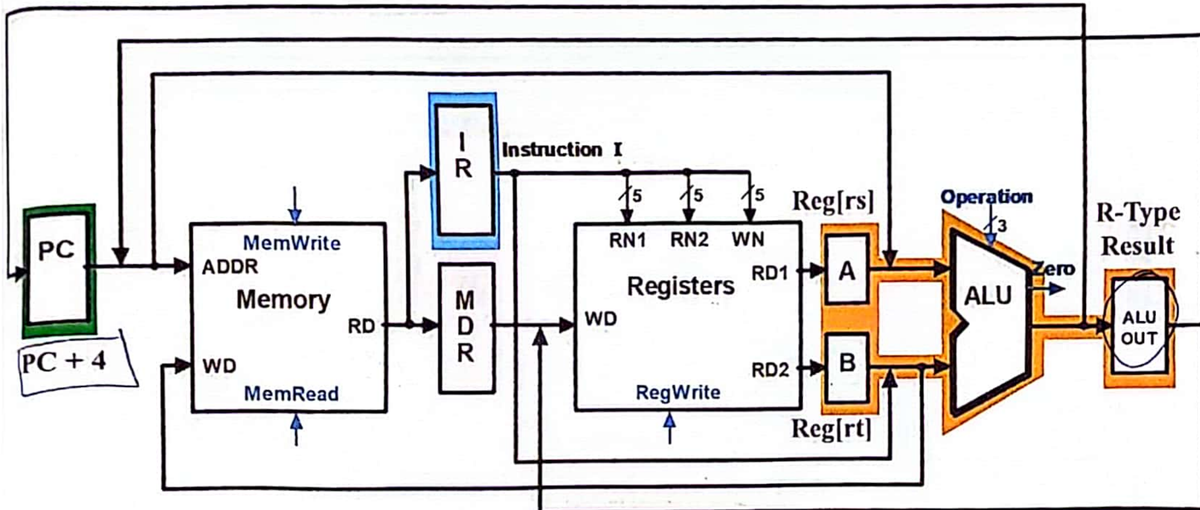
step2 = B.T
step3 = Mem Adre



المسوحة ضوئيا بـ CamScanner

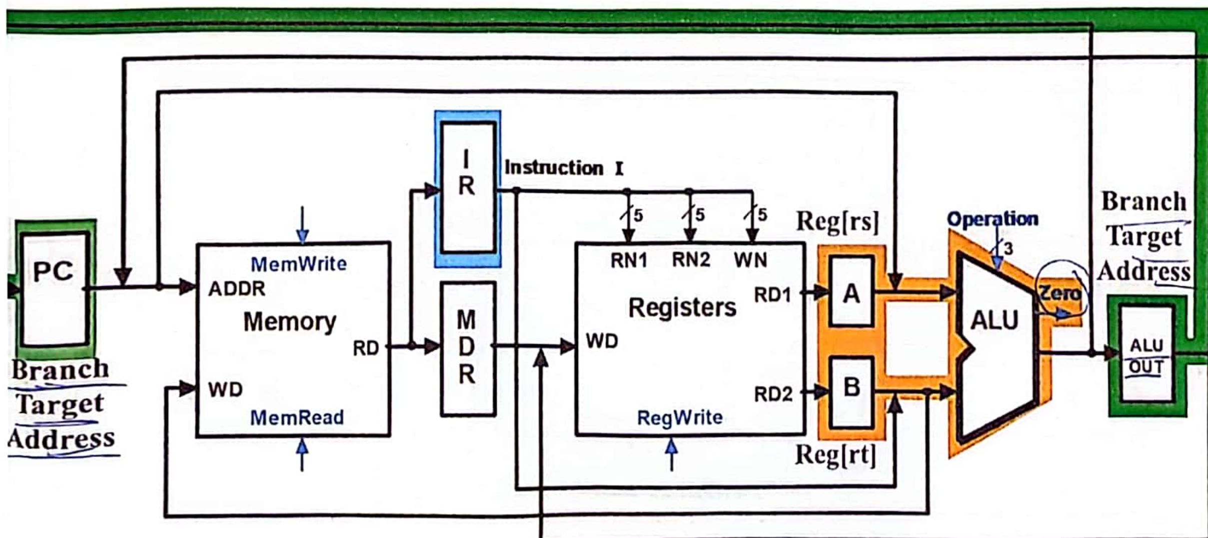
Multicycle Execution Step (3): ALU Instruction (R-Type)

```
ALUOut = A op B
```



Multicycle Execution Step (3): Branch Instructions

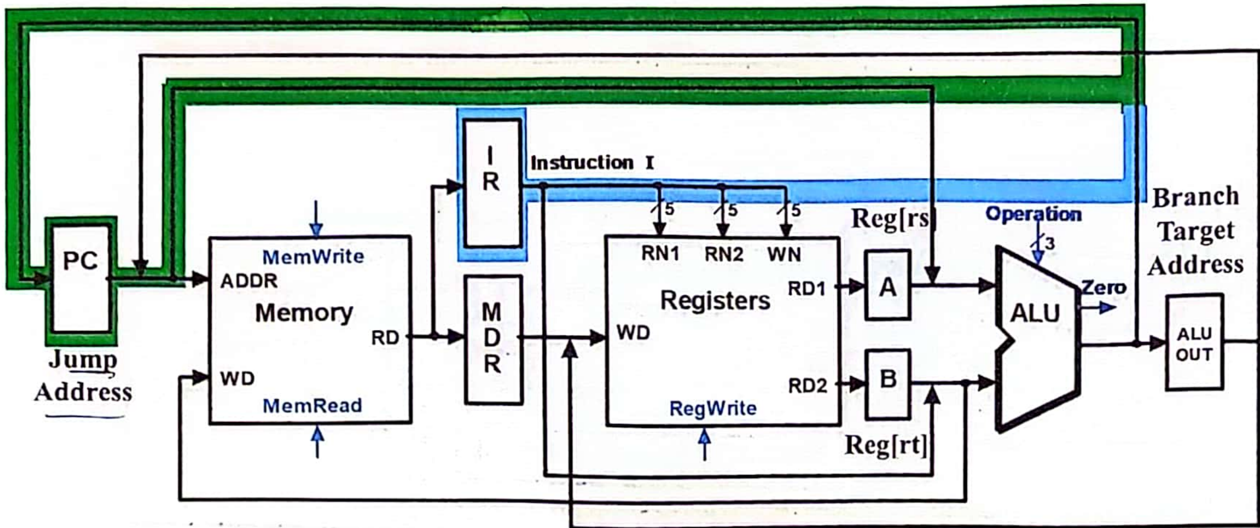
```
if (A == B) PC = ALUOut;
```



CamScanner المسوحة ضوئيا بـ

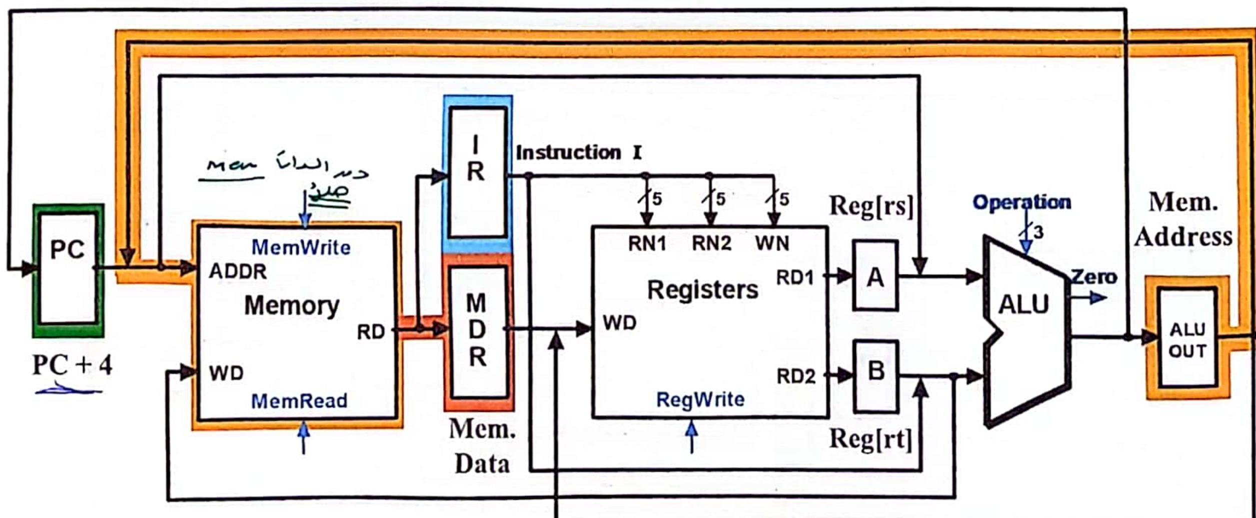
Multicycle Execution Step (3): Jump Instruction

PC = PC[31-28] concat (IR[25-0] << 2)



Multicycle Execution Step (4): Memory Access - Read (lw)

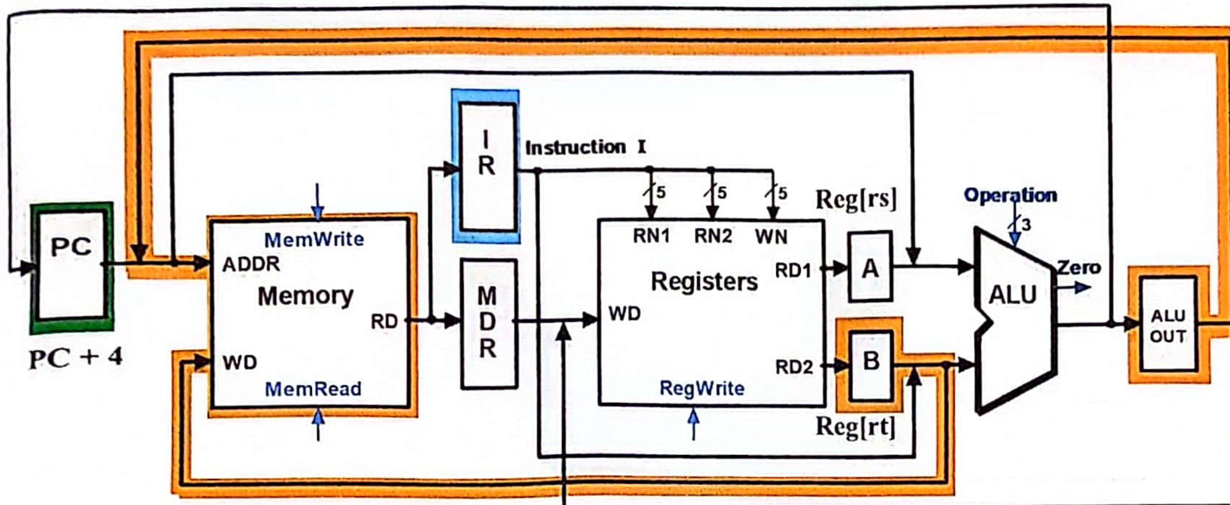
MDR = Memory[ALUOut];



المسوحة ضوئيا بـ CamScanner

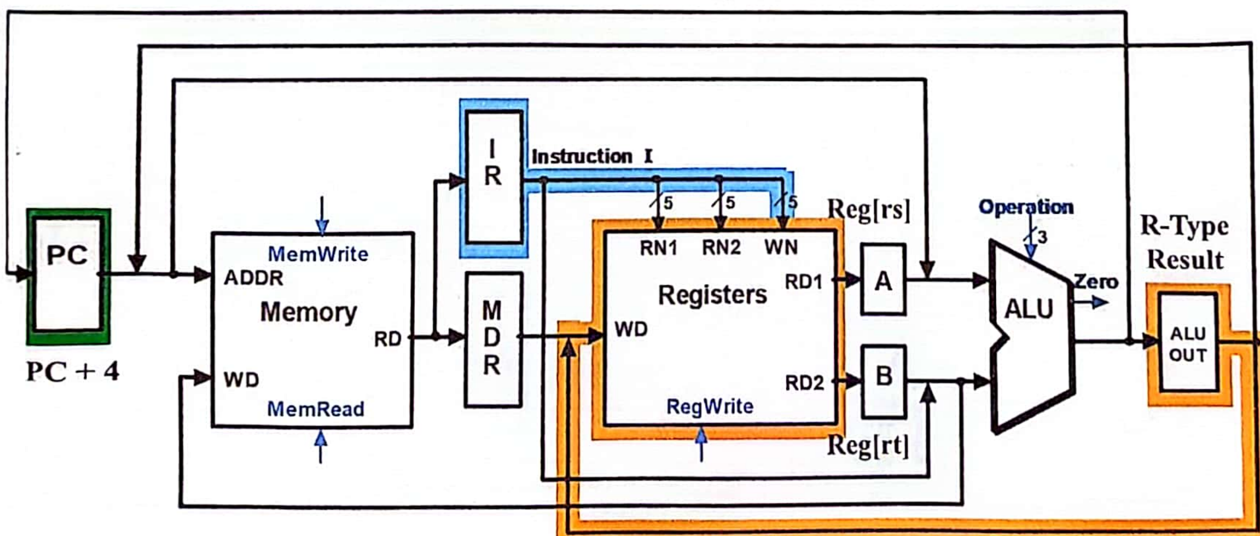
Multicycle Execution Step (4): Memory Access - Write (`sw`)

```
Memory[ALUOut] = B;
```



Multicycle Execution Step (4): ALU Instruction (R-Type)

```
Reg[IR[15:11]] = ALUOUT
```



CamScanner المسوحة ضوئيا بـ

Multicycle Execution Step (5): Memory Read Completion (lw)

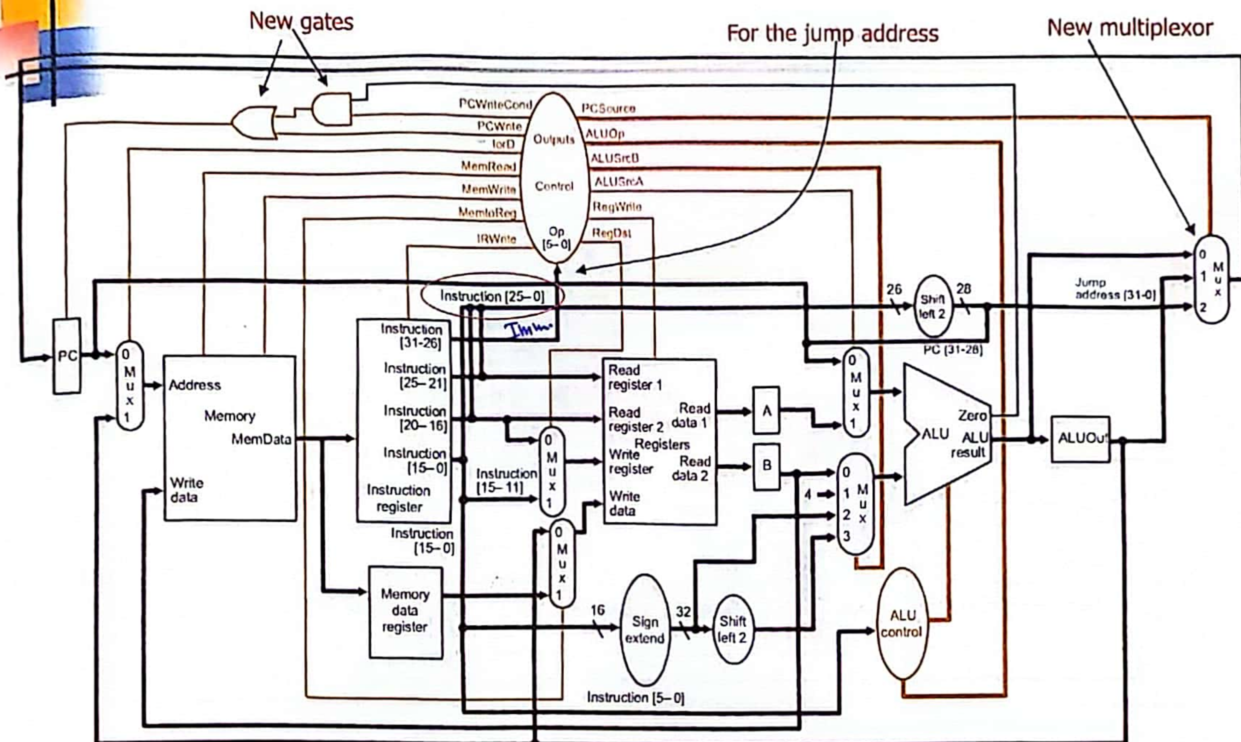
```
Reg[IR[20-16]] = MDR;
```

Multicycle Datapath with Control I

... with control lines and the ALU control block added – *not all* control lines are shown

الممسوحة ضوئيا بـ CamScanner

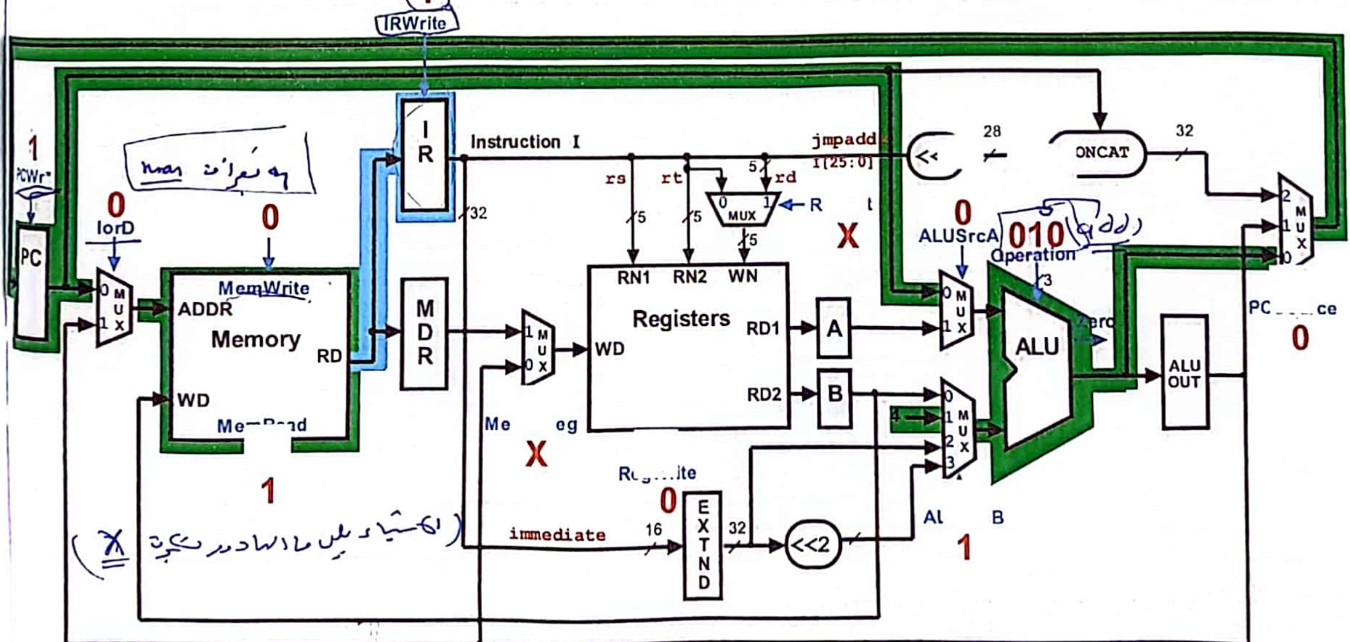
Multicycle Datapath with Control II



Complete multicycle MIPS datapath (with branch and jump capability) and showing the main control block and all control lines

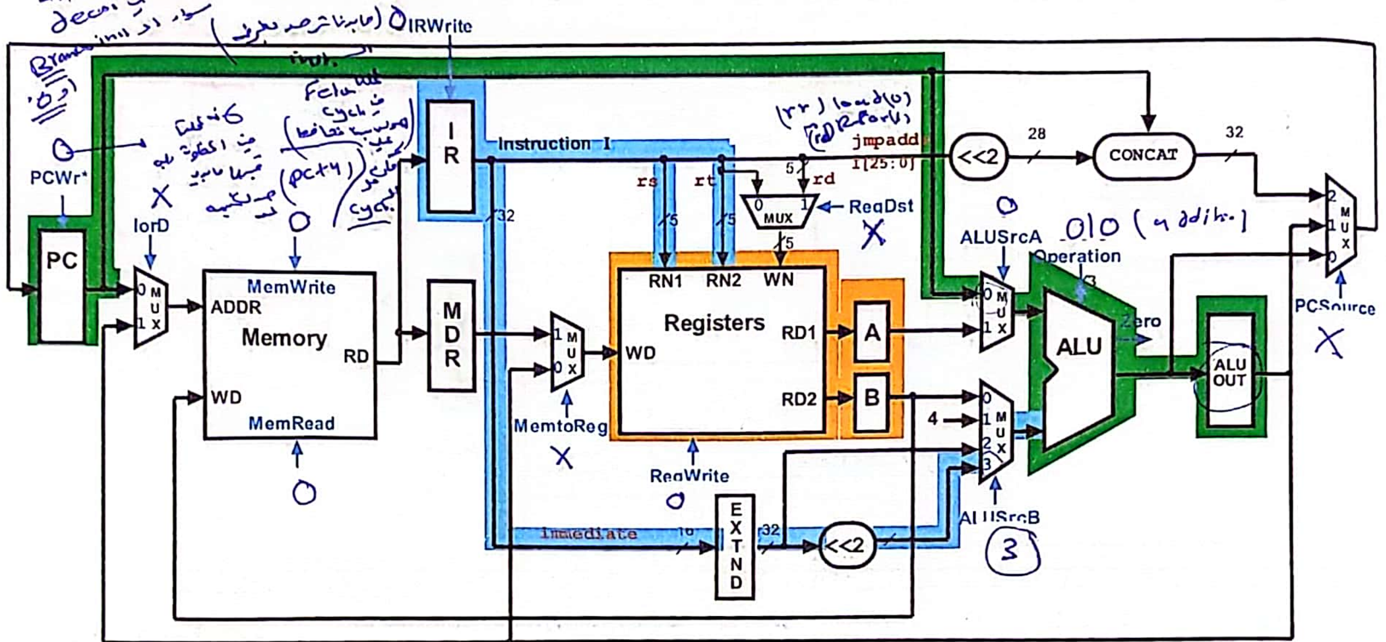
Multicycle Control Step (1): Fetch

```
IR = Memor  PC];
PC = PC + 4
```



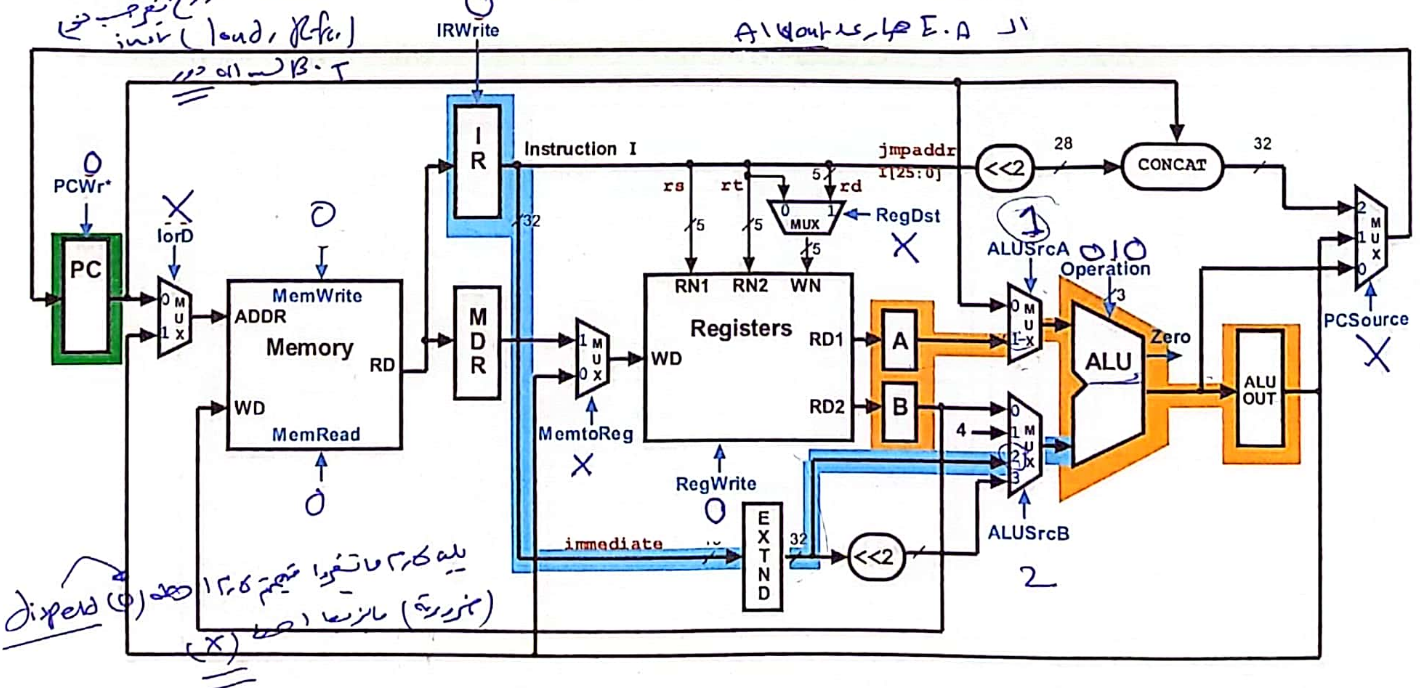
Multicycle Control Step (2): Instruction Decode & Register Fetch

```
A = Reg[IR[25-21]];        (A = Reg[rs])
B = Reg[IR[20-15]];        (B = Reg[rt])
ALUOut = (PC + sign-extend(IR[15-0]) << 2);
```



Multicycle Control Step (3): Memory Reference Instructions

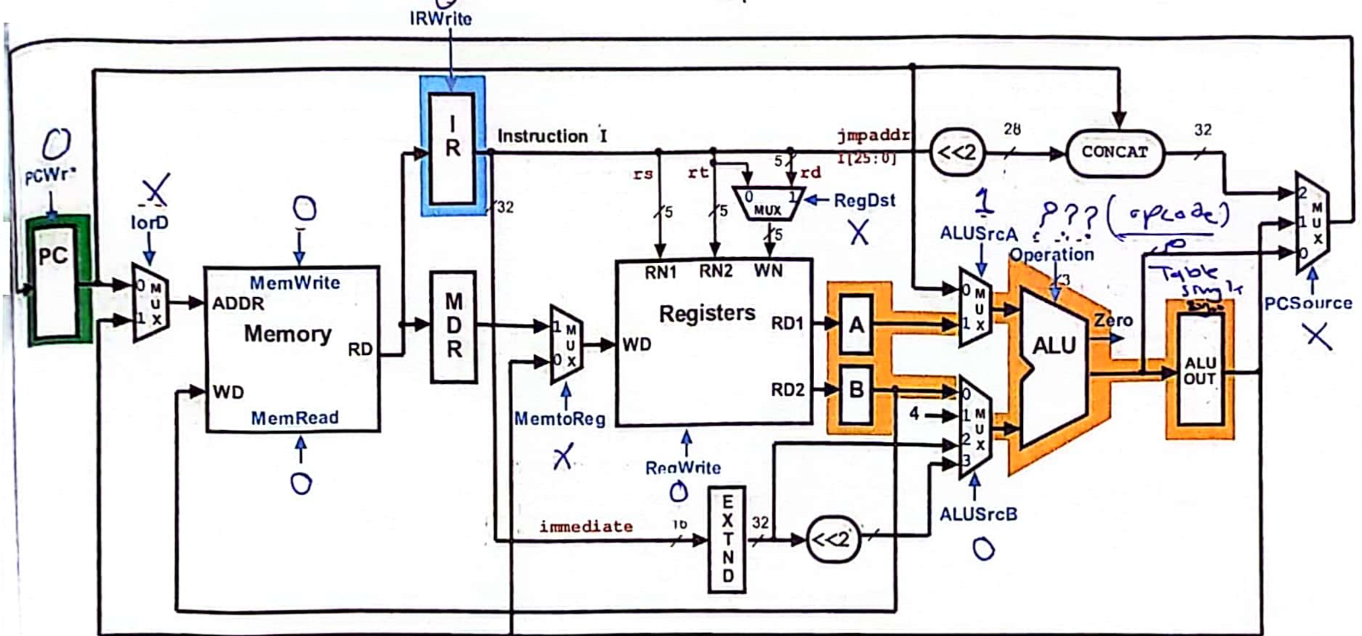
```
ALUOut = A + sign-extend(IR[15-0]);
```



CamScanner الممسوحة ضوئيا بـ

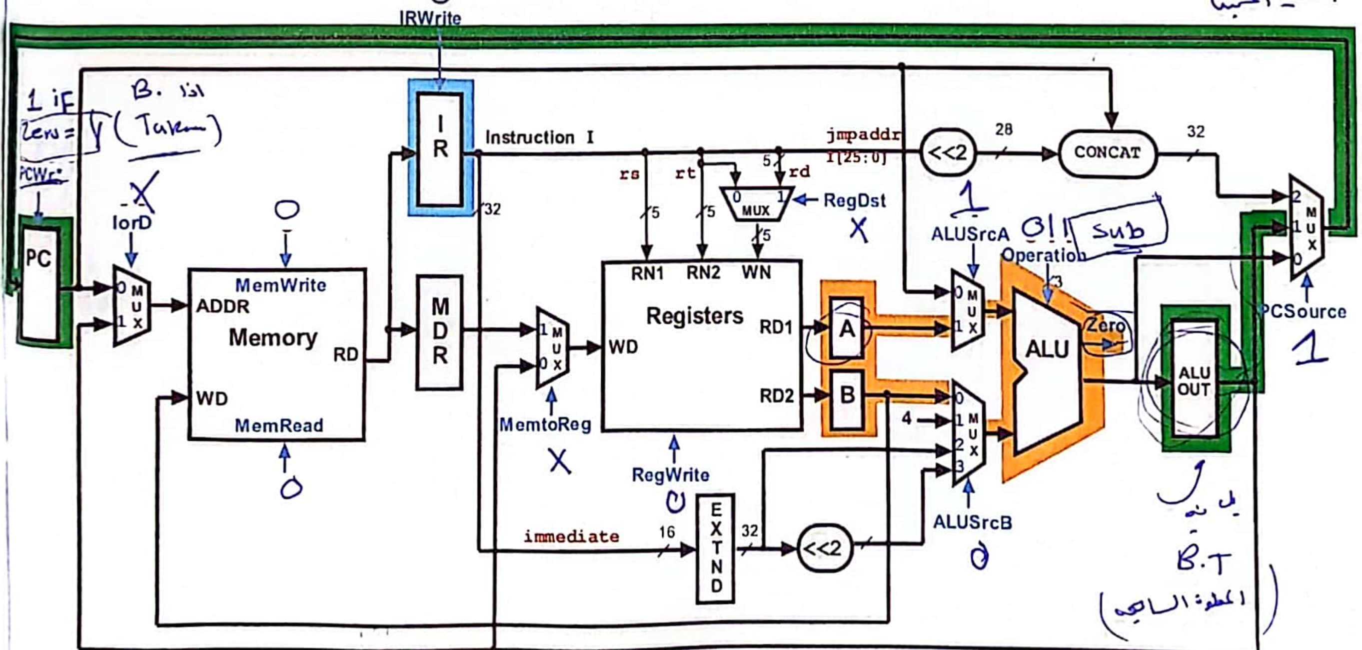
Multicycle Control Step (3): ALU Instruction (R-Type)

```
ALUOut = A op B;
```



