Inf2C - Computer Systems
Lecture 1
Course overview & the big picture

Boris Grot

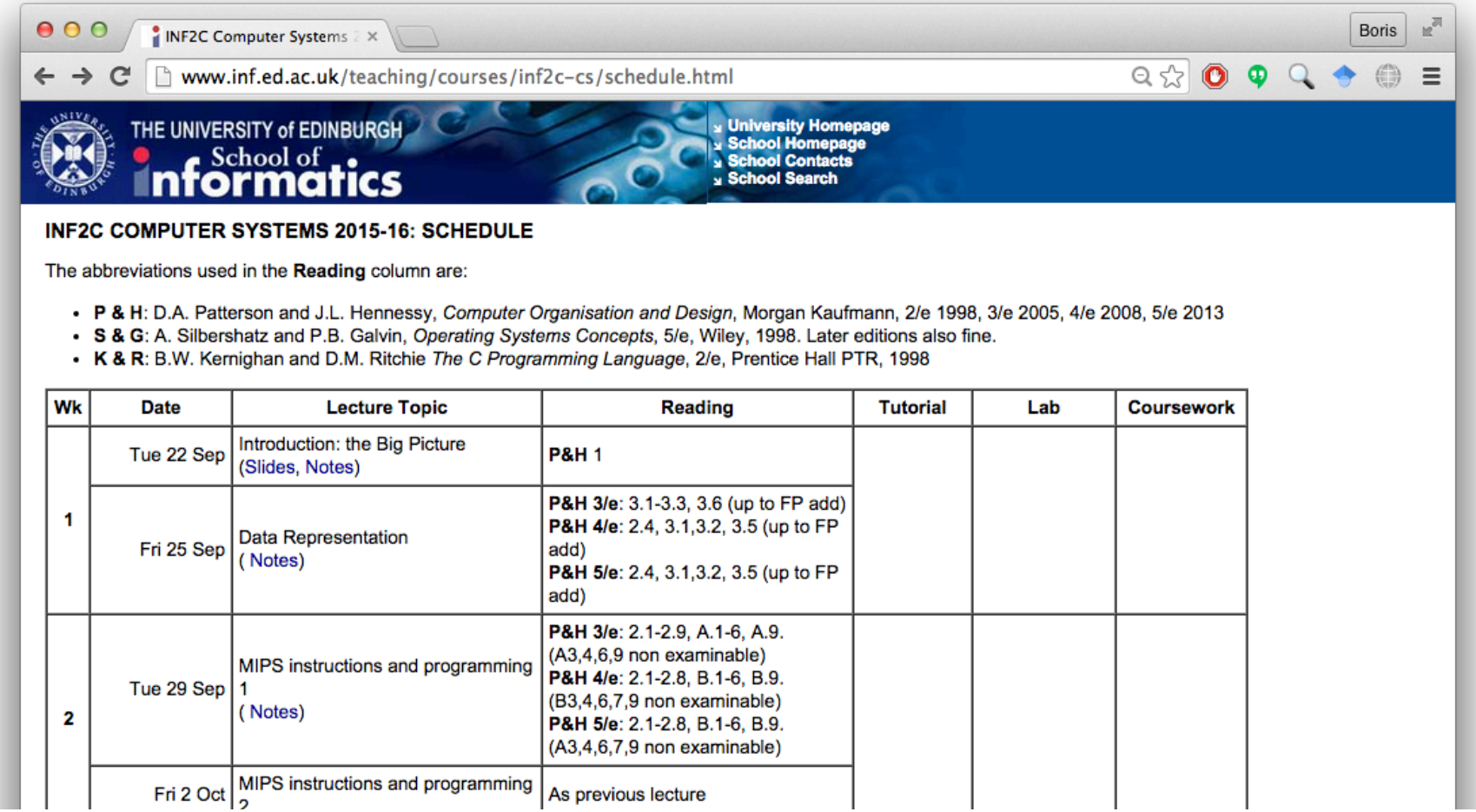
School of Informatics

University of Edinburgh

Practicalities

- Lectures:
  - Tue & Fri, David Hume Tower, Lec. Hall C @ 15:10– 16:00
- Tutorials: weeks 3, 5, 7, 9
- Drop-in labs: demonstrators available to help
- Online discussion forum: TBD
  - Primary means to Q&A outside of class.
- Notes are provided, but must read the book too
- All material are/will be on the course web-page: http://www.inf.ed.ac.uk/teaching/courses/inf2c-cs
  - Previous year's materials also online

Lecture schedule, slides, notes



INF2C Computer Systems

Boris

www.inf.ed.ac.uk/teaching/courses/inf2c-cs/schedule.html

THE UNIVERSITY of EDINBURGH School of informatics

University Homepage
School Homepage
School Contacts
School Search

INF2C COMPUTER SYSTEMS 2015-16: SCHEDULE

The abbreviations used in the **Reading** column are:

- **P & H**: D.A. Patterson and J.L. Hennessy, *Computer Organisation and Design*, Morgan Kaufmann, 2/e 1998, 3/e 2005, 4/e 2008, 5/e 2013
- **S & G**: A. Silbershatz and P.B. Galvin, *Operating Systems Concepts*, 5/e, Wiley, 1998. Later editions also fine.
- **K & R**: B.W. Kernighan and D.M. Ritchie *The C Programming Language*, 2/e, Prentice Hall PTR, 1998

Wk	Date	Lecture Topic	Reading	Tutorial	Lab	Coursework
1	Tue 22 Sep	Introduction: the Big Picture (Slides, Notes)	**P&H** 1			
	Fri 25 Sep	Data Representation (Notes)	**P&H 3/e**: 3.1-3.3, 3.6 (up to FP add) **P&H 4/e**: 2.4, 3.1,3.2, 3.5 (up to FP add) **P&H 5/e**: 2.4, 3.1,3.2, 3.5 (up to FP add)			
2	Tue 29 Sep	MIPS instructions and programming 1 (Notes)	**P&H 3/e**: 2.1-2.9, A.1-6, A.9. (A3,4,6,9 non examinable) **P&H 4/e**: 2.1-2.8, B.1-6, B.9. (B3,4,6,7,9 non examinable) **P&H 5/e**: 2.1-2.8, B.1-6, B.9. (A3,4,6,7,9 non examinable)			
	Fri 2 Oct	MIPS instructions and programming 2	As previous lecture			

Schedule will drift. It's OK.

Books

- **Required:** *Patterson & Hennessy: Computer Organization and Design, Morgan Kaufmann*
  - 5th or 4th ed recommended
  - Library has 2nd and 3rd ed (both OK, but try to get newer ed)
- *Silberschatz, Galvin, Gagne: Operating Systems Concepts, Wiley 9th ed*
  - Library has 5th and 7th ed ebook (both OK)
  - Only a few sections needed for this course
- *Kernighan and Ritchie. The C Programming Language, Prentice Hall 2nd ed*
  - Generally useful, but not mandatory for this course

Exam and Coursework

- Exam - 60%
  - In December; exact date not available yet.
  - Must achieve at least 35/100 to pass the course
- Coursework – 40%
  - Must achieve at least 25/100 to pass the course

1. MIPS assembly programming
   - Out: Tue 13 Oct (week 4)
   - Due: Tue 27 Oct (week 6) @ 4pm
2. TBD
   - Out: Tue 10 Nov (week 9)
   - Due: Tue 24 Nov (week 11) @ 4pm

Late coursework

- School-wide consistent policy:
  Normally, you will not be allowed to submit coursework late
- If you have a **good reason** to submit late, contact the ITO via their Support Form.
  - The ITO will log the report and pass it on to the UG2 Course/Year Organiser (Dr. Sharon Goldwater)
  - Only in exceptional circumstances (*e.g.*, illness that stopped you getting to email), would an extension be granted after a deadline has passed
- See the online Undergraduate Year 2 Handbook for details

Good reason

Something that, in the judgement of the member of staff responsible, would prevent a competent, well-organised, conscientious student from being able to submit on time.

Examples:

- Significant illness
- Serious personal problems

Non-examples:

- Difficult cluster of deadlines
- Last-minute computer problems, dog ate your homework, …

So what is this course about?

Syllabus Overview

- Hardware:
  - Data representation and operations
  - Design of (very) simple circuits
  - Processor organisation
  - Exceptions and interrupts
  - The memory subsystem
  - Input/Output (I/O)
- Software:
  - Low-level (assembly) programming
  - Operating systems basics
  - Introduction to C programming (2 lectures)

Words of wisdom & caution

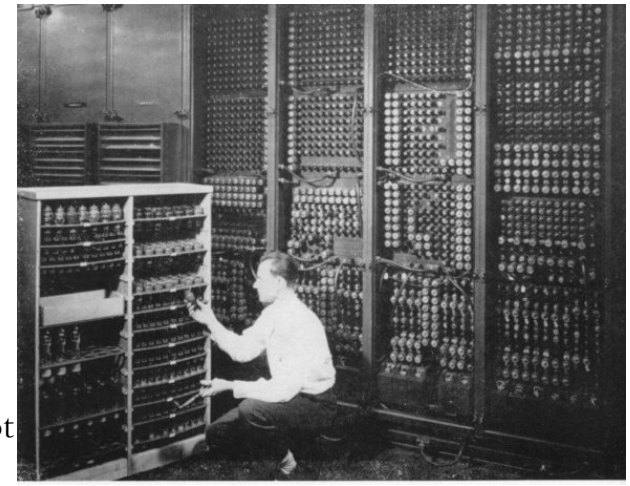
- This class covers a lot of material.
- Keeping up will require effort on your part.
  - This ain't no INF1!
- Attend all lectures, tutorials and labs.
- Seek help early.
- ASK QUESTIONS!

Reward: you will learn a lot!

Let's dig in…

Evolution of computers

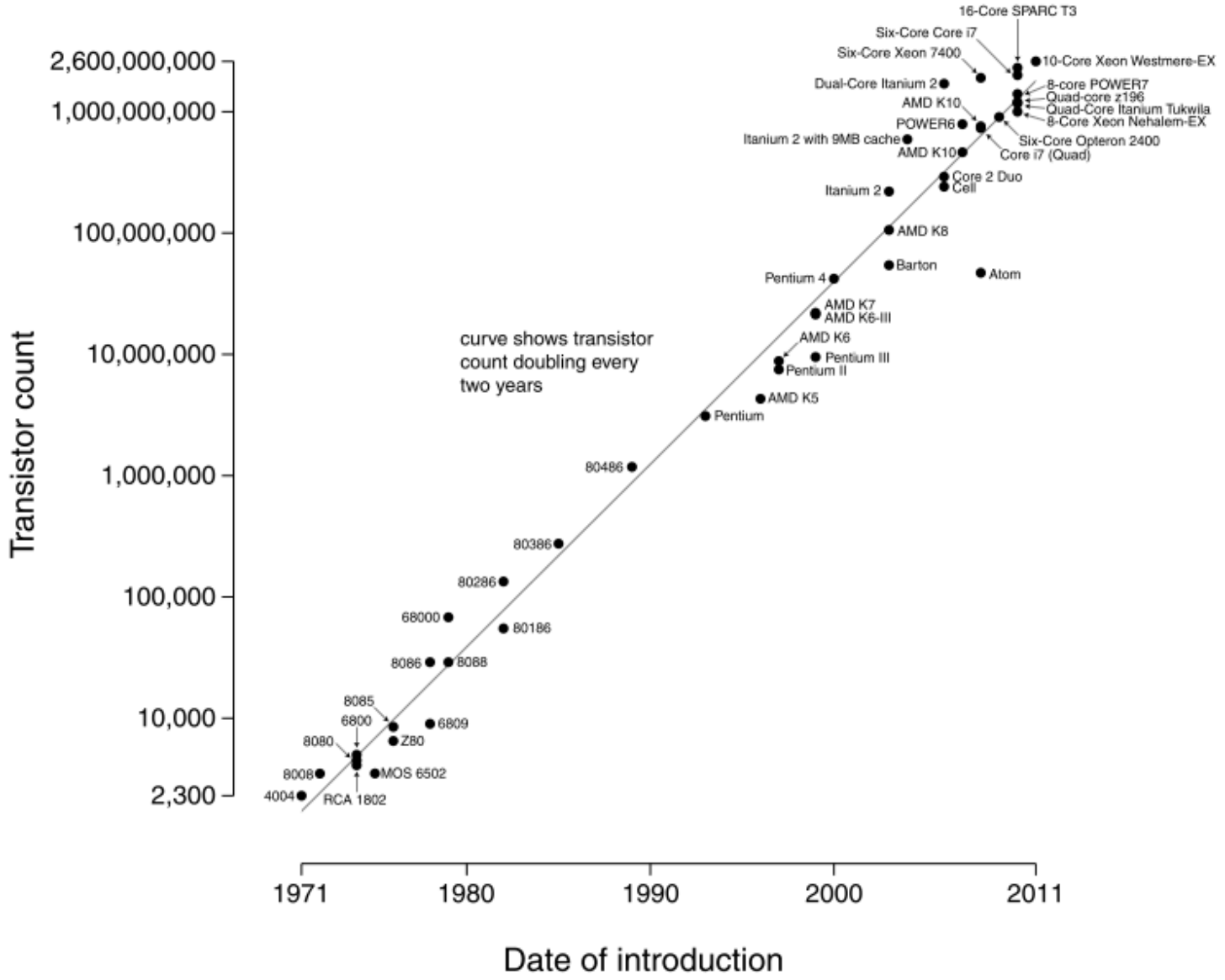
- Early computers had their programs set up by plugging cables and setting switches
- John von Neumann first proposed to store the program in the computer's memory
- All computers since then (~1945) are stored-program machines



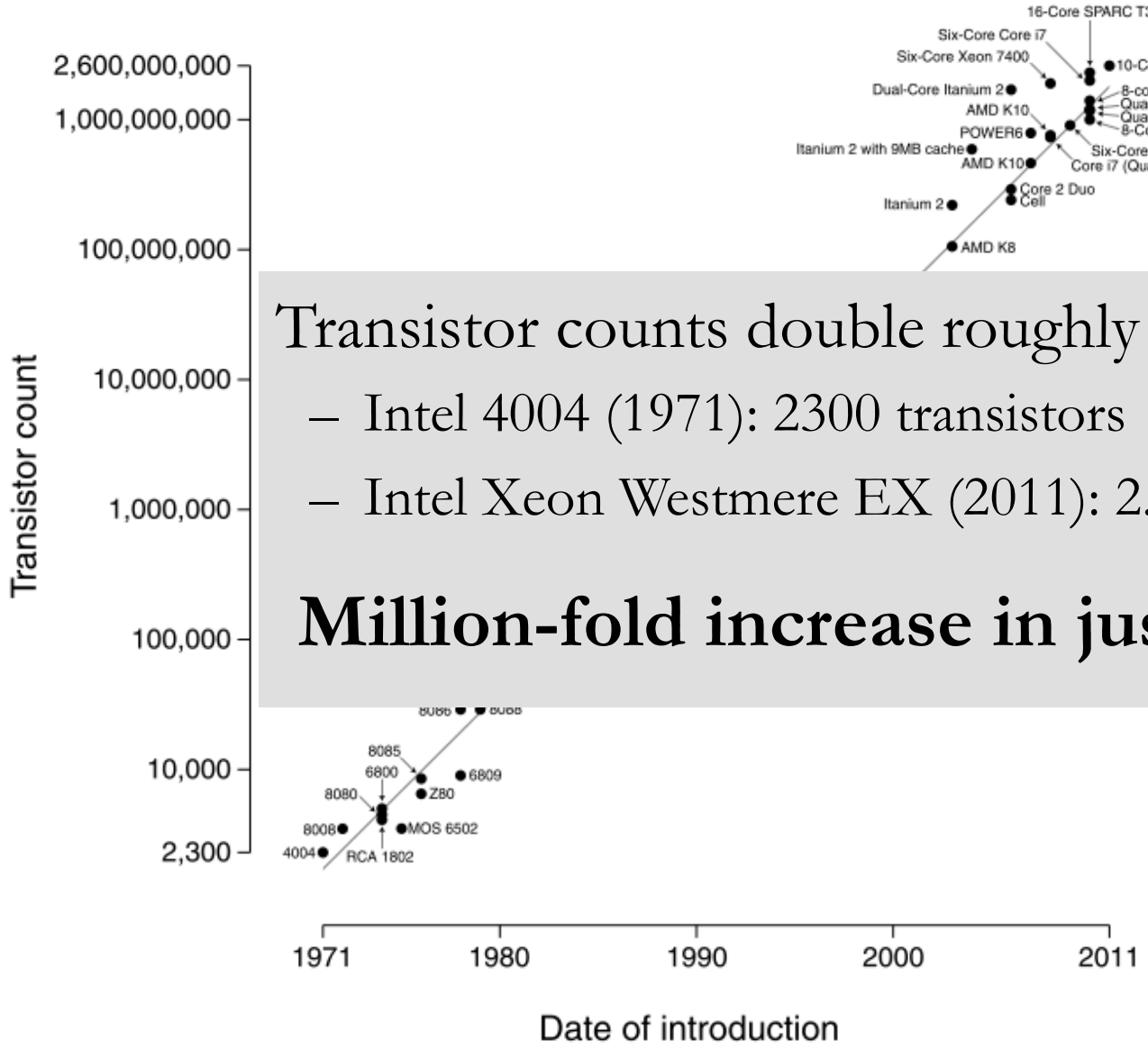
Evolution of computers

- What has changed is the number of transistors (electronic switches) and their speed
- Implementation technology progressed from vacuum tubes to discrete bipolar transistors to (eventually) Integrated Circuits (a.k.a. chips) made with complementary metal-oxide semiconductor (CMOS) technology.
- At the same time, the cost per transistor has been dropping

Moore's law


16-Core SPARC T3
Six-Core Core i7
Six-Core Xeon 7400
10-Core Xeon Westmere-EX
Dual-Core Itanium 2
8-core POWER7
Quad-core z196
AMD K10
Quad-Core Itanium Tukwila
POWER6
8-Core Xeon Nehalem-EX
Itanium 2 with 9MB cache
Six-Core Opteron 2400
AMD K10
Core i7 (Quad)
Core 2 Duo
Cell
Itanium 2
AMD K8
Barton
Atom
Pentium 4
AMD K7
AMD K6-III
AMD K6
Pentium III
Pentium II
AMD K5
Pentium
80486
80386
80286
68000
80186
8086
8088
8085
6800
6809
8080
Z80
8008
MOS 6502
4004
RCA 1802
curve shows transistor count doubling every two years
Transistor count
2,600,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
1971
1980
1990
2000
2011
Date of introduction

Moore's law



Transistor counts double roughly every 2 years

- Intel 4004 (1971): 2300 transistors
- Intel Xeon Westmere EX (2011): 2.3 bil. transistors

Million-fold increase in just 40 years!

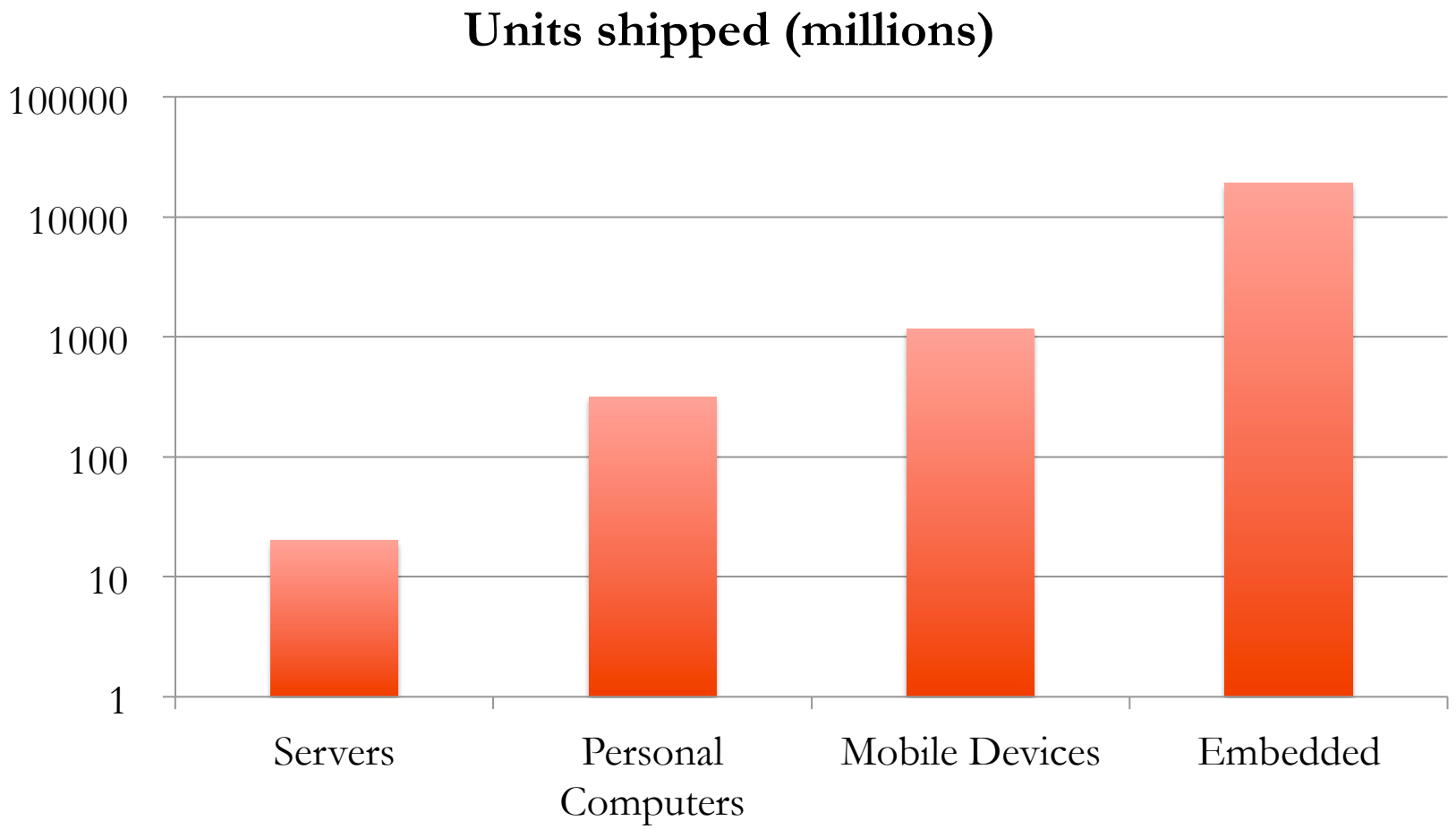
Types of computer systems

- Servers
  - Used for either few large tasks (e.g., engineering apps), or many small tasks (e.g., web server, Google)
  - Fast processors, lots of memory
  - Multi-user, multi-program
- Personal computers
  - Laptops, desktops
  - Balance cost, processing power
  - Few users, multi-program

Types of computer systems (con'd)

- Mobile devices
  - Smart phones, tablets
  - Highly integrated (multiple processors, GPU, GPS, media accelerators, etc), low-power
  - Single-user, multi-program
- Embedded:
  - Task specific: sensing, control, media playback, etc.
  - Low-cost, low-power
  - Single program

Which computer system category is the largest?

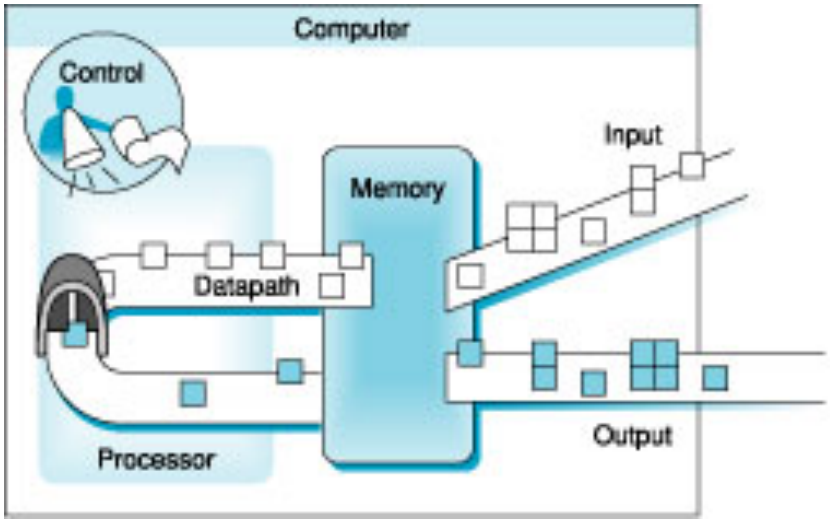


Computer components

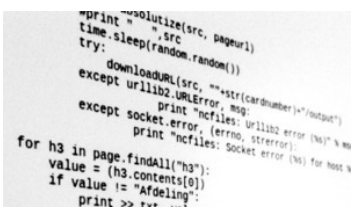
- Data path
  - Performs actual operations on data
- Control path
  - Fetches instructions from program in memory
  - Controls the flow of data through the data path

} Processor

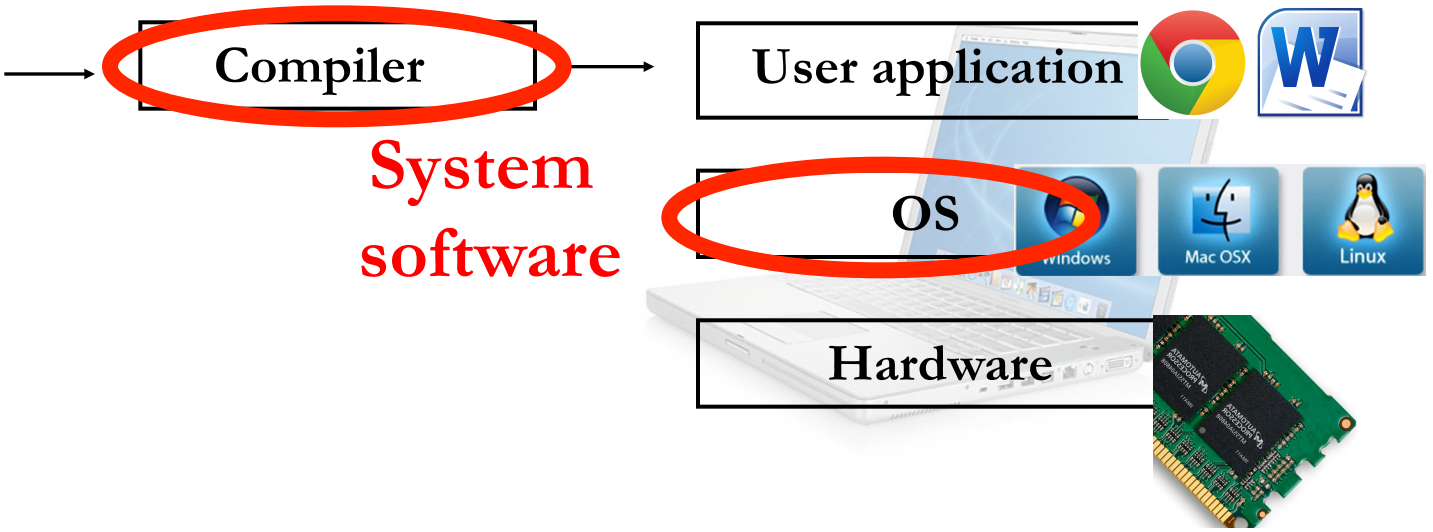
- Memory
  - Stores data and instructions
- Input/Output
  - Interfaces with other devices for getting/giving data



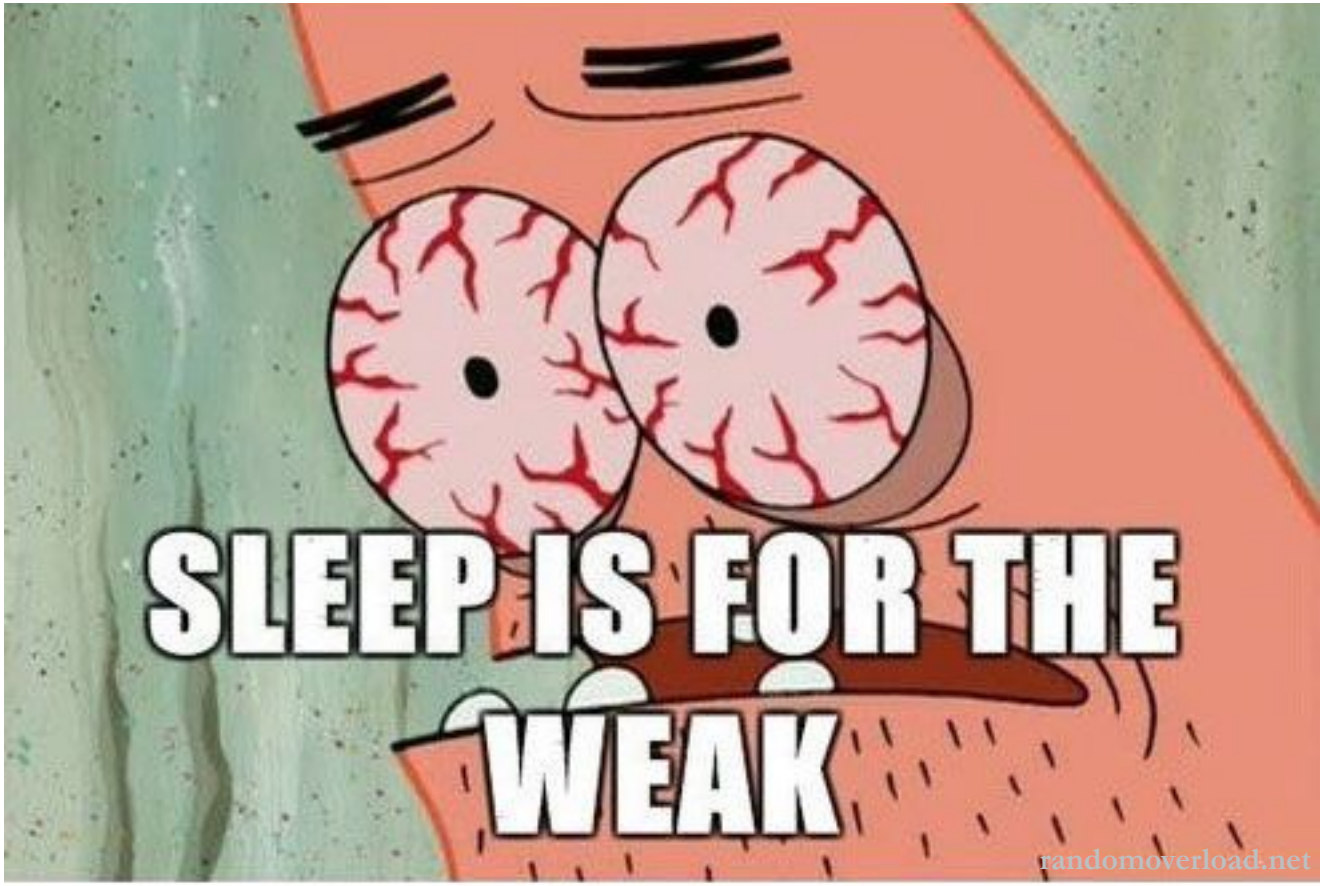
Modern computer system



Source code



- Compiler
  - Translates **High Level Language** (**HLL**) into **machine language** or **byte code**
- Operating System (OS)
  - Mediates access to hardware resources (CPU, Memory, I/O)
  - Schedules applications


SLEEP IS FOR THE
WEAK
randomoverload.net

Inf2C - Computer Systems
Lecture 2
Data Representation

Boris Grot

School of Informatics

University of Edinburgh

Last lecture

- Moore's law
- Types of computer systems
- Computer components
- Computer system stack

Lecture 2: Data Representation

- The way in which data is represented in computer hardware affects
  - complexity of circuits
  - cost
  - speed
  - reliability
- Must consider how to design hardware for
  - Storing data - memories
  - Manipulating data – e.g. adders, multipliers

Lecture outline

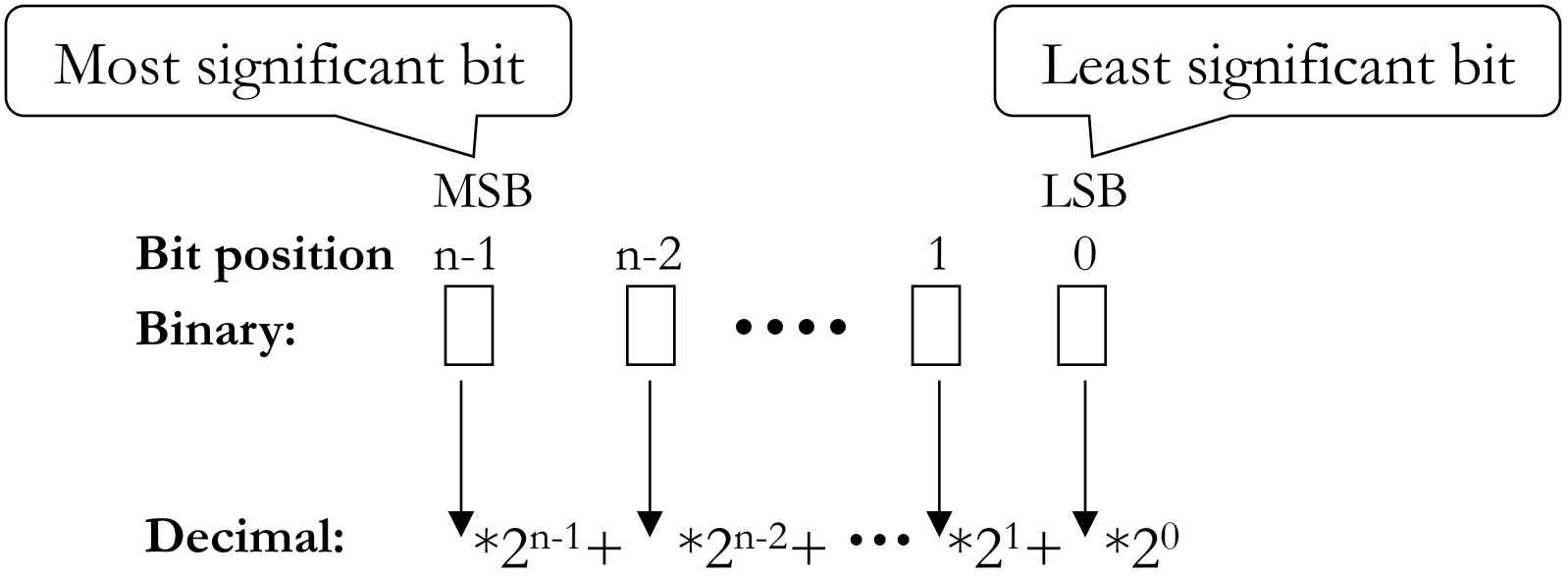
- The bit – atomic unit of data
- Representing numbers
- Representing text

The bit

- Information represented as sequences of symbols
  - In text, symbols are letters, numerals, punctuation, whitespace
  - With computers, we use just 0s and 1s, *bits*
- *Bit* is an acronym for Binary digiT
- Advantages: easy to do computation, very reliable, simple circuits
- Disadvantages: little information per bit, must use many of them. 512 ≡ 1 0000 0000, ‘A’ ≡ 0100 0001

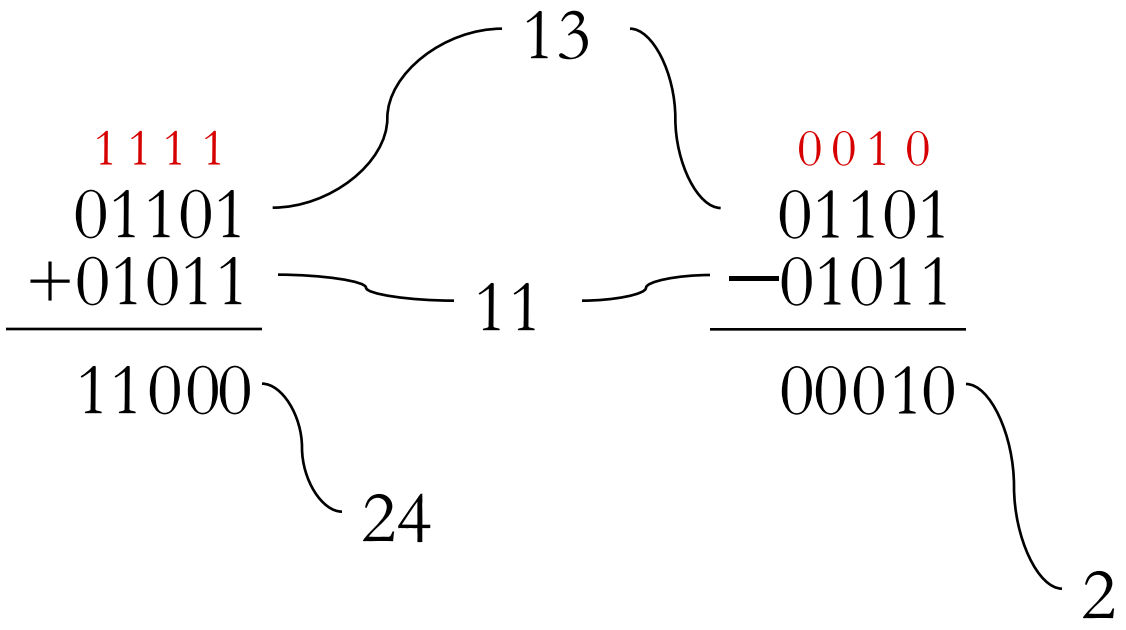
Natural numbers representation

- Non-negative (unsigned) integers are very simple to represent in binary



Basic operations

- Addition, subtraction with binary numbers is easy:



Fixed bit-length arithmetic

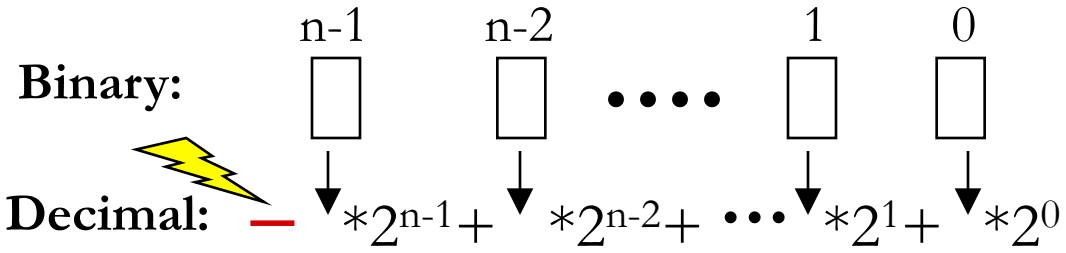
- Hardware cannot handle infinite long bit sequences
- We end up with a few fixed sized data types
  - Byte: always 8 bits
  - Word: the typical unit of data on which a processor operates (32 or 64 bits most common today)
- Overflow happens when a result does not fit
  - Numbers wrap-around when they become too large
  - Arithmetic is modulo 2^n, n=number of bits

What about negative numbers?

- Sign-magnitude representation:
  - Use 1st bit (MSB) as the sign: 1-negative, 0-positive
    0010 ≡ 2 1010 ≡ -2
- Complicates addition and subtraction
  - The actual operation depends on the sign
- Has positive and negative zero
  - 0000 ≡ 0 1000 ≡ -0
- There is a better way

Two's complement representation

- If doing mod 2k arithmetic on numbers 0…2k-1, treat numbers k…2k-1 as -k…-1
- To find the value of a binary number, consider the MSB as having negative weighting:



- To negate a number:

  Invert all bits (0 ↔ 1) and add 1, at the LSB

  – Note: −(−2n-1) overflows!