Multicycle Control Step (3): Branch Instructions

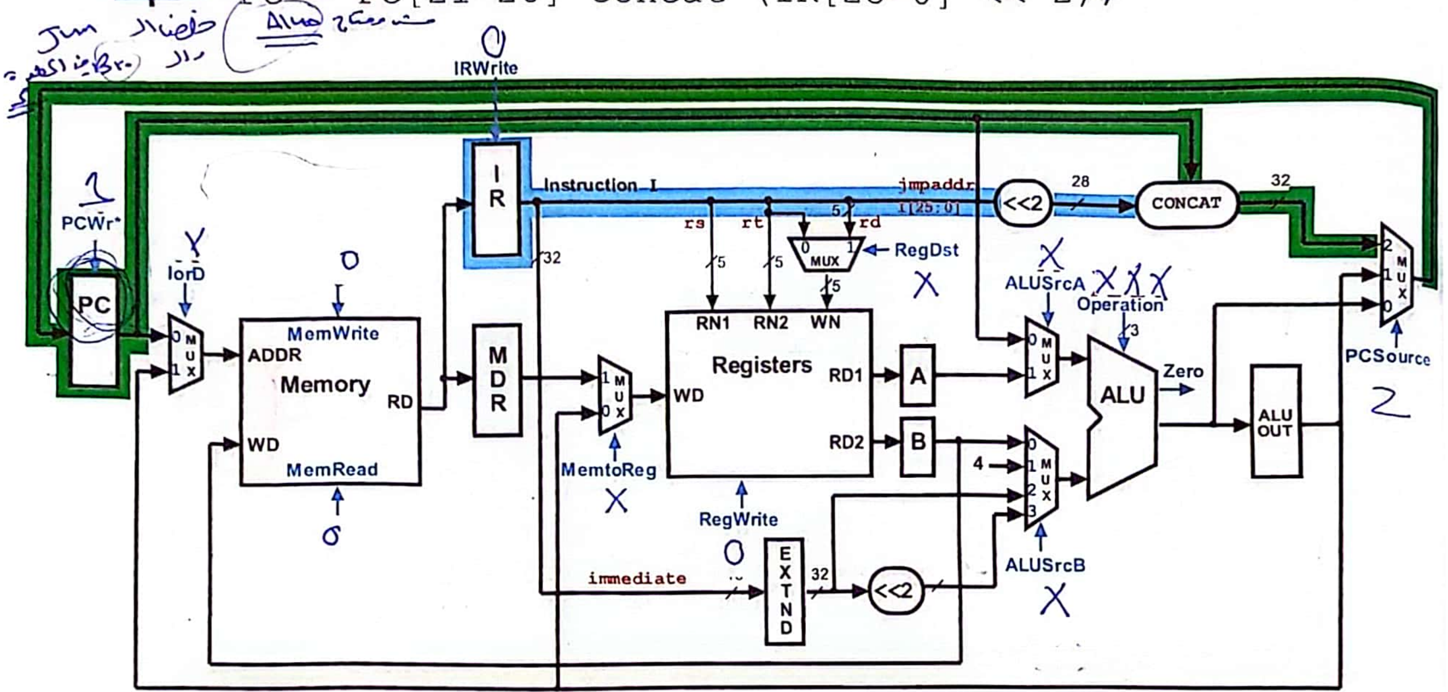
```
if (A == B) PC = ALUOut;
```



الممسوحة ضوئيا بـ CamScanner

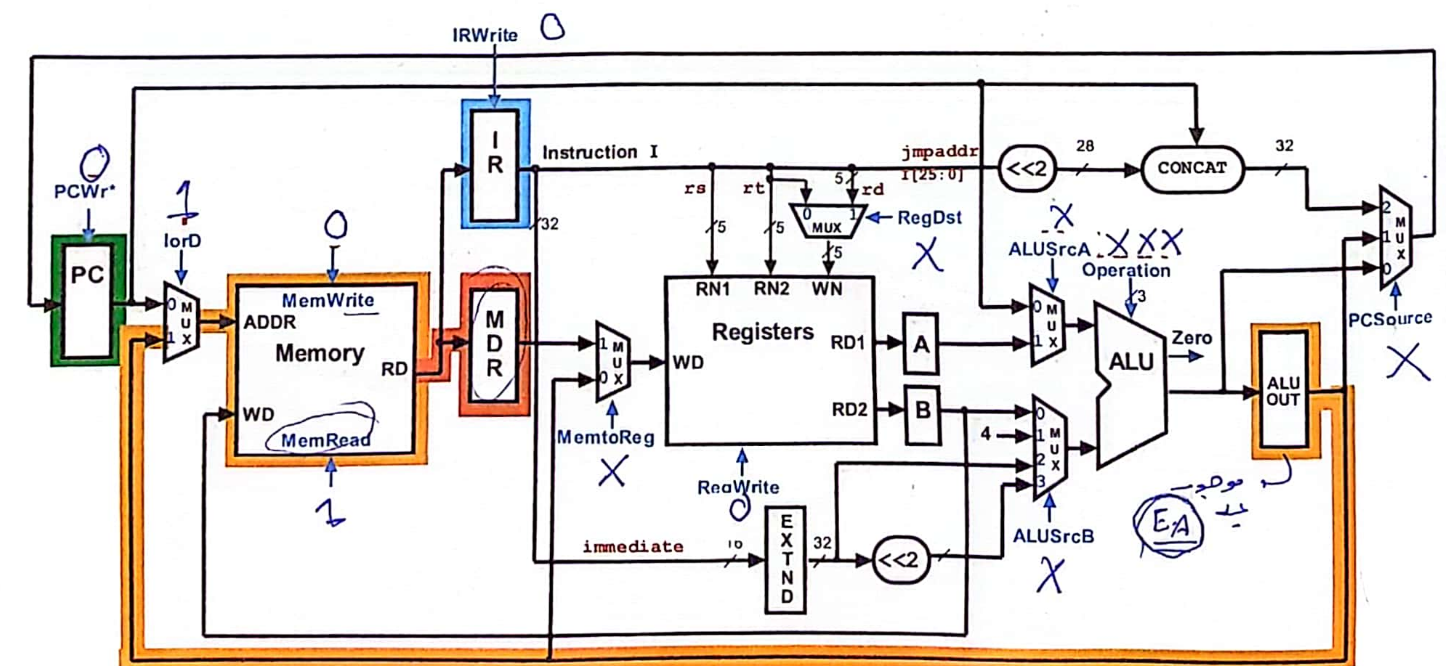
Multicycle Execution Step (3): Jump Instruction

`PC = PC[21-28] concat (IR[25-0] << 2);`



Multicycle Control Step (4): Memory Access - Read (lw)

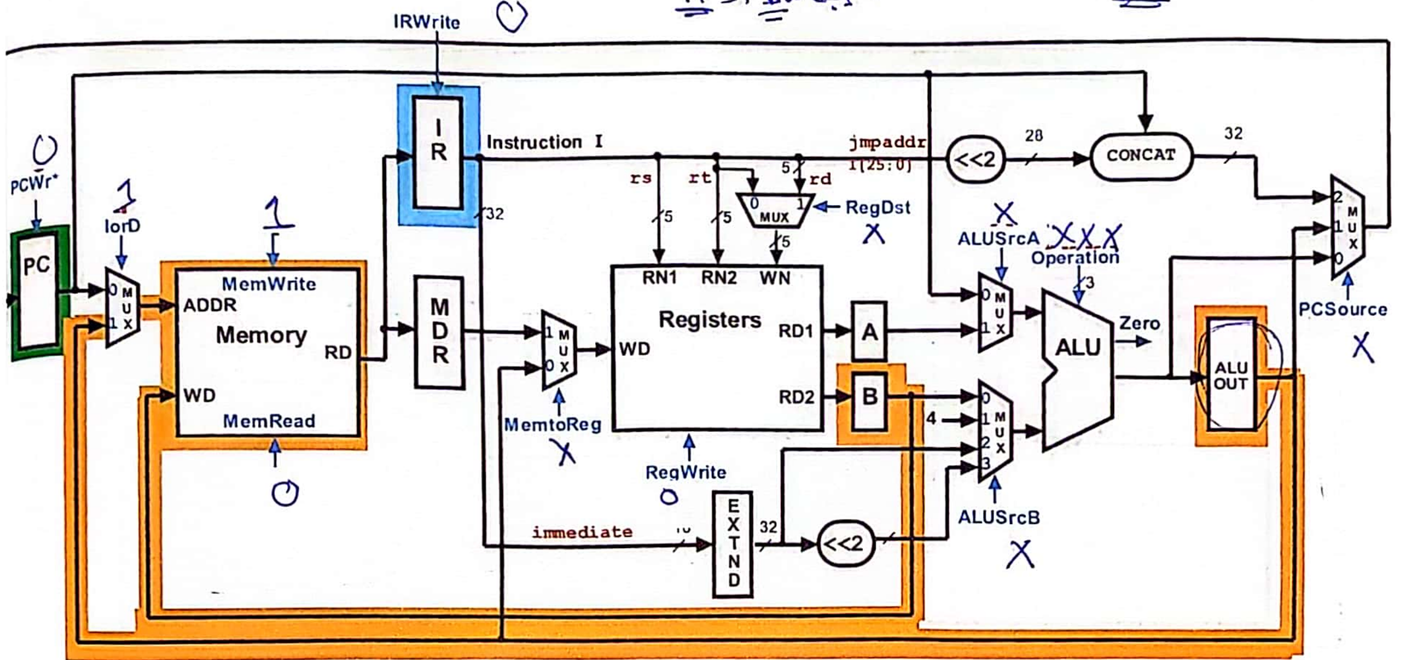
`MDR = Memory[ALUOut];`



CamScanner الممسوحة ضوئيا بـ

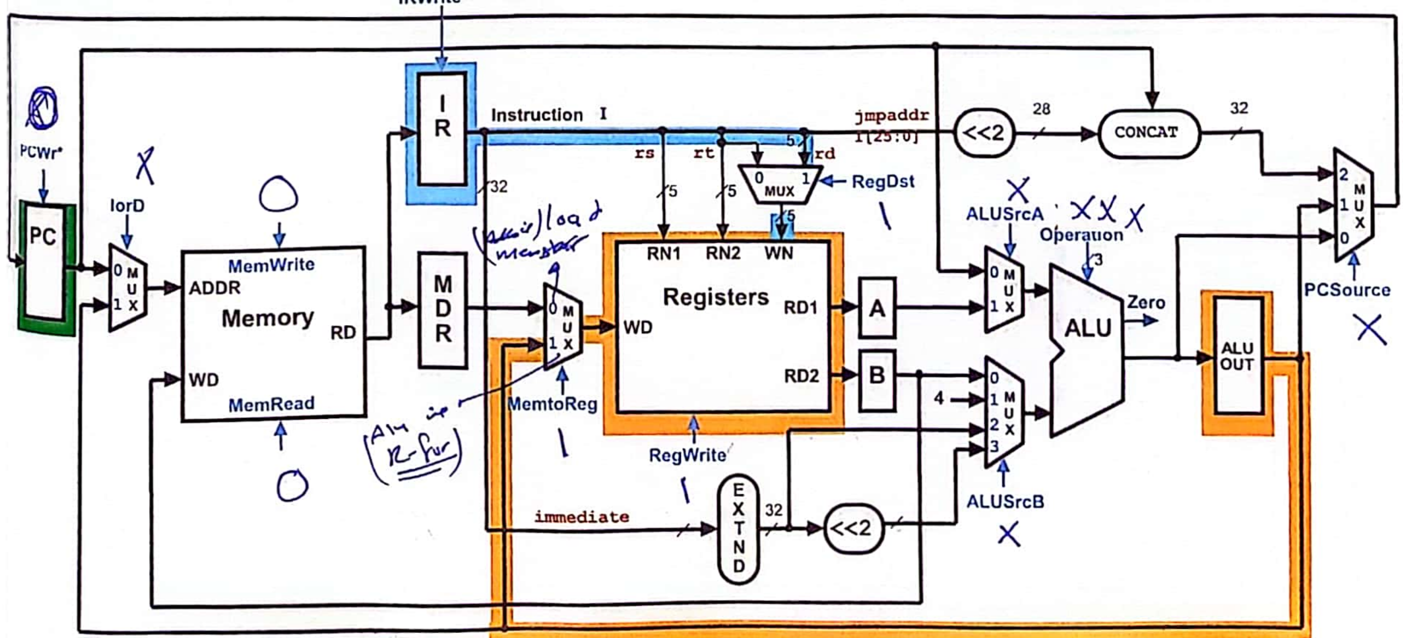
Multicycle Execution Steps (4)
Memory Access - Write (sw)

Memory[ALUOut] = B;



Multicycle Control Step (4): ALU Instruction (R-Type)

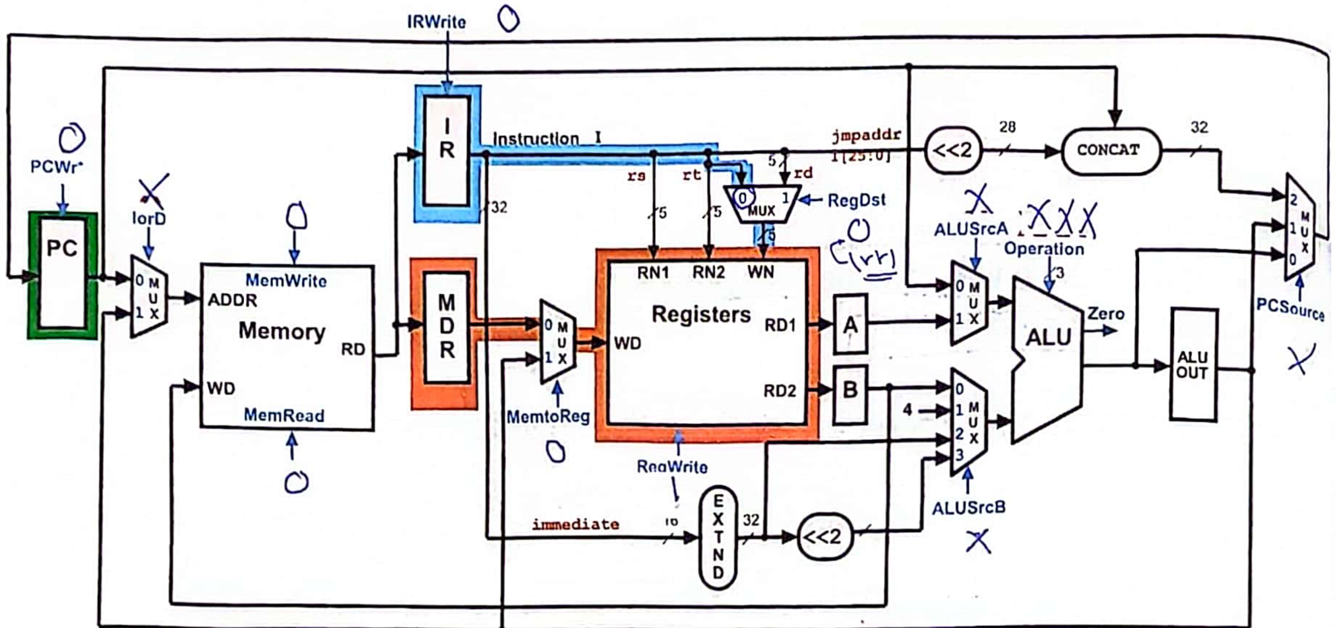
Reg[IR[15:11]] = ALUOut; (Reg[Rd] = ALUOut)



CamScanner الممسوحة ضوئيا بـ

Multicycle Execution Steps (5) Memory Read Completion (lw)

```
Reg[IR[20-16]] = MDR;
```



Simple Questions

- *How many cycles will it take to execute this code?*

```
        lw $t2, 0($t3)
        lw $t3, 4($t3)
        beq $t2, $t3, Label #assume not equal
        add $t5, $t2, $t3
        sw $t5, 8($t3)
Label:  ...
```

- *What is going on during the 8th cycle of execution?*

Clock time-line

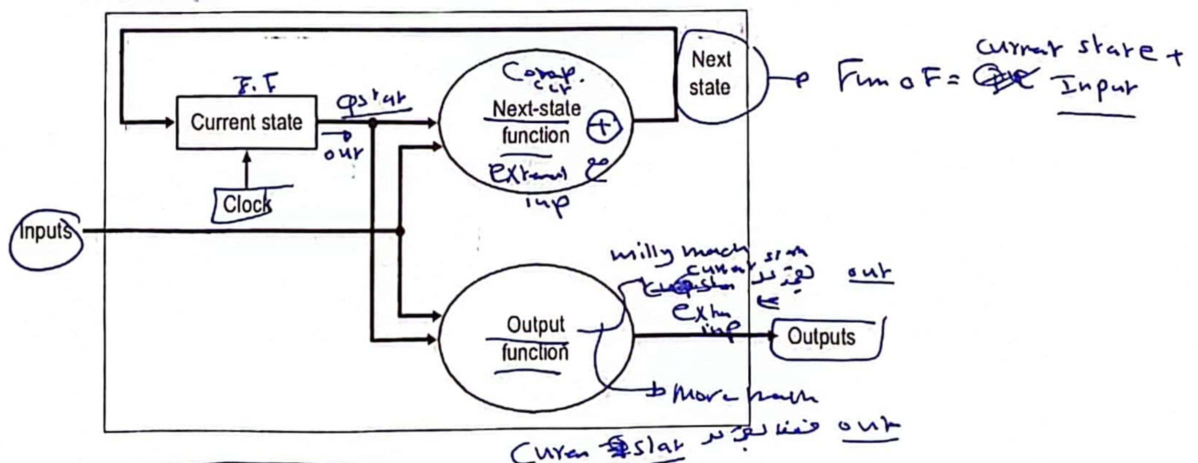
- *In what cycle does the actual addition of $t2 and $t3 takes place?*

Implementing Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we have accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation is then derived from the specification

Review: Finite State Machines

- Finite state machines (FSMs):
  - a set of states and
  - next state function, determined by current state and the input
  - output function, determined by current state and possibly input

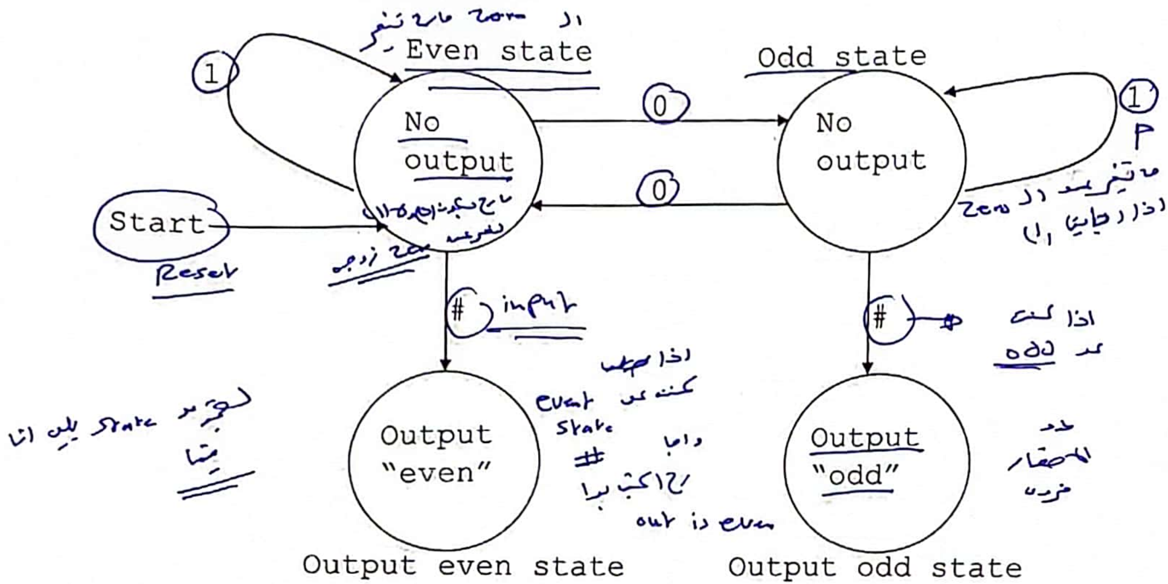


- We'll use a *Moore machine* – output based *only* on current state

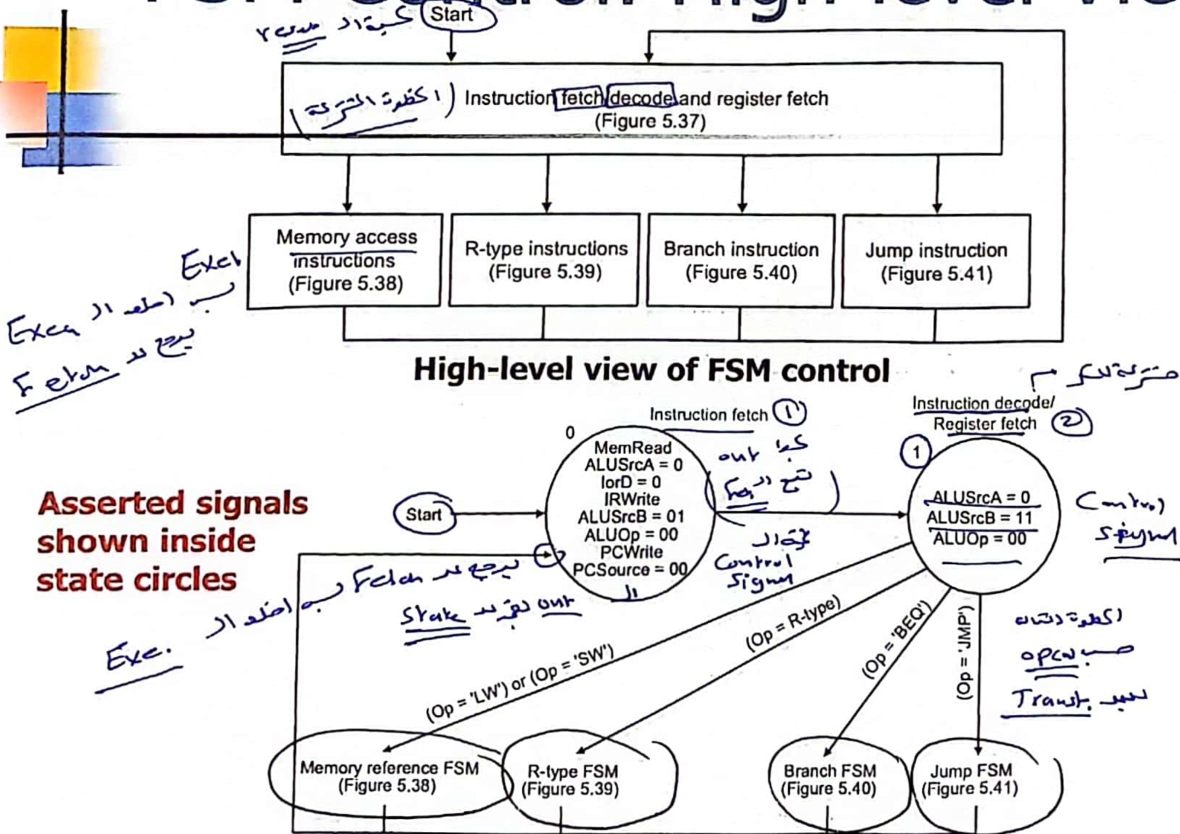
المسوحة ضوئيا بـ CamScanner

Example: Moore Machine

- The Moore machine below, given *input* a binary string terminated by "#", will *output* "even" if the string has an even number of 0's and "odd" if the string has an odd number of 0's



FSM Control: High-level View



High-level view of FSM control

Instruction fetch and decode steps of every instruction is identical

CamScanner المسوحة ضوئيا بـ

FSM Control: Memory Reference

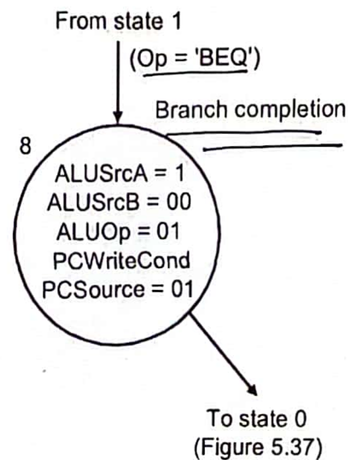
FSM control for memory-reference has 4 states

FSM Control: R-type Instruction

FSM control to implement R-type instructions has 2 states

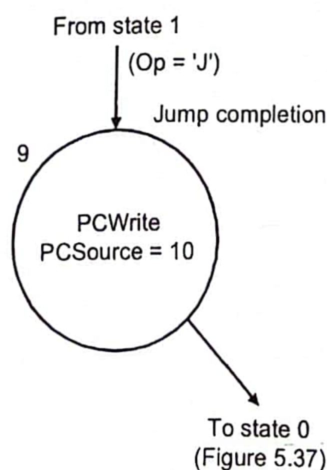
CamScanner الممسوحة ضوئيا بـ

FSM Control: Branch Instruction



FSM control to implement branches has 1 state

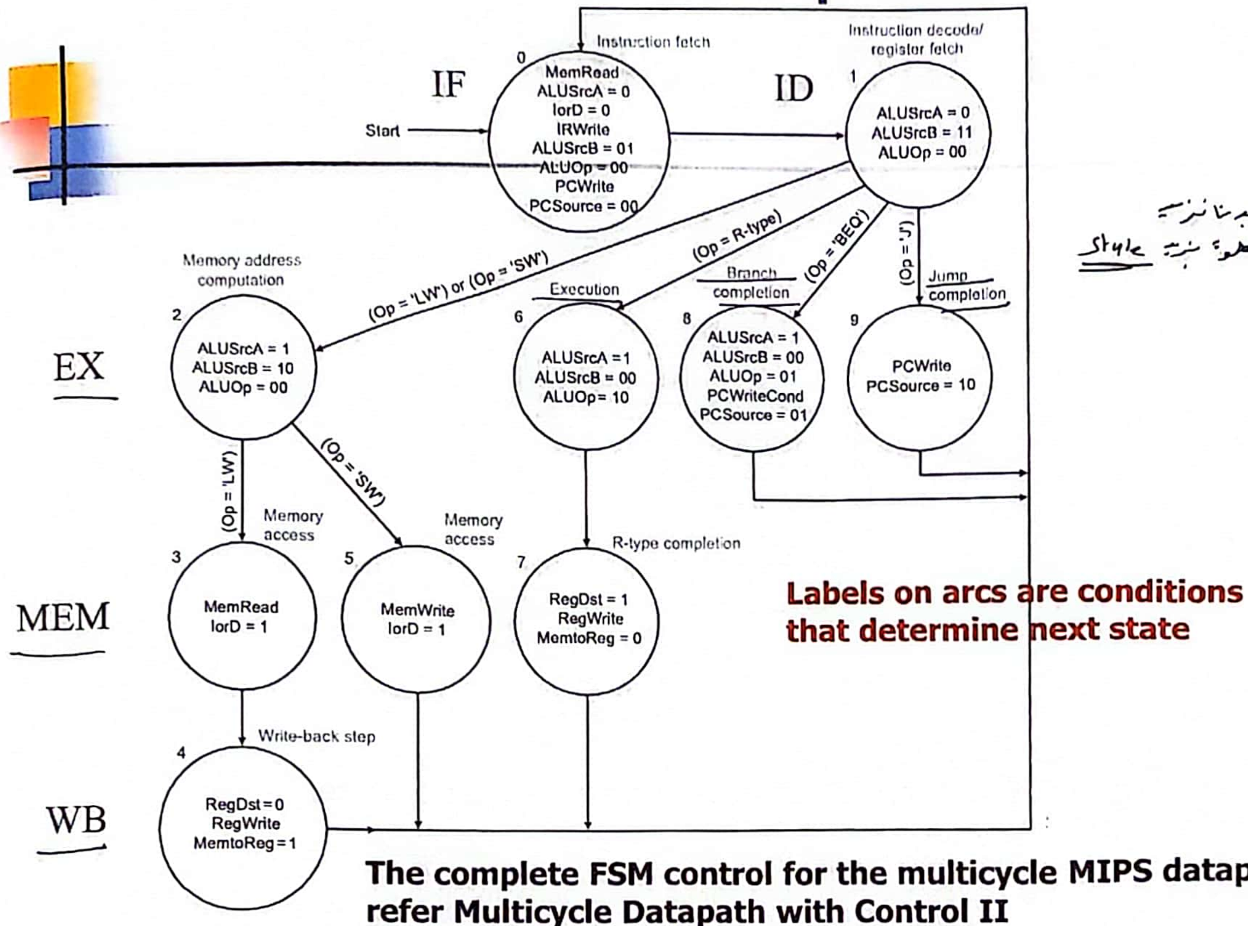
FSM Control: Jump Instruction



FSM control to implement jumps has 1 state

الممسوحة ضوئيا بـ CamScanner

FSM Control: Complete View



The complete FSM control for the multicycle MIPS datapath: refer Multicycle Datapath with Control II

Example: CPI in a multicycle CPU

- Assume
  - the control design of the previous slide
  - An instruction mix of 22% *loads*, 11% *stores*, 49% *R-type operations*, 16% *branches*, and 2% *jumps*
- *What is the CPI assuming each step requires 1 clock cycle?*
- Solution:
  - Number of clock cycles from previous slide for each instruction class:
    - *loads* 5, *stores* 4, *R-type instructions* 4, *branches* 3, *jumps* 3
  - CPI = CPU clock cycles / instruction count

$$= \Sigma\,(\text{instruction count}_{\text{class } i} \times \text{CPI}_{\text{class } i}) / \text{instruction count}$$

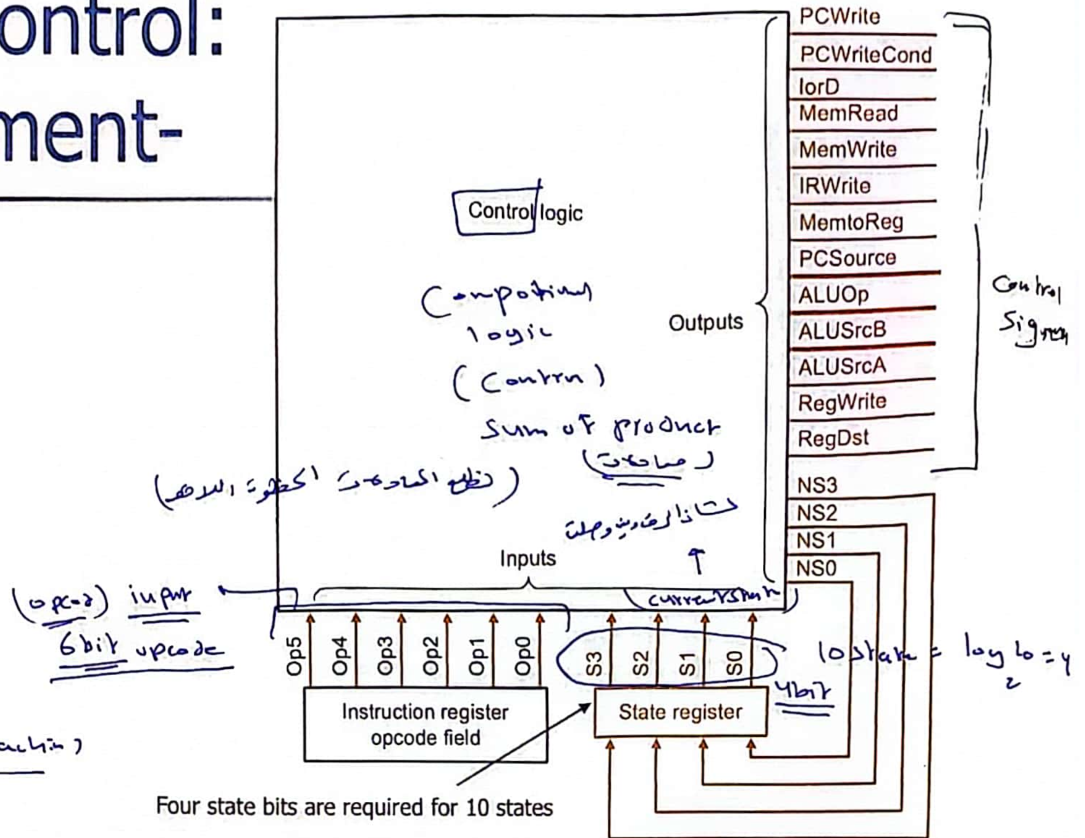
$$= \Sigma\,(\text{instruction count}_{\text{class } I} / \text{instruction count}) \times \text{CPI}_{\text{class } I}$$

$$= 0.22 \times 5 + 0.11 \times 4 + 0.49 \times 4 + 0.16 \times 3 + 0.02 \times 3$$

$$= 4.04$$

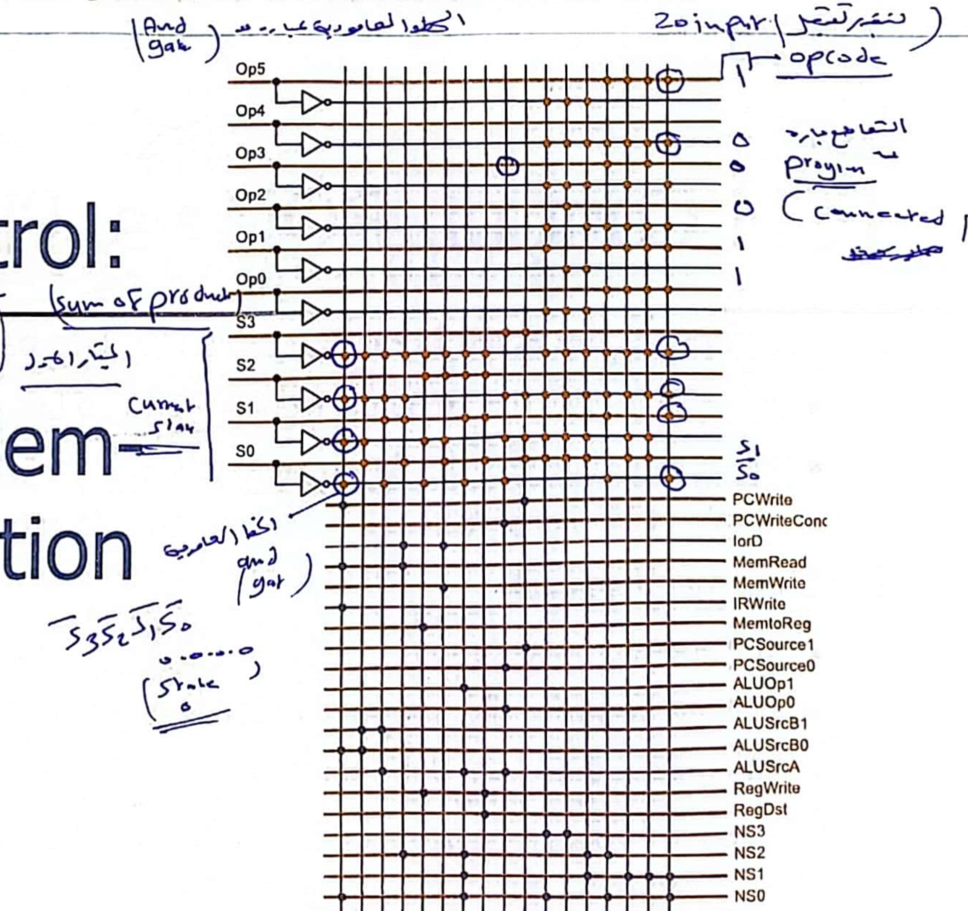
المسوحة ضوئيا بـ CamScanner

FSM Control: Implementation



High-level view of FSM implementation: inputs to the combinational logic block are the current state number and instruction opcode bits; outputs are the next state number and control signals to be asserted for the current state