2's complement details

- The MSB is the sign
- Range is asymmetric: -2^{n-1} to $2^{n-1}-1$
- There are two kinds of overflows:
  - Positive overflow produces a negative number
  - Negative underflow produces a positive number
- A – B = A + 2's complement of B
- Arithmetic operations do not depend on the operands' signs
  - 0010 ≡ 2 1010 ≡ -6

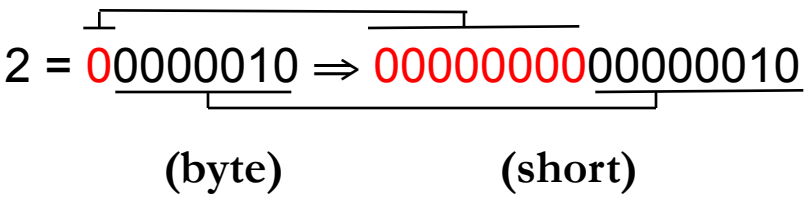
Converting between data types

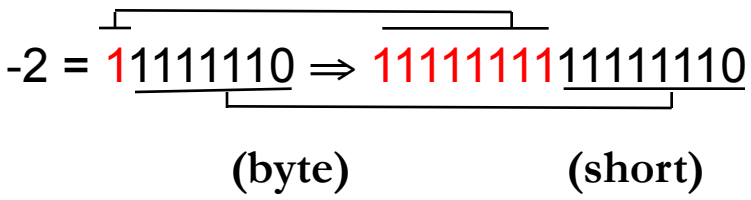
- Converting a 2's complement number from a smaller to a larger representation is done by sign extension

Example: from byte to short (16 bits):

2 = 0 0 0 0 0 0 1 0 ⇒ ? ? ? ? ? ? ? ? 0 0 0 0 0 0 1 0

-2 = 1 1 1 1 1 1 1 0 ⇒ ? ? ? ? ? ? ? ? 1 1 1 1 1 1 1 0





Shifting

- Shifting: move the bits of a data type left or right
  - Data bits falling off the edge are lost
- 0s fill up the empty bit places for left shifts
- For right shifts, two options:
  - Fill with 0: for non-numerical data (or positive integers)
  - Fill with the MSB: for 2's complement numbers
- Shift left by *n* is equivalent to multiplying by 2^n
- Shift right by *n* is equivalent to dividing by 2^n and rounding towards $-\infty$
- Example

  6 = 0 0 0 0 0 1 1 0 >> 2 → 0 0 0 0 0 0 0 1 = 1

  -6 = 1 1 1 1 1 0 1 0 >> 2 → 1 1 1 1 1 1 1 0 = -2

Hexadecimal notation

- Binary numbers (and other data) are too long and tedious for us to use
- Hexadecimal (base 16) is very commonly used in computer programming
- Hex digits: 0-9 and A-F
  - A=10, B=11, …, F=15
- Conversion to/from binary is very easy:

  Every 4 bits correspond to 1 hex digit:

  1 1 1 1 1 0 0 0 = 0xF8

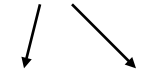
  F(15) 8

Hex is just a convenience, computers use the binary form

Real numbers - floating point

- Java's **`float` (32 bits)**
  `double` (64 bits)

- Binary representation:
  - example 0.75 in base 10 $\Rightarrow$ 0.11 in base 2



$(2^{-1} + 2^{-2} = 0.5 + 0.25 = 0.75)$

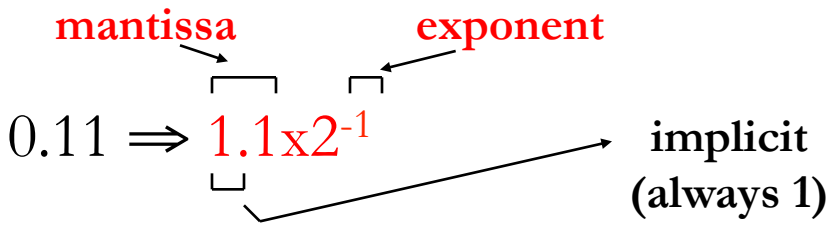
Real numbers - floating point

- Java's `float` **(32 bits)**
  `double` **(64 bits)**

- Binary representation:
  - example 0.75 in base 10 ⇒ 0.11 in base 2

$(2^{-1} + 2^{-2} = 0.5 + 0.25 = 0.75)$

- Normalization:



Why normalize?

Three reasons:

1. Simplifies machine representation
   (don't need to represent the fraction separator)
2. Simplifies comparisons
   - Which one is bigger: 0.001 or 1.01×2^{-2} ?
3. Is more compact (in some cases)
   - E.g., $0.0000000000000001 = 1.0 \times 2^{-16}$

   or can be made more compact (by rounding fraction)

Floating point conversion example #1

Convert the number 25 to floating point with normalization

1) 25 in base 10 ⇒ 11001 in base 2

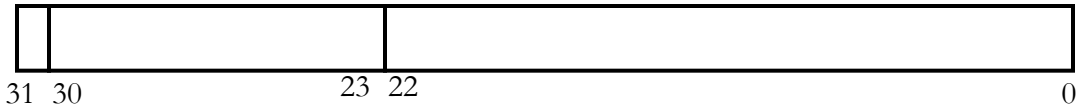
2) 11001 to normalized floating point ⇒ 1.1001×2^4

Understand that:

- 1.1001 is mantissa (aka significand)
- 4 is exponent
- sign is "+" (implicit here)

IEEE 754 Floating Point standard

- 32 bit:



sign (s) **exponent (exp)** **mantissa (m)**

$$(-1)^s \times (1.m) \times 2^{exp-127}$$ **Bias**

e.g.,

$(0.75)_{10} \rightarrow (0.11)_2 \rightarrow (1.1 \times 2^{-1})_2$

$\rightarrow$ s = 0, m = 1, exp = 126

$\rightarrow$ 0 01111110 10000000000000000000000

- 64 bit:
  - exponent = 11 bits; mantissa= 52 bits

IEEE 754 Floating Point standard

- Why bias?
  - Avoids the complexity of +/- exponents
  - Simplifies relative ordering of FP numbers

- Note: processors usually have specialized floating point units to perform FP arithmetic

IEEE 754 floating point conversion #2

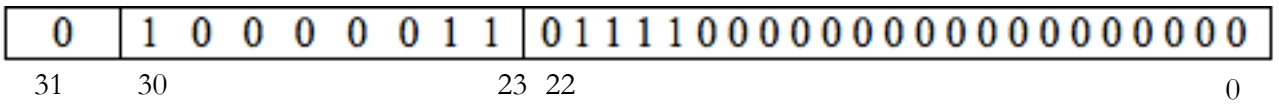
Example: Convert 23.5 (decimal) to IEEE 754 floating point

Start: 23 in base 10 → 10111 in base 2

IEEE 754 floating point conversion #2

Example: Convert 23.5 (decimal) to IEEE 754 floating point

1) 23.5 in base 10 ⇒ 10111.1 in base 2
2) 10111.1 to normalized floating point ⇒ 1.01111×2^4
3) S = 0
   M = 01111 is mantissa (remember: 1. is implicit)
   Exp = 4+127 = 131 in base 10 ⇒ 1000 0011 in base 2



sign (s) **exponent (exp)** **mantissa (m)**

IEEE 754 Floating Point notation

Exponent	Mantissa	Meaning
0	0	0
1-254	Anything	Floating point number
255	0	infinity
255	Non-zero	Not-a-number (NaN)

32-bit representation

Representing characters

- Characters need to be encoded in binary too
- Operations on characters have simpler requirements than on numbers, so the encoding choice is not crucial
- Most common representation is ASCII
  - Each character is held in a byte
  - E.g. '0' is 0x30, 'A' is 0x41, 'a' is 0x61
- Java uses Unicode which can encode characters from many (all?) languages
  - 16 bits per character required

Representing strings

- Words, sentences, etc. are just strings of characters
- How is the end of a string identified?
  - No common standard exists. Different programming languages use different encodings
  - In C: a special character, encoded as 0x00
  - In Java: string length is kept with the string itself (string is an object and length is one of the member variables)

Summary

- Computers use binary representation
- 2's complement
- Floating point
- Characters and strings

Inf2C - Computer Systems
Lectures 3-4
Assembly Language

Boris Grot

School of Informatics

University of Edinburgh

Announcements: Labs

- Purpose:
  - prep for courseworks by getting familiar with the tools
  - reinforce class concepts
- Regular schedule (starting next week):
  - Mon: 2-3pm, 5-6pm in FH 3.D02
  - Wed: 3-4pm in FH 3.D01
  - Thr: 11am-12pm in FH 1.B31
- Extras (whenever coursework is out)
  - Tue: 11am-12pm in FH 3.D01
  - Wed: 2-3pm in FH 3.D01
  - Fri: 1-2pm in FH 3.D01
- Drop-in format
  - You may attend any & all lab sessions!
  - Demonstrator (Kuba or Arpit) on-hand to answer questions

Announcements: Other

- Tutorial 1 running next week
  - Attempt the problems ahead of time
  - Come in with specific questions
  - You'll get out what you put in
- Online discussion forum: ASK
  - URL to be confirmed

Previous lecture

- Representing numbers
  - Binary encoding
  - Negative numbers
  - Floating point numbers
  - Characters & strings
- Other things
  - Binary addition
  - Shifting
  - Hex

Lectures 3-4: MIPS instructions

- Motivation: Learn how a processor's 'native' language looks like
- We will examine the MIPS ISA
  - MIPS: Microprocessor without Interlocked Pipeline Stages – a real-world ISA used by many different processors since the 80s.
  - Regular and representative → great for learning!
- ISA reference can be downloaded from:
  - `http://www.cs.wisc.edu/~larus/HP_AppA.pdf`

Outline

- Instruction set
- Basic arithmetic & logic instructions
- Processor registers
- Getting data from the memory
- Control-flow instructions
- Method calls

Processor instructions

- Instruction set architecture (ISA): the interface between the software and the hardware
  - Collection of all machine instructions recognized by a particular processor
  - Also, privilege levels, memory management, etc.
- The ISA abstracts away the hardware details from the programmer
  - Similar to how an object hides its implementation details from its users
  - Enables multiple implementations (called microarchitectures) of the same ISA.

Assembly language

- Instructions are represented internally as binary numbers
  - Example: 00000011110100100000001000001011
  - For a human, very hard to make out which instruction is which (sequence of 32-64 bits!)
- Assembly language: symbolic representation of machine instructions

MIPS assembly: a simple example

High-level language (HLL): `a[0]=b[0]+10`

MIPS assembly language:

```
lw  r4,0(r2)  # get the value of b[0] from memory
              #  and store it in register r4
add r5,r4,10  # compute b[0]+10 and store into r5
sw  r5,0(r1)  # store r5 into a[0]
```

Things to notice:

- Separate instructions to access data (in memory) & operate on it
  - Cannot operate on memory directly
- All instructions have similar format

MIPS arithmetic & logical operations

- Data processing instructions look like:

  `operation destination, 1st operand, 2nd operand`

  `add a,b,c` a = b+c

  `sub a,b,c` a = b−c

- Bit-wise logical instructions: `and, or, xor`

- Shift instructions:

  `sll a,b,shamt` a = b << shamt

  `srl a,b,shamt` a = b >> shamt (logical)

  `sra a,b,shamt` a = b >> shamt (arithmetic)

Registers

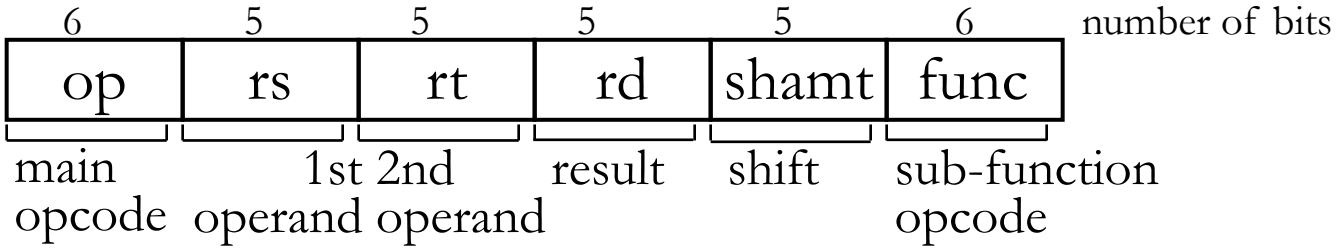
- ISA places restrictions on instruction operands
  - How many operands and where they come from
- MIPS processors operate on registers only
  - Registers are storage locations inside the processor that hold program variables
- Registers are sized to contain machine's word
  - 32 or 64 bits common today
- Processors have relatively few registers
  - MIPS has 32
  - x86 has 8-16

MIPS instruction example

- Assembly:

```
add $s1, $s2, $s3    # $s1 = $s2 + $s3
```

- Binary (R-format – used for arithmetic instructions):



0	18	19	17	0	32_{10}	`add $s1, $s2, $s3`
0	18	9	8	0	34_{10}	`sub $t0, $s2, $t1`

Each and every assembly instructions translates to exactly 1 machine instruction (no choice or ambiguity)

More on MIPS registers

- Most registers are available for any use
  - with a few important exceptions
- Program (C/Java) variables held in regs `$s0-$s7`
- Temporary variables: `$t0-$t9`
- Registers with special roles
  - Register 0 (`$zero`) is hardwired to 0
  - Program Counter (PC) holds address of next instruction to be executed (it is not a general purpose registers)
  - Other special-purpose registers exist

Immediate operands

- What if we need to operate on a constant?
  - Common in arithmetic operations, loop index updates (e.g., `i++`), or even character manipulations (e.g., changing the case of an ASCII character)
- MIPS has dedicated instructions with one constant (immediate) operand
  - e.g. `addi $r1, $r1, 1  # r1=r1+1`
- General form: *op*`i $r1, $r2, n`

Loading a constant operand

- Load a (small) constant into a register:
  - Constant is 16 bits and is signed

```
addi $s0,$zero,n  # $s0=n ($s0_31-16=sign ext; $s0_15-0=n)
```

- What if need a larger/smaller constant?
- Assembler pseudo-instruction `li reg,constant`
  - Translated into 1 instruction for immediates < 16bits and into more instructions for more complicated cases (e.g. 2 instructions for a 32-bit immediate)

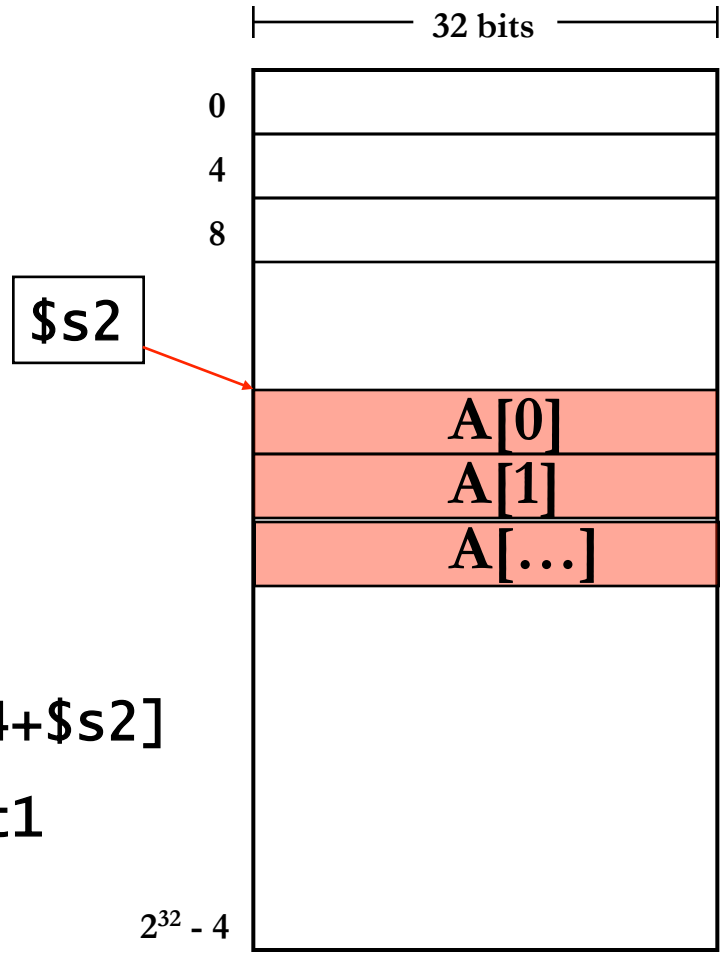
```
lui $s1,n1     # $s1_31-16=n1; $s1_15-0=0
ori $s1,$s1,n2 # $s1_31-16=n1; $s1_15-0=n2
```

Getting at the data

- Programming statement:

  `g = h + A[1]`

  where
  - h is in register $s1
  - A[0] is the first element of array A and is pointed to by $s2

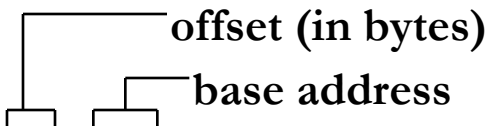


- MIPS:

offset

```
lw  $t1,4($s2)    # $t1=memory[4+$s2]
add $t2,$s1, $t1  # $t2 = h + $t1
```

Data-transfer instructions



- Load Word:

```
lw r1,n(r2)  # r1=memory[n+r2]
```

- Store Word:

```
sw r1,n(r2)  # memory[n+r2]=r1
```

- Load Byte:

```
lb r1,n(r2)  # r1_7-0= memory[n+r2]
               r1_31-8= sign extension
```

- Store Byte:

```
sb r1,n(r2)  # memory[n+r2]=r1_7-0
               no sign extension
```

Memory addressing

- Memory is byte addressable, but it is organised so that a whole word can be accessed directly

- Where can a word be stored?
  - Option 1: anywhere (unaligned)
  - Option 2: at an address that is a multiple of the word size (aligned)
  - Both options in use today: MIPS requires alignment, x86 doesn't
  - What are the trade-offs?

Memory addressing: Endianness

Given a memory address, Endianness tells us where to find the starting byte of a word

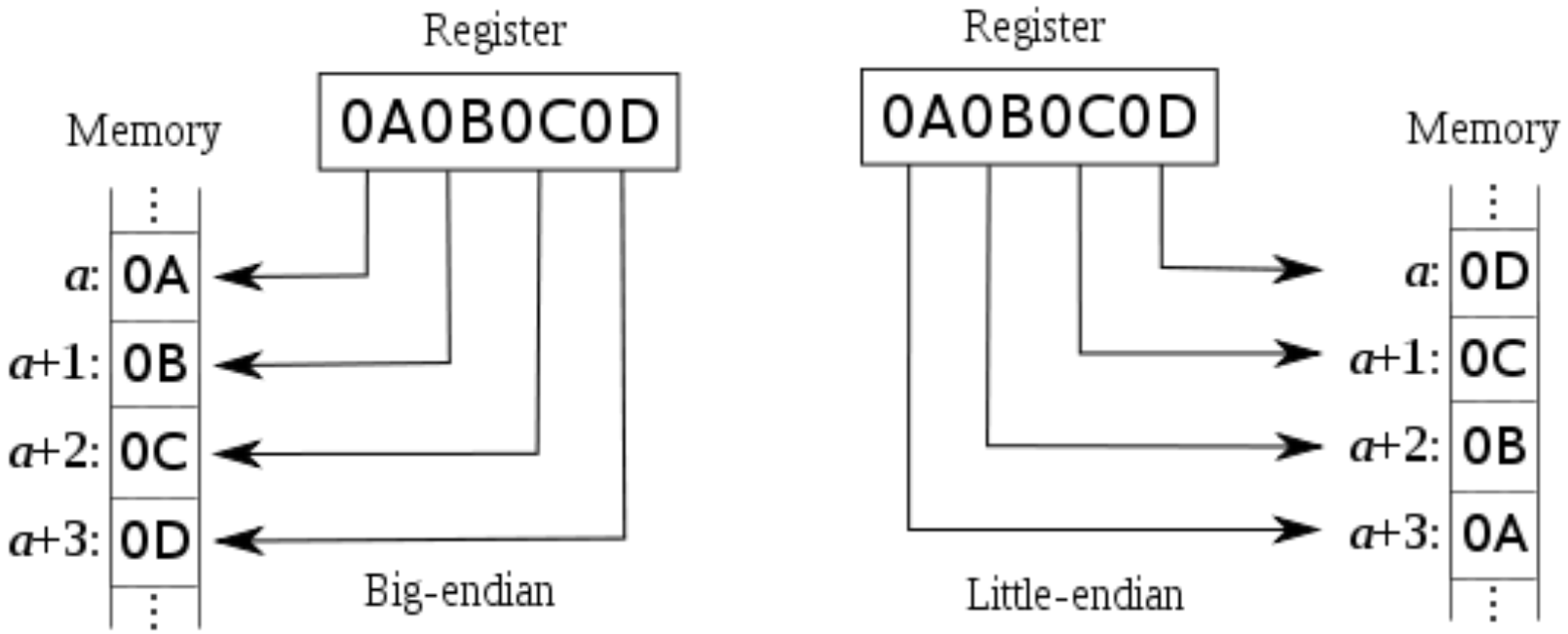
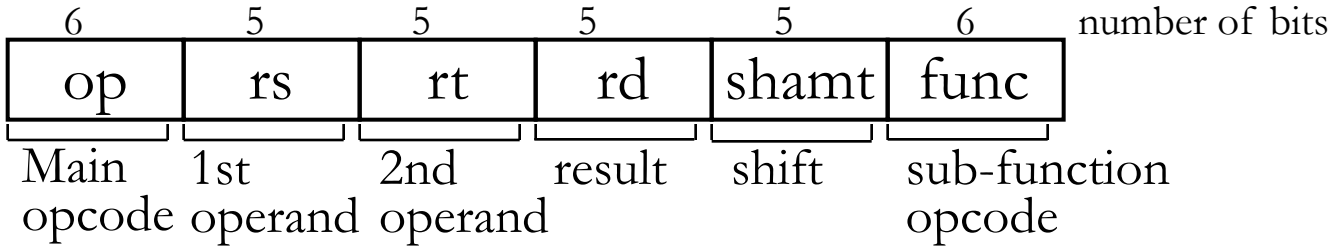


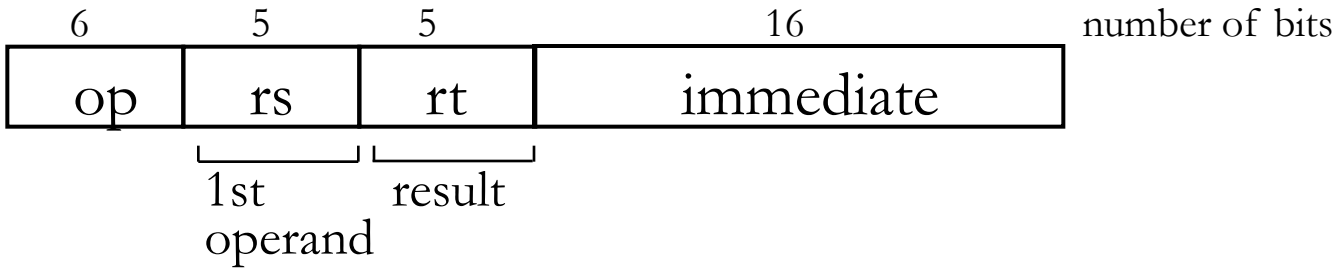
Image source: http://en.wikipedia.org/wiki/Endianness

Instruction formats

- Instruction representation composed of bit-fields
- Similar instructions have the same format
- MIPS instruction formats:
  - R-format (`add`, `sub`, …)



  - I-format (`addi`, `lw`, `sw`, …)



MIPS instructions – part 2

- Last time:
  - Data processing instructions: add, sub, and, …
    - Registers only and immediate types
  - Data transfer instructions: lw, sw, lb, sb
  - Instruction encoding
- Today:
  - Control transfer instructions

A simple program to swap array elements

```
1  int main(void) {
2    int size = 6;
3    int v[] = {1, 10, 2, 20, 3, 30};  // array w/ 6 elements
4
5    for (int i=0; i<size; i+=2)
6      swap(v, i);  // pass array (by reference) and index i
7  }
8
9  void swap(int v[], int idx) {
10   int temp;
11   temp = v[idx];
12   v[idx] = v[idx+1];
13   v[idx+1] = temp;
14 }
```

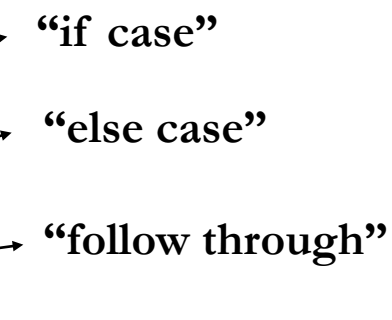
Swap() in MIPS

```
1   swap:
2         # inputs: $a0 - array base, $a1 - index
3         # Compute the address into the array
4         sll $t1, $a1, 2     # reg $t1 = idx * 4
5         add $t1, $a0, $t1   # reg $t1 = v + (idx*4)
6                             #  $t1 holds the addr of v[idx]
7
8         # Load the two values to be swapped
9         lw  $t0, 0($t1)     # reg $t0 = v[idx]
10        lw  $t2, 4($t1)     # reg $t2 = v[idx+1]
11
12        # Store the swapped values back to memory
13        sw  $t2, 0($t1)     # v[idx] = $t2
14        sw  $t0, 4($t1)     # v[idx+1] = $t0
```

Control transfers: If structures

Java/C:

```
if (i!=j)
   stmnt1
else
   stmnt2
stmnt3
```



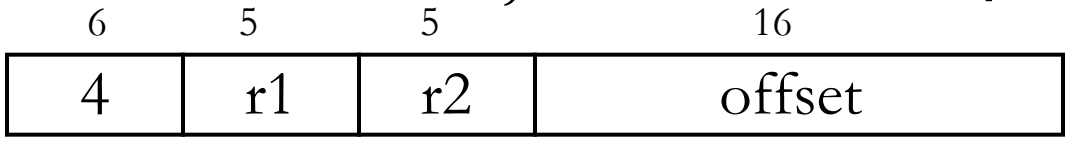
MIPS: "branch if equal" `beq $s1,$s2,label`

- compare value in $s1 with value in $s2
- if equal, branch to instruction marked label

```
          beq $s1,$s2,label2
          stmnt1
          j label3  # skip stmnt2
label2:   stmnt2
label3:   stmnt3
```

Control transfer instructions

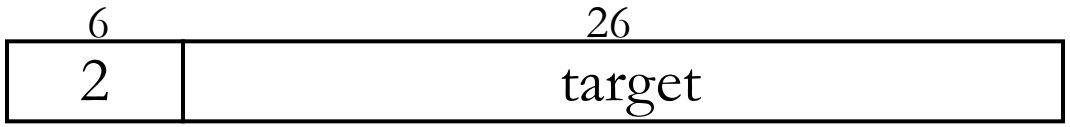
- Conditional branches, I-format: **`beq r1,r2,label`**



  - In assembly code label is usually a string
  - In machine code, label is obtained from immediate operand as: `branch target = PC + 4 * offset`

- Similarly: **`bne r1,r2,label  # if r1!=r2 go to label`**

- Unconditional jump, J-format: `j label`



Loops in assembly language

- Java/C: `while (count!=0) stmnt`
- MIPS:

```
loop:
        beq $s1,$zero,end  # $s1 holds count
        stmnt
        j loop  # branch back to loop
end:    …
```

Loops in assembly language

- Java/C: `while (flag1 && flag2) stmnt`
- MIPS:

```
loop:
        beq $s1,$zero,end  # $s1 holds flag1
        beq $s2,$zero,end  # $s2 holds flag2
        stmnt
        j loop  # branch back to loop
end:    …
```

Comparisons

- "Set if less than" (R-format): `slt r1,r2,r3`
  - set r1 to 1 if r2<r3, otherwise set r1 to 0
- Java/C: `while (i > j) stmnt`
- MIPS example:
  - assume that $s1 contains i and $s2 contains j

```
loop:
        slt $t0,$s2,$s1     # $t0 = (j < i)
        beq $t0,$zero,end   # branch if i <= j
        stmnt
        j loop  # jump back to loop
end:    …
```

MIPS Instruction Format Summary

- R-type (register to register)
  - three register operands
  - most arithmetic, logical and shift instructions
- I-type (register with immediate)
  - instructions which use two registers and a constant
  - arithmetic/logical with immediate operand
  - load and store
  - branch instructions with relative branch distance
- J-type (jump)
  - jump instructions with a 26 bit address

Practice problem:

What C code does the following piece of MIPS assembly code correspond to?

```
    slt $t0, $s1, $s2
    beq $t0, $zero, l1
    and $s3, $s1, $s2
    j l2
l1: or $s3, $s2, $s1
l2:
```

(a) `if (s1 < s2) s3 = s2 | s1; else s3 = s1 & s2;`

(b) `if (s1 <= s2) s3 = s2 | s1; else s3 = s1 & s2;`

(c) `if (s1 < s2) s1 = s3 | s2; else s2 = s3 & s1;`

(d) `if (s1 <= s2) s2 = s3 & s1; else s1 = s3 | s2;`

(e) `if (s1 < s2) s3 = s1 & s2; else s3 = s2 | s1;`

Common MIPS Arithmetic & Logical Operators

- Integer Arithmetic
  - `+` add
  - `-` sub
  - `*` mul
  - `/` div
  - `%` rem

- Shifts
  - `>>` (signed) shift-right-arithmetic
  - `>>` (unsigned) shift-right-logical
  - `<<` shift-left-logical

- Bit-wise logic
  - `|` or
  - `&` and
  - `^` xor
  - `~` not

- Boolean
  - `||` (src1 != 0 or src2 != 0)
  - `&&` (src1 != 0 and src2 != 0)

Common MIPS Relational Operators

- Relational
  - < slt, sltu
  - <= sle, sleu
  - > sgt, sgtu
  - >= sge, sgeu
  - == seq
  - != sne

C operator	Comparison	Reverse	Branch
==	seq	0	bne
!=	seq	0	beq
<	slt, sltu	0	bne
>=	slt, sltu	0	beq
>	slt, sltu	1	bne
<=	slt, sltu	1	beq

Method calls

- Method calls are essential even for a small program
- Most processors provide support for method calls
- Java/C:

```
…
foo();
…
foo();
…

void foo() {
…
return;
}
```

call to foo at line L1

call to foo at line L2

where do we return to?

MIPS support for method calls

- Jumping into the method: `jal label`
  - "jump and link":
    - set $ra to PC+4
    - set PC to label
  - Another J-format instruction

- Returning: `jr r1`
  - "jump register": set PC to value in register r1

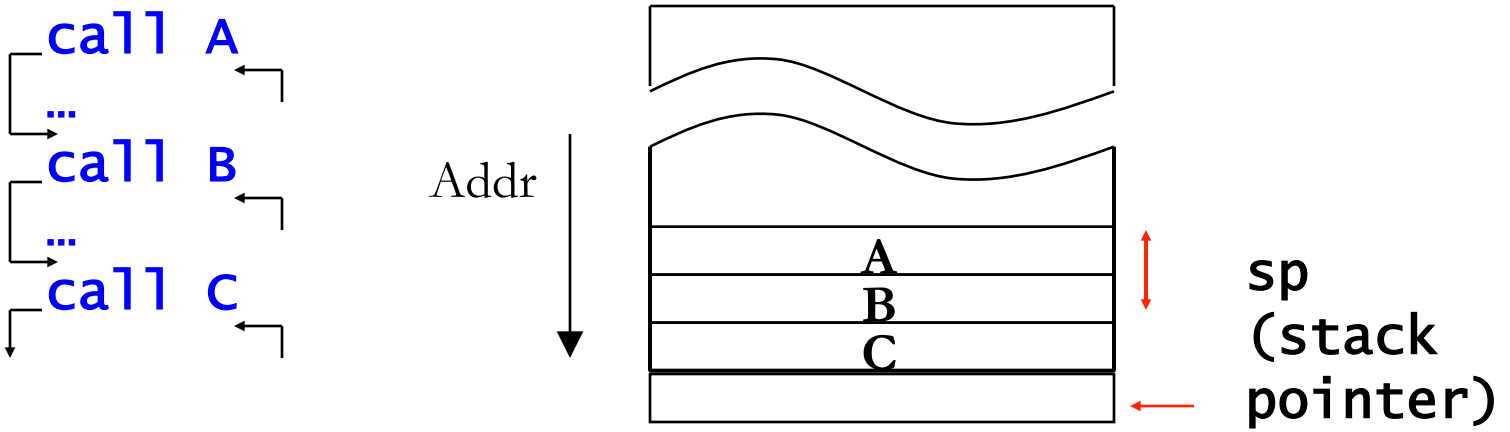
MIPS register convention on method calls

- Method parameters: $a0 - $a4
- Return values: $v0, $v1
- Regs preserved across call boundaries: $s0 - $s7
- Regs **not** preserved across call boundaries: $t0 - $t9

What about nested method calls?

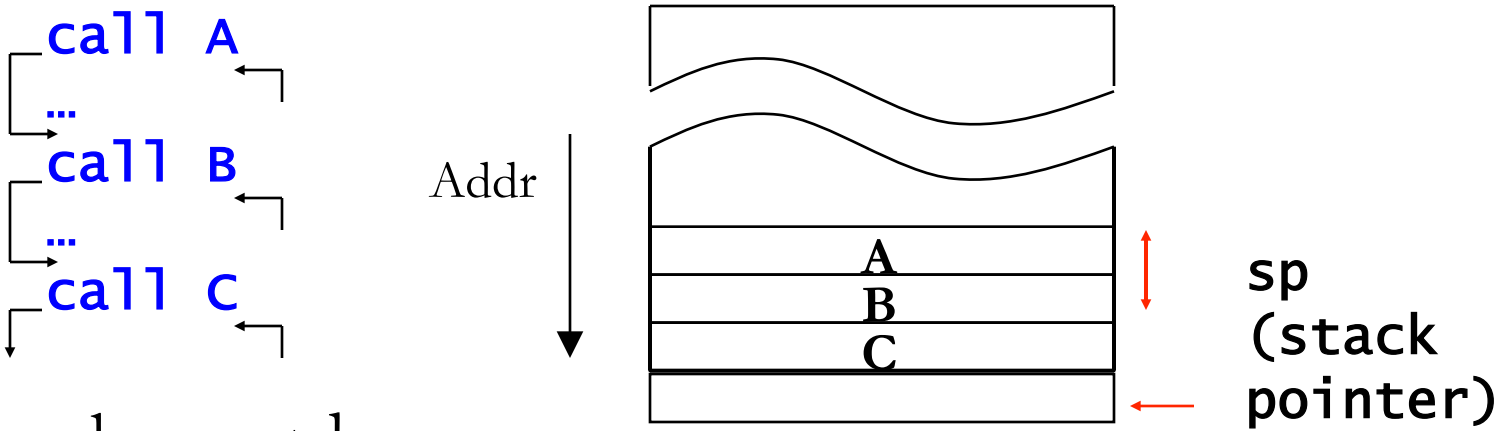
Using a stack for method calls

- Nested calls ⇒ must save return address & $t registers to prevent overwriting. Solution: use a stack in memory



Using a stack for method calls

- Nested calls ⇒ must save return address & $t registers to prevent overwriting. Solution: use a stack in memory



- to push a word:

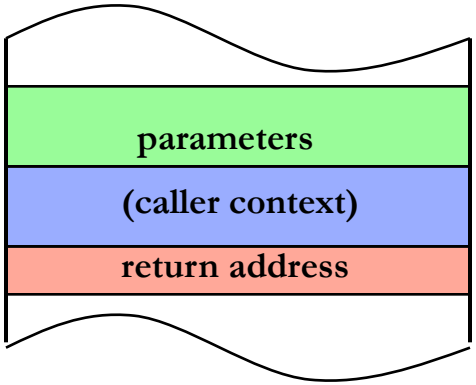
```
addi $sp,$sp,-4  # move sp down
sw   $ra,0($sp)  # save r1 on top of stack
```

- to pop a word:

```
lw   $ra,0($sp) # fetch value from stack
addi $sp,$sp,4  # move sp up
```

Other uses of the stack

- Stack used to save caller's registers, so that they can be used by the callee
  - "caller save" or "callee save" convention
- Stack can also be used to pass and return parameters
  - Gets around the limited number of parameter and return value registers



Main() in MIPS

```
1   main:
2         # Initialize variable
3         la   $s0, array     # $s0: base addr of the array
4         addi $s1, $zero, 0 # $s1: index into the array
5         addi $s2, $zero, 6 # $s1: array size
6
7   loop:
8         move $a0, $s0       # $a0 = $s0 (array base pointer)
9         move $a1, $s1       # $a1 = $s1 (index)
10        jal  swap           # call swap
11
12        addi $s1, $s1, 2    # increment the index
13        bne  $s1, $s2, loop
```

Swap(): what's missing?

```
1   swap:
2           # Compute the address into the array
3           sll $t1, $a1, 2     # reg $t1 = idx * 4
4           add $t1, $a0, $t1   # reg $t1 = v + (idx*4)
5                               #  $t1 holds the addr of v[idx]
6
7           # Load the two values to be swapped
8           lw  $t0, 0($t1)     # reg $t0 = v[idx]
9           lw  $t2, 4($t1)     # reg $t2 = v[idx+1]
10
11          # Store the swapped values back to memory
12          sw  $t2, 0($t1)     # v[idx] = $t2
13          sw  $t0, 4($t1)     # v[idx+1] = $t0
```

Swap(): complete version

```
1   swap:
2         # Compute the address into the array
3         sll $t1, $a1, 2     # reg $t1 = idx * 4
4         add $t1, $a0, $t1   # reg $t1 = v + (idx*4)
5                             #  $t1 holds the addr of v[idx]
6
7         # Load the two values to be swapped
8         lw  $t0, 0($t1)     # reg $t0 = v[idx]
9         lw  $t2, 4($t1)     # reg $t2 = v[idx+1]
10
11        # Store the swapped values back to memory
12        sw  $t2, 0($t1)     # v[idx] = $t2
13        sw  $t0, 4($t1)     # v[idx+1] = $t0
14
15        jr  $ra             # return to main
```

Should an ISA be simple or complex?

- ISAs range in complexity
- MIPS is a relatively simple ISA.
  - But is that the right design choice?
- Consider the earlier example:

  High-level language (HLL): `a=b+10`

  Assembly language:

  - MIPS:
```
lw   r4,0(r2)   # r4=memory[r2+0]
add  r5,r4,10   # r5=r4+10
sw   r5,0(r3)   # memory[r3+0]=r5
```
  - x86:
```
ADDW3 (R5),(R2),10
```

Which one seems better?