FSM Control: PLA Implementation

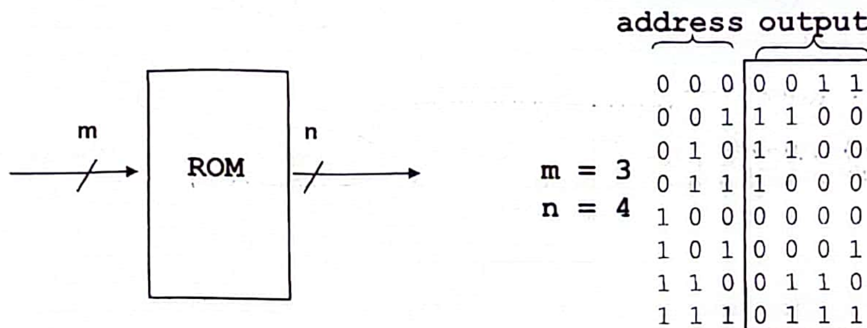


Upper half is the AND plane that computes all the products. The products are carried to the lower OR plane by the vertical lines. The sum terms for each output is given by the corresponding horizontal line
E.g., IorD = $\overline{S0}.\overline{S1}.S2.\overline{S3}$ + S0.S1.$\overline{S2}$.S3

CamScanner الممسوحة ضوئيا بـ

FSM Control: ROM Implementation

- ROM (Read Only Memory)
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address 2^m entries in the ROM
  - outputs are the bits of the entry the address points to



address	output
0 0 0	0 0 1 1
0 0 1	1 1 0 0
0 1 0	1 1 0 0
0 1 1	1 0 0 0
1 0 0	0 0 0 0
1 0 1	0 0 0 1
1 1 0	0 1 1 0
1 1 1	0 1 1 1

The size of an m-input n-output ROM is 2^m x n bits – such a ROM can be thought of as an array of size 2^m with each entry in the array being n bits

(H.W control unit)

FSM Control: ROM vs. PLA

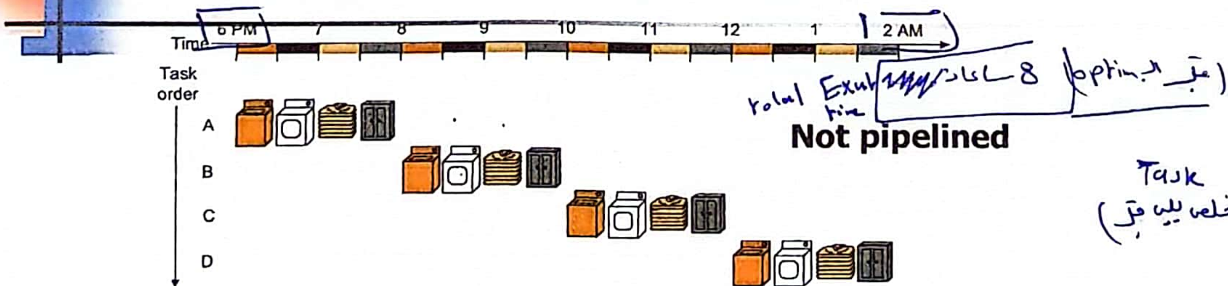
- First improve the ROM: break the table into two parts
  - 4 state bits give the 16 output signals – 2^4 x 16 bits of ROM
  - all 10 input bits give the 4 next state bits – 2^{10} x 4 bits of ROM
  - Total – 4.3K bits of ROM
- PLA is much smaller
  - can share product terms
  - only need entries that produce an active output
  - can take into account don't cares
- PLA size = (#inputs × #product-terms) + (#outputs × #product-terms)
  - FSM control PLA = (10x17)+(20x17) = 460 PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)

المسموحة ضوئيا بـ CamScanner

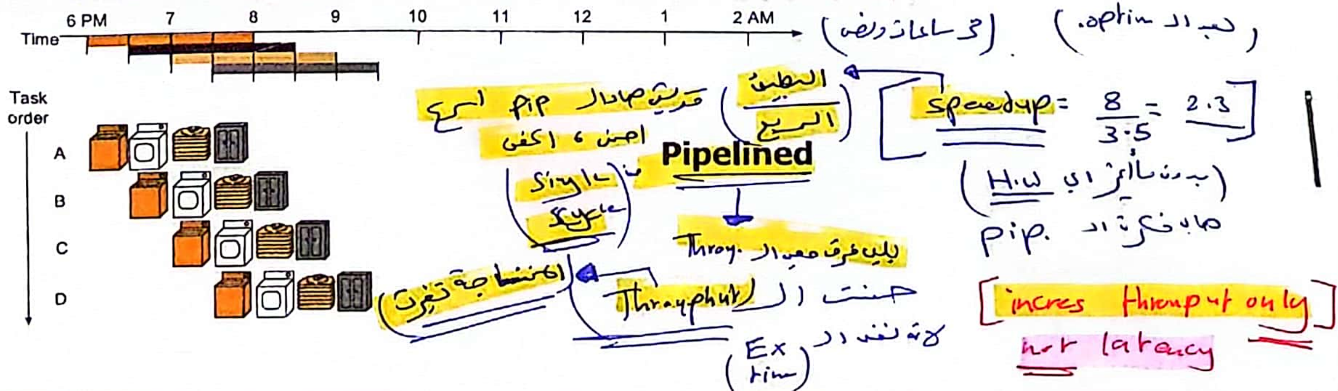
COD Ch. 6 Enhancing Performance with Pipelining

Pipelining

- Start work ASAP!! Do not waste time!



Assume 30 min. each task – wash, dry, fold, store – and that separate tasks use separate hardware and so can be overlapped

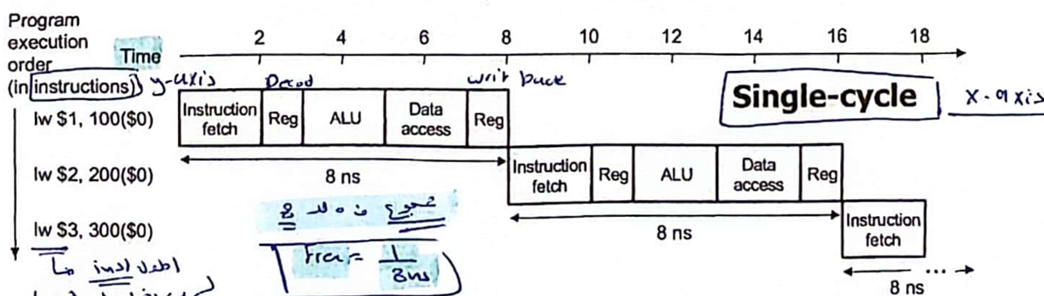


CamScanner المسوحة ضوئيا بـ

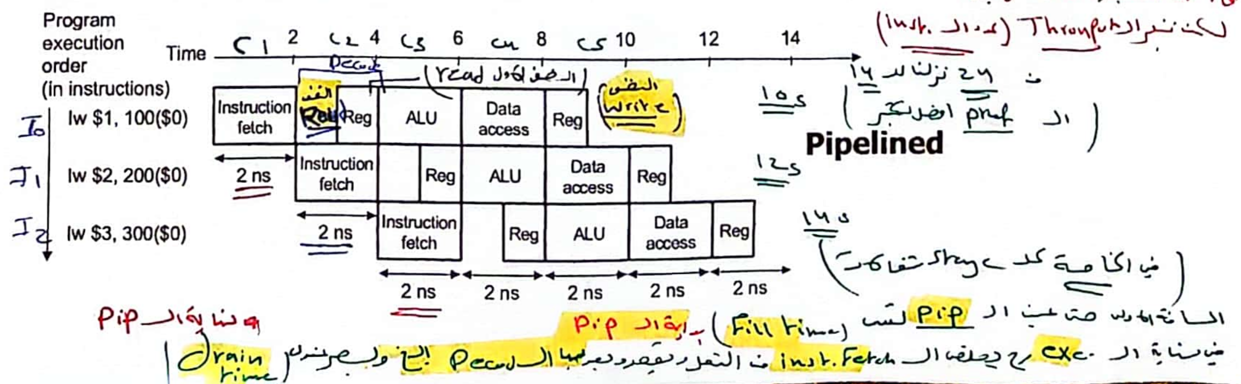
MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Pipelined vs. Single-Cycle Instruction Execution: the Plan



Assume 2 ns for memory access, ALU operation; 1 ns for register access: therefore, single cycle clock 8 ns; pipelined clock cycle 2 ns.



Pipelining: Keep in Mind

- Pipelining *does not reduce latency* of a single task, it *increases throughput* of entire workload
- Pipeline rate *limited by longest stage*
  - *potential* speedup = number pipe stages
  - *unbalanced lengths* of pipe stages reduces speedup
- Time to *fill* pipeline and time to *drain* it – when there is *slack* in the pipeline – reduces speedup

Example Problem

- *Problem: for the laundry fill in the following table when*
  1. *the stage lengths are 30, 30, 30 30 min., resp.*
  2. *the stage lengths are 20, 20, 60, 20 min., resp.*

Person	Unpipelined finish time	Pipeline 1 finish time	Ratio unpipelined to pipeline 1	Pipeline 2 finish time	Ratio unpiplelined to pipeline 2
1	**120**	**120**	**1**	**120**	**1**
2	240	150	1.6	180	1.3
3	360	180	2	240	1.5
4	480	210	2.2	300	1.6
	600	240	2.5	360	1.7
n	120n	120+30(n-1)	4 as n→∞	120+60(n-1)	2 as n→∞

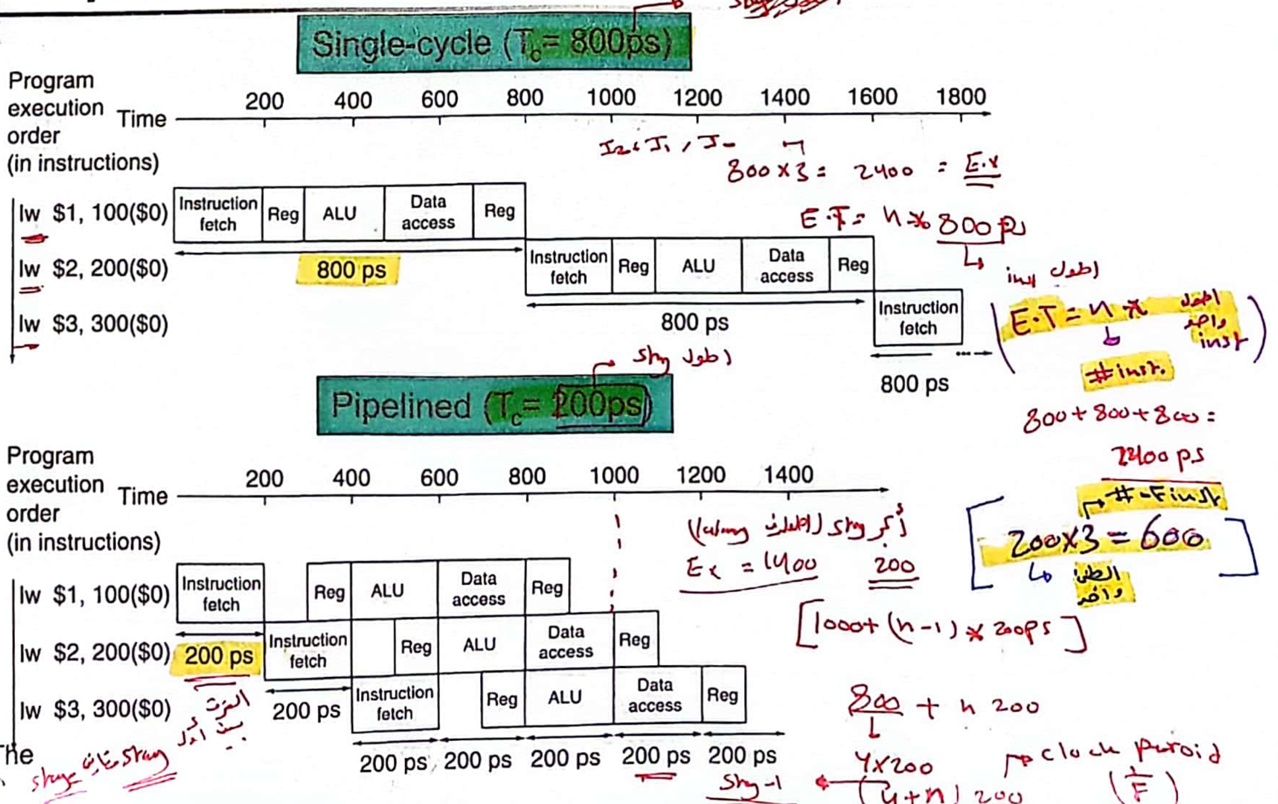
الممسوحة ضوئيا بـ CamScanner

Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance



Pipeline Speedup

$$\text{Speedup} = \frac{\text{CPU_Time}_{scalar}}{\text{CPU_Time}_{pipeline}}$$

CPU_Time_{scalar} = IC * CPI * Cyc_Time
= IC * Time/Instr.

$\text{CPU_Time}_{pipeline}$ = (Fill_Time + IC)* CPI * Cyc_Time

←----------------→
number of cycles

(Note: Number of instructions > # stages)

Fill_Time = # of stages - 1 = S - 1

$$\text{Speedup} = \frac{\text{CPU_Time}_{scalar}}{\text{CPU_Time}_{pipeline}}$$

If IC >> S, Speedup approaches S (max speedup)

Speedup Example

- Assume a program with N instructions: 10% (loads), 10%(stores), 50%(ALU) and 30%(branch). Assume the stage timing of:
- CPU_Time_{scalar}= N*0.1*800 + N*0.1*700 + N*0.5*600 + N*0.3*500
= N* 600 ps
- $\text{CPU_Time}_{pipeline}$= (S- 1+ N)* 1* 200
- Speedup = N*600 / [(S- 1+ N)* 200]
- If N >> S
- Speedup = 600N/200*N= 3

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

CamScanner المسوحة ضوئيا بـ

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$

$$= \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$$

- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining MIPS

- What makes it easy with MIPS?
  - all *instructions are same length*
    - so fetch and decode stages are similar for all instructions
  - just a *few instruction formats*
    - simplifies instruction decode and makes it possible in one stage
  - *memory operands appear only in load/stores*
    - so memory access can be deferred to exactly one later stage
  - *operands are aligned in memory*
    - one data transfer instruction requires one memory access stage

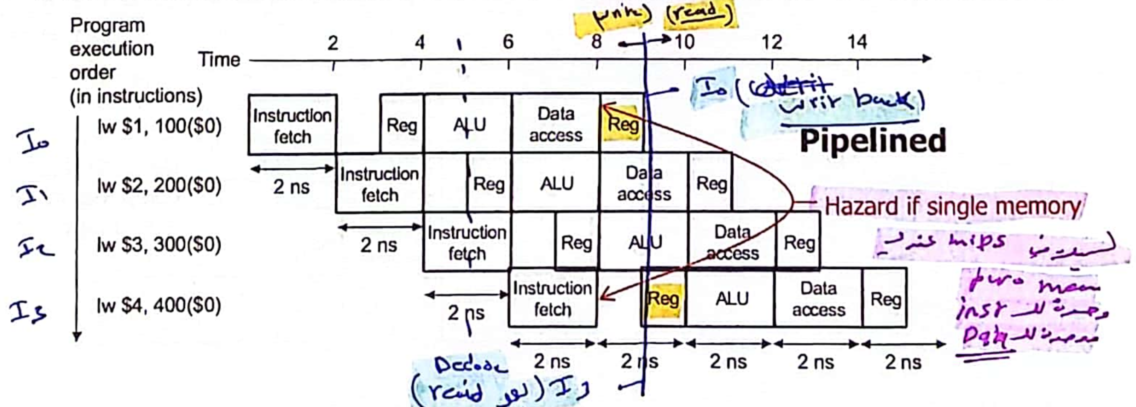
المسوحة ضوئيا بـ CamScanner

Pipelining MIPS (Hazards)

- What makes it hard?
  - *structural hazards*: different instructions, at different stages, in the pipeline want to use the same hardware resource
  - *control hazards*: succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction, already in pipeline
  - *data hazards*: an instruction in the pipeline requires data to be computed by a previous instruction still in the pipeline
- Before actually building the pipelined datapath and control we first briefly examine these potential hazards individually…

Structural Hazards

- *Structural hazard*: inadequate hardware to simultaneously support all instructions in the pipeline in the same clock cycle
- E.g., suppose *single – not separate –* instruction and data memory in pipeline below with *one read port*
  - then a structural hazard between first and fourth `lw` instructions



- *MIPS was designed to be pipelined*: structural hazards are easy to avoid!

CamScanner المسموحة ضوئيا بـ

Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Control Hazards

- *Control hazard*: need to make a decision based on the result of a previous instruction still executing in pipeline
- Solution 1. *Stall* the pipeline

Program execution order (in instructions)

Time: 2, 4, 6, 8, 10, 12, 14, 16

add $4, $5, $6 — Instruction fetch | Reg | ALU | Data access | Reg

beq $1, $2, 40 — (2ns) Instruction fetch | Reg | ALU | Data access | Reg

lw $3, 300($0) — (4 ns) bubble | Instruction fetch | Reg | ALU | Data access | Reg (2ns)

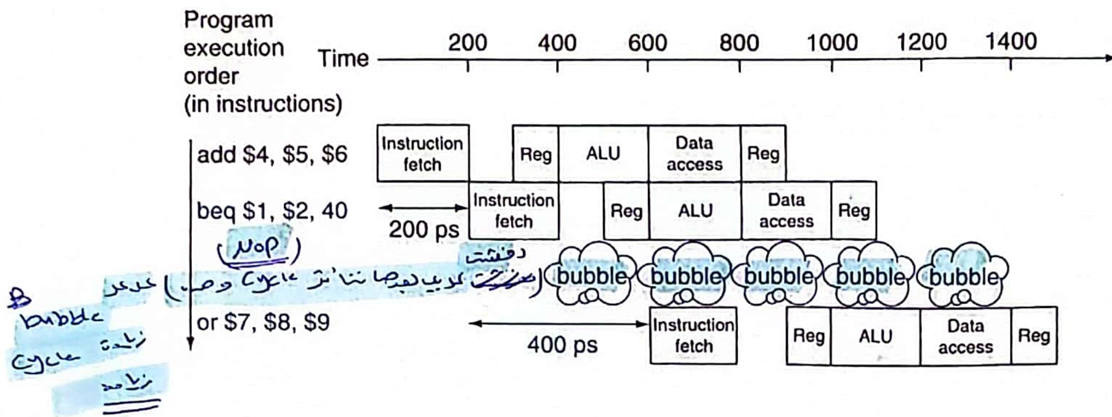
Note that branch outcome is computed in ID stage with added hardware (later...)

Pipeline stall

CamScanner المسموحة ضوئيا بـ

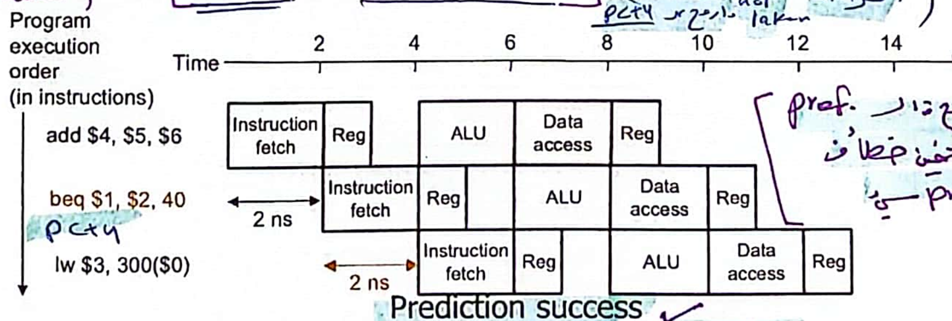
Stall on Branch

- Wait until branch outcome determined before fetching next instruction

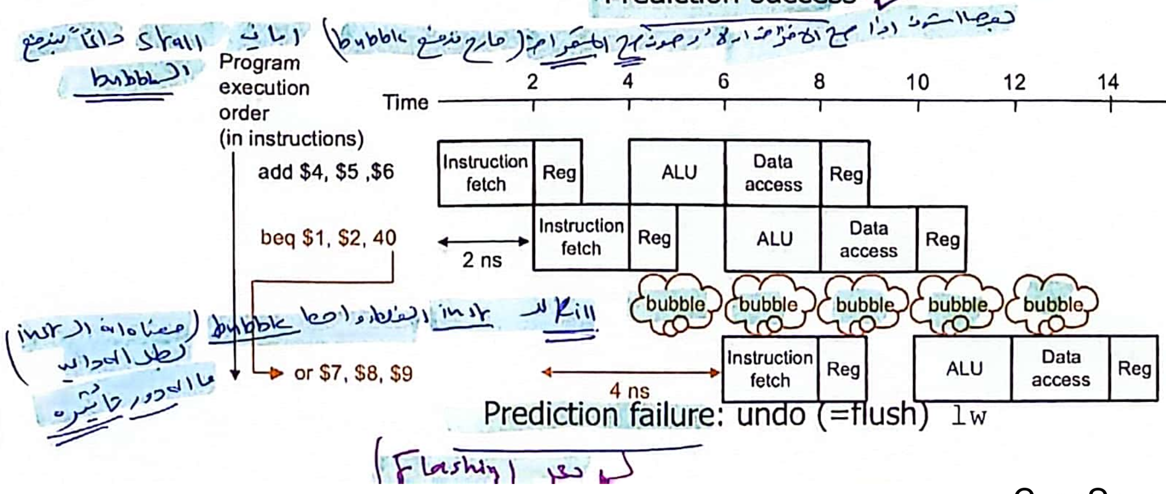


Control Hazards

- Solution 2 *Predict* branch outcome
  - e.g. predict *branch-not-taken*:



Prediction success



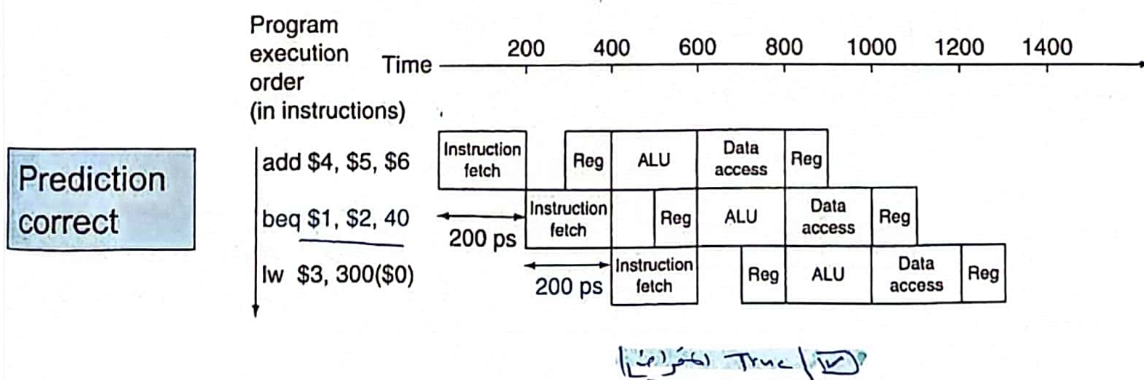
Prediction failure: undo (=flush) `lw`

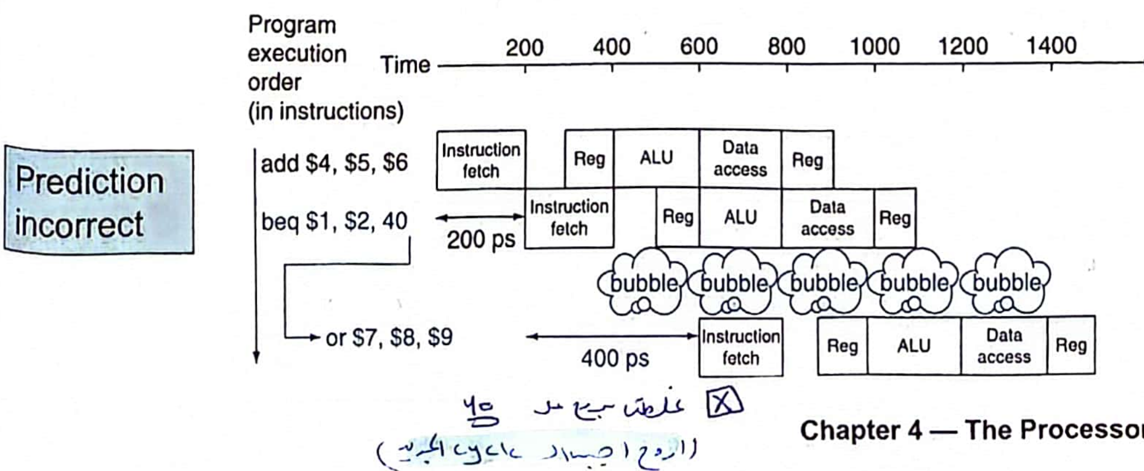
CamScanner الممسوحة ضوئيا بـ

Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken





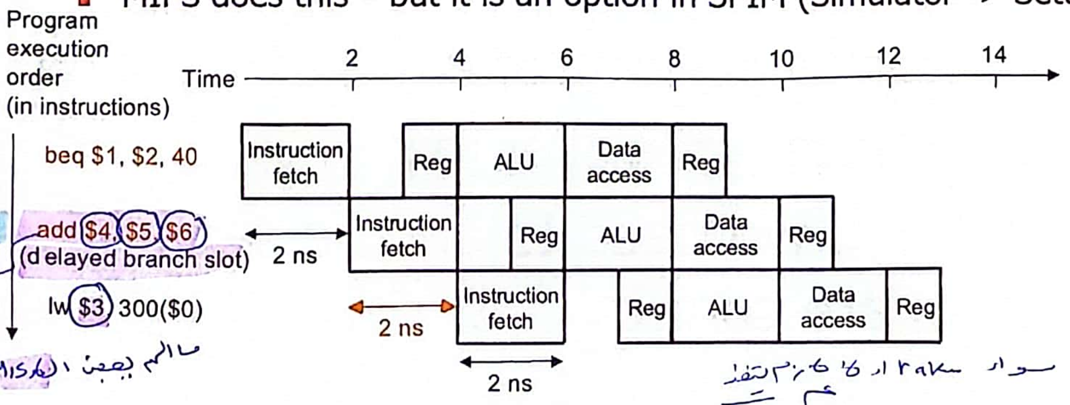
المسوحة ضوئيا بـ CamScanner

More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Control Hazards

- Solution 3 *Delayed branch*: always execute the sequentially next statement with the branch executing after one instruction delay – compiler's job to find a statement that can be put in the slot that is independent of branch outcome
  - MIPS does this – but it is an option in SPIM (Simulator -> Settings)



Delayed branch `beq` **is followed by** `add` **that is independent of branch outcome**

CamScanner المسوحة ضوئيا بـ

Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add $s0, $t0, $t1
  - sub $t2, $s0, $t3

Time: 200 400 600 800 1000 1200 1400 1600

add $s0, $t0, $t1: IF → ID → EX → MEM → WB

bubble bubble bubble bubble bubble

bubble bubble bubble bubble bubble

sub $t2, $s0, $t3: IF → ID → EX → MEM → WB

Data Dependences

Types of data dependences

- Flow dependence (true data dependence – read after write)
- Output dependence (write after write)
- Anti dependence (write after read)

For all of them, we need to ensure semantics of the program are correct

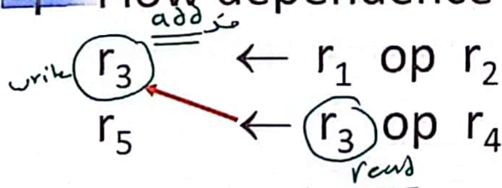
CamScanner الممسوحة ضوئيا بـ

Data Dependences

- Flow dependences always need to be obeyed because they constitute true dependence on a value
- Anti and output dependences exist due to limited number of architectural registers
  - They are dependence on a name, not a value

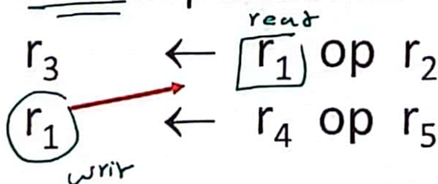
Data Dependence Types

Flow dependence



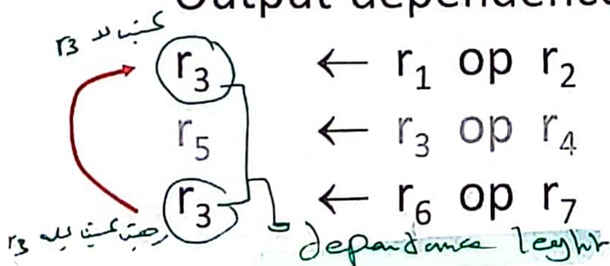
Read-after-Write (RAW)

Anti dependence



Write-after-Read (WAR)

Output-dependence

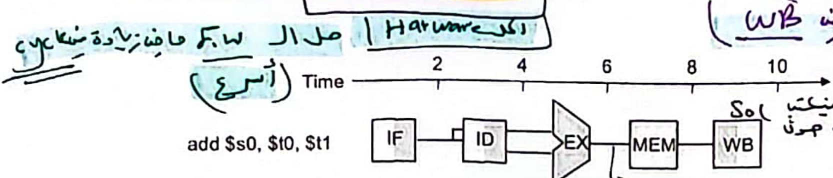


Write-after-Write (WAW)

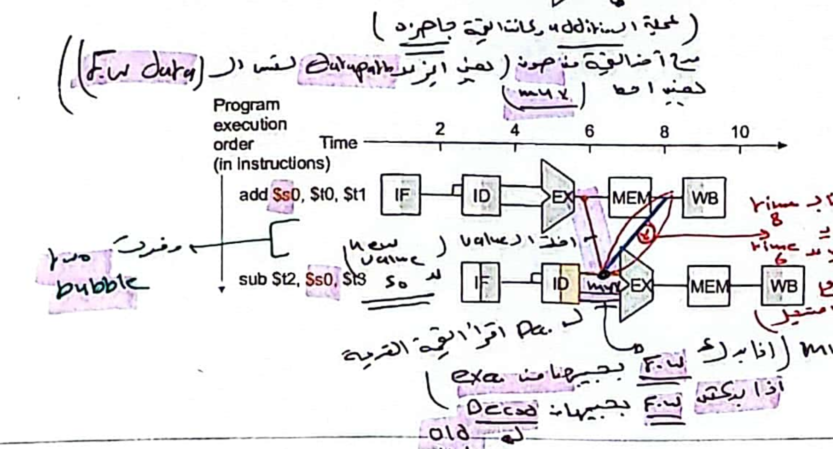
الممسوحة ضوئيا بـ CamScanner

Data Hazards

- *Data hazard*: instruction needs data from the result of a previous instruction still executing in pipeline
- Solution *Forward* data if possible…



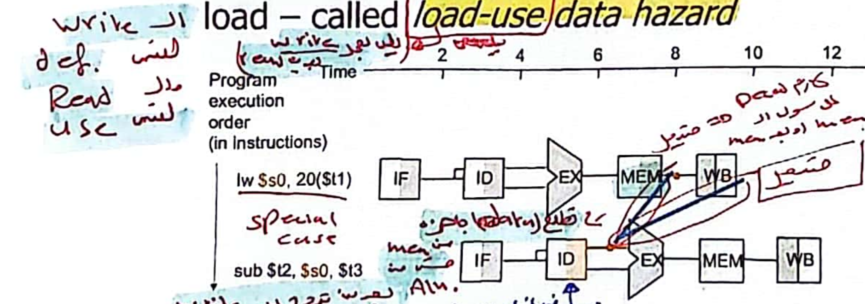
Instruction pipeline diagram: shade indicates use – left=write, right=read



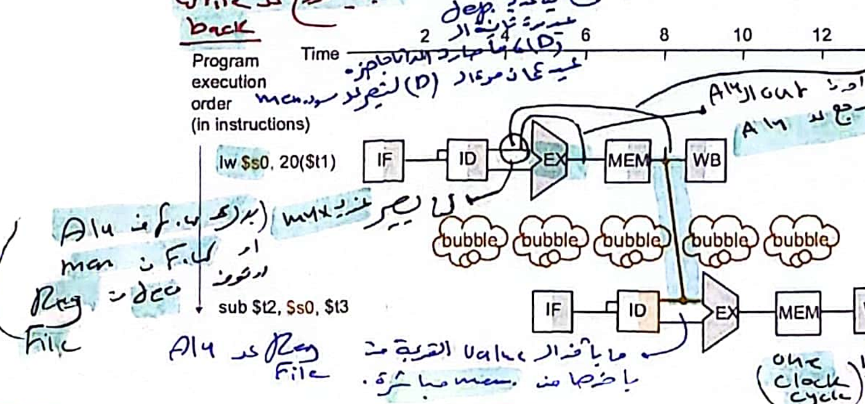
Without forwarding – blue line – data has to go back in time; with forwarding – red line – data is available in time

Data Hazards

- Forwarding may not be enough
  - e.g., if an R-type instruction following a load uses the result of the load – called *load-use data hazard*



Without a stall it is impossible to provide input to the `sub` instruction in time



With a one-stage stall, forwarding can get the data to the `sub` instruction in time

Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!