CISC vs RISC ISAs

- Complex Instruction Set (CISC)
  - Appeared in early computers, including x86
  - Computers programmed in assembly → high-level language features as instructions
  - Very few registers → operands can be in memory
  - Very little memory → variable length instructions to minimize code size

- Reduced Instruction Set (RISC)
  - Appeared in the 80s. Used today in ARM, MIPS, and SPARC ISAs.
  - Compilers → Simple instructions
  - More registers → load-store architecture
  - More memory & faster clock frequency → fixed length, fixed format instructions for easy, fast decoding logic

Lectures 5-7 Inf2C - Computer Systems: Intro to C

Boris Grot

School of Informatics

University of Edinburgh

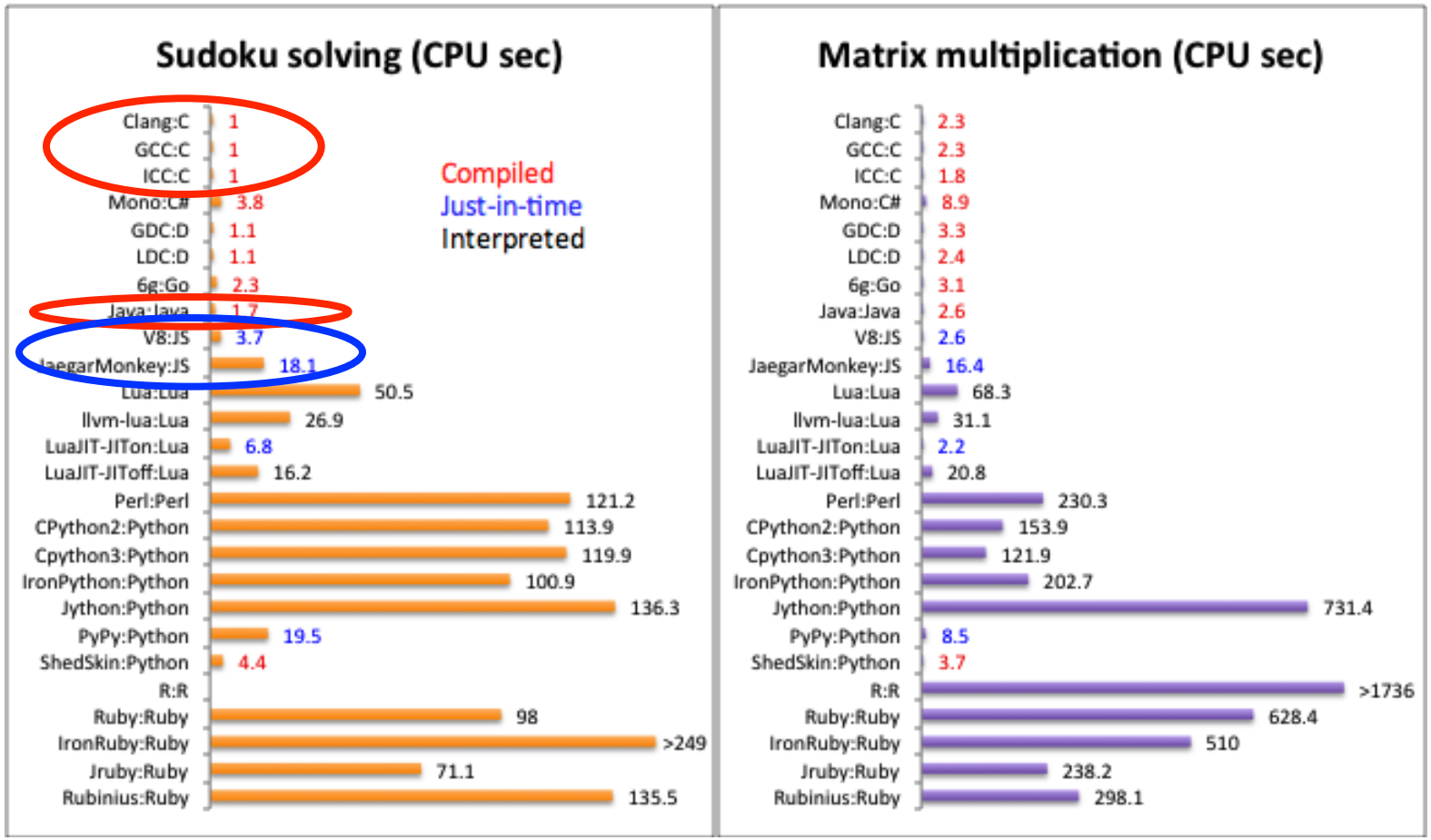
Previous lectures

- MIPS
  - Arithmetic and memory
  - Control flow: branches and jumps
  - Function calls and the stack

Lectures 5-7: Intro to C

- Motivation:
  - C is both a high and a low-level language
  - Very useful for systems programming
  - Fast! (next slide)
- This intro assumes knowledge of Java
  - Focus is on differences
  - Most of the syntax is the same
  - Most statements, expressions are the same

Performance: C vs. the rest



Source: http://attractivechaos.github.io/plb/

Outline

- A simple program; how to compile and run
- Major differences with Java
- Data types and composite data structures
- Arrays and strings
- Pointers
- Other issues
  - Memory regions
  - C Preprocessor
  - Portability

The hello world program

```
#include<stdio.h>

int main(void)
{   // This is a comment
    printf("Hello world!\n");
    return 0;
}
```

Linux/DICE shell commands

Compile: `gcc hello.c`

Run: `./a.out`

Major differences with Java

- C is not object oriented
  - C programs are collections of functions, like Java methods, but not class-based.
  - No inheritance, subtyping, dynamic dispatch in C
- C is not interpreted
  - A C program is compiled into an executable machine code program, which runs directly on the processor
  - Java programs are compiled into a byte code, which is read and executed by the Java interpreter (which is just another program)

C is less "safe"

- Run-time errors are not 'caught' in C
  - The Java interpreter catches these errors before they are executed by the processor
    - Example: array out-of-bounds exception
  - C run-time errors happen for real and the program crashes
- The C compiler trusts the programmer!
  - Many mistakes go un-noticed, causing run-time errors and leaving systems vulnerable to security exploits

Memory management is different

- In Java
  - All objects dynamically allocated
  - Unusable objects recycled automatically by garbage collection
- In C
  - No objects, only data structures
  - Some data structures statically allocated, others dynamically
  - Dynamically-allocated storage must be reclaimed (or freed) once the data structures there are no longer needed.
    - Major source of error, particularly when the programmer forgets to free the memory, resulting in memory leaks.

C has pointers …

- Pointers are special variables that reference (or point to) another variable
  - Similar to Java references
- We have already seen pointers in assembly:

```
lw $t1,0($s2)
```

  - `$s2` is a pointer
  - C pointers are the same thing! (more later)

Built-in data types

- The usual basic data types are there:

char	8 bits
short	16
int	16, 32, 64 (same as machine word size)
long	32, 64
float	32
double	64

- Data type sizes are machine dependent
  - Unlike Java where an int is always 32 bits
- Normally signed, unsigned available too
- No boolean type exists
  - for any number (int, char,…): 0 false, other true

Composite data structures - `struct`

- Structures are like objects, but their types have no methods, unlike classes:

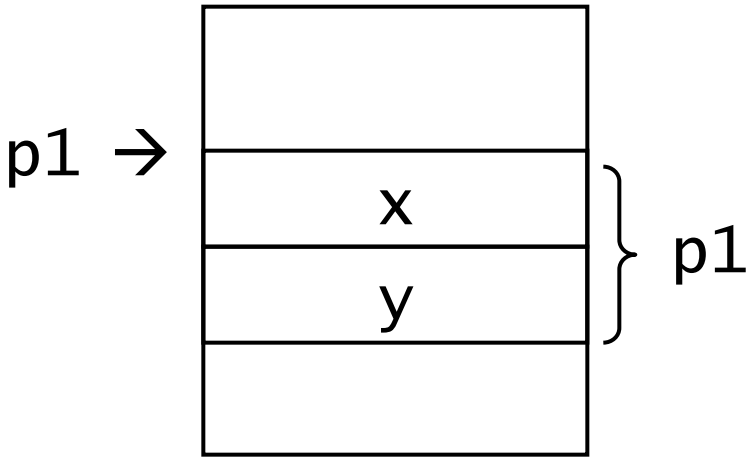
```
struct point {
    int x, y;
    // can include other structs
} p1;
struct point p2;
```

- Components accessed using "." operator

```
p1.x = 2;
```

In memory: structures

```
struct point {
   int x;
   int y;
} p1;
```



`sizeof(point) = 8`

What does `p1.y` translate into in MIPS?

```
addi $t0, $s0, 4  // $s0 points to the starting addr of p1
lw   $t4, $t0     // load p1.y into $t4
```

User-defined types

- Define names for new or built-in types

```
typedef <type> <name>;
```

- Example:

```
typedef unsigned char byte;

typedef struct {
   inx x;
   int y;
} point;
...
point p1, p2;
```

Arrays

- Syntax of C arrays similar to Java
- As in Java, C arrays have fixed size
- Example declarations of array:

```
int m[] = {5, 8, 10};  // size fixed to 3
int n[2][10]; // two-dimensional array
              //  with 2 rows and 10 cols
point p[4]; // array of 4 structs
```

- C arrays have no knowledge of their length
  - No checking that indexes are within bounds
- In C, close relationship between arrays and pointers
  - Pointers commonly used to pass arrays between functions

Strings

- C strings are simply arrays of type `char`
  - Encoded in 8 bits using ASCII
- They end with `'\0'`, the null character

```
char s[10]; // up to 9 characters long
```

- String initialisation

```
char s[10] = "string";      // '\0' implied
char s1[] = "string, too"; // length= ?
```

- Usual C rule for arrays apply:
  - Cannot store more chars than reserved at declaration
  - But bounds are not checked!

Strings – common operations

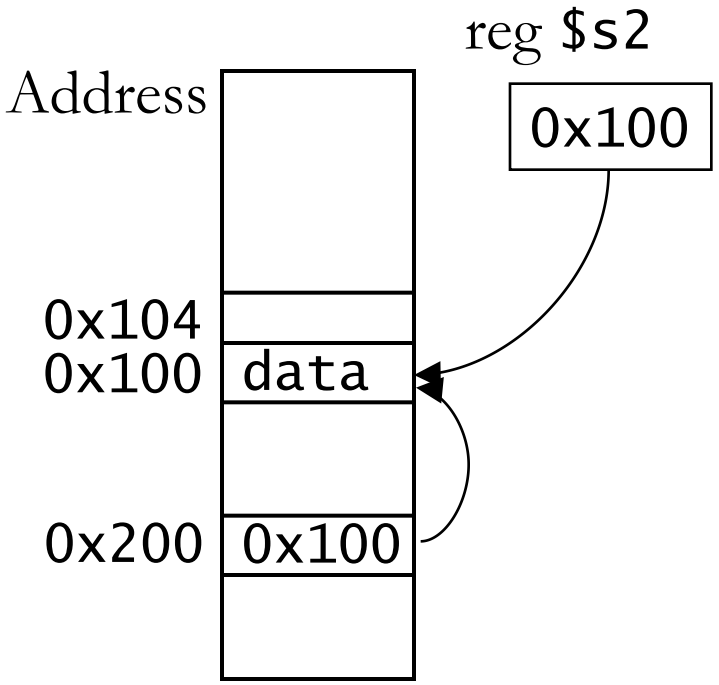
- Assignment: `strcpy(s, "string");`
- Length: `strlen(s)`
- To get the 6th character: `s[5]`
  - First char at position 0, as in Java arrays
- Comparison, `strcmp(s1, s2)` returns:
  - 0 when equal
  - Negative number when lexicographically s1 < s2
  - Positive when s1>s2
- Must `#include<string.h>` to call the functions
  - Type: `man string` to see what's available

Pointers

- We have seen pointers in assembly:

```
lw $t1,0($s2)
```

- `$s2` points to the location in memory where the "real" data is kept
- `$s2` is a register, but there's nothing stopping us to have pointers stored in memory like "normal" variables



C pointers

- A C pointer is a variable that holds the address of a piece of data
- Declaration:

```
int *p; // p is a pointer to an int
```

  – The compiler must know what data type the pointer points to

- Basic pointer usage:

```
p = &i; // p points to i now
*p = 5; // *p is another name for I
```

- & - *address of* operator. * *dereference* operator

Pointers as function arguments

- In Java
  - an argument with primitive type is passed by value (function gets copy of value)
  - an argument with class type is passed by reference (function gets reference to value)
- In C
  - All arguments passed by value
  - To get effect of `pass by reference', use an argument with a pointer type

Example – the swap function

```
void swap_wrong(int a, int b) {
  int t=a;
  a=b; b=t;
}
```

`swap_wrong` swaps the local variables `a,b` which are unknown outside of the function

```
void swap(int *a, int *b) {
  int t=*a;
  *a=*b; *b=t;
}
```

Function call: `swap(&x, &y);`

Pointer arithmetic and arrays

C allows arithmetic on pointers:

```
int a[10];
int *p;
p = a;  // p points to a[0]. Same as p = &a[0]
```

`p+1` points to `a[1]`

- Note that `&a[1] = &a[0]+1`
- The compiler multiplies +1 with the data type size

In general: `p+i` points to `a[i]`, `*(p+i)` is `a[i]`

Also valid: `*(a+i)` and `p[i]`

- but cannot change what `a` points to. It's not a variable

More pointer arithmetic

Common expressions:

`*p++` use value pointed by `p`, make `p` point to next element

`*++p` as above, but increment `p` first

`(*p)++` increment value pointed by `p`, `p` is unchanged

- Special value NULL used to show that a pointer is not pointing to anything
  - NULL is typically 0, so statements like `if (!p)` are common
- Dereferencing a NULL pointer is a very common cause of C program crashes

Example – pointer arithmetic

Return the length of a string:

```
int strlen(char *s)
{
  char *p=s;
  while (*s++ !='\0');
  return s-p-1;
}
```

- Argument/variable `s` is local, so we can change it
- Pointer increment, dereference and comparison all in one! No statement in the loop body
- Note pointer subtraction at return statement

More fun with strings & pointers

```
char s1[10] = "Bob";
char s2[10] = "Bob";

if (s1 =="Bob")
  // do x
else if (s1 == s2)
  // do y
else
  // do z
```

Which statement (x, y, or z) is executed?

Dynamic memory allocation

- Pointers are not much use with statically allocated data
- Library function `malloc` allocates a chunk of memory at run time and returns the address

```
int *p;
if ((p = malloc(n*sizeof(int))) == NULL) {
   // Error
}
...
free(p); // release the allocated memory
```

Pointers to pointers

- Consider an array of strings:
  ```
  char *strTable[10];
  ```
- The strings are dynamically allocated ⇒ any size
- But the table size is fixed to 10 strings
- What if we don't know the number of strings ahead of time?
  - Need to be able to provision array size on demand
  - That is, need to dynamically allocate the storage for the array of strings

```
char **strTable;
```

Pointers to pointers - details

Space must be allocated both for the table and the strings themselves

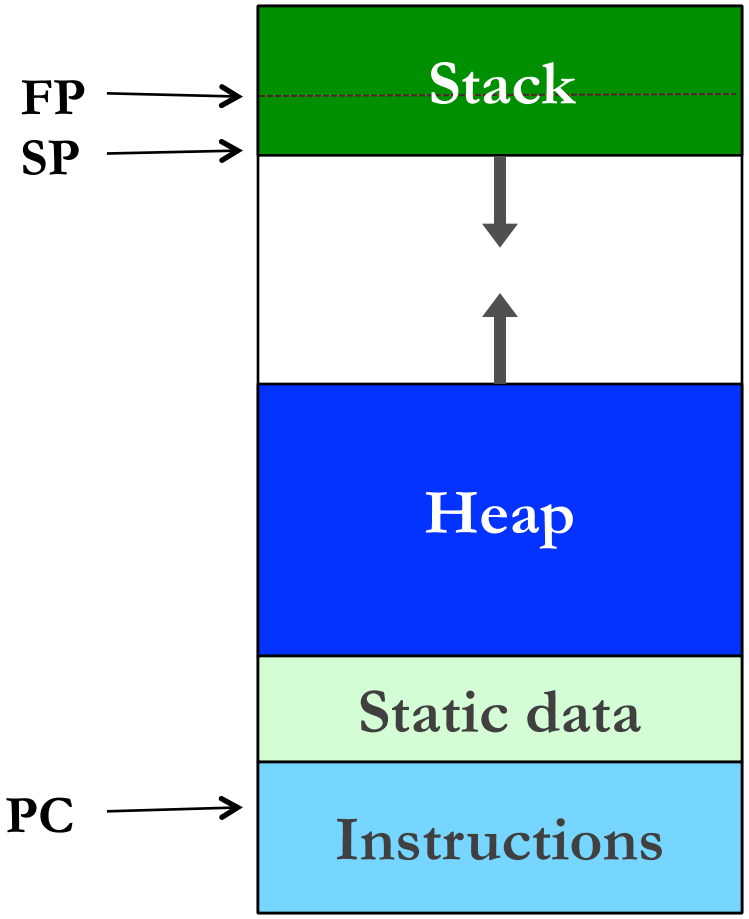
– Pointer to pointer!

```
1  char **strTable;
2  strTable = malloc(n*sizeof(char *));
3  for (i=0; i < n; i++) {
4     // s gets a string of length l
5     *(strTable+i) = malloc(l*sizeof(char));
6     strcpy(strTable[i], s);
7  }
8  // strTable[i][j] == *(*(strTable+i)+j)
```

Memory regions and management

- Memory areas
  - *Heap*: dynamically allocated storage
  - *Stack*: for function/method local variables
  - *Static*: for data living program lifetime
- In Java
  - All objects on heap
  - Unusable objects on heap recycled automatically by garbage collection
- In C
  - Data structures in all 3 areas
  - Programs must explicitly free-up heap storage that is no longer needed

Memory regions in detail



Managed by the compiler

Managed by the programmer (in C) or the system (in Java)

Initialized when the process starts

Categories of variables in C

- Global variables (statically allocated)
  - Defined outside of functions
  - Have *lifetime* of program and *scope* to file end
  - `extern` declarations extend scope before definition and to other files
  - Declare `static` to hide from other files
- Local (*automatic*) variables (allocated on stack)
  - Defined inside a function
  - Not available outside function
  - Distinct storage for each function invocation
  - Declare `static` for same storage for all invocations

Compilation units

- Programs are divided into *compilation units*
  - Provide degree of modularity
  - Each commonly has main file (.c) for source code
  - *Header* files (.h) declare public interfaces of units
- Each compiled separately to relocatable object code
  - Allows creation of object-code libraries
- A *linker* assembles these into an *executable*, resolving references between units
- A *loader* sets up the executable program in memory and initialises data areas, prior to program being run
  - Loader also computes addresses for Jump instructions

Declaration vs Definition

- Declaration: inform the compiler of the existence of a variable or function

```
void swap(int *a, int *b);    // in .h file
```

- Definition: provide function body; allocate memory for globals

```
void swap(int *a, int *b) {   // in .c file
    int temp = *a;
    *b = a;
    *a = temp;
}
```

Compilation units example

A.h:

```
int array_len;        // global
extern int MAX_SIZE; // global, defined elsewhere

// function declarations
void swap(int *a, int *b);
```

A.c:

```
#include "A.h"

// function definition
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

main.c:

```
#include <stdio.h>
#include "A.h"

int main(void) {
  int a = 5;
  int b = 15;
  swap(a, b);
}
```

Error?

The C pre-processor: cpp

- Includes – imports header files

```
#include <stdio.h>
#include "A.h"
```

- Text substitution, e.g. define constants

```
#define NAME value
```

- Macros (inline functions)

```
#define MAX(X,Y) (X>Y ? X : Y)
```

- Conditional compilation

```
#ifdef DEBUG
Printf("Debugging message");
#endif

> gcc –DDEBUG ...
```

That's all folks

- Not all C features have been covered, but this introduction should be enough to get you started
- Useful things to learn on your own:
  - Standard input/output: `printf, scanf, getc`, …
  - File handling: `fopen, fscanf, fprintf`, …
- Look over past exam papers for simple C programming exercises

Coursework 1

- Assigned "now", due in 2 weeks
  - **Deadline: Tue, 27 Oct, 16:00h**
- Task A: split a character string into words
  - Given: a C implementation
  - Your job: convert it to MIPS
- Task B: find single-word palindromes in a string
  - Given: C and MIPS implementations of Task A
  - Your job: write C and MIPS code for Task B

Coursework 1 (con'd)

- Task A example:

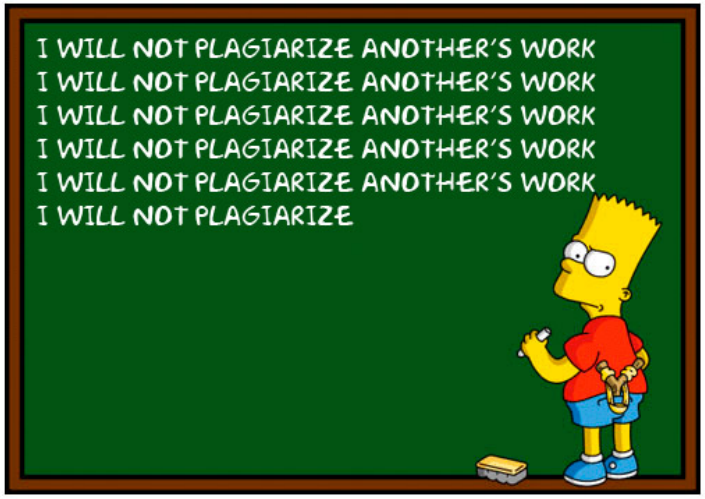
```
input: The first INF2C-CS coursework
output:
The
first
INF2C
CS
coursework
```

- Task B example:

```
input: I got my Honda Civic in 2002.
output:
Civic
2002
```

A (friendly) note on plagiarism

- **Don't do it!!!!**
- We use MOSS to electronically cross-check all submissions
  - MOSS is unaffected by variable renaming, code reshuffling, etc.
- Two plagiarism instances (4 students in total) were detected and prosecuted last year.
  - Remember: if you're sharing your code, you're just as guilty as the person taking it.



Inf2C - Computer Systems
Lecture 8
Logic Design

Boris Grot

School of Informatics

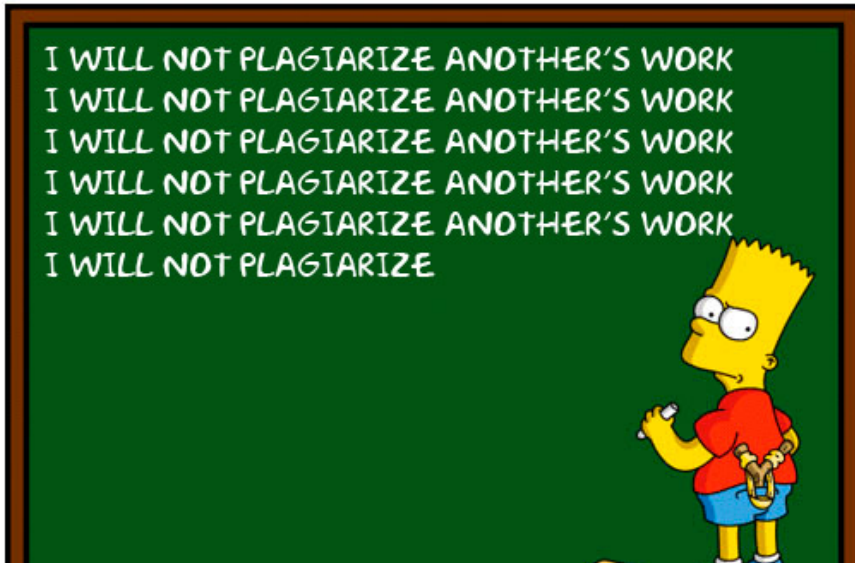
University of Edinburgh

Reminder

- **Coursework 1: due Tues @ 4pm**
- Do:
  - Have correct code
    - Compiles/builds/runs without warnings or errors
    - MIPS & C syntax & semantics are followed
  - Have well-structured code
    - Use functions; no goto's
  - Have readable code
    - Meaningful comments
    - Meaningful names for functions, labels, C variables, etc.

Reminder

- **Coursework 1: due Tues @ 4pm**
- Don't:
  - Be late!
  - Ask me for extensions
    - UG2 organizer – Dr. Sharon Goldwater – handles these
  - Plagiarize!

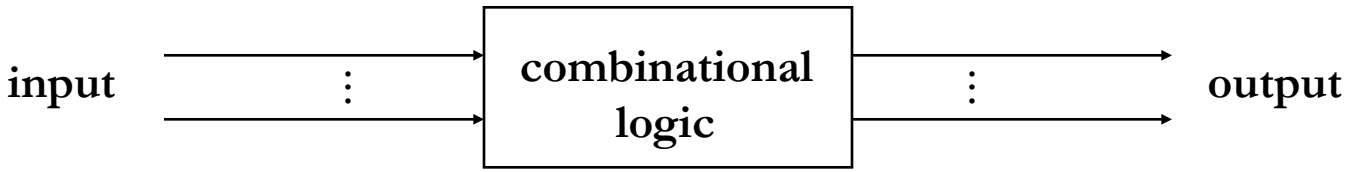


Logic design overview

Binary digital logic circuits:

- Two voltage levels (ground and supply voltage) for 0 and 1
  - Built from transistors used as on/off switches
  - Analog circuits not very suitable for generic computing
  - Digital logic with more than two states is not practical

Combinational logic: output depends only on the current inputs (no memory of past inputs)

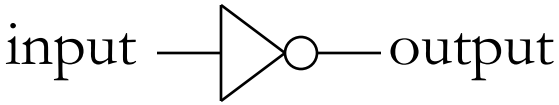


Sequential logic: output depends on the current inputs as well as (some) previous inputs → requires "memory"

Combinational logic circuits

- Inverter (or NOT gate): 1 input and 1 output

  "invert the input signal"



IN	OUT
0	1
1	0

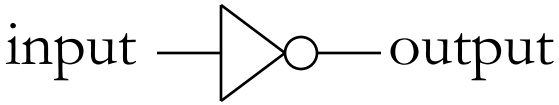
Truth table

$$OUT = \overline{IN}$$

Combinational logic circuits

- Inverter (or NOT gate): 1 input and 1 output

  "invert the input signal"

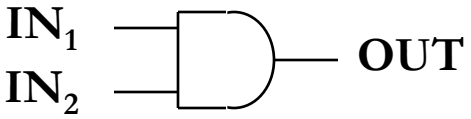


IN	OUT
0	1
1	0

$$OUT = \overline{IN}$$

- AND gate: 2 inputs and 1 output

  "output 1 only if both inputs are 1"

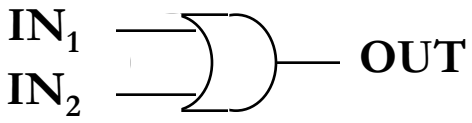


IN_1	IN_2	OUT
0	0	0
0	1	0
1	0	0
1	1	1

$$OUT = IN_1 \,.\, IN_2$$

Combinational logic circuits

- OR gate: "output 1 if at least one input is 1"

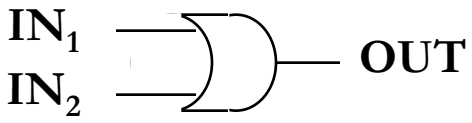


IN_1	IN_2	OUT
0	0	0
0	1	1
1	0	1
1	1	1

$OUT = IN_1 + IN_2$

Combinational logic circuits

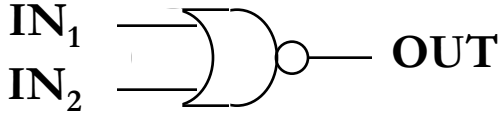
- OR gate: "output 1 if at least one input is 1"



IN_1	IN_2	OUT
0	0	0
0	1	1
1	0	1
1	1	1

$$OUT = IN_1 + IN_2$$

- NOR gate: "output 1 if no input is 1" (NOT OR)

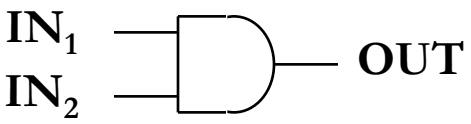


IN_1	IN_2	OUT
0	0	1
0	1	0
1	0	0
1	1	0

$$OUT = \overline{IN_1 + IN_2}$$

Combinational logic circuits

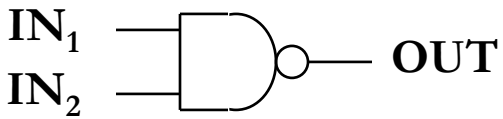
- AND gate: "output 1 if both inputs are 1"



IN_1	IN_2	OUT
0	0	0
0	1	0
1	0	0
1	1	1

$$OUT = IN_1 \,.\, IN_2$$

- NAND gate: "output 1 if both inputs are <u>not</u> 1" (NOT AND)

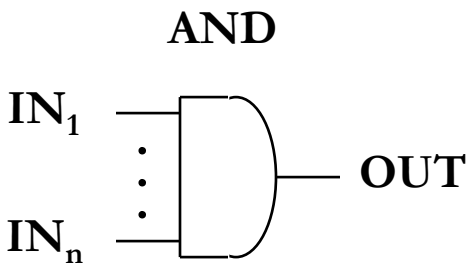


IN_1	IN_2	OUT
0	0	1
0	1	1
1	0	1
1	1	0

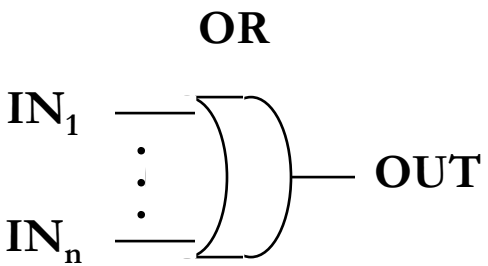
$$OUT = \overline{IN_1 \,.\, IN_2}$$

Combinational logic circuits

- Multiple-input gates:



OUT = 1 if <u>all</u> IN_i=1



OUT = 1 if <u>any</u> IN_i=1

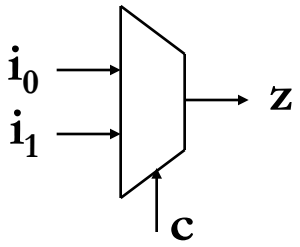
Combinational logic circuits

- Functional completeness:
  - Set of gates that is sufficient to express any boolean function

- Examples:
  - AND + OR + NOT
  - NAND
  - NOR

Multiplexer

- Multiplexer: a circuit for selecting one of multiple inputs



$$z = \begin{cases} i_0, \text{ if } c=0 \\ i_1, \text{ if } c=1 \end{cases}$$