Program execution order (in instructions) — Time: 200, 400, 600, 800, 1000, 1200, 1400

lw $s0, 20($t1): IF, ID, EX, MEM, WB

bubble bubble bubble bubble bubble

sub $t2, $s0, $t3: IF, ID, EX, MEM, WB

Reordering Code to Avoid Pipeline Stall (Software Solution)

- Example:

```
lw  $t0, 0($t1)
lw  $t2, 4($t1)
sw  $t2, 0($t1)
sw  $t0, 4($t1)
```

Data hazard

- Reordered code:

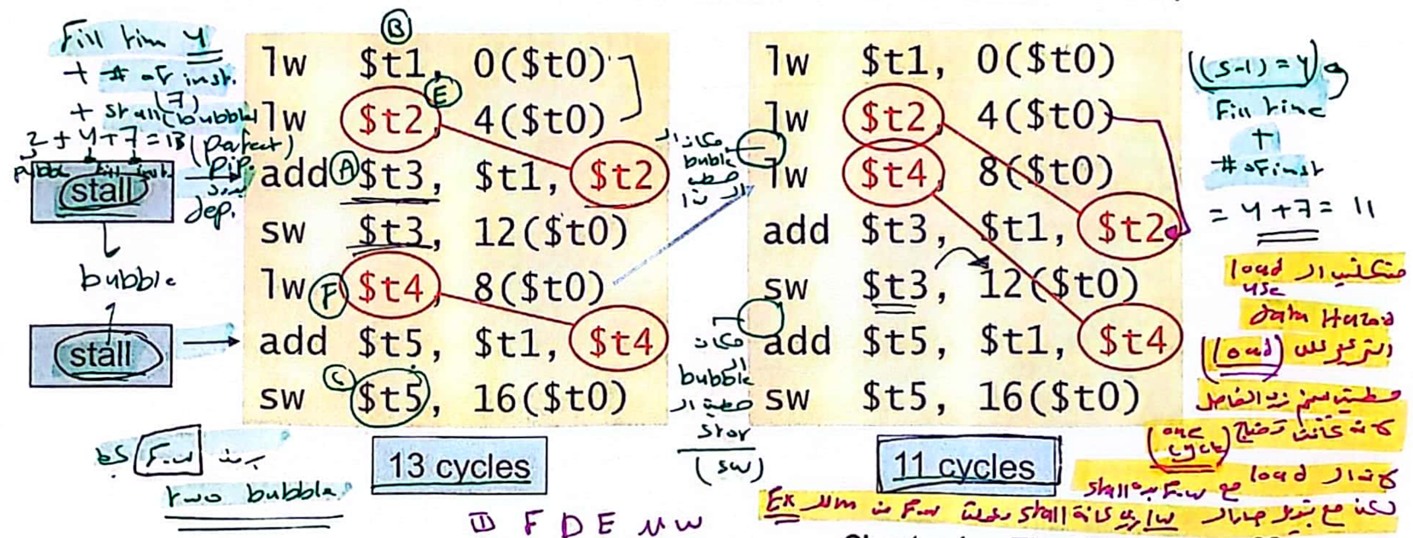
```
lw  $t0, 0($t1)
lw  $t2, 4($t1)
sw  $t0, 4($t1)
sw  $t2, 0($t1)
```

Interchanged

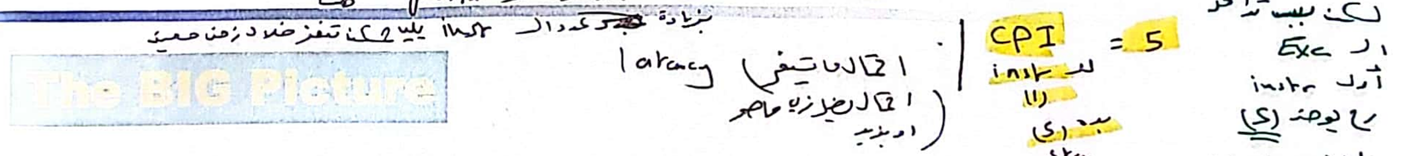
الممسوحة ضوئيا بـ CamScanner

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`



Pipeline Summary



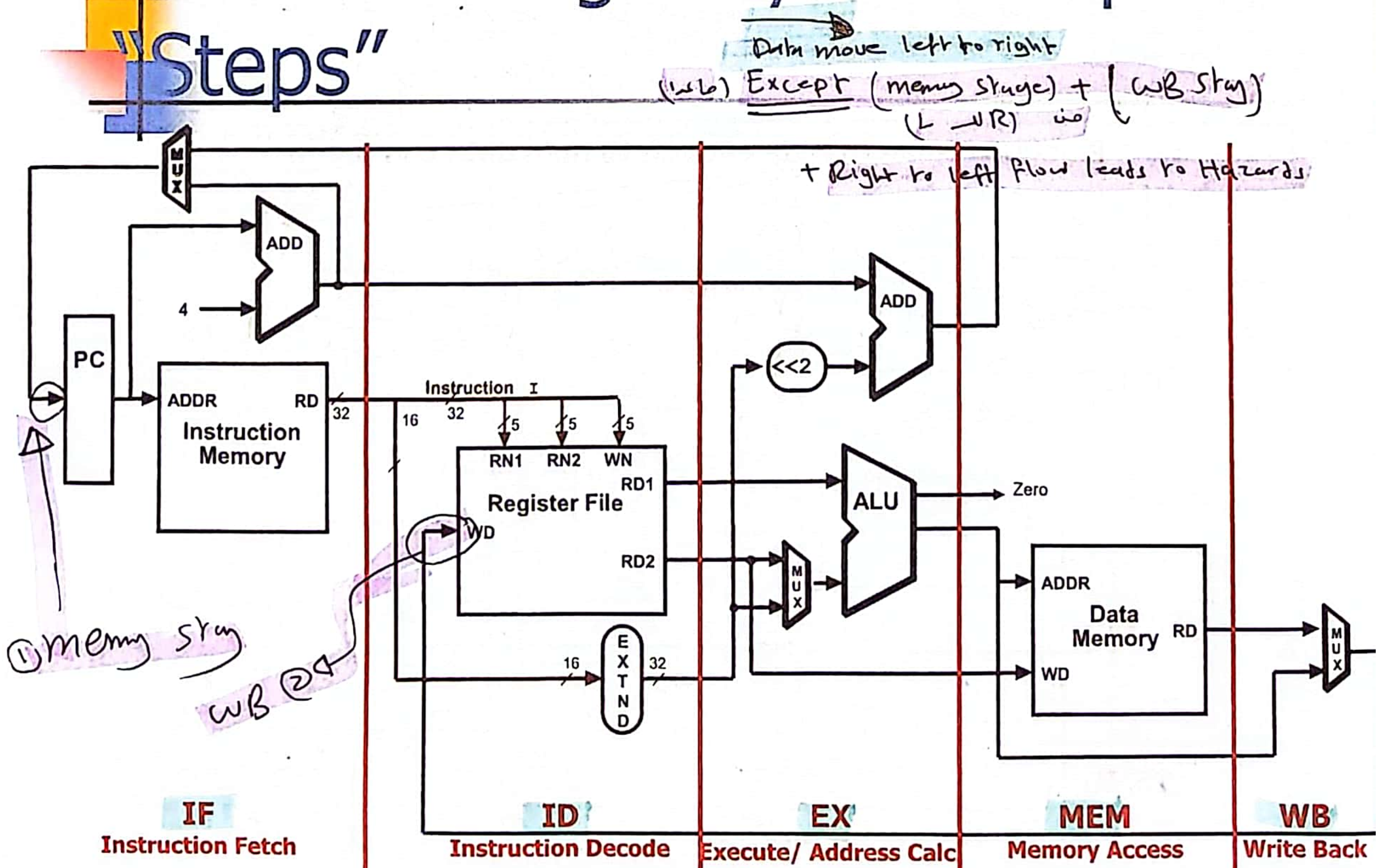
- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

CamScanner الممسوحة ضوئيا بـ

Pipelined Datapath

- We now move to actually building a pipelined datapath
- First recall the 5 steps in instruction execution
  1. Instruction Fetch & PC Increment (IF)
  2. Instruction Decode and Register Read (ID)
  3. Execution or calculate address (EX)
  4. Memory access (MEM)
  5. Write result into register (WB)
- Review: single-cycle processor
  - all 5 steps done in a single clock cycle
  - dedicated hardware required for each step
- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*

Review - Single-Cycle Datapath "Steps"



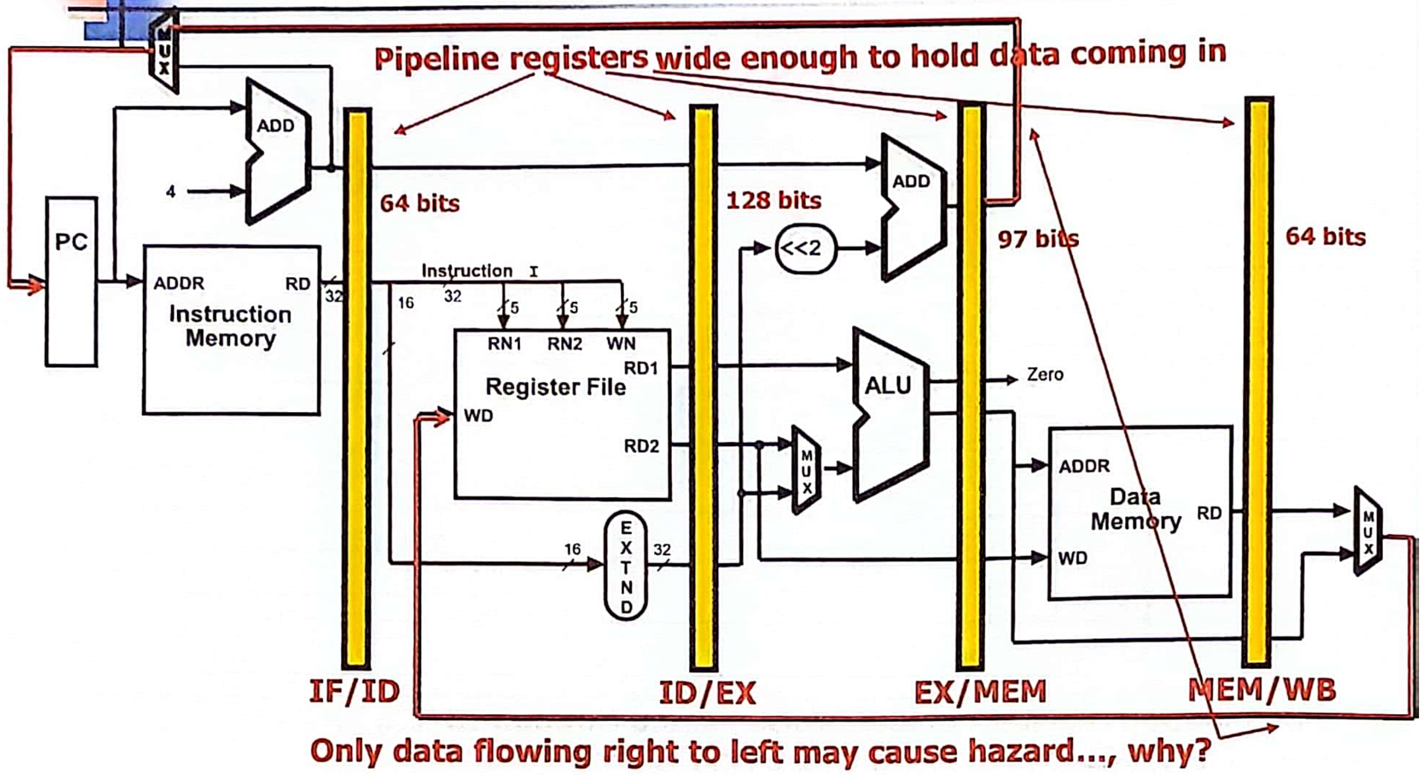
CamScanner الممسوحة ضوئيا بـ

Pipelined Datapath – Key Idea

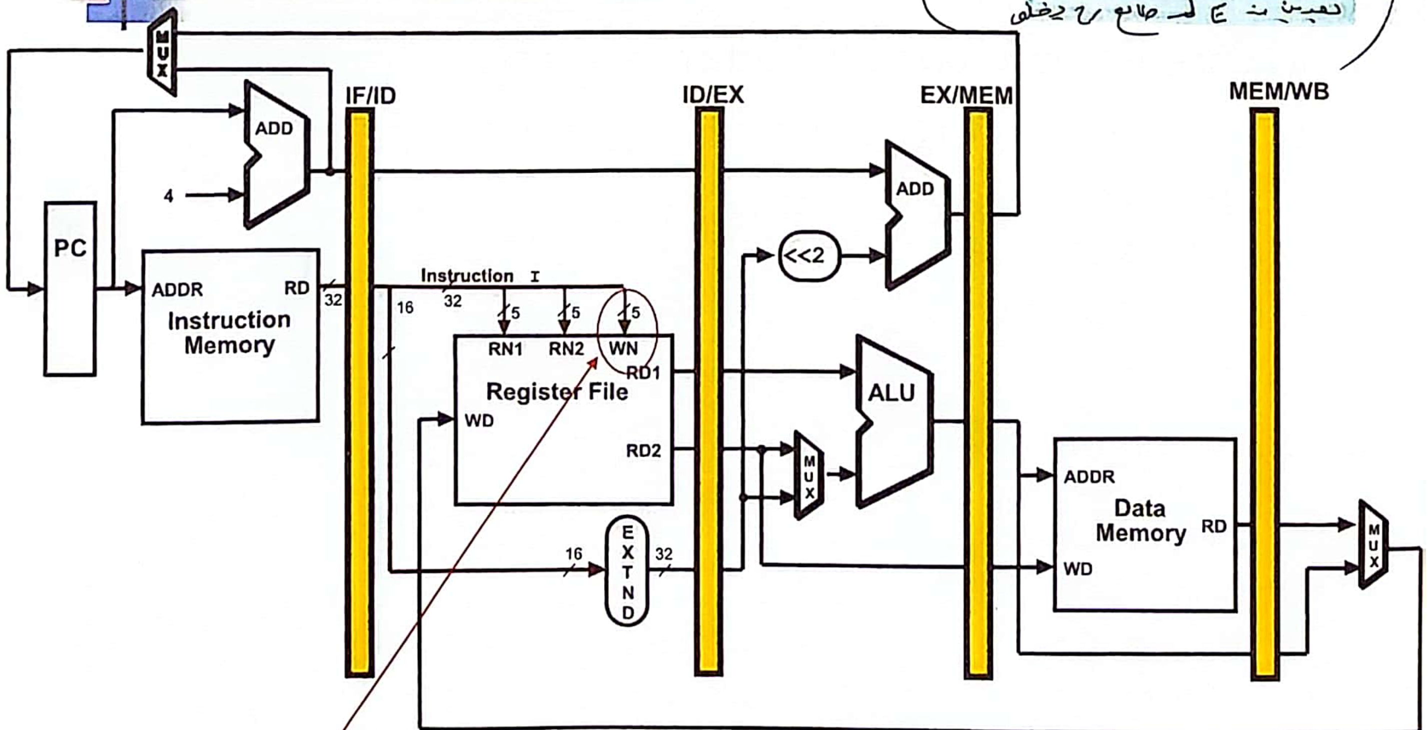
- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*
  - Answer: *We may be able to start executing a new instruction at each clock cycle* - pipelining
- ...but we shall need *extra* registers to hold data between cycles – *pipeline registers*

Pipelined Datapath

Pipelined Datapath



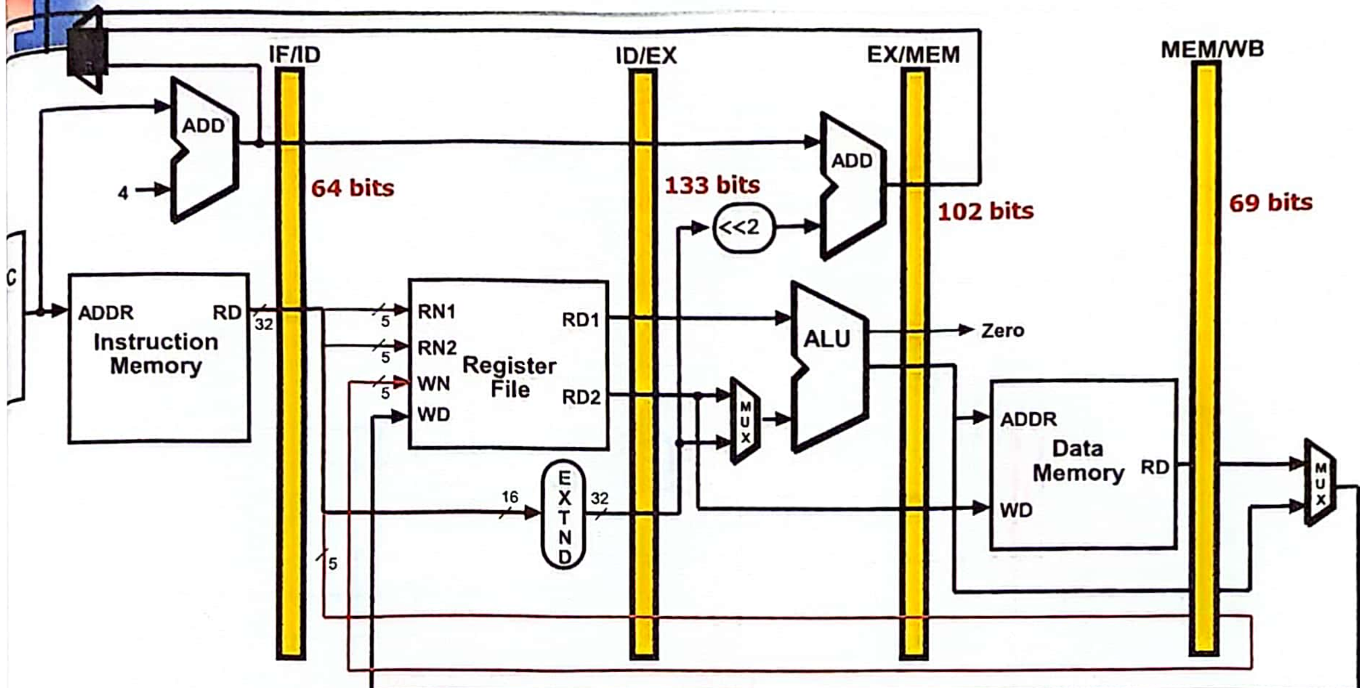
Bug in the Datapath



Write register number comes from another *later* instruction!

CamScanner الممسوحة ضوئيا بـ

Corrected Datapath



Destination register number is also passed through ID/EX, EX/MEM and MEM/WB registers, which are now wider by 5 bits

Pipelined Example

- Consider the following instruction sequence:

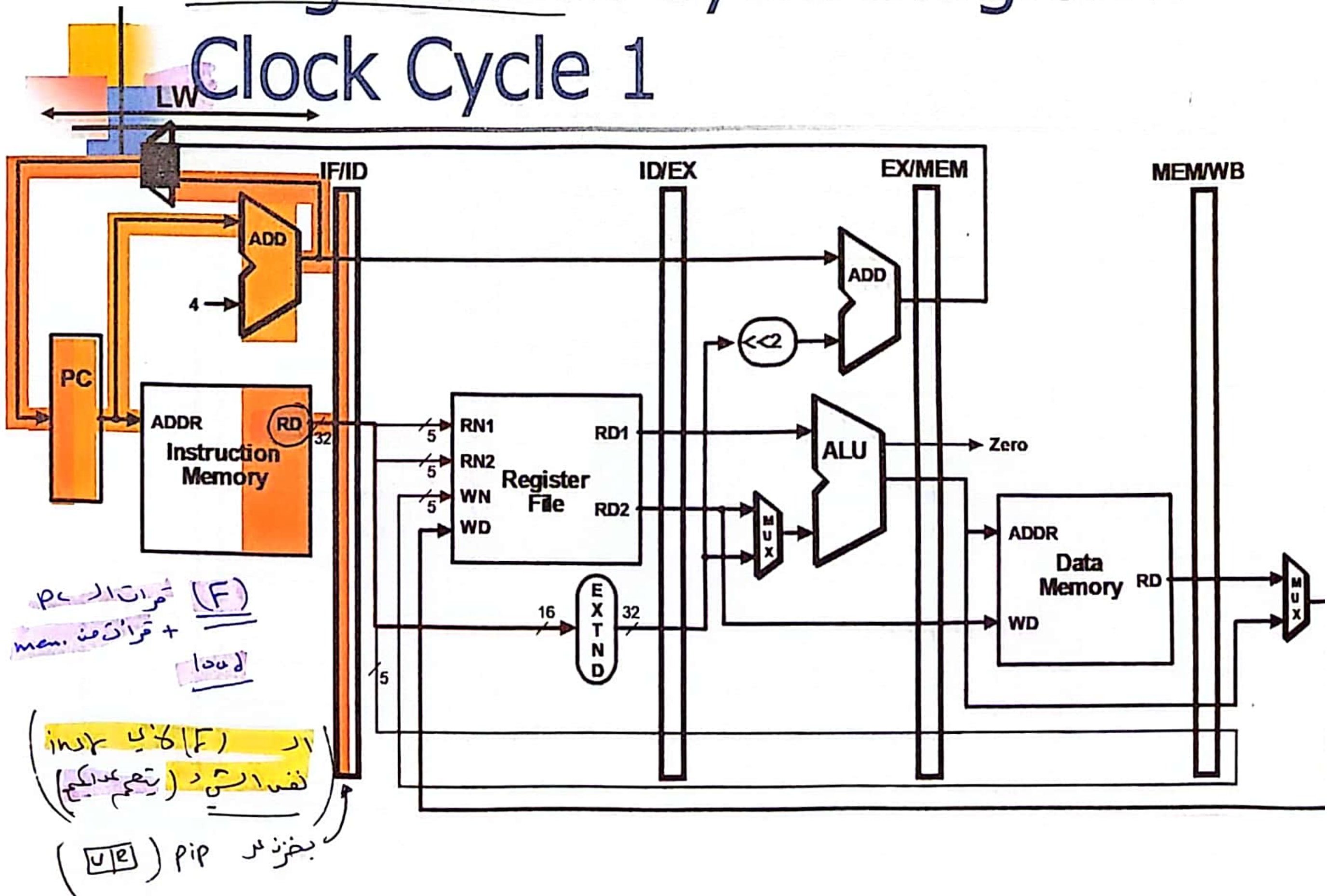
```
lw  $t0,  10($t1)
sw  $t3, 20($t4)
add $t5,  $t6,  $t7
sub $t8,  $t9,  $t10
```

CamScanner الممسوحة ضوئيا بـ

Pipeline Operation

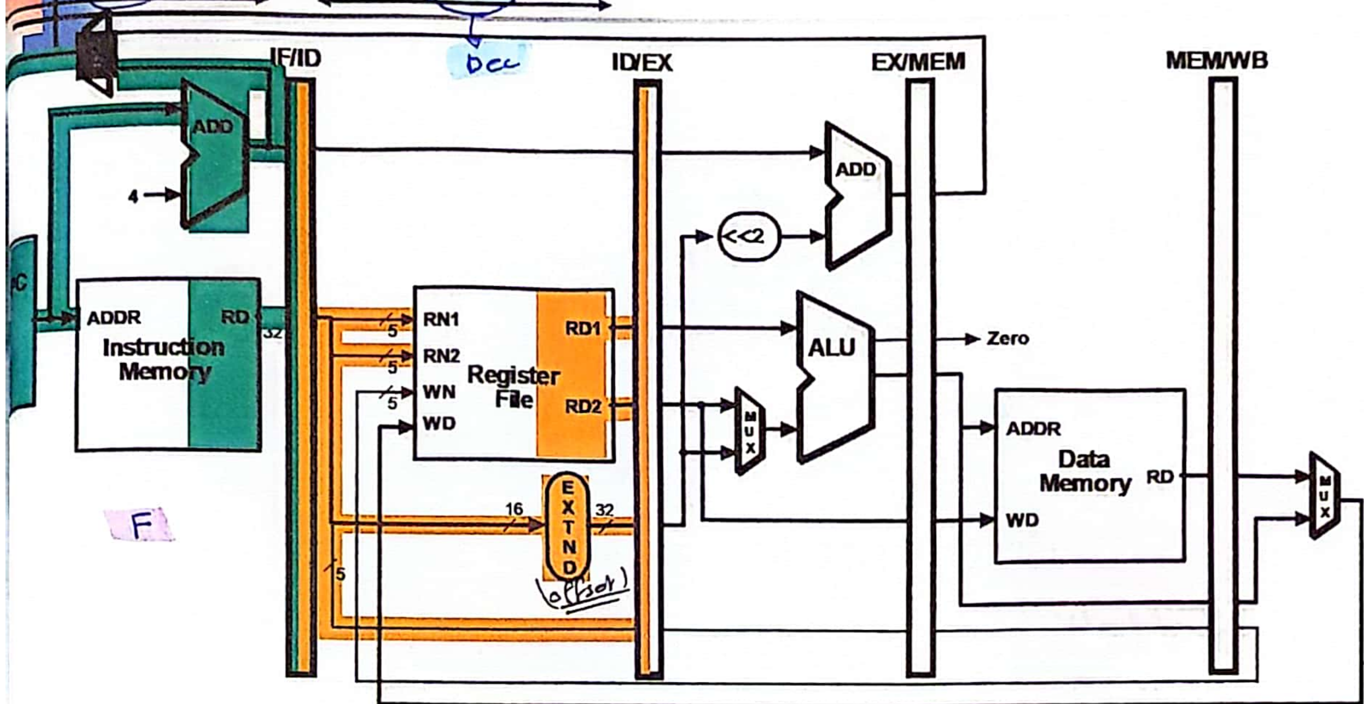
- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store

Single-Clock-Cycle Diagram: Clock Cycle 1

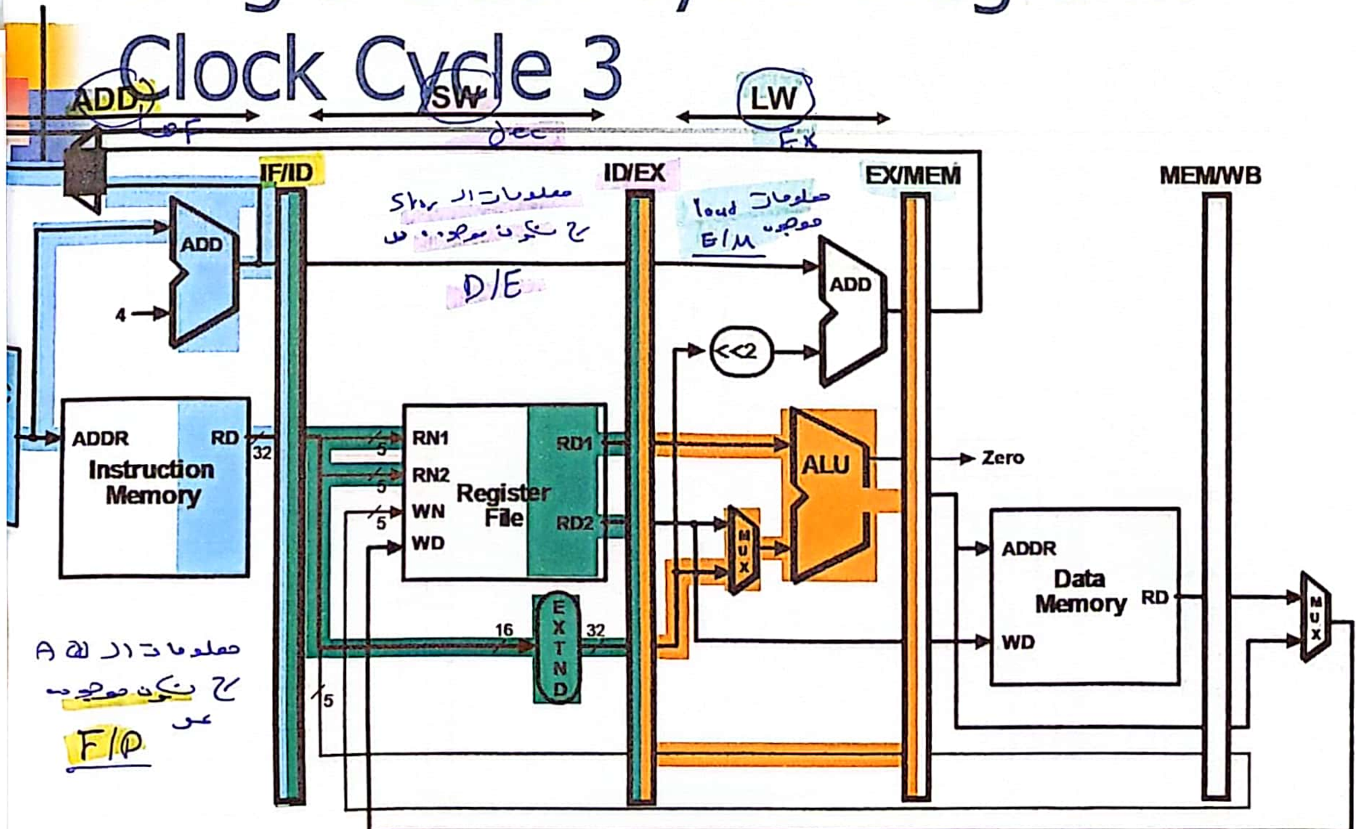


المسوحة ضوئيا بـ CamScanner

Single-Clock-Cycle Diagram: Clock Cycle 2

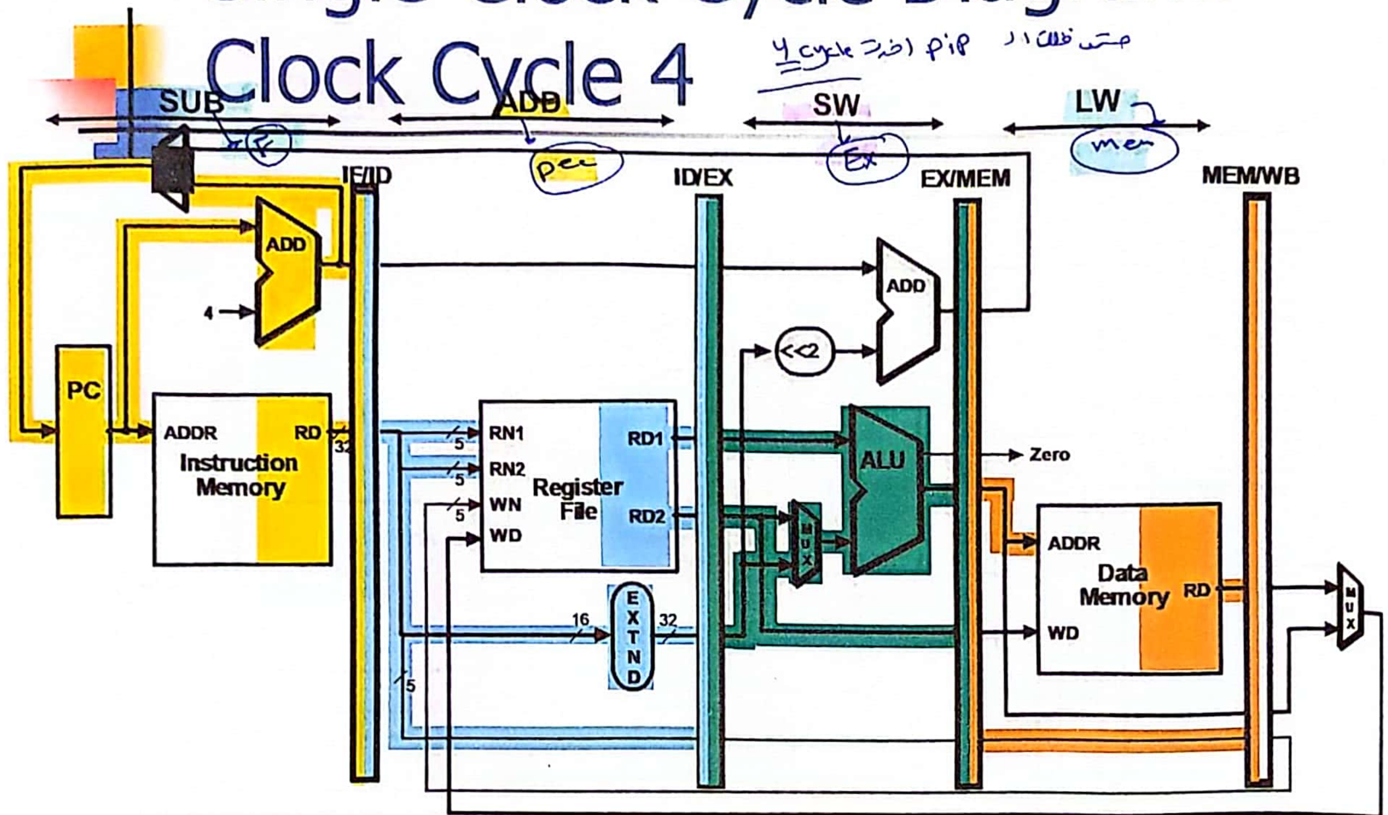


Single-Clock-Cycle Diagram: Clock Cycle 3



الممسوحة ضوئيا بـ CamScanner

Single-Clock-Cycle Diagram: Clock Cycle 4



(Fill time = stages − 1
= 5 − 1 = 4)

Single-Clock-Cycle Diagram: Clock Cycle 5

SUB ADD SW LW

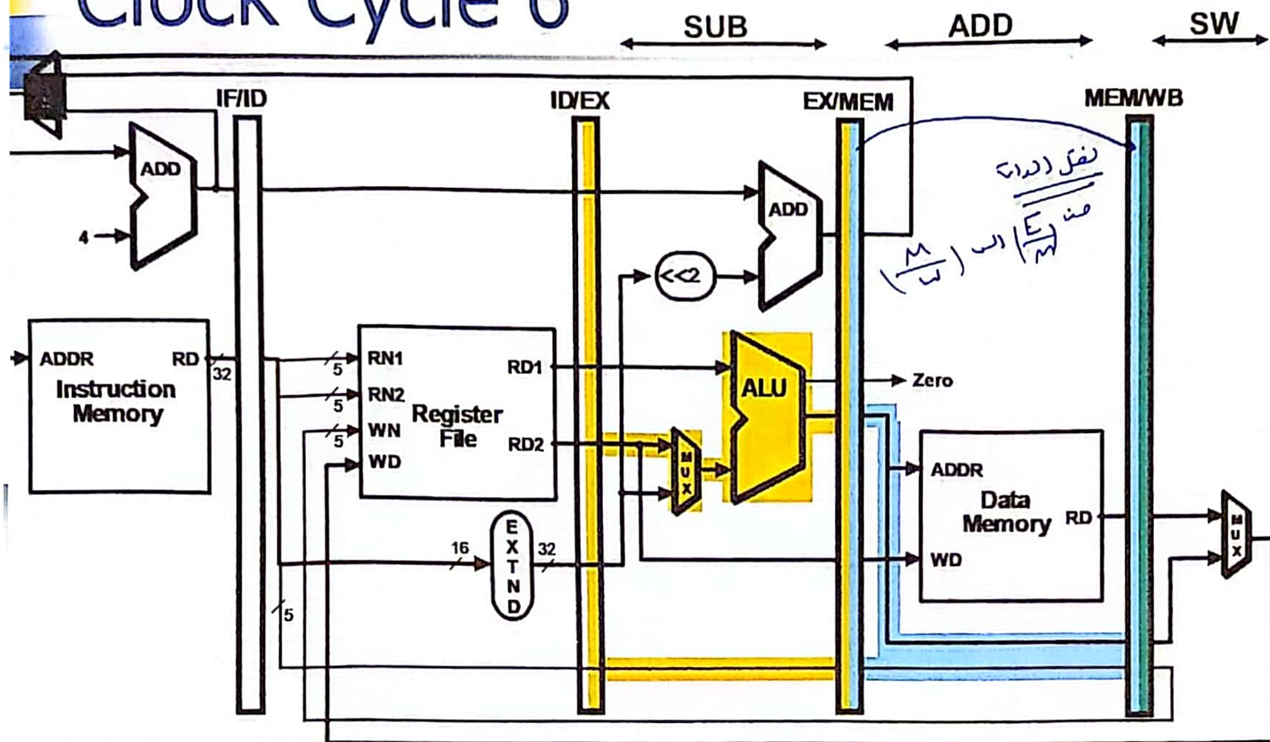
IF/ID ID/EX EX/MEM MEM/WB

ADD 4 PC ADDR RD 32 Instruction Memory RN1 RN2 WN WD Register File RD1 RD2 5 16 EXTND 32 <<2 ADD ALU Zero ADDR Data Memory RD WD MUX

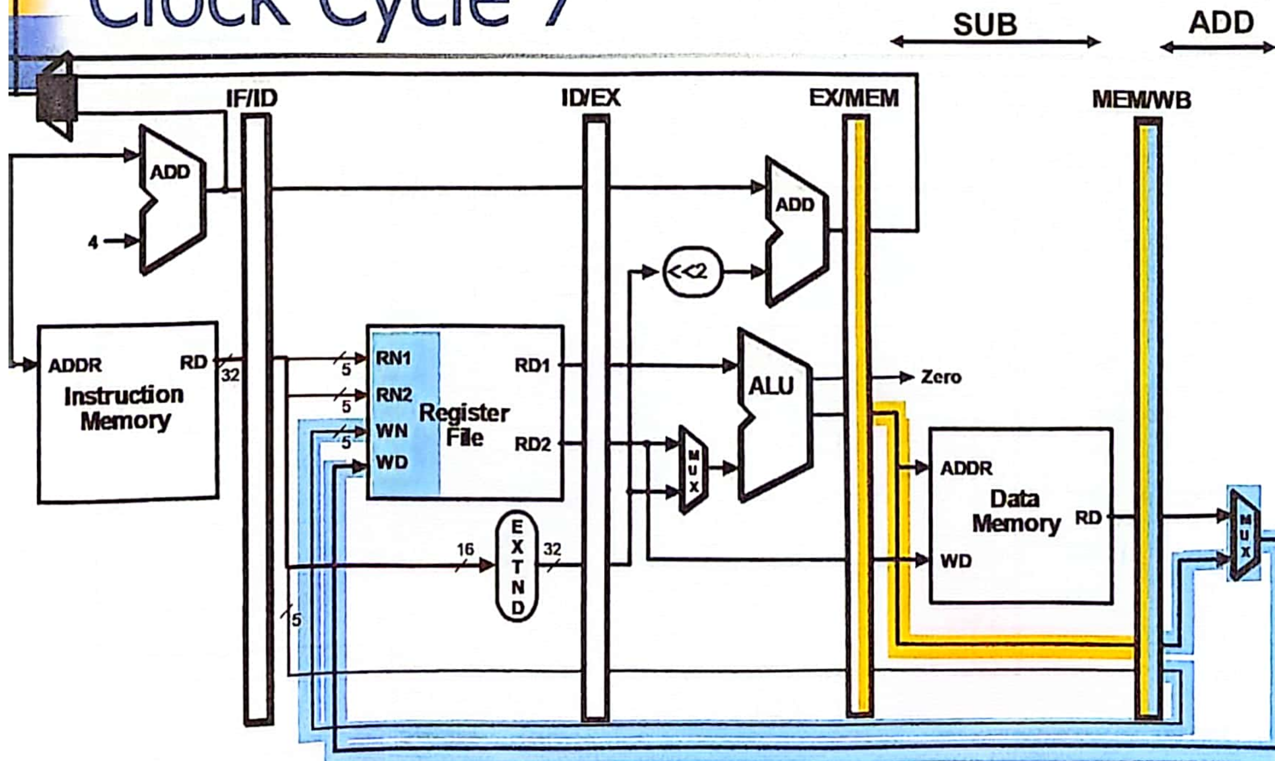
Wrong Reg number?! why

5bit Reg #

Single-Clock-Cycle Diagram: Clock Cycle 6

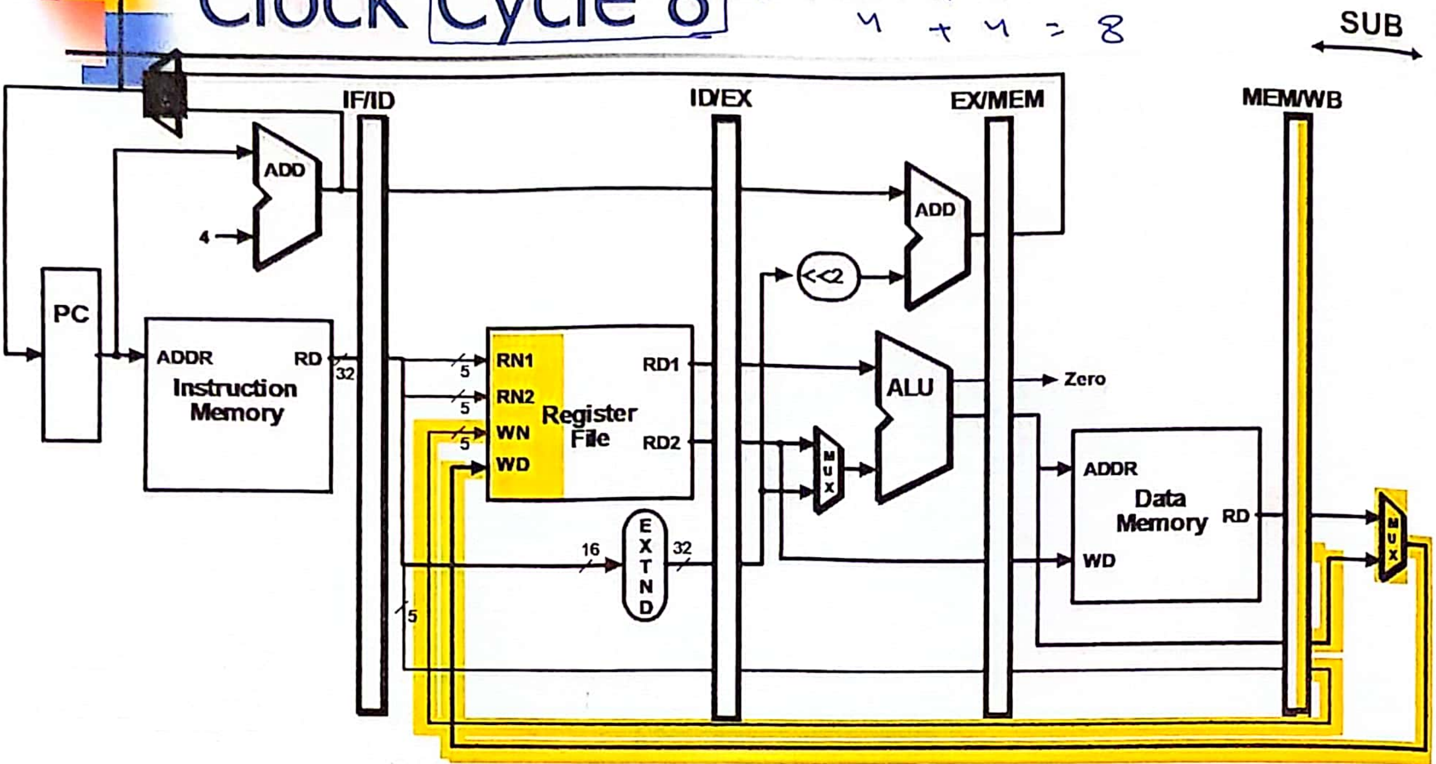


Single-Clock-Cycle Diagram: Clock Cycle 7

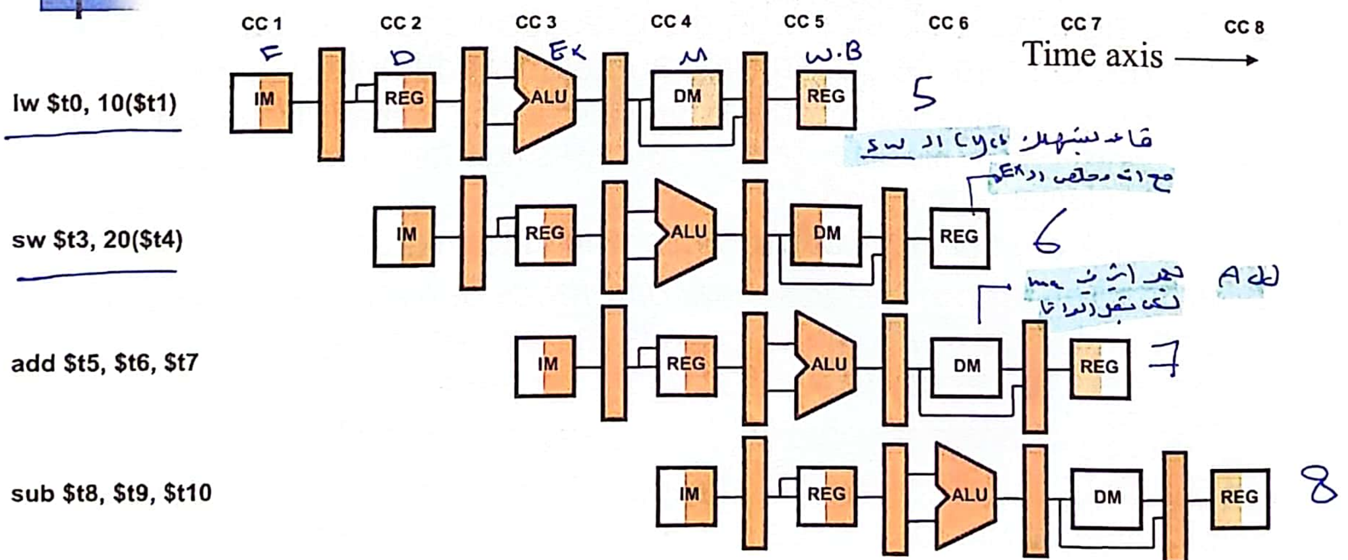


المسوحة ضوئيا بـ CamScanner

Single-Clock-Cycle Diagram: Clock Cycle 8



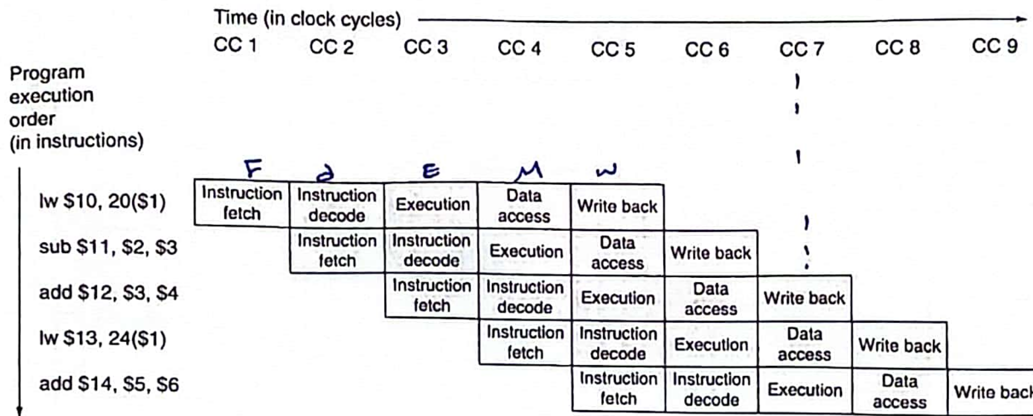
Alternative View – Multiple-Clock-Cycle Diagram



المسوحة ضوئيا بـ CamScanner

Multi-Cycle Pipeline Diagram

- Traditional form

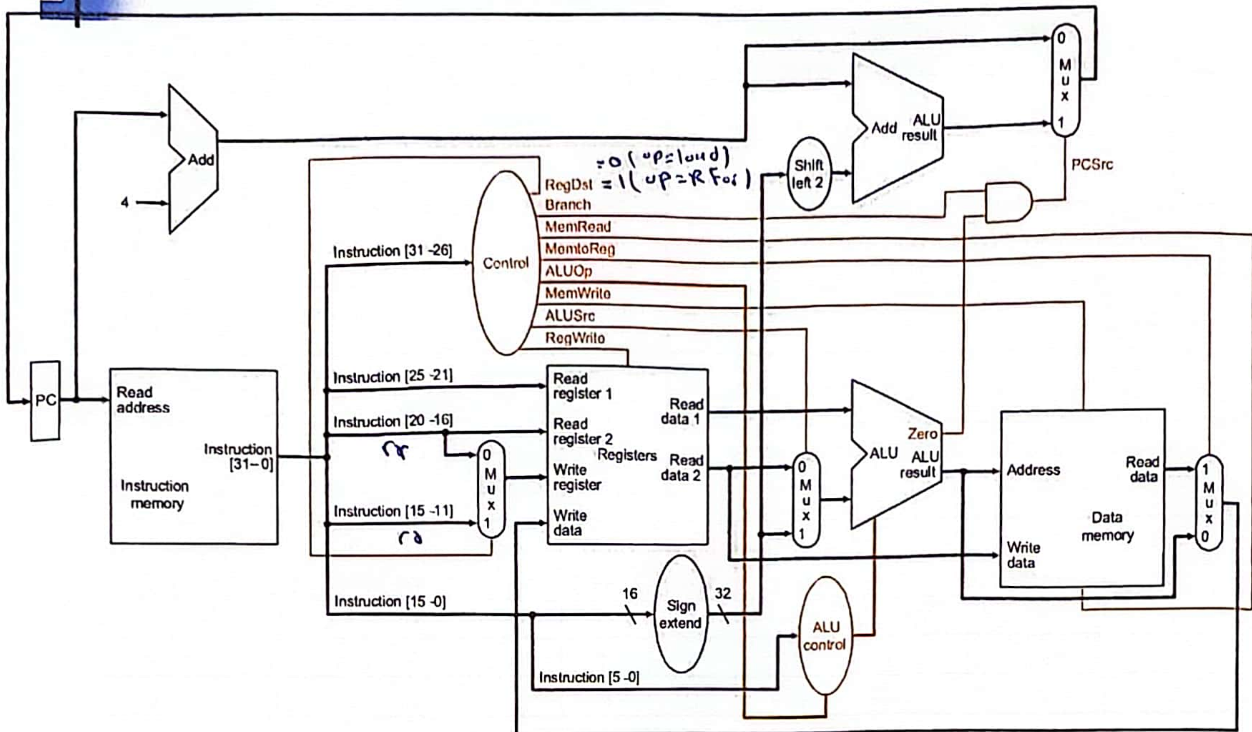


Notes

- One significant difference in the execution of an R-type instruction between multicycle and pipelined implementations:
  - register write-back for the R-type instruction is the 5th (the last write-back) pipeline stage vs. the 4th stage for the multicycle implementation. *Why?*
  - think of *structural hazards* when writing to the register file…
- Worth repeating: the *essential difference* between the pipeline and multicycle implementations is the insertion of pipeline registers to *decouple the 5 stages*
- The CPI of an *ideal pipeline* (no stalls) is 1. *Why?*
- The RaVi Architecture Visualization Project of Dortmund U. has pipeline simulations – see link in our Additional Resources page
- As we develop control for the pipeline keep in mind that the text *does not consider* `jump` – should not be too hard to implement!

CamScanner الممسوحة ضوئيا بـ

Recall Single-Cycle Control – the Datapath



Recall Single-Cycle – ALU Control

Instruction opcode	AluOp	Instruction operation	Funct Field	Desired ALU action	ALU control input
LW	00	load word	xxxxxx	add	010
SW	00	store word	xxxxxx	add	010
Branch eq	01	branch eq	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less	101010	set on less	111

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Truth table for ALU control bits

CamScanner الممسوحة ضوئيا بـ

ecall Single-Cycle – Control Signals

Effect of control bits

ignal Name	Effect when deasserted	Effect when asserted
egDst	The register destination number for the Write register comes from the rt field (bits 20-16)	The register destination number for the Write register comes from the rd field (bits 15-11)
egWrite	None	The register on the Write register input is written with the value on the Write data input
LUSrc	The second ALU operand comes from the second register file output (Read data 2)	The second ALU operand is the sign-extended, lower 16 bits of the instruction
Src	The PC is replaced by the output of the adder that computes the value of PC + 4	The PC is replaced by the output of the adder that computes the branch target
emRead	None	Data memory contents designated by the address input are put on the first Read data output
emWrite	None	Data memory contents designated by the address input are replaced by the value of the Write data input
emtoReg	The value fed to the register Write data input comes from the ALU	The value fed to the register Write data input comes from the data memory

er-
ing
trol
s

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Pipeline Control

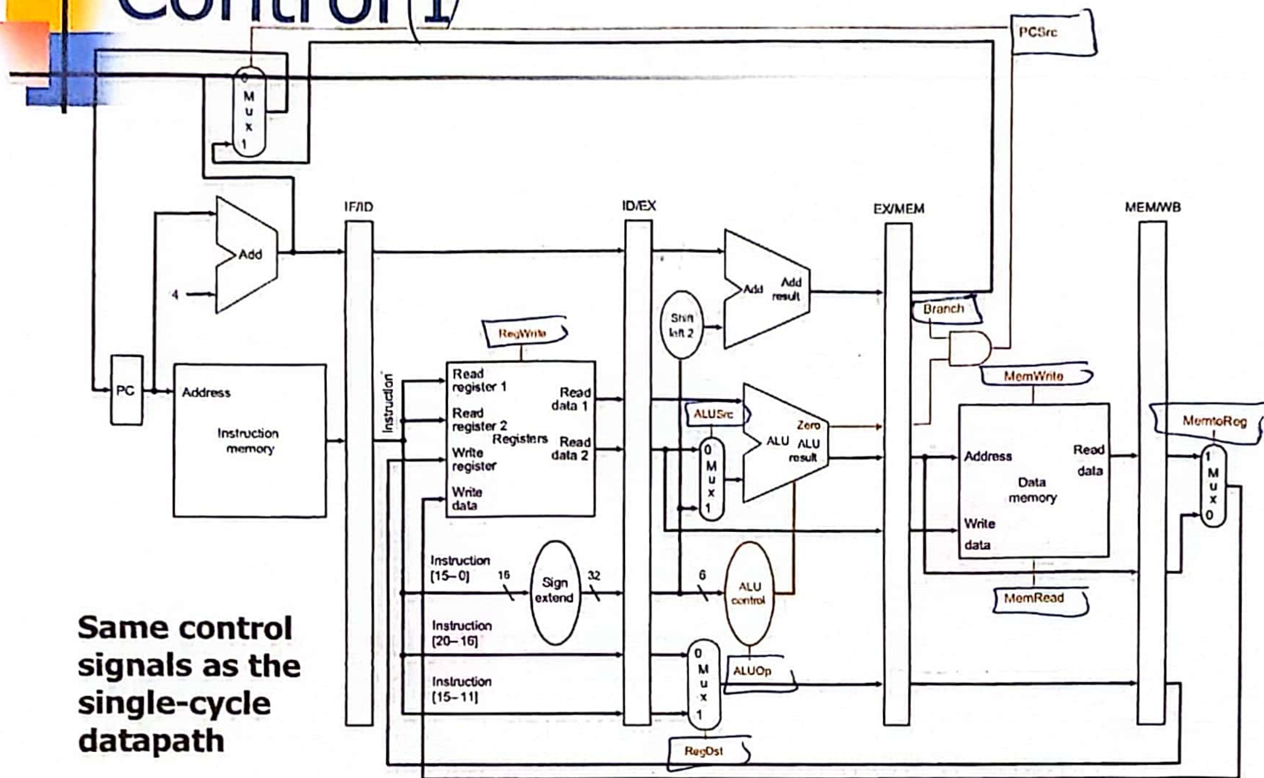
- Initial design – *motivated by single-cycle datapath control* – use the *same* control signals
- Observe:
  - No separate write signal for the PC as it is written every cycle
  - No separate write signals for the pipeline registers as they are written every cycle
  - *No separate read signal for instruction memory* as it is read every clock cycle
  - *No separate read signal for register file* as it is read every clock cycle

Will be modified by hazard detection unit!!

- Need to *set control signals during each pipeline stage*
- Since control signals are associated with components active during a single pipeline stage, can *group control lines into five groups according to pipeline stage*

CamScanner الممسوحة ضوئيا بـ

Pipelined Datapath with Control I



Same control signals as the single-cycle datapath

Pipeline Control Signals

- There are five stages in the pipeline
  - *instruction fetch / PC increment*
  - *instruction decode / register fetch*
  - *execution / address calculation*
  - *memory access*
  - *write back*

Nothing to control as instruction memory read and PC write are always enabled

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

الممسوحة ضوئيا بـ CamScanner

Pipeline Control Implementation

- *Pass control signals along just like the data* – extend each pipeline register to hold needed control bits for succeeding stages

Instruction
Control
WB
M
EX
WB
M
WB
IF/ID
ID/EX
EX/MEM
MEM/WB
64

- *Note*: The 6-bit *funct field* of the instruction required in the EX stage to generate ALU control can be retrieved as the 6 least significant bits of the immediate field which is sign-extended and passed from the IF/ID register to the ID/EX register

Pipelined Datapath with Control II

PCSrc
Mux
Add
4
PC
Address
Instruction memory
IF/ID
ID/EX
Control
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALUSrc
Zero
ALU
ALU result
Branch
MemWrite
Address
Data memory
Read data
Write data
MemtoReg
MemRead
Instruction [15–0]
16
Sign extend
32
6
ALU control
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst

Control signals emanate from the control portions of the pipeline registers

المسوحة ضوئيا بـ CamScanner

Pipelined Execution and Control

- Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```

Label "before<i>" means i th instruction before lw

IF: lw $10, 20($1) | ID: before<1> | EX: before<2> | MEM: before<3> | WB: before<4>

Clock cycle 1

IF: sub $11, $2, $3 | ID: lw $10, 20($1) | EX: before<1> | MEM: before<2> | WB: before<3>

Clock cycle 2

Pipelined Execution and Control

- Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```

IF: and $12, $4, $5 | ID: sub $11, $2, $3 | EX: lw $10, . . . | MEM: before<1> | WB: before<2>

Clock cycle 3

IF: or $13, $6, $7 | ID: and $12, $2, $3 | EX: sub $11, . . . | MEM: lw $10, . . . | WB: before<1>

Clock cycle 4

المسوحة ضوئيا بـ CamScanner

Pipelined Execution and Control

- Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```

Label "after<i>" means i th instruction after add

IF: add $14, $8, $0 | ID: or $13, $6, $7 | EX: and $12, . . . | MEM: sub $11, . . . | WB: lw $10, . . .

Clock cycle 5

IF: after<1> | ID: add $14, $8, $9 | EX: or $13, . . . | MEM: and $12, . . . | WB: sub $11, . . .

Clock cycle 6

Pipelined Execution and Control

- Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```

IF: after<2> | ID: after<1> | EX: add $14, . . . | MEM: or $13, . . . | WB: and $12, . . .

Clock cycle 7

IF: after<3> | ID: after<2> | EX: after<1> | MEM: add $14, . . . | WB: or $13, . . .

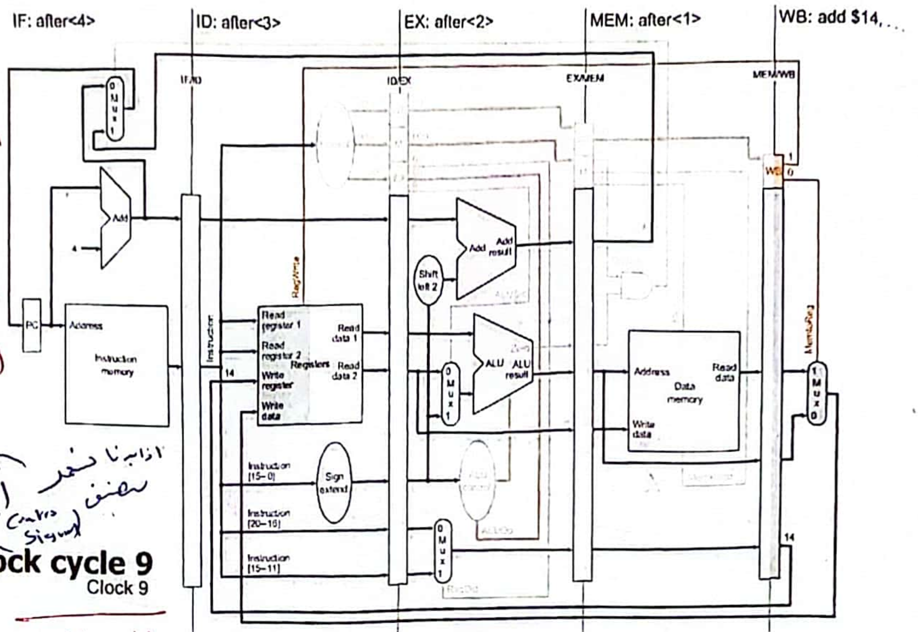
Clock cycle 8

الممسوحة ضوئيا بـ CamScanner

Pipelined Execution and Control

- Instruction sequence:

```
lw   $10, 20($1)
sub  $11, $2, $3
and  $12, $4, $7
or   $13, $6, $7
add  $14, $8, $9
```



Revisiting Hazards

- So far our datapath and control have ignored hazards
- We shall revisit *data hazards* and *control hazards* and enhance our datapath and control to handle them in *hardware*...