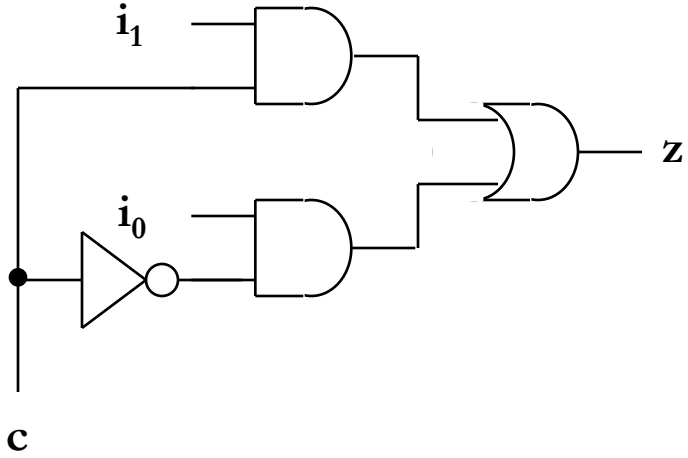
c	i_0	i_1	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$$z = \bar{c}.i_0.\bar{i_1} + \bar{c}.i_0.i_1 + c.\bar{i_0}.i_1 + c.i_0.i_1$$
$$= \bar{c}.i_0.(\bar{i_1} + i_1) + c.(\bar{i_0} + i_0).i_1$$
$$= \bar{c}.i_0 + c.i_1$$

"sum of products form"

A multiplexer implementation

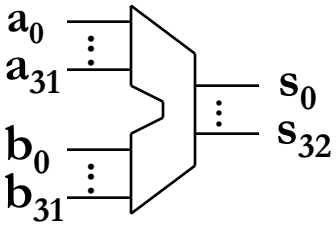
- Sum of products form: $\mathbf{i_1.c + i_0.\overline{c}}$
  - Can be implemented with 1 inverter, 2 AND gates & 1 OR gate:



- Sum of products is not practical for circuits with large number of inputs (n)
  - The number of possible products can be proportional to 2^n

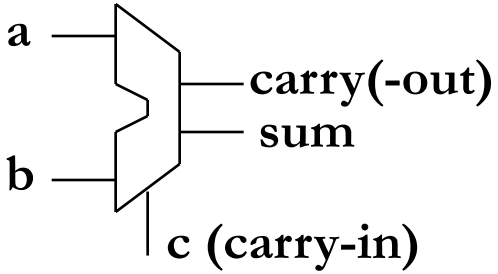
Arithmetic circuits

- 32-bit adder



64 inputs → too complex for sum of products

- Full adder:



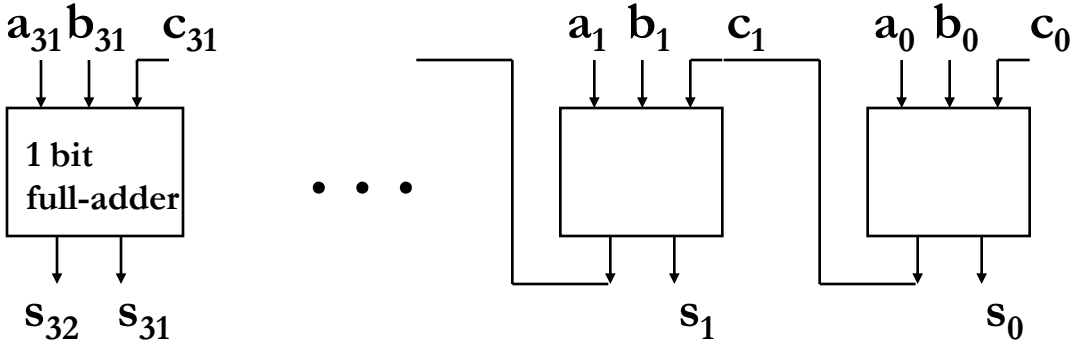
$$sum = \bar{a}.\bar{b}.c + \bar{a}.b.\bar{c} + a.\bar{b}.\bar{c} + a.b.c$$

$$carry = b.c + a.c + a.b$$

a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Ripple carry adder

- 32-bit adder: chain of 32 full adders



  - Carry bits c_i are computed in sequence c_1, c_2, … , c_{32} (where $c_{32} = s_{32}$), as c_i depends on c_{i-1}
  - Since sum bits s_i also depend on c_i, they too are computed in sequence

Propagation delays

- Propagation delay = time delay between input signal change and output signal change at the other end
- Delay depends on:
  1. technology (transistor parameters, wire capacitance, etc.)
  2. number of gates driven by a gate's output (**fan out**)
- e.g.: Half-adder circuit: 3 gate delays → fast! (inverter, AND, OR)
- e.g.: 32-bit ripple carry adder:
  - 65 gate delays → slow
  - 1 AND + 1 OR for each of 31 carries to propagate; followed by 1 inverter + 1 AND + 1 OR for S_{31}

Practice problem:

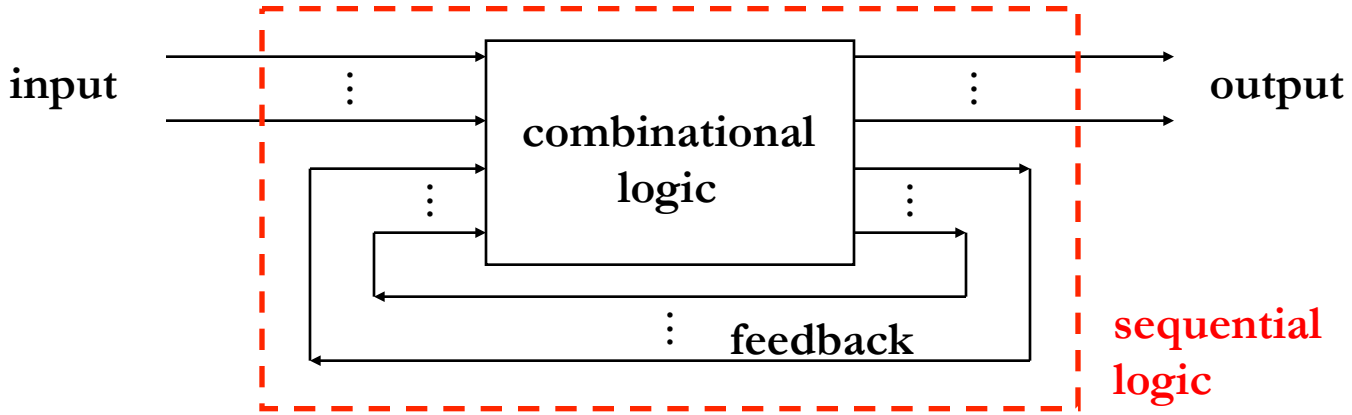
Design a circuit that, given a 4-bit hex character, outputs

1 - if the character is ≥9

0 - otherwise

What is the propagation delay of the circuit?

Sequential logic circuits

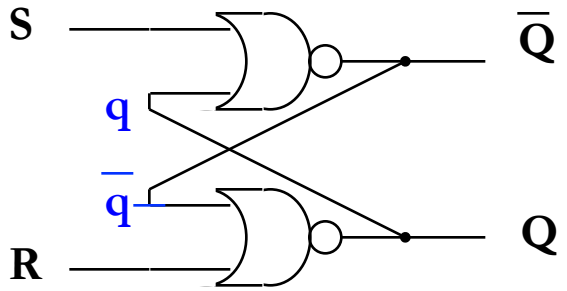


- Output depends on current AND past inputs
  - The circuit has memory
- Sequences of inputs generate sequences of outputs ⇒ sequential logic
  - With n feedback signals → up to 2^n stable states

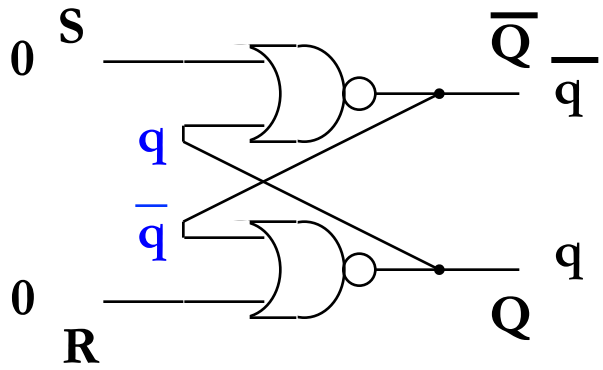
SR Latch: the basic state element

SR latch

- Inputs: R, S
- Feedback: q, $\overline{q}$
- Output: Q



SR Latch

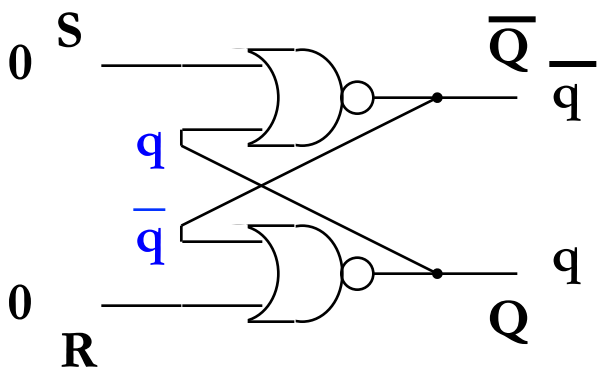


SR Latch

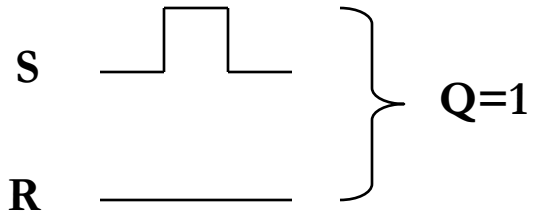
- Truth table:

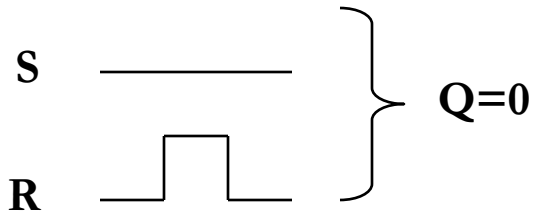
S	R	Q_i
0	0	Q_{i-1}
0	1	0
1	0	1
1	1	inv

inv=invalid



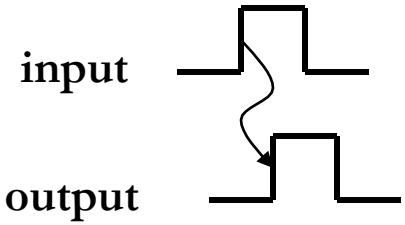
- Usage: 1-bit memory
  - Keep the value in memory by maintaining S=0 and R=0
  - Set the value in memory to 0 (or 1) by setting R=1 (or S=1) for a short time



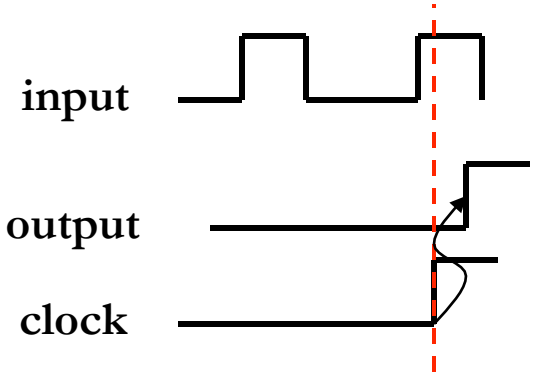


Timing of events

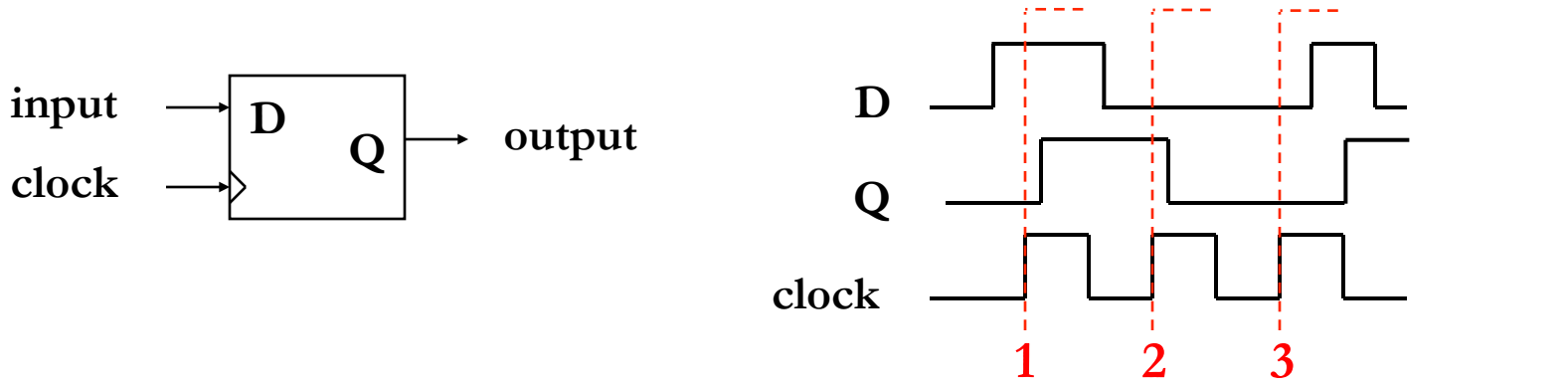
- Asynchronous sequential logic
  - State (and possibly output) of circuit changes whenever inputs change



- Synchronous sequential logic
  - State (and possibly output) can only change at times synchronized to an external signal → the **clock**

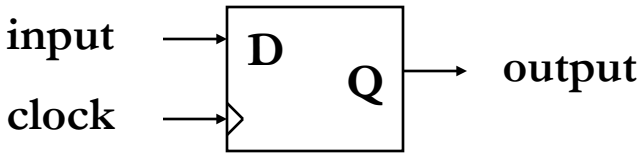


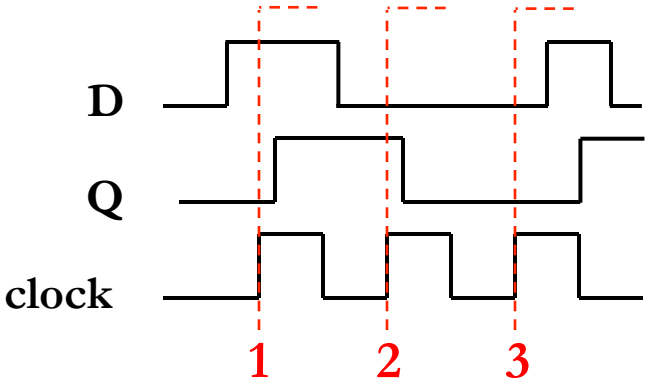
Using clock to build a D latch



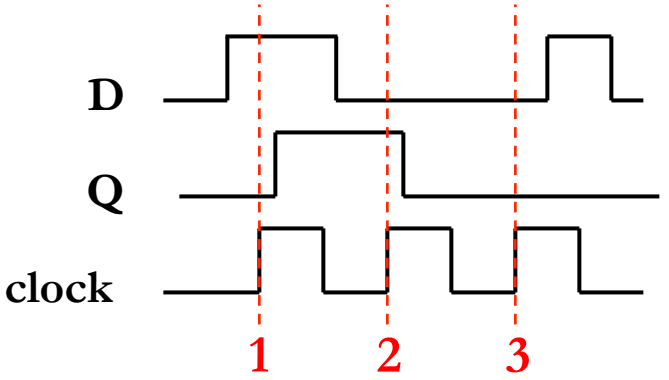
- Level-triggered latch: whenever clock is 1, D is propagated to Q

D flip-flop



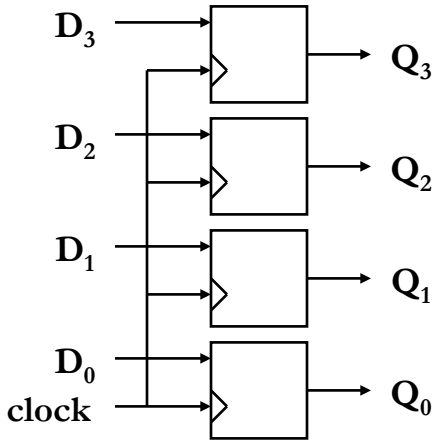


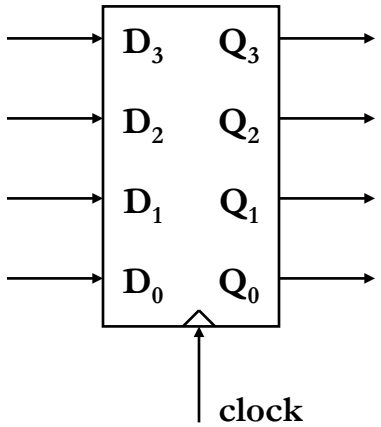
- Level-triggered latch: whenever clock is 1, D is propagated to Q
- Edge-triggered flip-flop: on a positive clock edge, D is propagated to Q



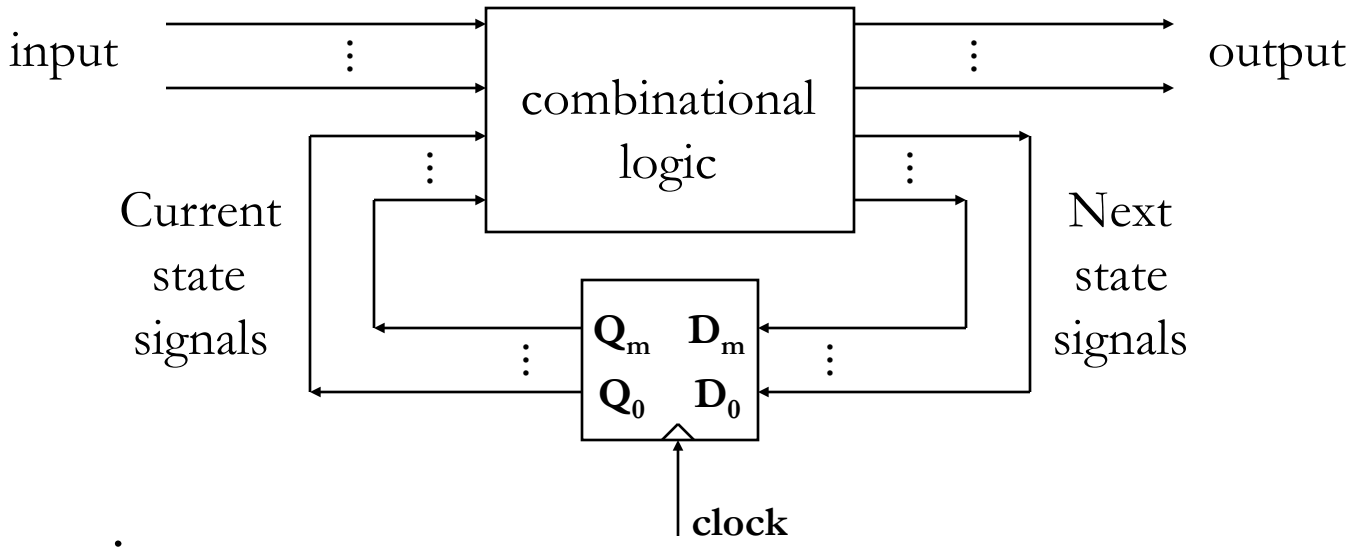