CamScanner المسوحة ضوئيا بـ

Data Hazards and Forwarding

- Problem with starting an instruction before previous are finished:
  - data dependencies that go backward in time – called *data hazards*

$2 = 10 before sub;
$2 = -20 after sub

Time (in clock cycles)

Value of register $2:	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
	10	10	10	10	10/–20	–20	–20	–20	–20

Program execution order (in instructions)

sub $2, $1, $3 — IM, Reg, ALU, DM, Reg
and $12, $2, $5 — IM, Reg, ALU, DM, Reg
or $13, $6, $2 — IM, Reg, ALU, DM, Reg
add $14, $2, $2 — IM, Reg, ALU, DM, Reg
sw $15, 100($2) — IM, Reg, ALU, DM, Reg

```
sub  $2,   $1,  $3
and  $12,  $2,  $5
or   $13,  $6,  $2
add  $14,  $2,  $2
sw   $15,  100($2)
```

Software Solution

- Have compiler guarantee *never* any data hazards!
  - by *rearranging instructions to insert independent instructions between instructions* that would otherwise have a data hazard between them,
  - or, if such rearrangement is not possible, *insert* nops

```
sub  $2,  $1,  $3              sub  $2,  $1,  $3
lw   $10, 40($3)               nop
slt  $5,  $6,  $7              nop
and  $12, $2,  $5       or     and  $12, $2,  $5
or   $13, $6,  $2              or   $13, $6,  $2
add  $14, $2,  $2              add  $14, $2,  $2
sw   $15, 100($2)              sw   $15, 100($2)
```

- Such compiler solutions may not always be possible, and nops slow the machine down

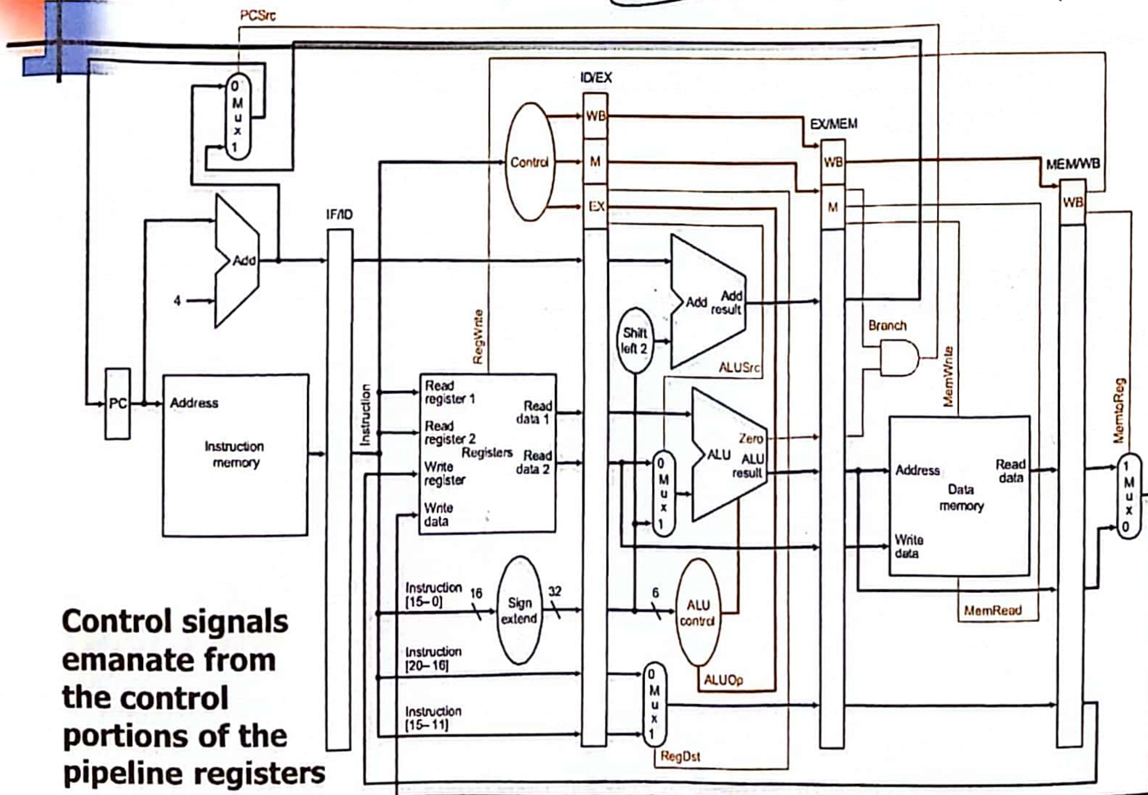
MIPS: nop = "no operation" = 00…0 (32bits) = sll $0, $0, 0

CamScanner المسموحة ضوئيا بـ

Hardware Solution: Forwarding

- Idea: *use intermediate data,* do not wait for result to be finally written to the destination register. Two steps:
  1. *Detect* data hazard
  2. *Forward* intermediate data to resolve hazard

Pipelined Datapath with Control II (as before)



Control signals emanate from the control portions of the pipeline registers

الممسوحة ضوئيا بـ CamScanner

Hazard Detection

- Hazard conditions:

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

- Eg., in the earlier example, first hazard between `sub $2, $1, $3` and `and $12, $2, $5` is detected when the `and` is in EX stage and the `sub` is in MEM stage because
  - `EX/MEM.RegisterRd = ID/EX.RegisterRs = $2 (1a)`

- Whether to forward also depends on:
  - *if the later instruction is going to write a register* – if *not*, no need to forward, even if there is register number match as in conditions above
  - *if the destination register of the later instruction is $0* – in which case there is no need to forward value ($0 is always 0 and never overwritten)

Data Forwarding

- Plan:
  - *allow inputs to the ALU not just from ID/EX, but also later pipeline registers,* and
  - *use multiplexors and control signals to choose appropriate inputs* to ALU

```
sub $2,  $1,  $3
and $12, $2,  $5
or  $13, $6,  $2
add $14, $2,  $2
sw  $15, 100($2)
```

Time (in clock cycles)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM :	X	X	X	–20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	–20	X	X	X	X

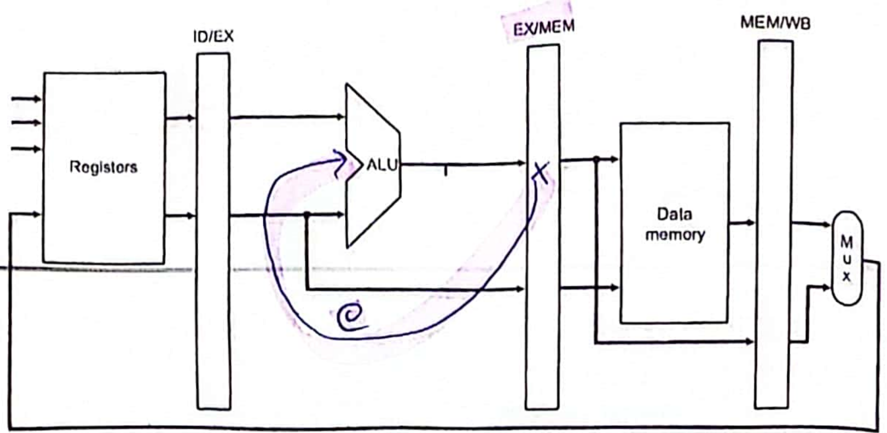
Program execution order (in instructions)

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)

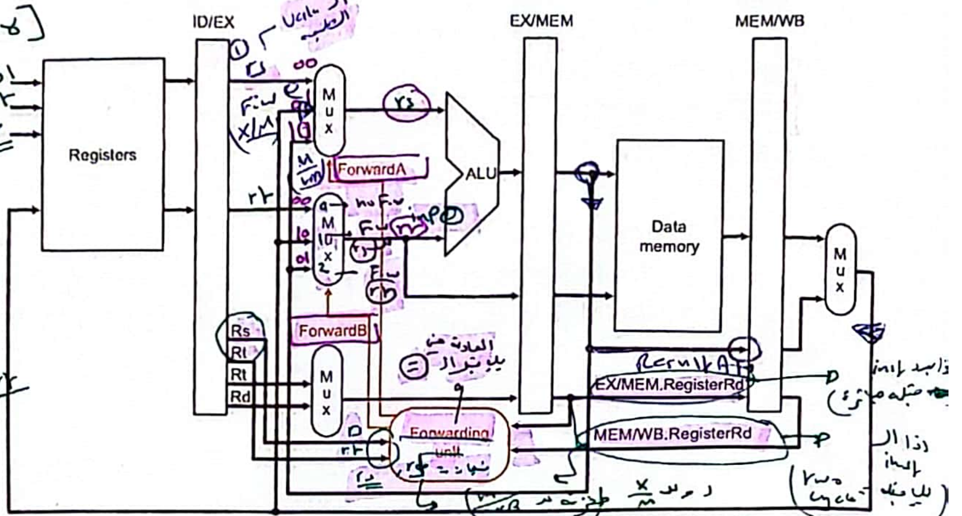
Dependencies between pipelines move forward in time

CamScanner المسوحة ضوئيا بـ

Forwarding Hardware



a. No forwarding **Datapath before adding forwarding hardware**



b. With forwarding **Datapath after adding forwarding hardware**

Forwarding Hardware: Multiplexor Control

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from prior ALU result
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result
ForwardB = 00	ID/EX	The second ALU operand comes from the register file
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from prior ALU result
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result

Depending on the selection in the rightmost multiplexor (see datapath with control diagram)

CamScanner المسوحة ضوئيا بـ

Data Hazard: Detection and Forwarding

- Forwarding unit determines multiplexor control according to the following rules:

1. **EX hazard**

```
if (      EX/MEM.RegWrite                                // if there is a write...
    and ( EX/MEM.RegisterRd ≠ 0 )                        // to a non-$0 register...
    and ( EX/MEM.RegisterRd = ID/EX.RegisterRs ) )       // which matches, then...
ForwardA = 10

if (      EX/MEM.RegWrite                                // if there is a write...
    and ( EX/MEM.RegisterRd ≠ 0 )                        // to a non-$0 register...
    and ( EX/MEM.RegisterRd = ID/EX.RegisterRt ) )       // which matches, then...
ForwardB = 10
```

Data Hazard: Detection and Forwarding

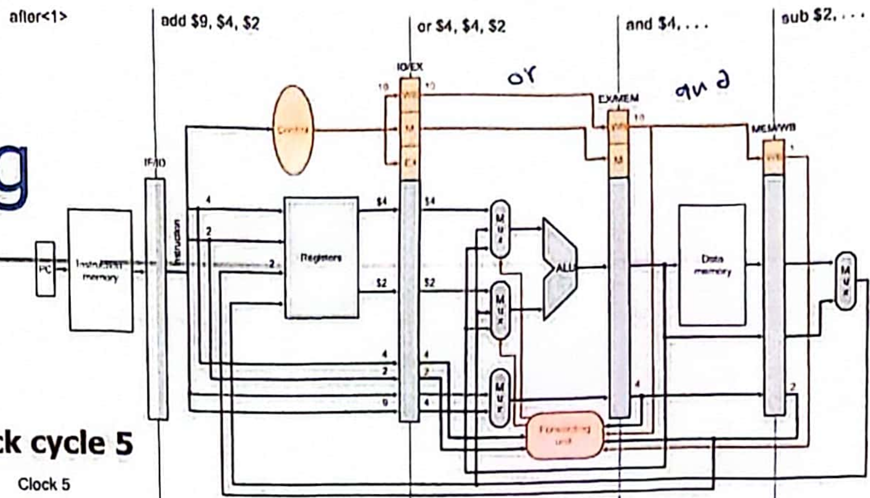
MEM hazard

```
if (      MEM/WB.RegWrite                                // if there is a write...
    and ( MEM/WB.RegisterRd ≠ 0 )                        // to a non-$0 register...
    and ( EX/MEM.RegisterRd ≠ ID/EX.RegisterRs )         // and not already a register match
                                                         // with earlier pipeline register...
    and ( MEM/WB.RegisterRd = ID/EX.RegisterRs ) )       // but match with later pipeline
                                                            register, then...
ForwardA = 01

if (      MEM/WB.RegWrite                                // if there is a write...
    and ( MEM/WB.RegisterRd ≠ 0 )                        // to a non-$0 register...
    and ( EX/MEM.RegisterRd ≠ ID/EX.RegisterRt )         // and not already a register match
                                                         // with earlier pipeline register...
    and ( MEM/WB.RegisterRd = ID/EX.RegisterRt ) )       // but match with later pipeline
                                                            register, then...
ForwardB = 01
```

This check is necessary, e.g., for sequences such as add \$1, \$1, \$2; add \$1, \$1, \$3; add \$1, \$1, \$4; (array summing?), where an earlier pipeline (EX/MEM) register has more recent data

CamScanner الممسوحة ضوئيا بـ

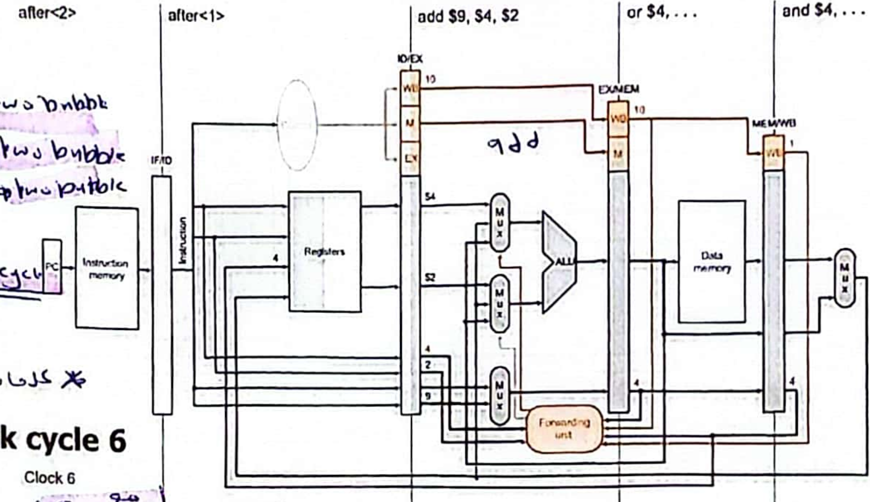
Forwarding



- Execution example (cont.):

Clock cycle 5

```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

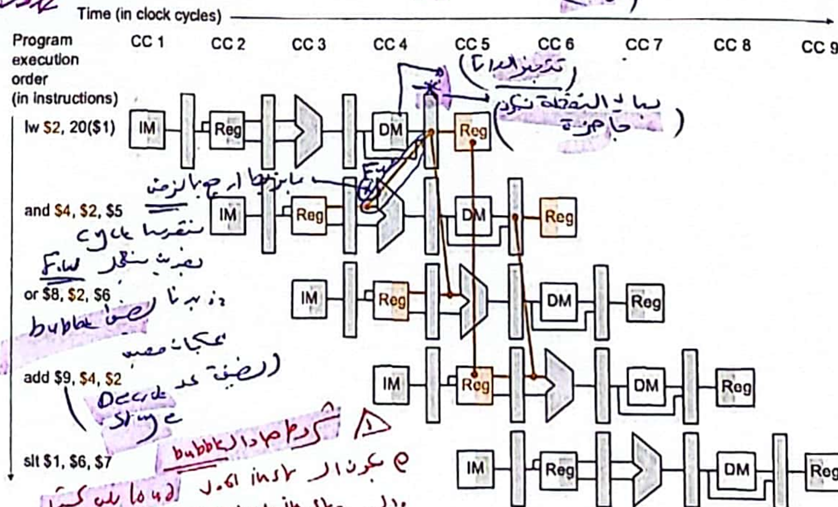


Clock cycle 6

Data Hazards and Stalls

- Load word can still cause a hazard:
  - ~~an instruction tries to read a register following a load instruction that~~ writes to the same register

```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
```

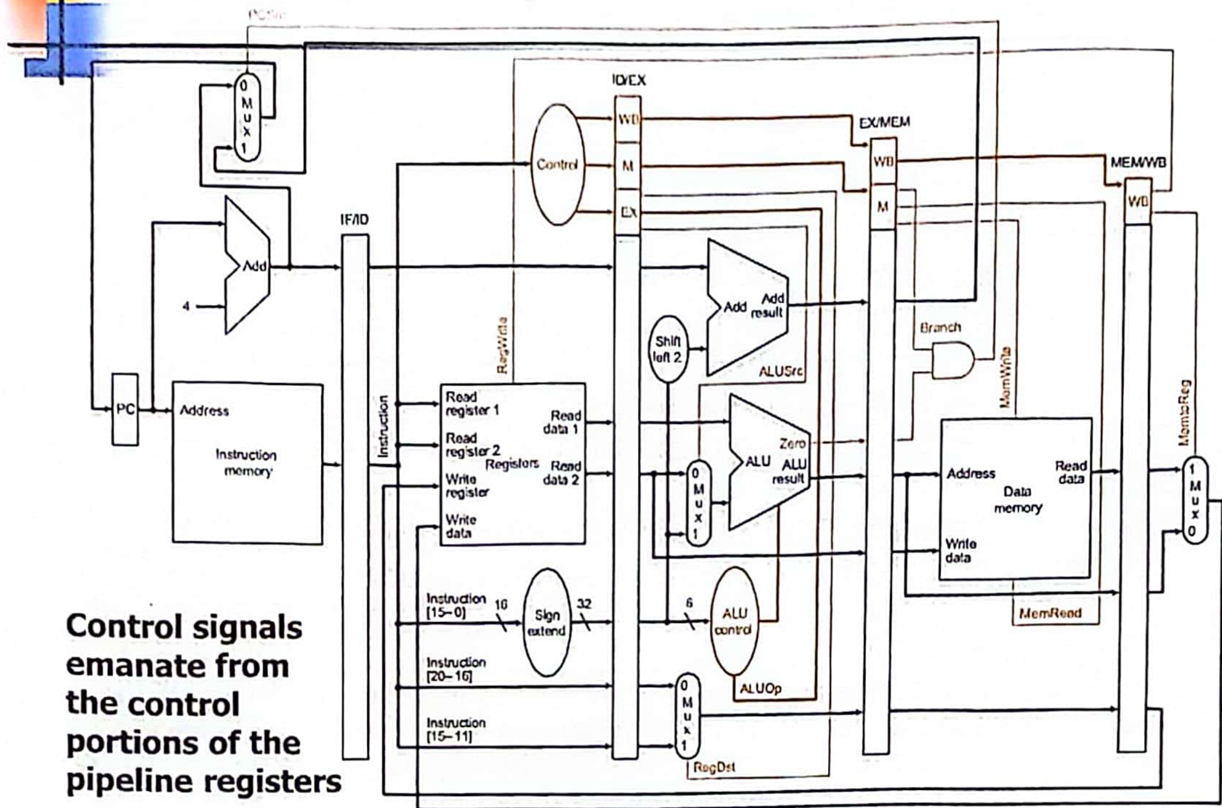


As even a pipeline dependency goes backward in time forwarding will not solve the hazard

- therefore, we need a *hazard detection unit* to *stall* the pipeline after the load instruction

CamScanner الممسوحة ضوئيا بـ

Pipelined Datapath with Control II (as before)



Control signals emanate from the control portions of the pipeline registers

Hazard Detection Logic to Stall

- Hazard detection unit implements the following check if to stall

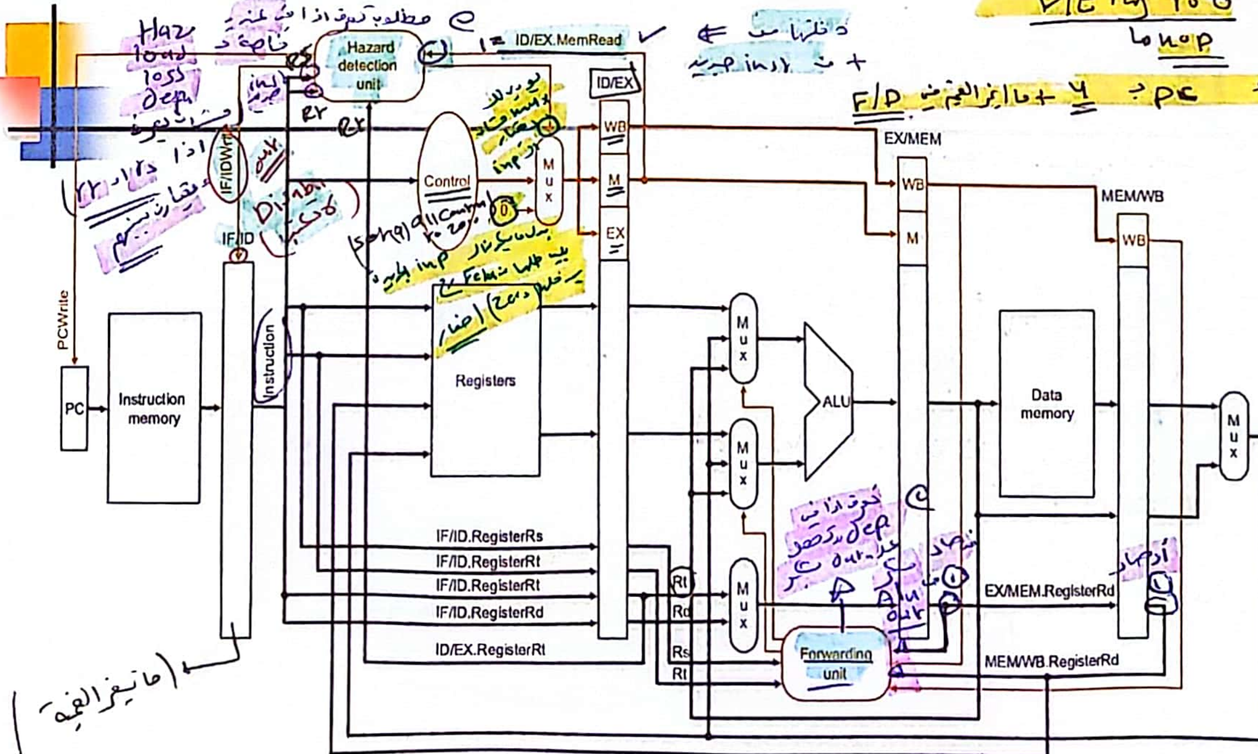
```
if ( ID/EX.MemRead            // if the instruction in the EX stage is a load...
   and ( ( ID/EX.RegisterRt = IF/ID.RegisterRs )   // and the destination register
   or ( ID/EX.RegisterRt = IF/ID.RegisterRt ) ) )  // matches either source register
                              // of the instruction in the ID stage, then...
stall the pipeline
```

الممسوحة ضوئيا بـ CamScanner

Mechanics of Stalling

- If the check to stall verifies, then the *pipeline needs to stall only 1 clock cycle* after the load as after that the forwarding unit can resolve the dependency
- What the hardware does to stall the pipeline 1 cycle:
  - *does not let the IF/ID register change* (*disable write!*) – this will cause the instruction in the ID stage to repeat, i.e., *stall*
  - therefore, the instruction, just behind, in the IF stage must be stalled as well – so hardware *does not let the PC change* (*disable write!*) – this will cause the instruction in the IF stage to repeat, i.e., *stall*
  - *changes all the EX, MEM and WB control fields in the ID/EX pipeline register to 0*, so effectively the instruction just behind the load becomes a `nop` – a *bubble* is said to have been inserted into the pipeline
    - note that we cannot turn that instruction into an `nop` by 0ing all the bits in the instruction itself – recall `nop` = 00…0 (32 bits) – because it has already been decoded and control signals generated

Hazard Detection Unit



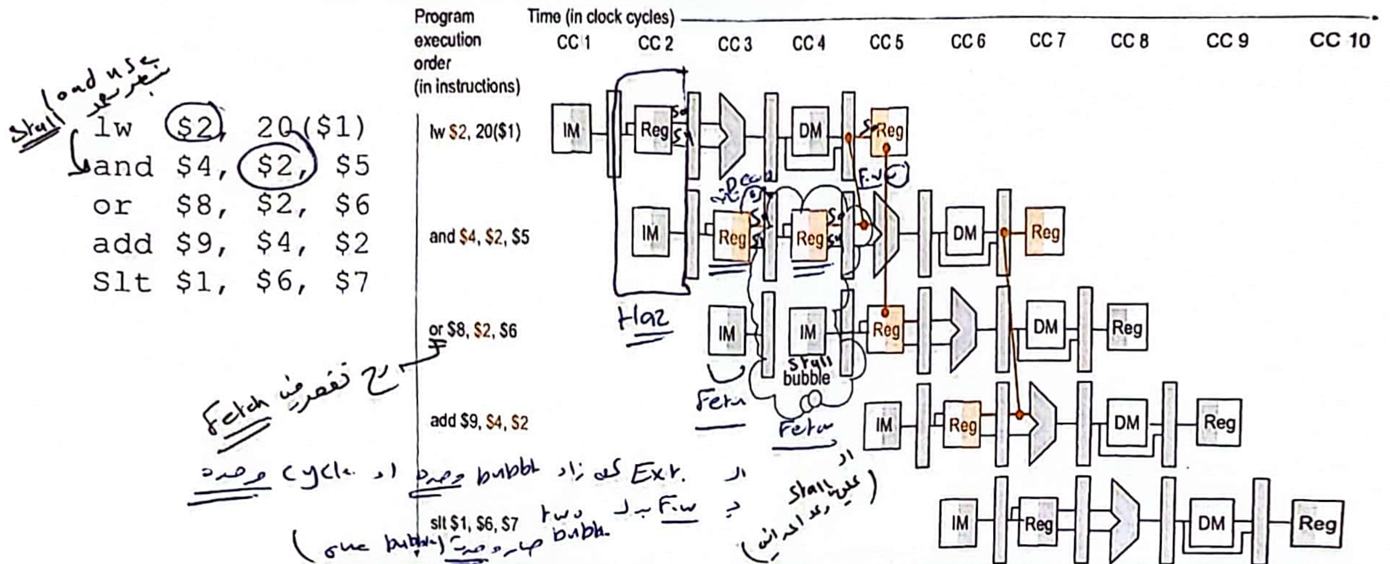
Datapath with forwarding hardware, the hazard detection unit and controls wires – certain details, e.g., branching hardware are omitted to simplify the drawing

الممسوحة ضوئيا بـ CamScanner

Stalling Resolves a Hazard

- Same instruction sequence as before for which forwarding by itself could not resolve the hazard:

```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
Slt $1, $6, $7
```

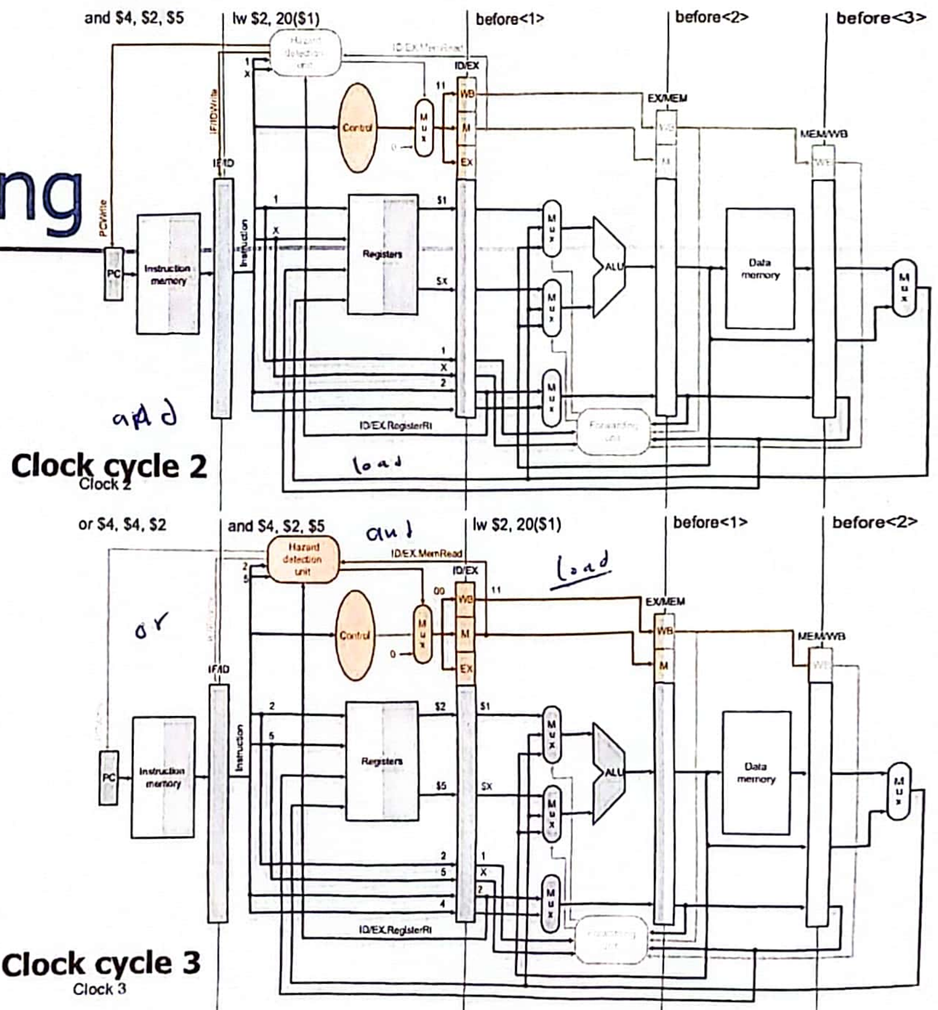


Hazard detection unit inserts a 1-cycle bubble in the pipeline, after which all pipeline register dependencies go forward so then the forwarding unit can handle them and there are no more hazards

Stalling

- Execution example:

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

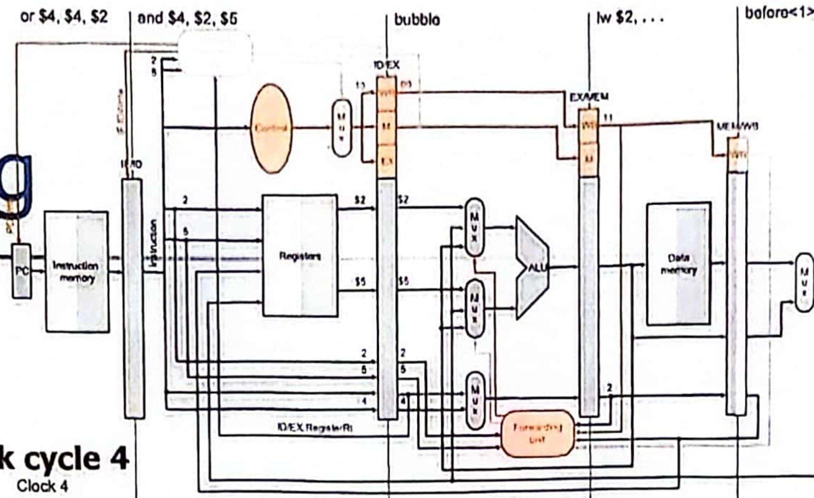


المسوحة ضوئيا بـ CamScanner

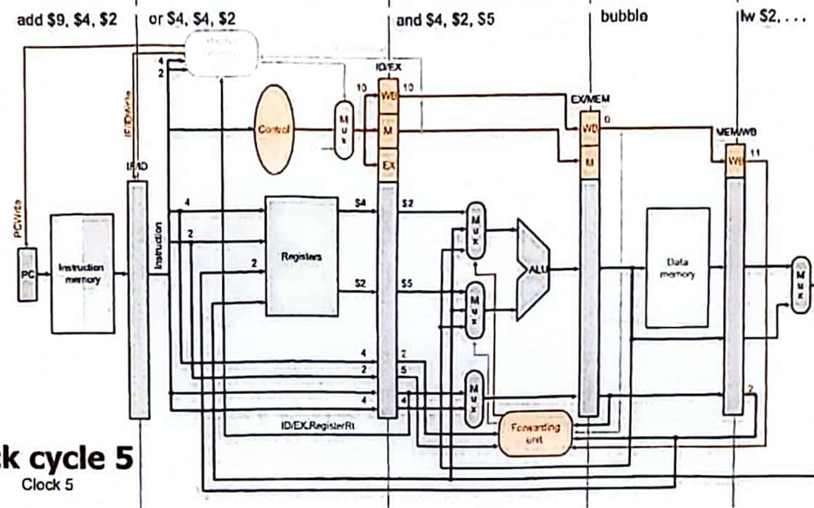
Stalling

- Execution example (cont.):

```
lw   $2, 20($1)
and  $4, $2, $5
or   $4, $4, $2
add  $9, $4, $2
```



Clock cycle 4
Clock 4



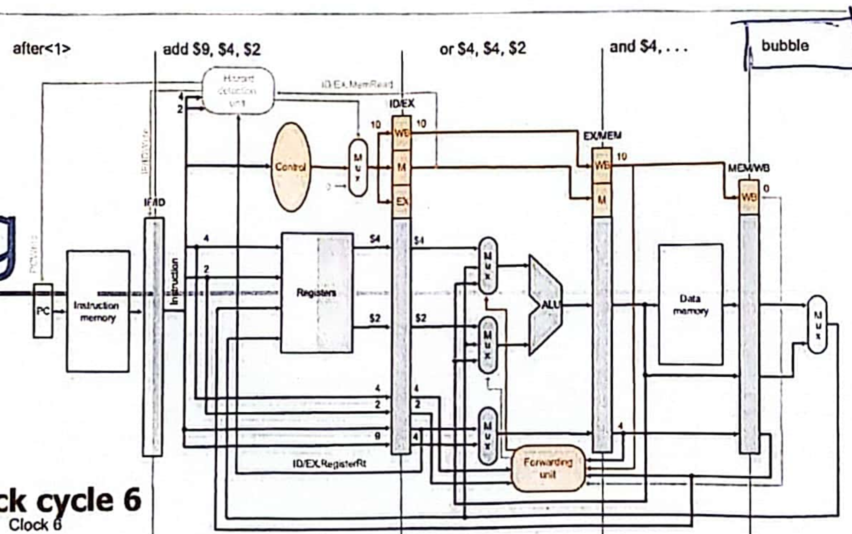
Clock cycle 5
Clock 5

Stalling

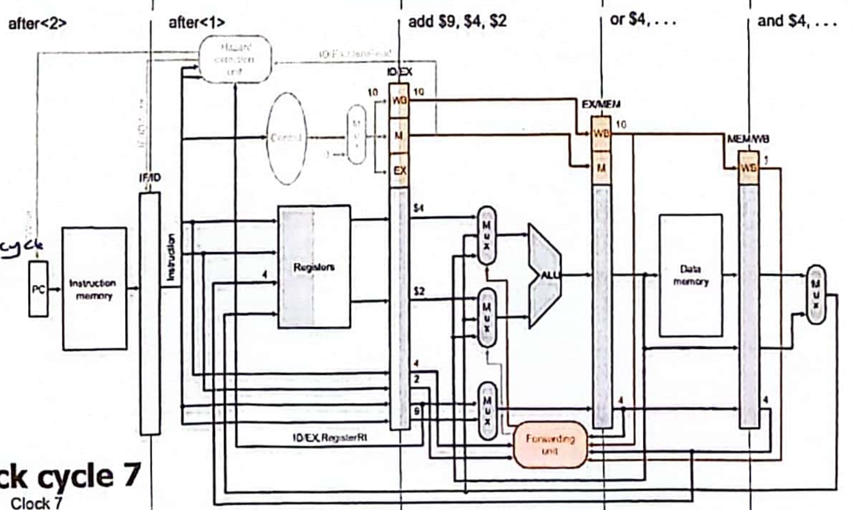
- Execution example (cont.):

```
lw   $2, 20($1)
and  $4, $2, $5
or   $4, $4, $2
add  $9, $4, $2
```

8 + 1 (bubble) = 9 cycle



Clock cycle 6
Clock 6



Clock cycle 7
Clock 7

المسوحة ضوئيا بـ CamScanner

Control (or Branch) Hazards

- Problem with branches in the pipeline we have so far is that the *branch decision is not made till the MEM stage* – so *what instructions, if at all, should we insert into the pipeline following the branch instructions*?

- Possible solution: *stall* the pipeline till branch decision is known
  - not efficient, slow the pipeline significantly!

- Another solution: *predict* the branch outcome
  - e.g., always predict *branch-not-taken – continue with next sequential instructions*
  - if the prediction is wrong have to *flush* the pipeline behind the branch – discard instructions already fetched or decoded – and *continue execution at the branch target*

Doing Better than Stalling Fetch

...

- Rather than waiting for true-dependence on PC to resolve, just guess nextPC = PC+4 to keep fetching every cycle
  - Is this a good guess?
  - What do you lose if you guessed incorrectly?

- ~20% of the instruction mix is control flow
  - ~50 % of "forward" control flow (i.e., if-then-else) is taken
  - ~90% of "backward" control flow (i.e., loop back) is taken

  Overall, typically ~70% taken and ~30% not taken

  [Lee and Smith, 1984]

- Expect "nextPC = PC+4" ~86% of the time, but what about the remaining 14%?

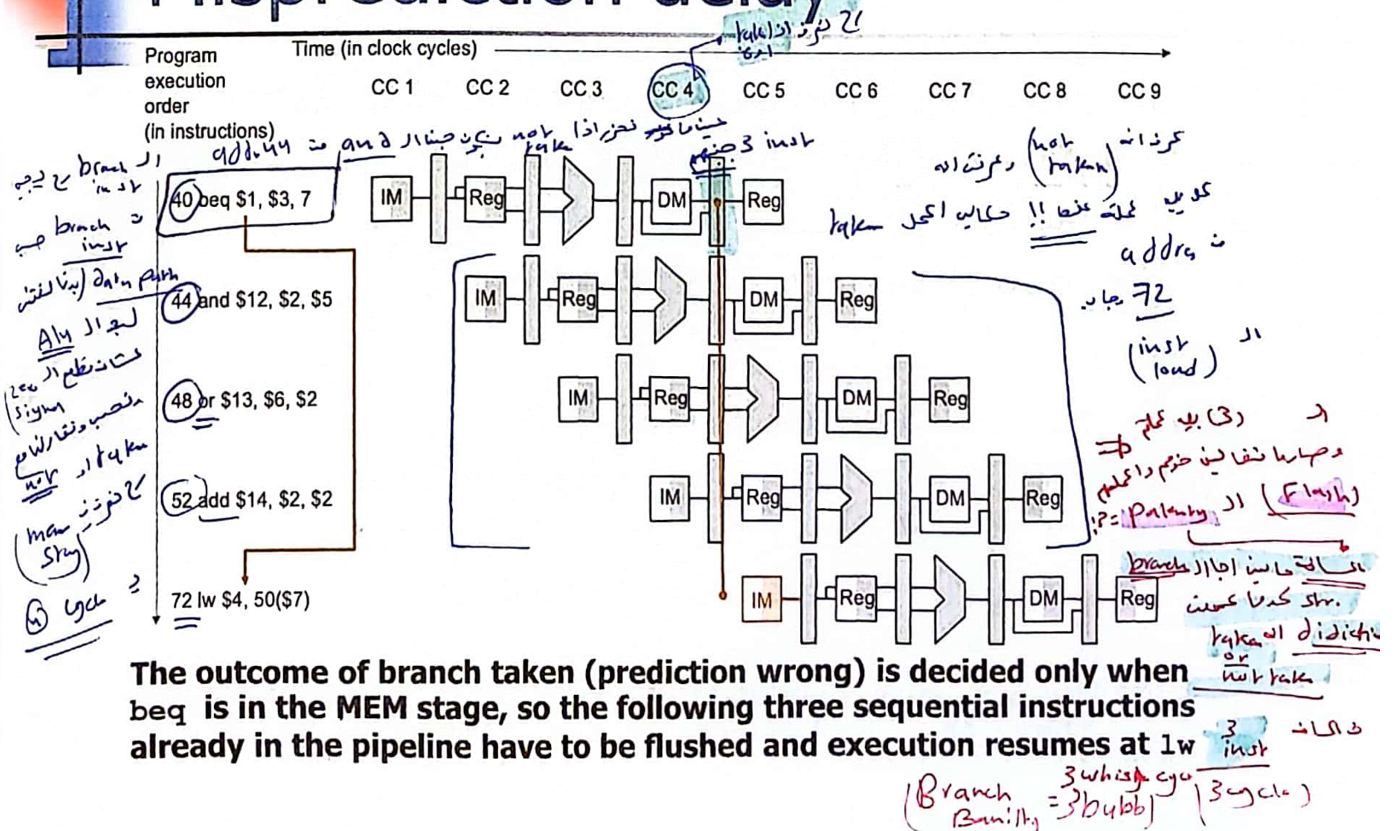
CamScanner المسوحة ضوئيا بـ

More Sophisticated Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid

Predicting Branch-not-taken: Misprediction delay



The outcome of branch taken (prediction wrong) is decided only when `beq` is in the MEM stage, so the following three sequential instructions already in the pipeline have to be flushed and execution resumes at `lw`

الممسوحة ضوئيا بـ CamScanner

Optimizing the Pipeline to Reduce Branch Delay

- *Move the branch decision* from the MEM stage (as in our current pipeline) *earlier to the ID stage*
  - *calculating the branch target address* involves moving the branch adder from the MEM stage to the ID stage – inputs to this adder, the PC value and the immediate fields are already available in the IF/ID pipeline register
  - *calculating the branch decision* is efficiently done, e.g., for equality test, by XORing respective bits and then ORing all the results and inverting, rather than using the ALU to subtract and then test for zero (when there is a carry delay)
    - with the more efficient equality test we can put it in the ID stage without significantly lengthening this stage – remember an objective of pipeline design is to keep pipeline stages balanced
  - *we must correspondingly make additions to the forwarding and hazard detection units* to forward to or stall the branch at the ID stage in case the branch decision depends on an earlier result

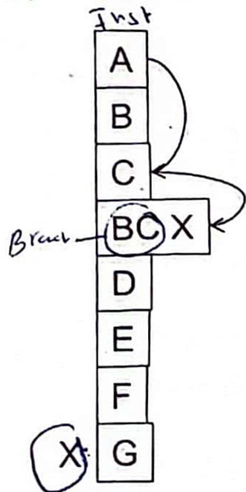
Delayed Branching (I)

- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles
- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are always executed regardless of branch direction.
- Problem: How do you find instructions to fill the delay slots?
  - Branch must be independent of delay slot instructions
- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot

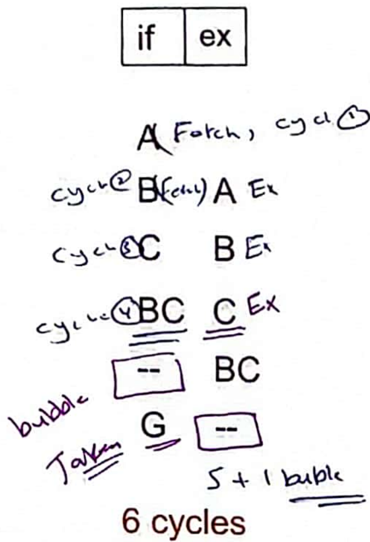
الممسوحة ضوئيا بـ CamScanner

Delayed Branching (II)

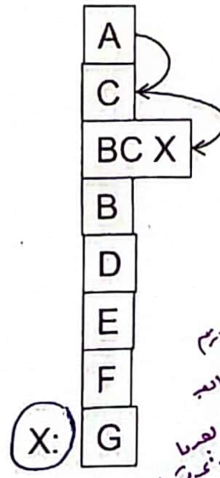
Normal code:



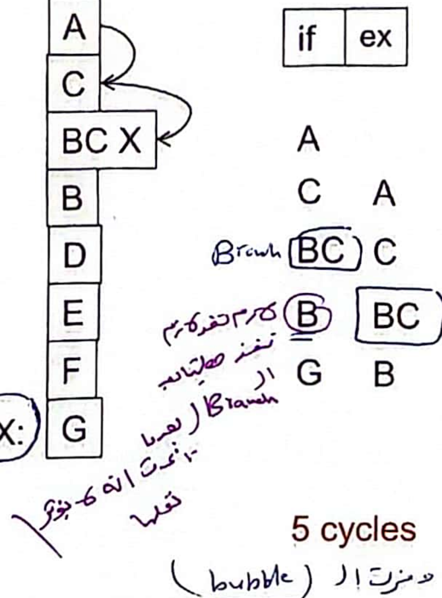
Timeline:



Delayed branch code:



Timeline:

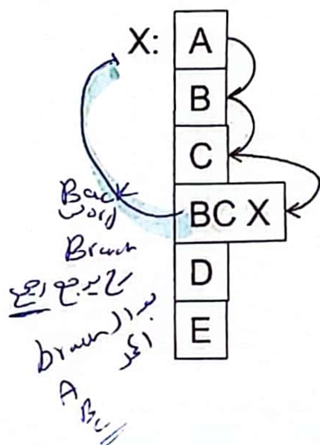


96

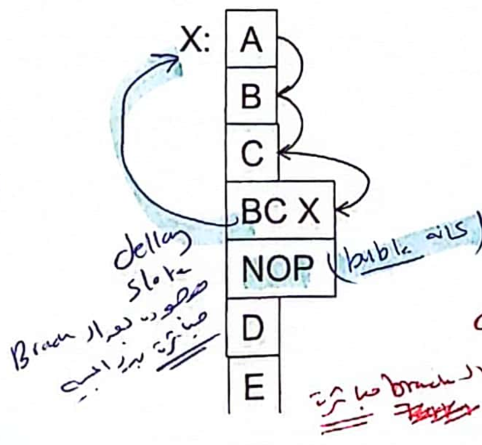
Fancy Delayed Branching (III)

- Delayed branch with squashing(In SPARC)
    - If the branch falls through (not taken), the delay slot instruction is not executed
    - Why could this help?

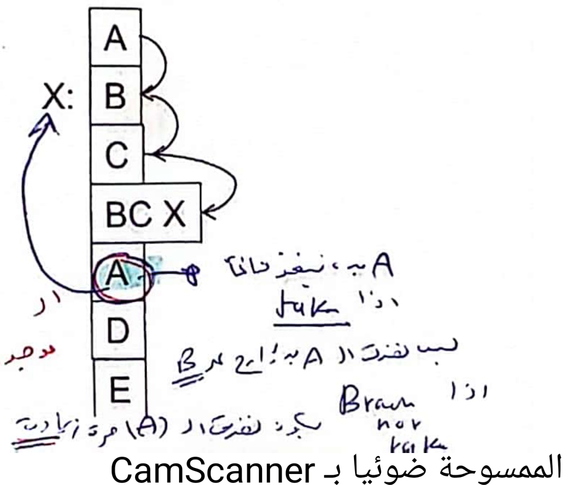
Normal code:



Delayed branch code:



Delayed branch w/ squashing:



CamScanner الممسوحة ضوئيا بـ

Delayed Branching (IV)

- **Advantages:**

  + Keeps the pipeline full with useful instructions in a simple way assuming

    1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves
    2. All delay slots can be filled with useful instructions

- Disadvantages:

  -- Not easy to fill the delay slots (even with a 2-stage pipeline)

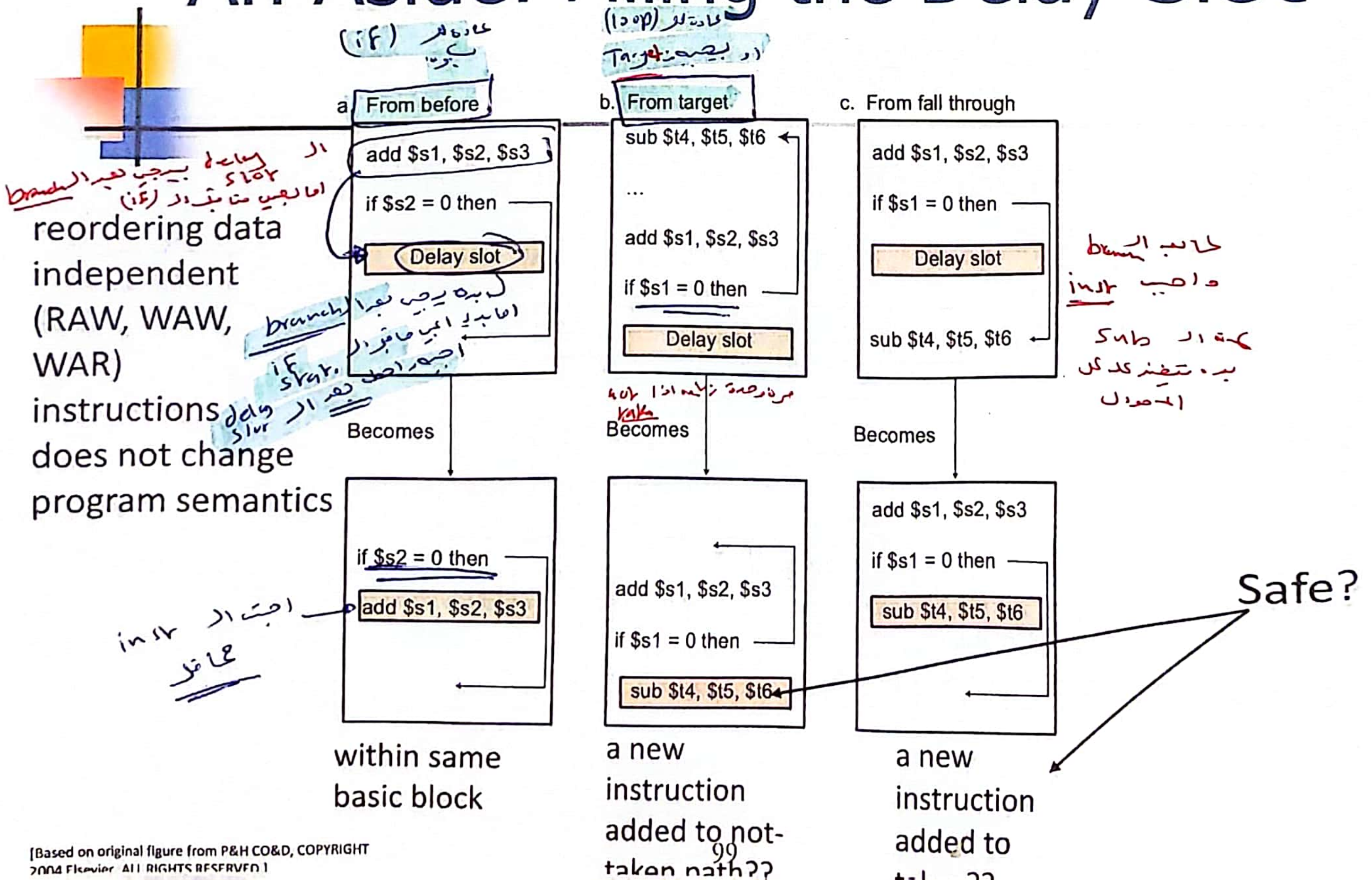
    1. Number of delay slots increases with pipeline depth, superscalar execution width
    2. Number of delay slots should be variable with variable latency operations. Why?

  -- Ties ISA semantics to hardware implementation

    -- SPARC, MIPS, HP-PA: 1 delay slot

    -- What if pipeline implementation changes with the next design?

An Aside: Filling the Delay Slot



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

CamScanner المسوحة ضوئيا بـ

Reducing Branch Delay

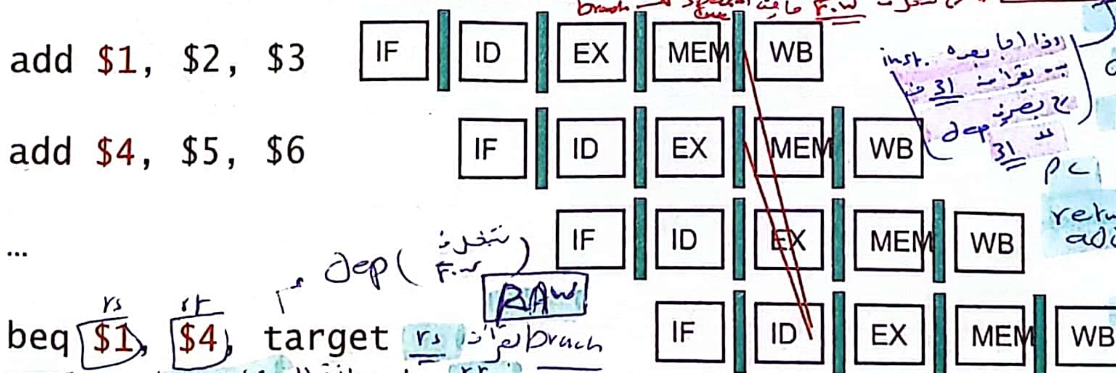
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 7
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
     ...
72:  lw   $4, 50($7)
```

TargetAddress = PC + Offeset = 44+ 7*4= 72

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction



- Can resolve using forwarding

الممسوحة ضوئيا بـ CamScanner

Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

Instruction								
`lw $1, addr`	IF	ID	EX	MEM	WB			
`add $4, $5, $6`		IF	ID	EX	MEM	WB		
`beq stalled`			IF	ID				
`beq $1, $4, target`					ID	EX	MEM	WB

Flushing on Misprediction

- Same strategy as for stalling on load-use data hazard…
- Zero out all the control values (or the instruction itself) in pipeline registers for the instructions following the branch that are already in the pipeline – effectively turning them into `nops` – so they are flushed
  - in the optimized pipeline, with branch decision made in the ID stage, we have to flush only one instruction in the IF stage – the branch delay penalty is then only one clock cycle

المسوحة ضوئيا بـ CamScanner

Optimized Datapath for Branch

IF.Flush control zeros out the instruction in the IF/ID pipeline register (which follows the branch)

Branch decision is moved from the MEM stage to the ID stage – simplified drawing not showing enhancements to the forwarding and hazard detection units

Pipelined Branch

- Execution example:

```
36 sub $10, $4, $8
40 beq $1,  $3,  7
44 and $12  $2, $5
48 or  $13  $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
...
72 lw  $4,  50($7)
```

Optimized pipeline with only one bubble as a result of the taken branch

Clock cycle 3

Clock cycle 4

الممسوحة ضوئيا بـ CamScanner

Impact of Stall on Performance

- Each stall cycle corresponds to 1 lost ALU cycle
- For a program with N instructions and S stall cycles, Average CPI=(N+S)/N
- S depends on
  - frequency of RAW dependences
  - exact distance between the dependent instructions
  - distance between dependences

Simple Example: Comparing Performance I

- *Compare performance for single-cycle, multicycle, and pipelined datapaths using the gcc instruction mix*
  - assume 2 ns for memory access, 2 ns for ALU operation, 1 ns for register read or write
  - assume gcc instruction mix 23% loads, 13% stores, 19% branches, 2% jumps, 43% ALU
  - for pipelined execution assume
    - 50% of the loads are followed immediately by an instruction that uses the result of the load
    - 25% of branches are mispredicted
    - branch delay on misprediction is 1 clock cycle
    - jumps always incur 1 clock cycle delay so their average time is 2 clock cycles

الممسوحة ضوئيا بـ CamScanner

Simple Example: Comparing PerformanceII

- *Single-cycle* (p. 373): average instruction time 8 ns
  - Cycle period = 2 + 1 + 2 + 2 + 1 = 8ns
  - CPI=1
  - No Hazards
- *Multicycle* (p. 397): average instruction time 8.04 ns
  - Cycle period =max(2,1,2,2,1)=2ns
  - No Hazards
  - CPI= 5,4,3
  - LD instruction time =5*2=10 ns
  - SW,Rtype instruction time =4*2=8ns
  - Br,J instruction time =3*2 = 6 ns
  - average instruction time = 0.23*10 + 0.13*8 + 0.19* 6 + 0.02*6 + 0.43*8 = 8.04

Simple Example: Comparing Performance III

Pipelined:

- loads use 1 cc (clock cycle) when no load-use dependency and 2 cc when there is dependency – given 50% of loads are followed by dependency the average cc per load is 1.5
- stores use 1 cc each
- branches use 1 cc when predicted correctly and 2 cc when not – given 25% misprediction average cc per branch is 1.25
  - CPI = .25*2 + .75*1 = 1.25
- jumps use 2 cc each
- ALU instructions use 1 cc each
- therefore, average CPI is

$1.5 \times 23\% + 1 \times 13\% + 1.25 \times 19\% + 2 \times 2\% + 1 \times 43\% = 1.18$

- therefore, average instruction time is $1.18 \times 2 = 2.36$ ns

الممسوحة ضوئيا بـ CamScanner

Performance Analysis (Prediction)

- correct guess ⇒ no penalty ~86% of the time
- incorrect guess ⇒ 2 bubbles
- Assume
  - no data hazards
  - 20% control flow instructions
  - 70% of control flow instructions are taken
  - CPI = [1 + (0.20*0.7) * 2] =

$$[1 + 0.14 * 2] = 1.28$$

probability of a wrong guess

penalty for a wrong guess

Can we reduce either of the two penalty terms?

Computer Organization

Slide Sources: Patterson & Hennessy COD book site (copyright Morgan Kaufmann) adapted and supplemented

الممسوحة ضوئيا بـ CamScanner

COD Ch. 7
Large and Fast: Exploiting Memory Hierarchy

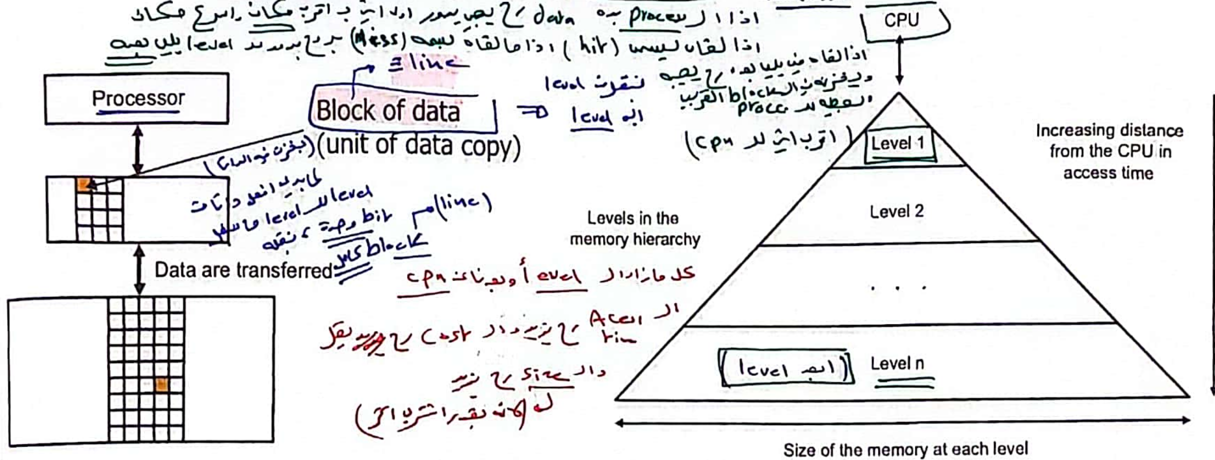
Memories: Review

- DRAM (Dynamic Random Access Memory):
  - value is stored as a charge on capacitor that must be *periodically refreshed*, which is why it is called *dynamic*
  - very small – 1 transistor per bit – but factor of 5 to 10 slower than SRAM
  - used for *main memory*
- SRAM (Static Random Access Memory):
  - value is stored on a pair of inverting gates that will *exist indefinitely* as long as there is power, which is why it is called *static*
  - very fast but takes up more space than DRAM – 4 to 6 transistors per bit
  - used for *cache*

المسوحة ضوئيا بـ CamScanner

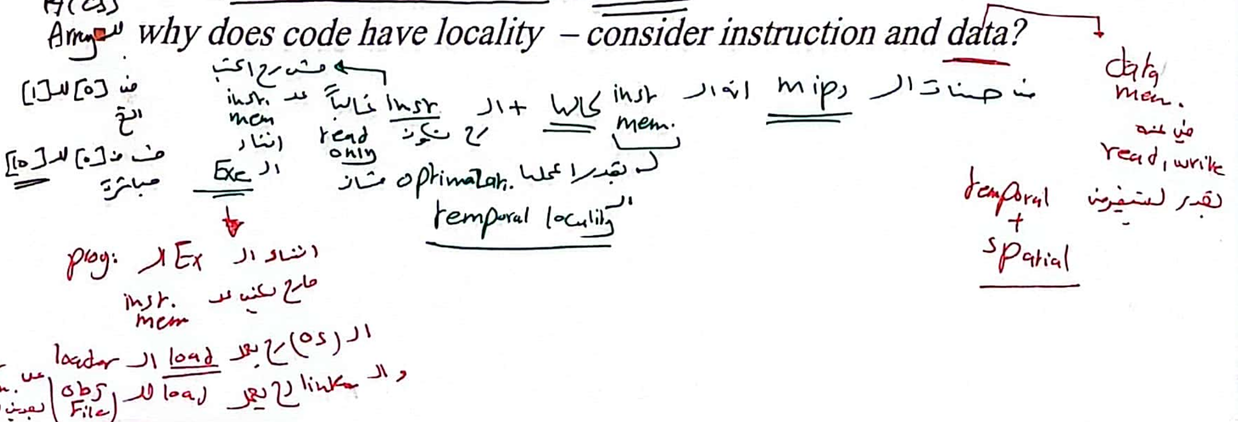
Memory Hierarchy

- Users want large and fast memories...
  - expensive and they don't like to pay...
- Make it seem like they have what they want...
  - memory hierarchy
  - hierarchy is *inclusive*, every level is *subset* of lower level
  - performance depends on *hit rates*



Locality

- *Locality* is a principle that makes having a memory hierarchy a good idea
- If an item is referenced then because of
  - *temporal locality*: it will tend to be *again* referenced soon
    - e.g., instructions in a loop, induction variables
  - *spatial locality*: *nearby items* will tend to be referenced soon
    - E.g., sequential instruction access, array data
  - *why does code have locality – consider instruction and data?*



الممسوحة ضوئيا بـ CamScanner

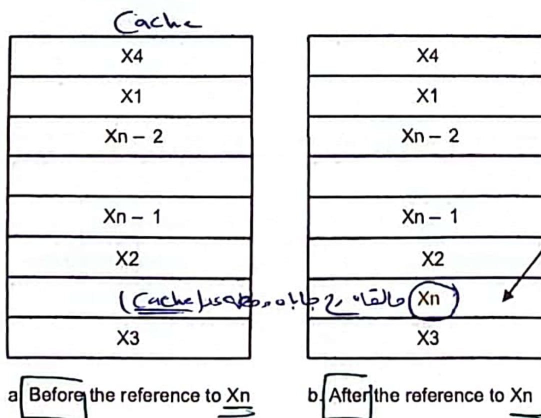
Hit and Miss

- Focus on *any two adjacent* levels – called, *upper* (closer to CPU) and *lower* (farther from CPU) – in the memory hierarchy, because each block copy is always between two adjacent levels
- Terminology:
  - *block*: minimum unit of data to move between levels
  - *hit*: data requested is in upper level
  - *miss*: data requested is not in upper level
  - *hit rate*: fraction of memory accesses that are hits (i.e., found at upper level)
  - *miss rate*: fraction of memory accesses that are not hits
    - miss rate = 1 – hit rate
  - *hit time*: time to determine if the access is indeed a hit + time to access and deliver the data from the upper level to the CPU
  - *miss penalty*: time to determine if the access is a miss + time to replace block at upper level with corresponding block at lower level + time to deliver the block to the CPU

Caches

By simple example

- assume block size = one word of data



a. Before the reference to Xn   b. After the reference to Xn

Reference to X_n causes miss so it is fetched from memory

- Issues:
  - *how do we know if a data item is in the cache?*
  - *if it is, how do we find it?*
  - *if not, what do we do?*
- Solution depends on *cache addressing scheme…*

CamScanner الممسوحة ضوئيا بـ

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!)

الممسوحة ضوئيا بـ CamScanner

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

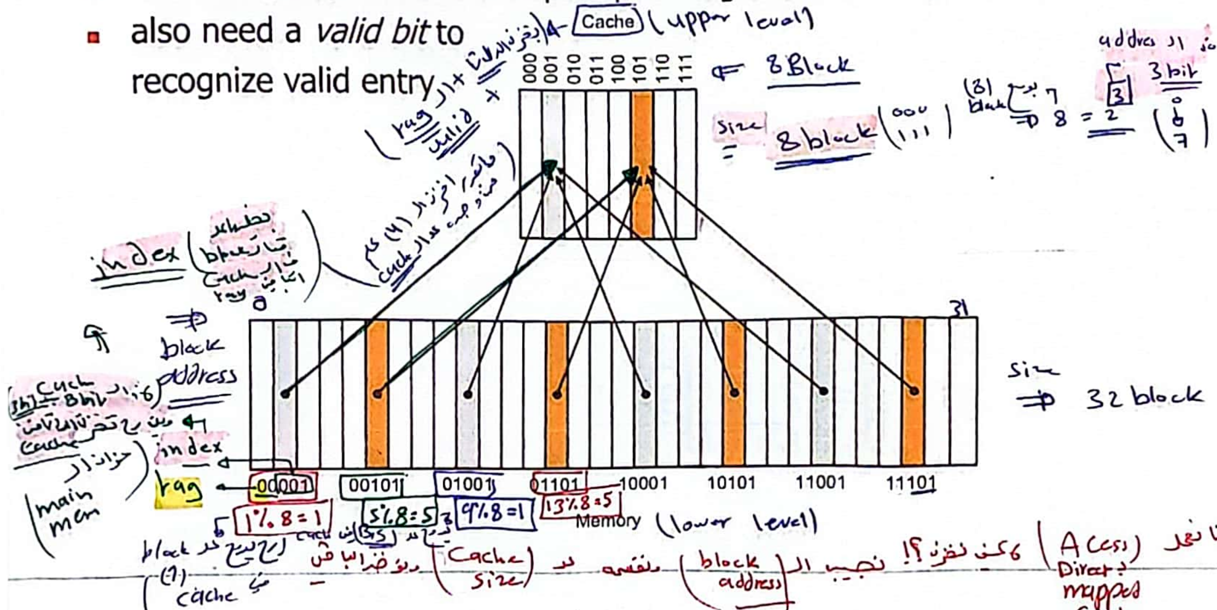
Tags and Valid Bits

- 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

الممسوحة ضوئيا بـ CamScanner

Direct Mapped Cache

- Addressing scheme in *direct mapped* cache:
  - cache block address = memory block address **mod** cache size (*unique*)
  - if cache size = 2^m, cache address = lower m bits of n-bit memory address
  - remaining upper n-m bits kept kept as *tag bits* at each cache block
  - also need a *valid bit* to recognize valid entry



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		

الممسوحة ضوئيا بـ CamScanner

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		

الممسوحة ضوئيا بـ CamScanner

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		

الممسوحة ضوئيا بـ CamScanner

Cache Example

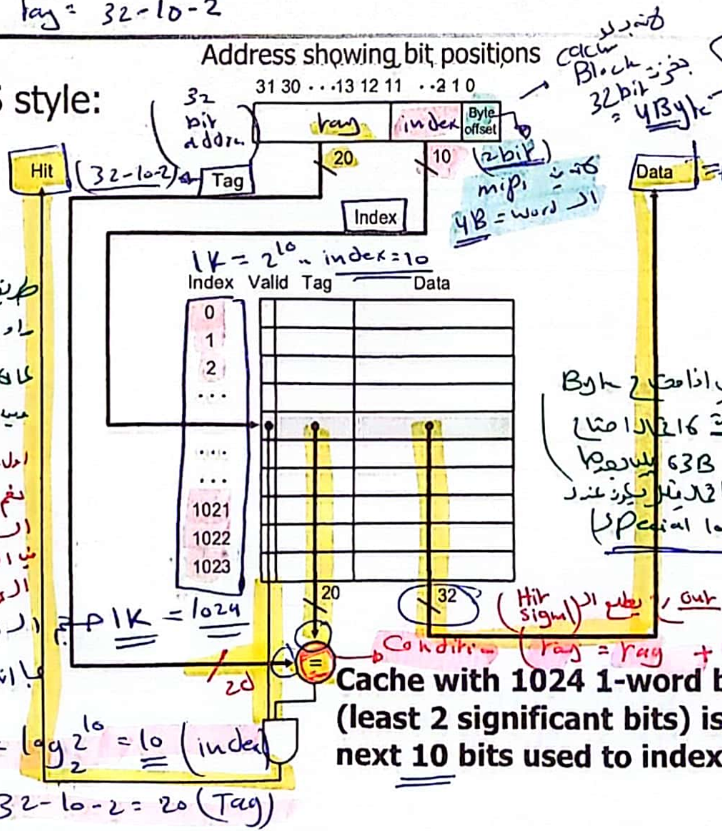
Word addr	Binary addr	Hit/miss	Cache block
18	10010	Miss	010

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

Direct Mapped Cache

- MIPS style:



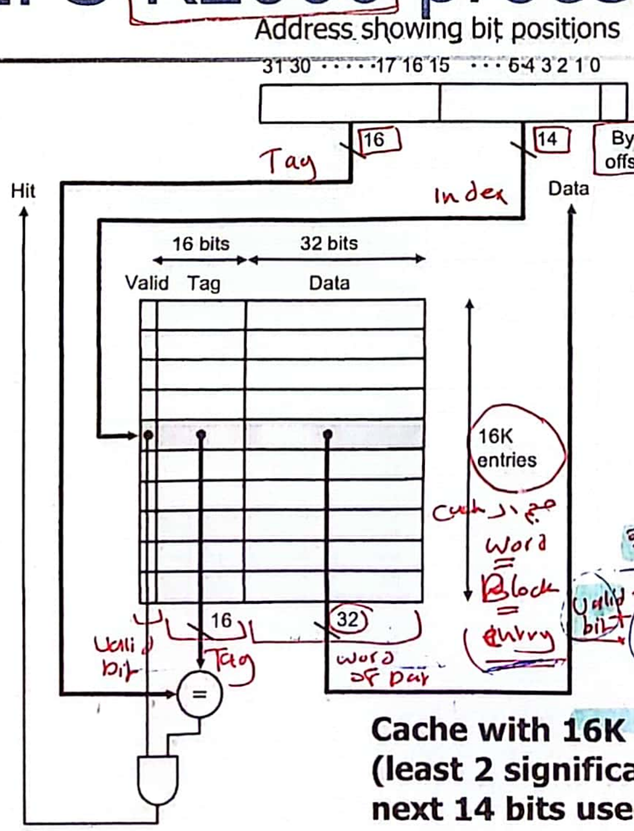
What kind of locality are we taking advantage of?

Cache with 1024 1-word blocks: *byte offset* (least 2 significant bits) is ignored and next 10 bits used to index into cache

Cache Read Hit/Miss

- *Cache read hit*: no action needed
- *Instruction cache read miss*:
  1. *Send original PC value* (*current PC – 4*, as PC has already been incremented in first step of instruction cycle) to memory
  2. Instruct main memory to perform read and wait for memory to complete access – *stall* on read
  3. After read completes *write cache* entry
  4. *Restart* instruction execution at first step to refetch instruction
- *Data cache read miss*:
  - Similar to instruction cache miss
  - To reduce data miss penalty allow processor to execute instructions while waiting for the read to complete *until* the word is required – *stall on use* (why won't this work for instruction misses?)

DECStation 3100 Cache (MIPS R2000 processor)



Cache with 16K 1-word blocks: ***byte offset*** **(least 2 significant bits) is ignored and next 14 bits used to index into cache**

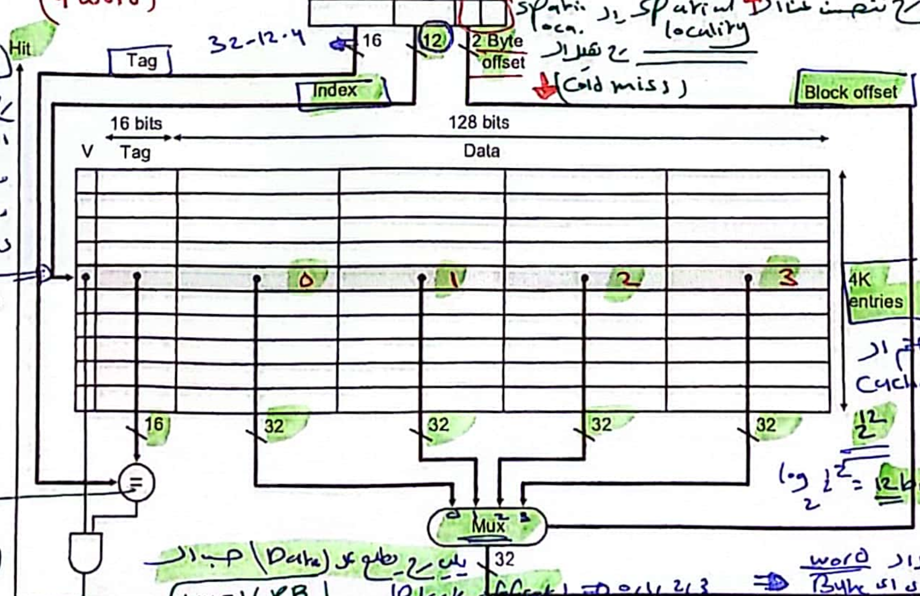
الممسوحة ضوئيا بـ CamScanner

Cache Write Hit/Miss

- *Write-through* scheme
  - on *write hit*: replace data in cache *and* memory with *every* write hit to avoid *inconsistency*
  - on *write miss*: write the word into cache *and* memory – obviously no need to read missed word from memory!
  - Write-through is slow because of always required memory write
    - performance is improved with a *write buffer* where words are stored while waiting to be written to memory – processor can continue execution until write buffer is full
    - when a word in the write buffer completes writing into main that buffer slot is freed and becomes available for future writes
    - DEC 3100 write buffer has 4 words
- *Write-back* scheme
  - write the data block *only* into the cache and *write-back* the block to main *only when* it is replaced in cache
  - more efficient than write-through, more complex to implement

Direct Mapped Cache: Taking Advantage of Spatial Locality

- Taking advantage of spatial locality with *larger* blocks:



Cache with 4K 4-word blocks: ***byte offset*** **(least 2 significant bits) is ignored, next 2 bits are** ***block offset*****, and the next 12 bits are used to index into cache**

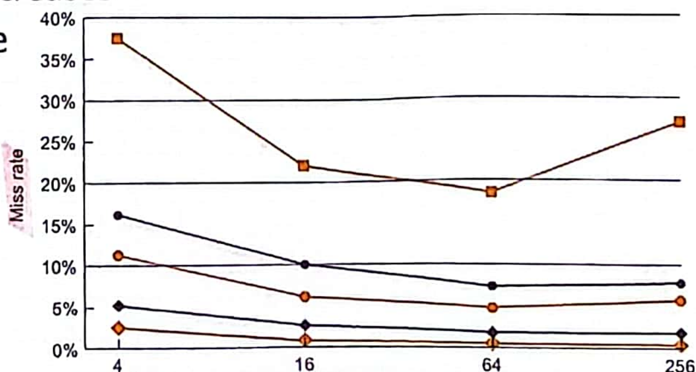
الممسوحة ضوئيا بـ CamScanner

Direct Mapped Cache: Taking Advantage of Spatial Locality

- *Cache replacement* in large (multiword) blocks:
  - word *read miss*: read entire block from main memory
  - word *write miss*: *cannot* simply write word and tag! *Why*?!
  - writing in a *write-through* cache:
    - if *write hit*, i.e., tag of requested address and and cache entry are equal, continue as for 1-word blocks by replacing word and writing block to both cache and memory
    - if *write miss*, i.e., tags are unequal, fetch block from memory, replace word that caused miss, and write block to both cache and memory
    - therefore, unlike case of 1-word blocks, a write miss with a multiword block causes a memory read

Direct Mapped Cache: Taking Advantage of Spatial Locality

- Miss rate falls at first with increasing block size as expected, but, as block size becomes a large fraction of total cache size, miss rate may go up because
  - there are few blocks
  - competition for blocks increases
  - blocks get ejected before most of their words are accessed (*thrashing* in cache)



Miss rate vs. block size for various cache sizes

الممسوحة ضوئيا بـ CamScanner

Example Problem

- *How many total bits are required for a direct-mapped cache with 128 KB of data and 1-word block size, assuming a 32-bit address?*

- Cache data = 128 KB = 2^{17} bytes = 2^{15} words = 2^{15} blocks
- Cache entry size = block data bits + tag bits + valid bit

  = 32 + (32 – 15 – 2) + 1 = 48 bits
- Therefore, cache size = 2^{15} × 48 bits = 2^{15} × (1.5 × 32) bits = 1.5 × 2^{20} bits = 1.5 Mbits
  - data bits in cache = 128 KB × 8 = 1 Mbits
  - total cache size/actual cache data = 1.5

Example Problem

- *How many total bits are required for a direct-mapped cache with 128 KB of data and 4-word block size, assuming a 32-bit address?*

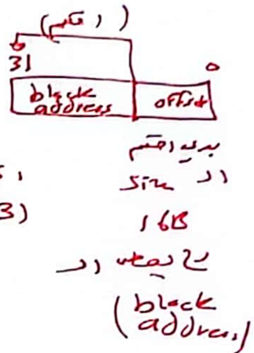
- Cache size = 128 KB = 2^{17} bytes = 2^{15} words = 2^{13} blocks
- Cache entry size = block data bits + tag bits + valid bit

  = 128 + (32 – 13 – 2 – 2) + 1 = 144 bits
- Therefore, cache size = 2^{13} × 144 bits = 2^{13} × (1.25 × 128) bits = 1.25 × 2^{20} bits = 1.25 Mbits
  - data bits in cache = 128 KB × 8 = 1 Mbits
  - total cache size/actual cache data = 1.25

CamScanner الممسوحة ضوئيا بـ

Example Problem

Consider a cache with 64 blocks and a block size of 16 bytes. What block number does byte address 1200 map to?

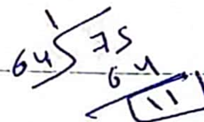


As block size = 16 bytes:

byte address 1200 ⇒ block address $\lfloor 1200/16 \rfloor = 75$

As cache size = 64 blocks:

block address 75 ⇒ cache block (75 *mod* 64) = 11



Improving Cache Performance

- Use split caches for instruction and data because there is more spatial locality in instruction references:

Program	Block size in words	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

Miss rates for gcc and spice in a MIPS R2000 with one and four word block sizes

- Make reading multiple words (higher bandwidth) possible by increasing physical or logical width of the system…

الممسوحة ضوئيا بـ CamScanner

Improving Cache Performance by Increasing Bandwidth

- Assume:
  - cache block of 4 words
  - 1 clock cycle to send address to memory address buffer (1 bus trip)
  - 15 clock cycles for each memory data access
  - 1 clock cycle to send data to memory data buffer (1 bus trip)

CPU / Bus / Cache / Bus / Memory

a. One-word-wide memory organization

1 + 4*15 + 4*1 = 65 cycles

CPU / Multiplexor / Cache / Bus / Memory

b. Wide memory organization

4 word wide memory and bus

1 + 1*15 +1*1 = 17 cycles

CPU / Cache / Bus / Memory bank 0 / Memory bank 1 / Memory bank 2 / Memory bank 3

c. Interleaved memory organization

4 word wide memory only

1 +1*15 + 4*1 = 20 cycles

Bus / Processor / Memory bank 0 / Memory bank 1 / Memory bank 2 / Memory bank 3

Interleaved memory units compete for bus

Miss penalties

Performance

- Simplified model assuming equal read and write miss penalties:
  - CPU time = (execution cycles + memory stall cycles) × cycle time
  - memory stall cycles = memory accesses × miss rate × miss penalty
- Therefore, two ways to improve performance in cache:
  - decrease miss rate
  - decrease miss penalty
  - *what happens if we increase block size?*

الممسوحة ضوئيا بـ CamScanner

Example Problems

- Assume for a given machine and program:
  - instruction cache miss rate 2%
  - data cache miss rate 4%
  - miss penalty always 40 cycles
  - CPI of 2 without memory stalls
  - frequency of load/stores 36% of instructions

1. *How much faster is a machine with a perfect cache that never misses?*
2. *What happens if we speed up the machine by reducing its CPI to 1 without changing the clock rate?*
3. *What happens if we speed up the machine by doubling its clock rate, but if the absolute time for a miss penalty remains same?*

Solution

Assume instruction count = I
Instruction miss cycles = I × 2% × 40 = 0.8 × I
Data miss cycles = I × 36% × 4% × 40 = 0.576 × I
So, total memory-stall cycles = 0.8 × I + 0.576 × I = 1.376 × I

- in other words, 1.376 stall cycles per instruction

Therefore, CPI with memory stalls = 2 + 1.376 = 3.376
Assuming instruction count and clock rate remain same for a perfect cache and a cache that misses:
CPU time with stalls / CPU time with perfect cache
= 3.376 / 2 = 1.688
Performance with a perfect cache is better by a factor of 1.688

CamScanner الممسوحة ضوئيا بـ

Solution (cont.)

2.

- CPI without stall = 1
- CPI with stall = 1 + 1.376 = 2.376
  - (clock has not changed so stall cycles per instruction remains same)
- CPU time with stalls / CPU time with perfect cache

  = CPI with stall / CPI without stall

  = 2.376
- Performance with a perfect cache is better by a factor of 2.376
- Conclusion: with higher CPI cache misses "hurt more" than with lower CPI

Solution (cont.)

3.

- With doubled clock rate, miss penalty = 2 × 40 = 80 clockcycles
- Stall cycles per instruction =

  (I × 2% × 80) + (I × 36% × 4% × 80) = 2.752 × I
- faster machine with cache miss has CPI = 2 + 2.752 = 4.752
- CPU time with stalls / CPU time with perfect cache

  = CPI with stall / CPI without stall

  = 4.752 / 2 = 2.376
- Performance with a perfect cache is better by a factor of 2.376
- Conclusion: with higher clock rate cache misses "hurt more" than with lower clock rate

CamScanner المسوحة ضوئيا بـ

Decreasing Miss Rates with Associative Block Placment

- *Direct mapped*: one *unique* cache location for each memory block
  - cache block address = memory block address *mod* cache size
- *Fully associative*: each memory block can locate *anywhere* in cache
  - *all* cache entries are searched (in parallel) to locate block
- *Set associative*: each memory block can place in a *unique set* of cache locations – if the set is of size n it is n-way set-associative
  - cache set address = memory block address *mod* num of sets in cache
  - all cache entries in the corresponding set are searched (*in parallel*) to locate block
- Increasing degree of associativity
  - *reduces miss rate*
  - *increases hit time* because of the parallel search and then fetch

Decreasing Miss Rates with Associative Block Placment

Direct Mapped | 2-way Set Associative | Fully Associative

Direct Mapped: Block # 0 1 2 3 4 5 6 7; Data; 12 mod 8 = 4; Tag 1 2; Search

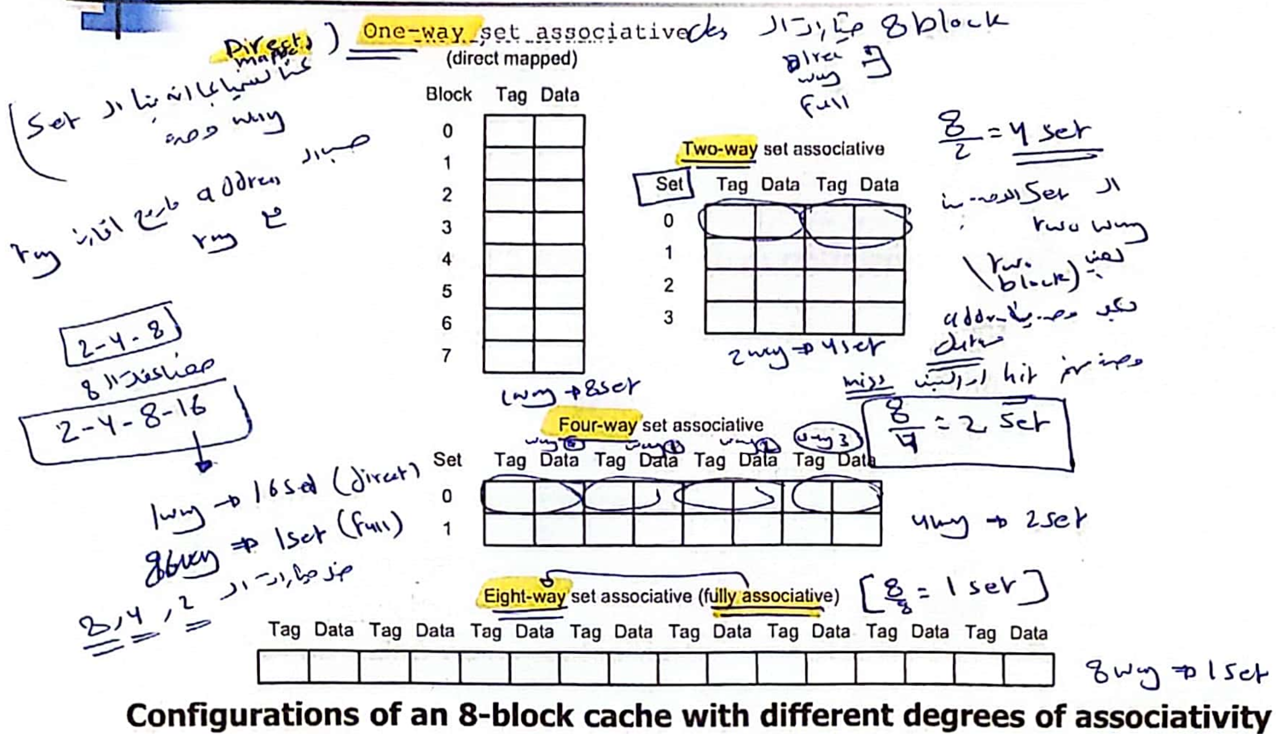
2-way Set Associative: Set # 0 1 2 3; Data; 12 mod 4 = 0; Tag 1 2; Search

Fully Associative: Data; Tag 1 2; Search

Location of a memory block with address 12 in a cache with 8 blocks with different degrees of associativity

CamScanner المسوحة ضوئيا بـ

Decreasing Miss Rates with Associative Block Placment



Configurations of an 8-block cache with different degrees of associativity

Example Problems

- *Find the number of misses for a cache with four 1-word blocks given the following sequence of memory block accesses:*

  0, 8, 0, 6, 8,

  for each of the following cache configurations

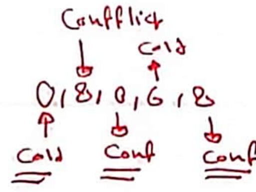
  1. *direct mapped*
  2. *2-way set associative (use LRU replacement policy)*
  3. *fully associative*

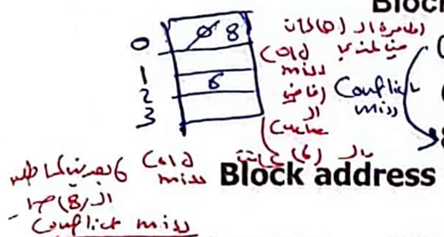
- Note about LRU replacement
  - in a 2-way set associative cache LRU replacement can be implemented with one bit at each set whose value indicates the mostly recently referenced block

الممسوحة ضوئيا بـ CamScanner

Solution

- 1 (direct-mapped)





Block address	Cache block
0	0 (= 0 *mod* 4)
6	2 (= 6 *mod* 4)
8	0 (= 8 *mod* 4)

Block address translation in direct-mapped cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference 0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Cache contents after each reference – red indicates new entry added

- 5 misses

Solution (cont.)

- 2 (two-way set-associative)

Block address	Cache set
0	0 (= 0 *mod* 2)
6	0 (= 6 *mod* 2)
8	0 (= 8 *mod* 2)

Block address translation in a two-way set-associative cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Cache contents after each reference – red indicates new entry added

- Four misses

الممسوحة ضوئيا بـ CamScanner

Solution (cont.)

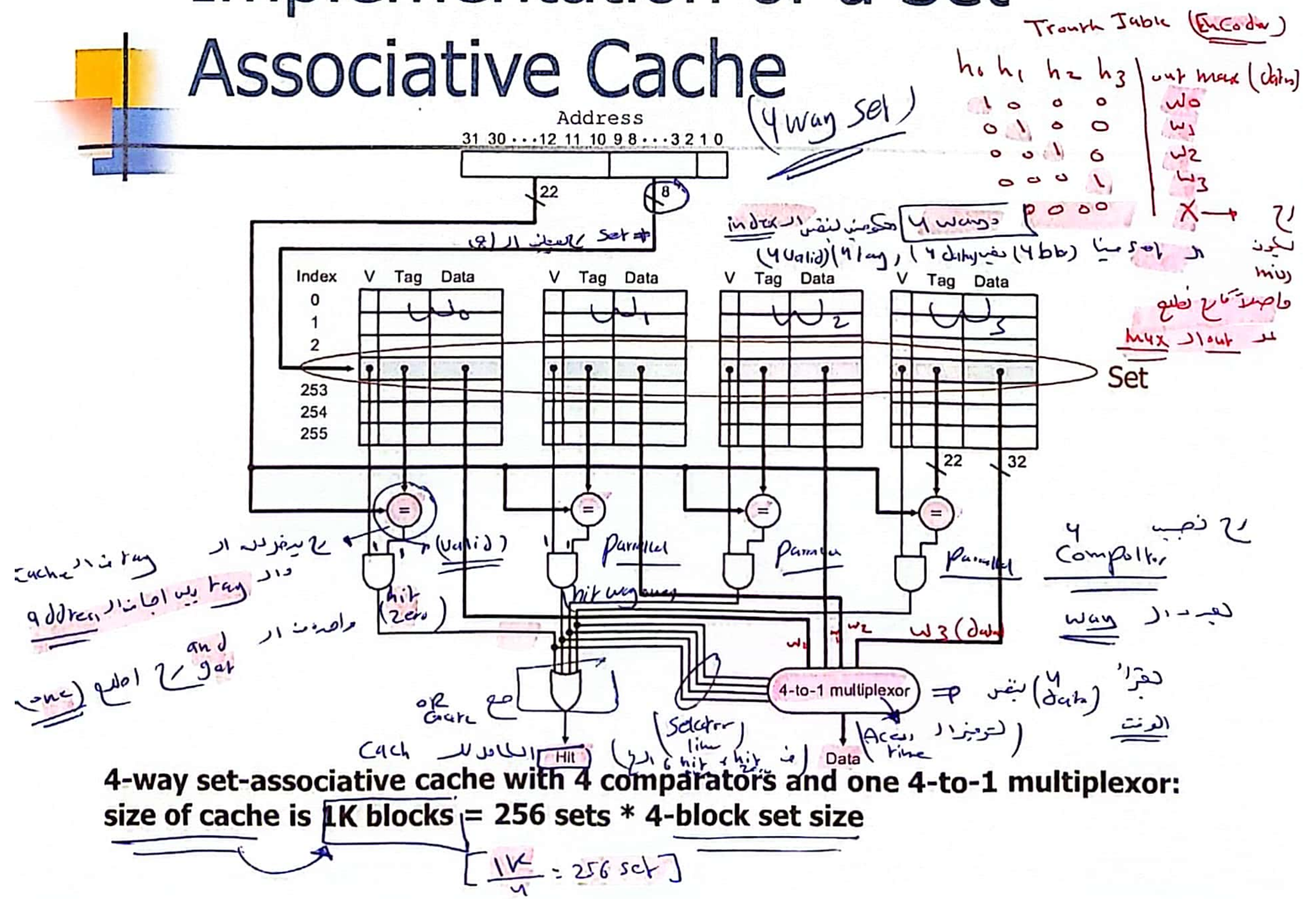
- 3 (fully associative)

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

Cache contents after each reference – red indicates new entry added

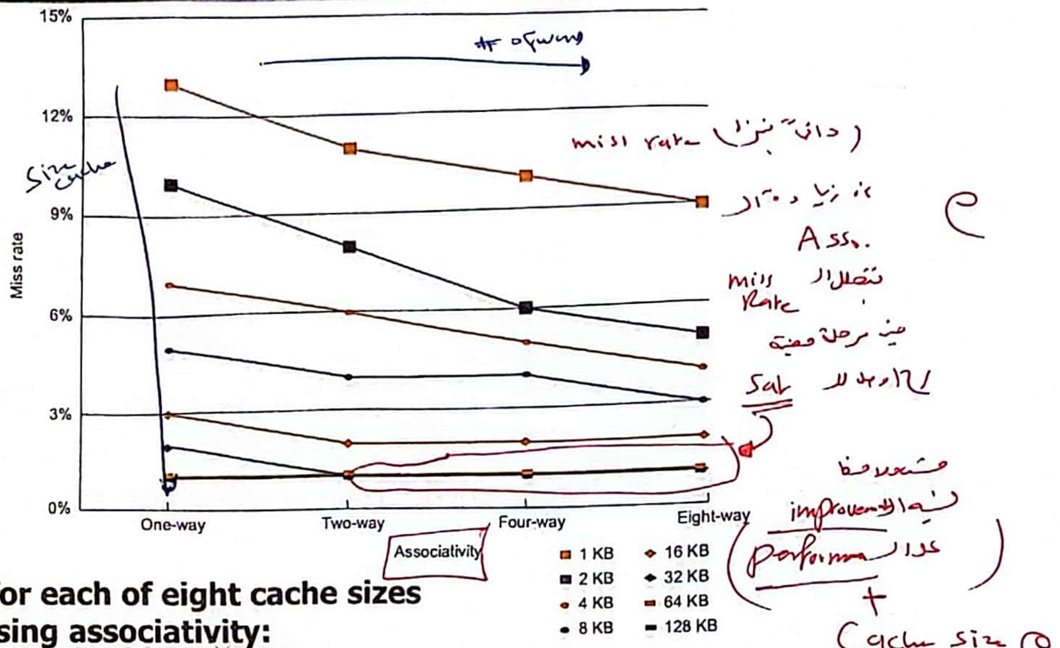
- 3 misses

Implementation of a Set-Associative Cache



4-way set-associative cache with 4 comparators and one 4-to-1 multiplexor: size of cache is 1K blocks = 256 sets * 4-block set size

Performance with Set-Associative Caches



Miss rates for each of eight cache sizes with increasing associativity: data generated from SPEC92 benchmarks with 32 byte block size for all caches

Decreasing Miss Penalty with Multilevel Caches

Add a *second-level* cache

- primary cache is on the same chip as the processor
- use SRAMs to add a second-level cache, sometimes off-chip, *between main memory and the first-level cache*
- if miss occurs in primary cache second-level cache is accessed
- if data is found in second-level cache miss penalty is access time of second-level cache which is much less than main memory access time
- if miss occurs again at second-level then main memory access is required and large miss penalty is incurred

Design considerations using two levels of caches:

- try and optimize the *hit time on the 1st level cache* to reduce clock cycle
- try and optimize the *miss rate on the 2nd level cache* to reduce memory access penalties
- In other words, 2nd level allows 1st level to go for speed without "worrying" about failure...

CamScanner الممسوحة ضوئيا بـ

Example Problem

- Assume a 500 MHz machine with
  - base CPI 1.0
  - main memory access time 200 ns.
  - miss rate 5%
- *How much faster will the machine be if we add a second-level cache with 20ns access time that decreases the miss rate to 2%?*

Solution

- Miss penalty to main = 200 ns / (2 ns / clock cycle) = 100 clock cycles
- Effective CPI with one level of cache
  = Base CPI + Memory-stall cycles per instruction
  = 1.0 + 5% × 100 = 6.0
- With two levels of cache, miss penalty to second-level cache
  = 20 ns / (2 ns / clock cycle) = 10 clock cycles
- Effective CPI with two levels of cache
  = Base CPI + Primary stalls per instruction
  + Secondary stall per instruction
  = 1 + 5% × 10 + 2% × 100 = 3.5
- Therefore, machine with secondary cache is faster by a factor of
  6.0 / 3.5 = 1.71

CamScanner الممسوحة ضوئيا بـ

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.

المسوحة ضوئيا بـ CamScanner