Registers out of flip-flops

- Tie multiple D flip-flops together using a common clock
- E.g., 4-bit register:





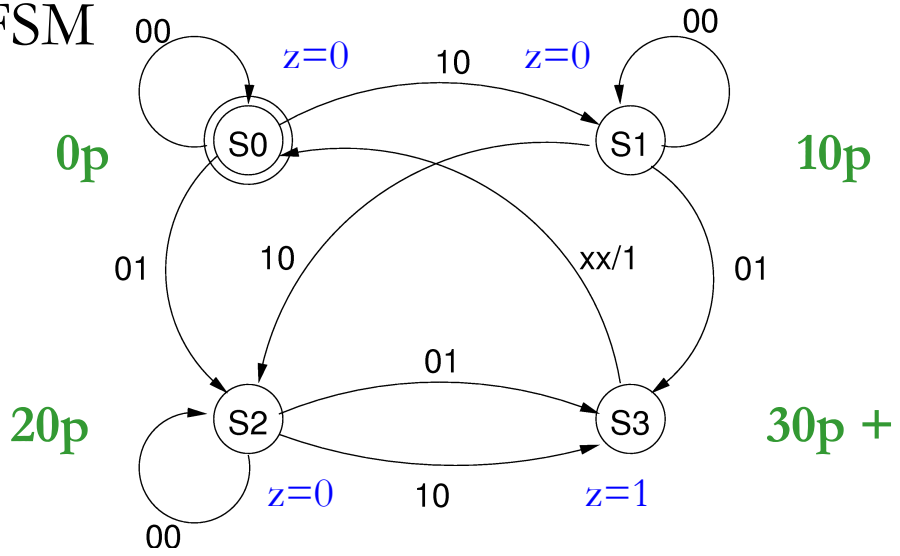
General sequential logic circuit



- Operation:
  - At every rising clock edge next state signals are propagated to current state signals
  - Current state signals plus inputs work through combinational logic and generate output and next state signals

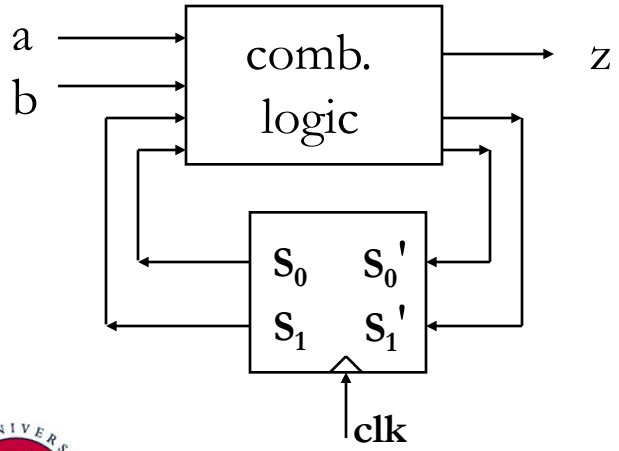
Hardware FSM

- A sequential circuit is a (deterministic) Finite State Machine – FSM
- Example: Vending machine
  - Accepts 10p, 20p coins, sells one product costing 30p, no change given
  - Coin reader has 2 signals: *a, b* for 10p, 20p coins respectively. These are the inputs to our FSM
  - Output *z* asserted when 30p or more has been paid in



FSM implementation

- Methodology:
  - Choose encoding for states, e.g S0=00, …, S3=11
  - Build truth table for the next state s_1', s_0' and output z
  - Generate logic equations for s_1', s_0', z
  - Design comb logic from logic equations and add state-holding register



s_1	s_0	a	b	s_1'	s_0'	z
0	0	0	0	0	0	
0	0	0	1	1	0	0
0	0	1	0	0	1	
0	1	0	0	0	1	
0	1	0	1	1	1	0
0	1	1	0	1	0	
•••••••				•••••••		

Inf2C - Computer Systems
Lecture 9
Processor Design – Single Cycle

Boris Grot

School of Informatics

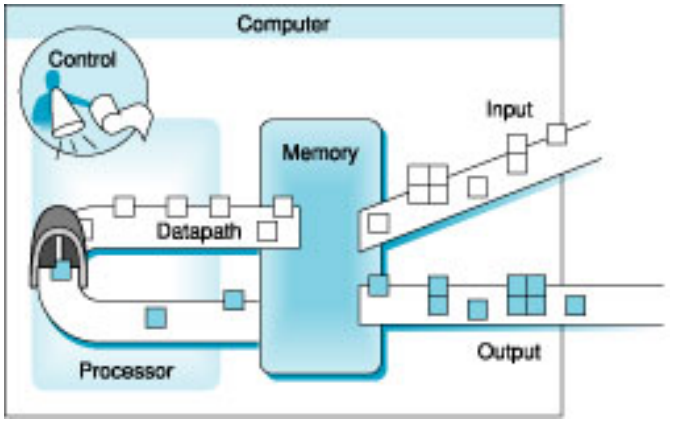
University of Edinburgh

Previous lectures

- Combinational circuits
  - Combinations of gates (INV, AND, OR, NOR..)
  - Output: function of the input only (memory-less)
- Sequential circuits
  - Output: function of the input and prev inputs
  - Basic memory element: SR-latch (cross-coupled NORs)
  - Clock: synchronizes the operation of the circuit
  - Operation: current states + inputs go through combinational logic. Next states stored in a register on a rising clock edge.
- Hardware Finite State Machines (FSMs)
  - Registers for states + comb'l logic for transitions & outputs

Lecture 9: Processor design – single cycle

- Motivation:
  - Learn how to design a simple processor
- Two main parts:
  - Datapath: performs the data operations as commanded by the program instructions
  - Control: controls the datapath, memory and I/O according to the program instructions
- Using:
  - Combinational and sequential circuits described in previous lectures



Design steps / Lecture outline

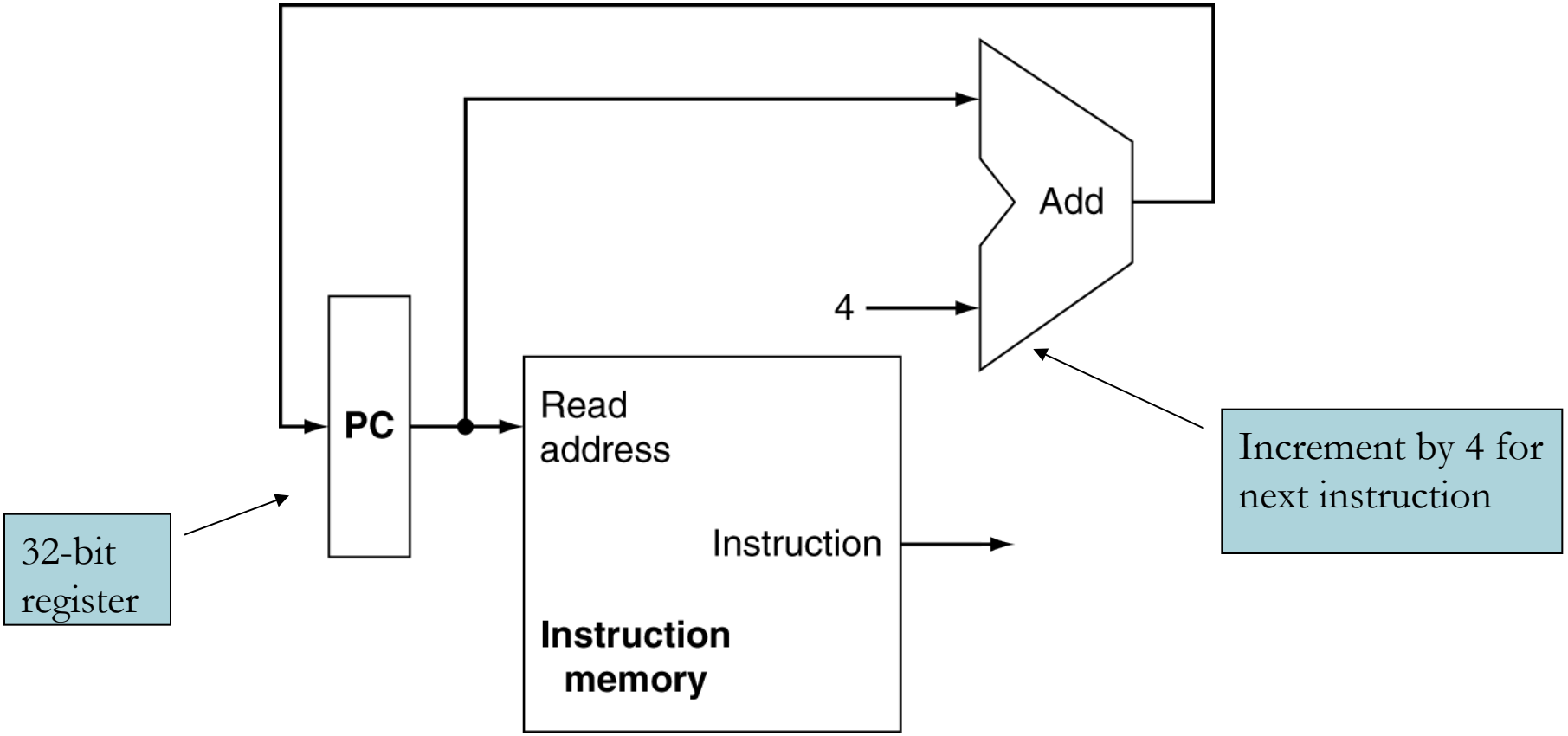
- Step 1: Determine the components required by understanding main processor functions
- Step 2: Build the datapath
- Step 3: Build the control

- Show the execution of a few instructions on the designed machine

Main processor functions

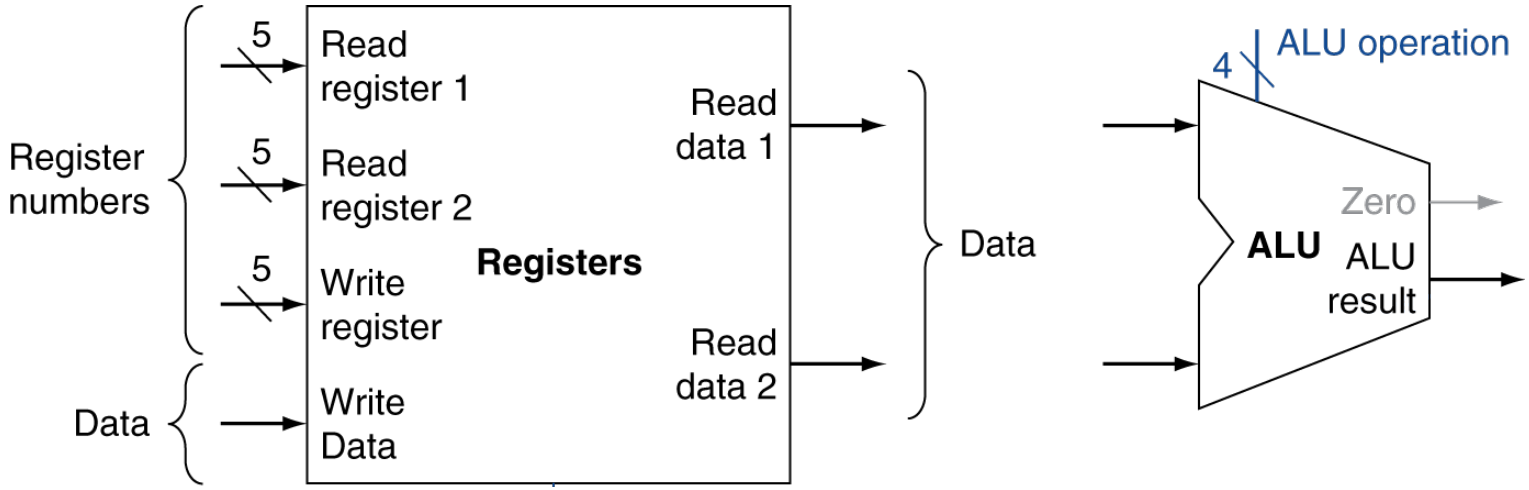
- **Fetch** instruction from instruction memory
- Read the register operands
- Use the ALU for computation
  - Arithmetic, memory address, branch target address
- Access data memory for load/store
- Store the result of computation or loaded data into the destination register
- Update the Program Counter (PC)

Instruction Fetch



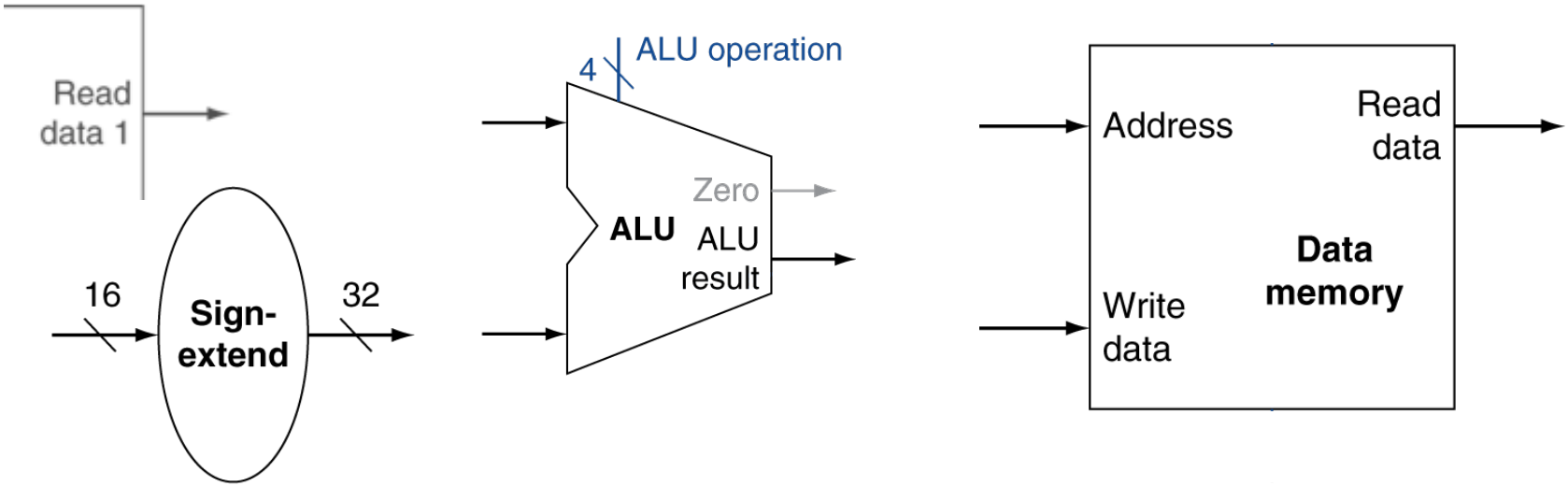
R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



Load/Store Instructions

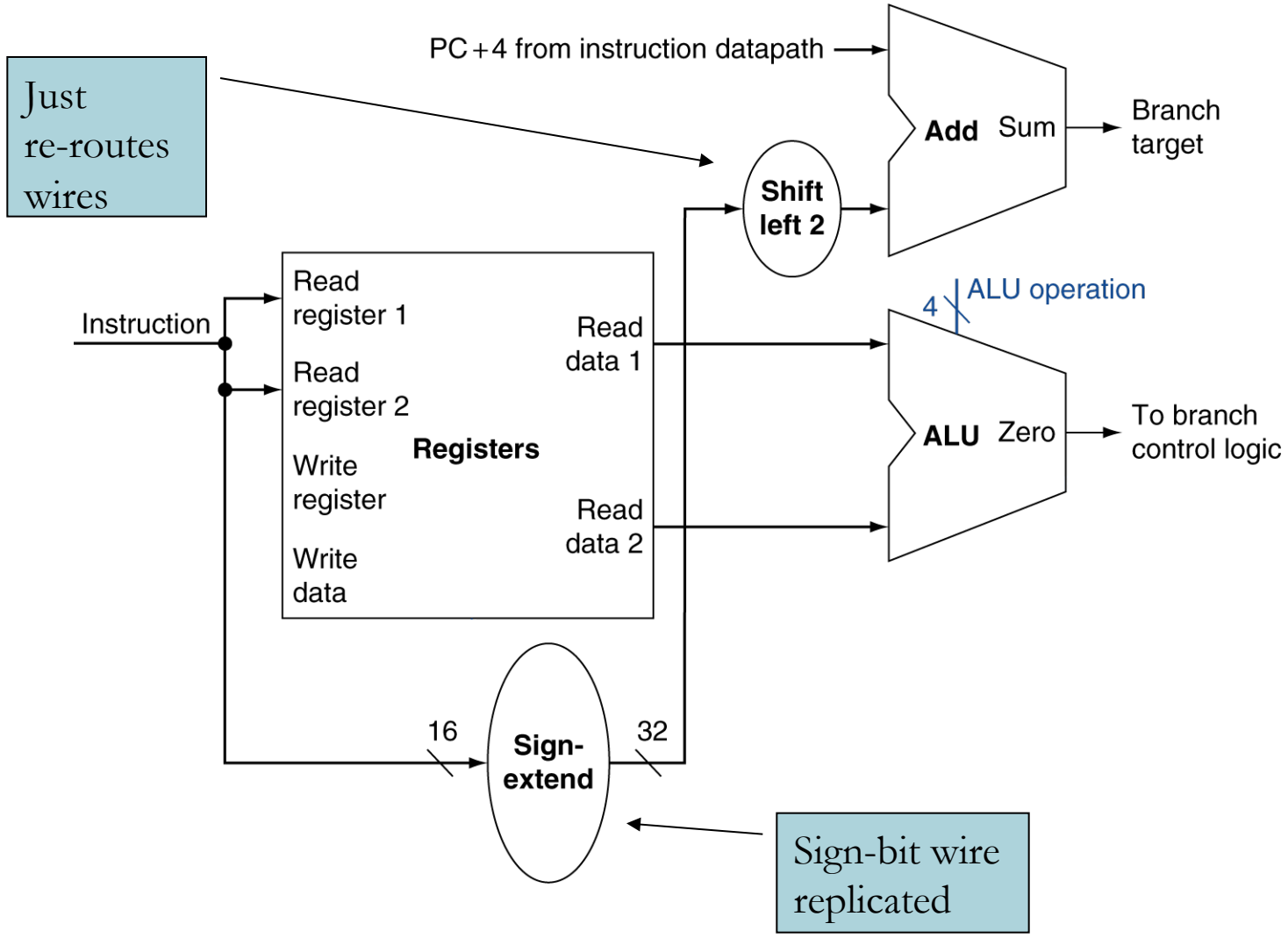
- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



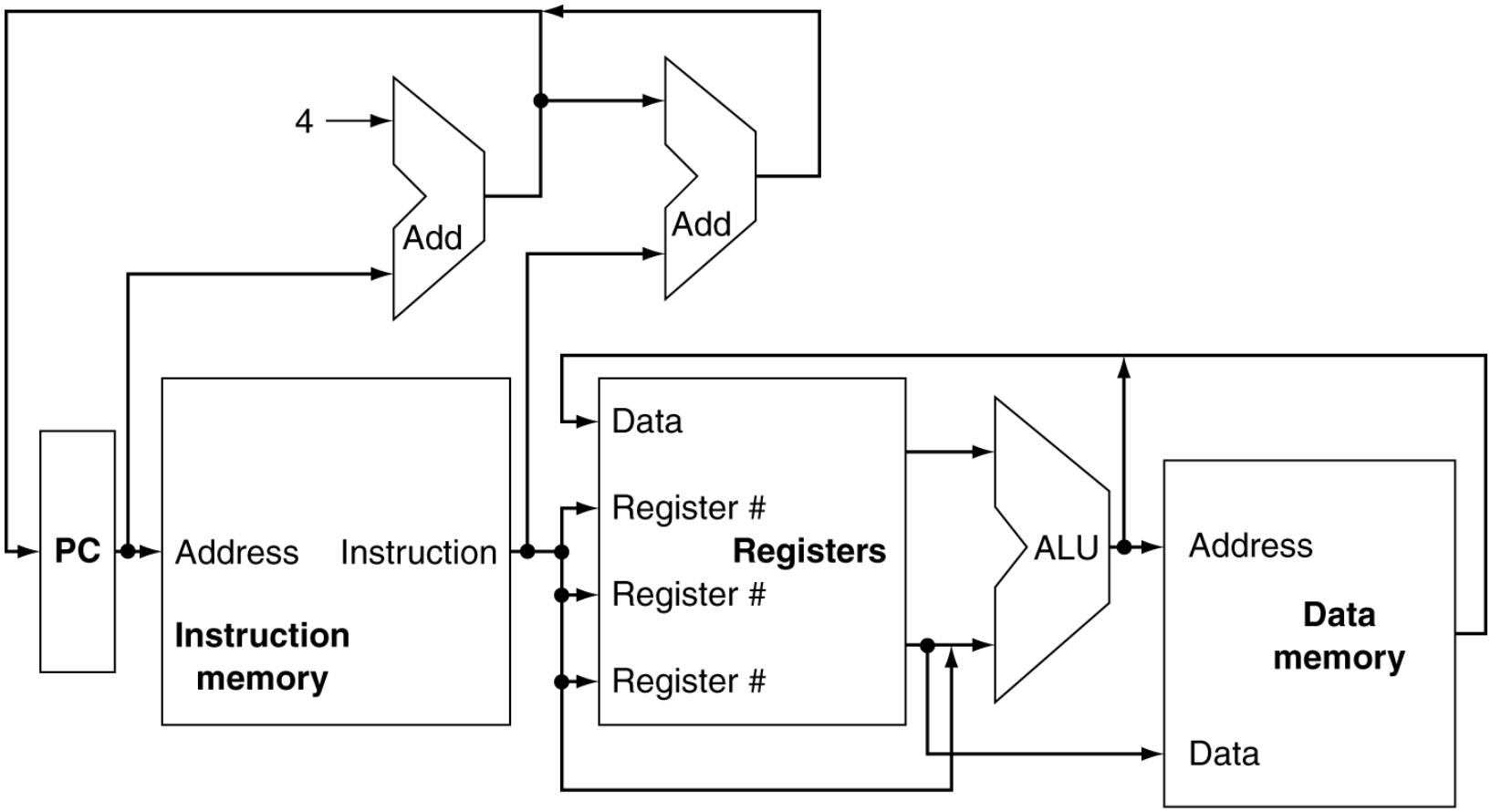
Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend the immediate (offset)
  - Shift left 2 places (word align)
  - Add to PC + 4
    - Already calculated by instruction fetch

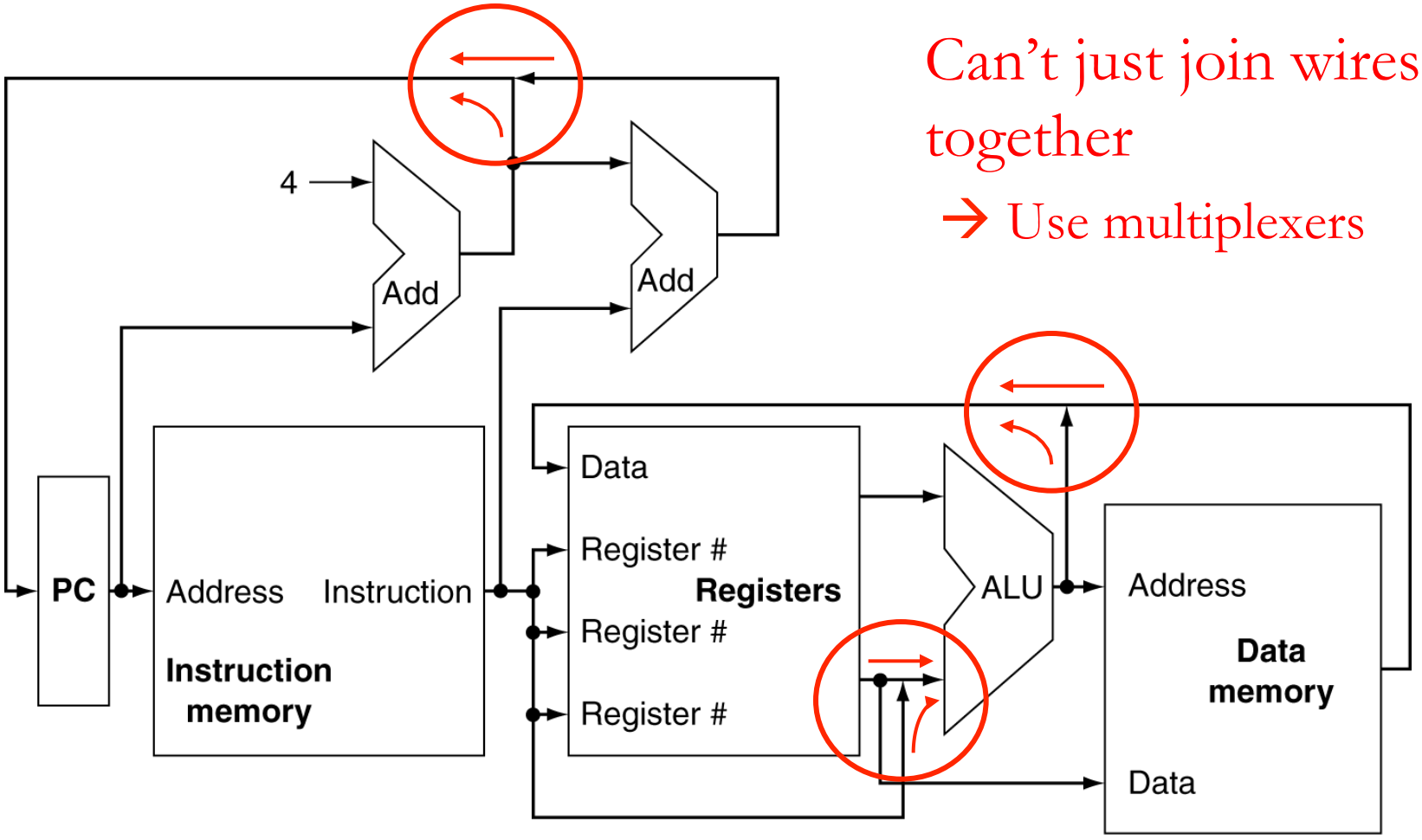
Branch Instructions



Simplified Datapath



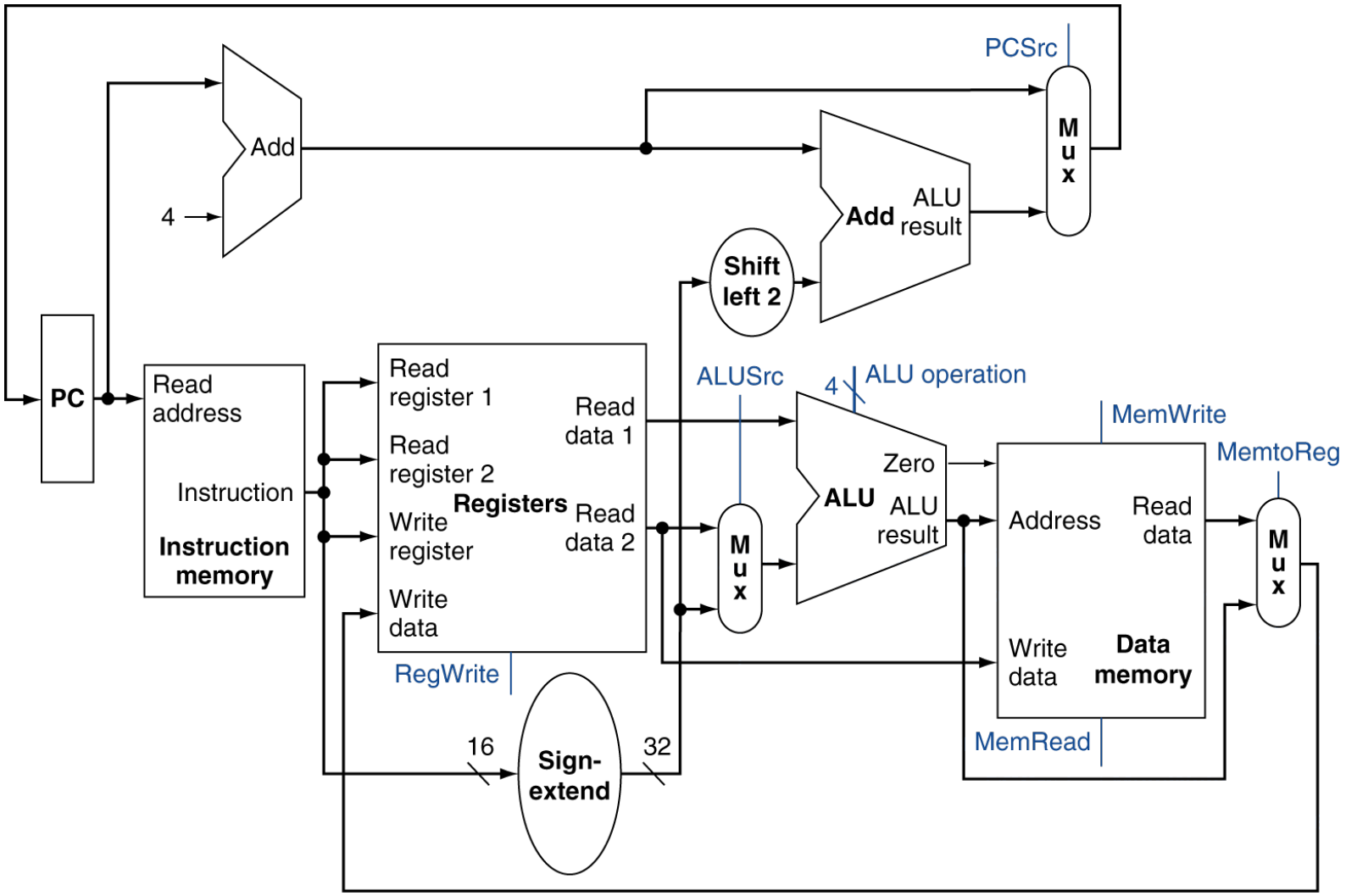
Simplified Datapath



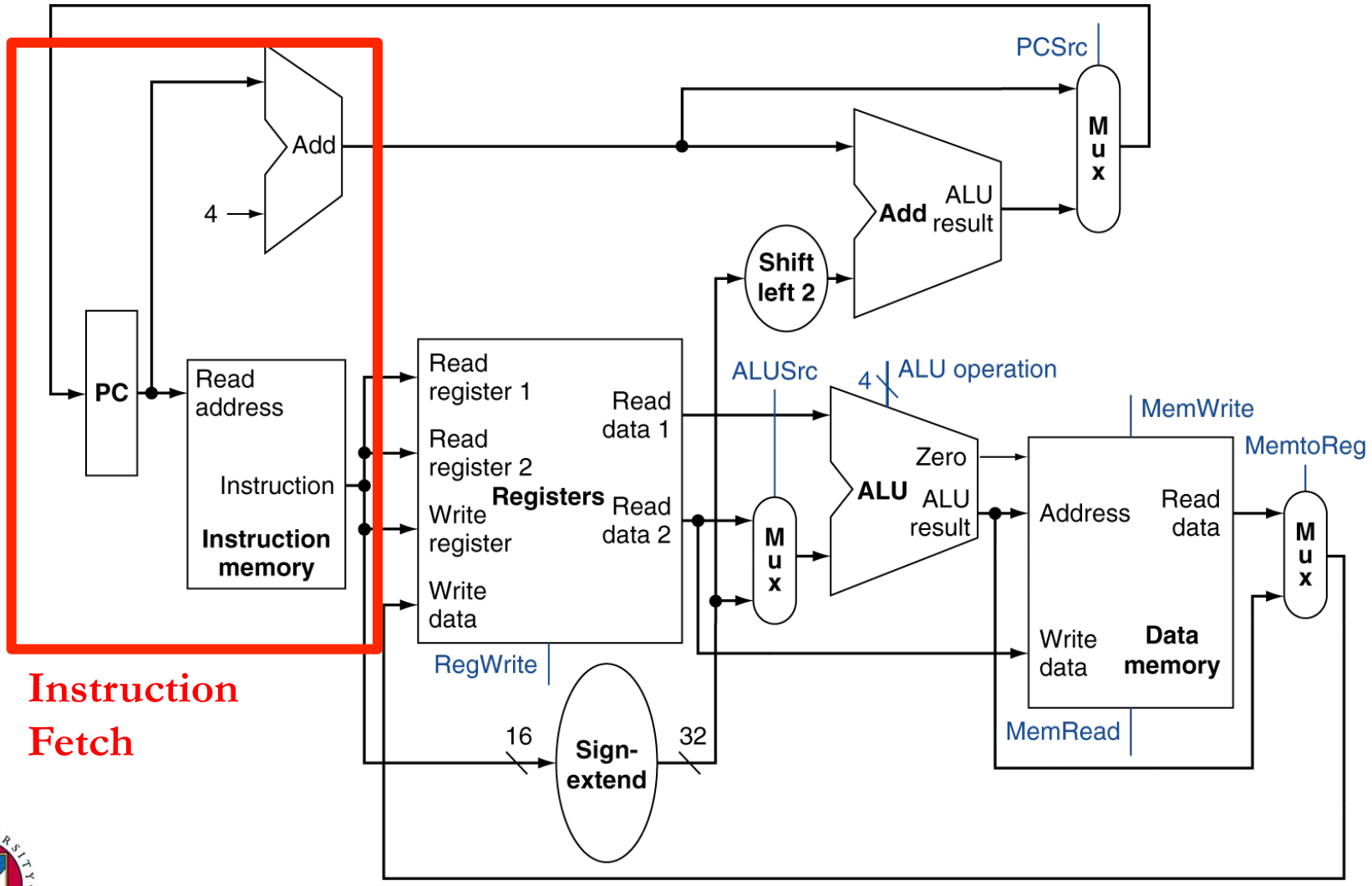
Can't just join wires together

→ Use multiplexers

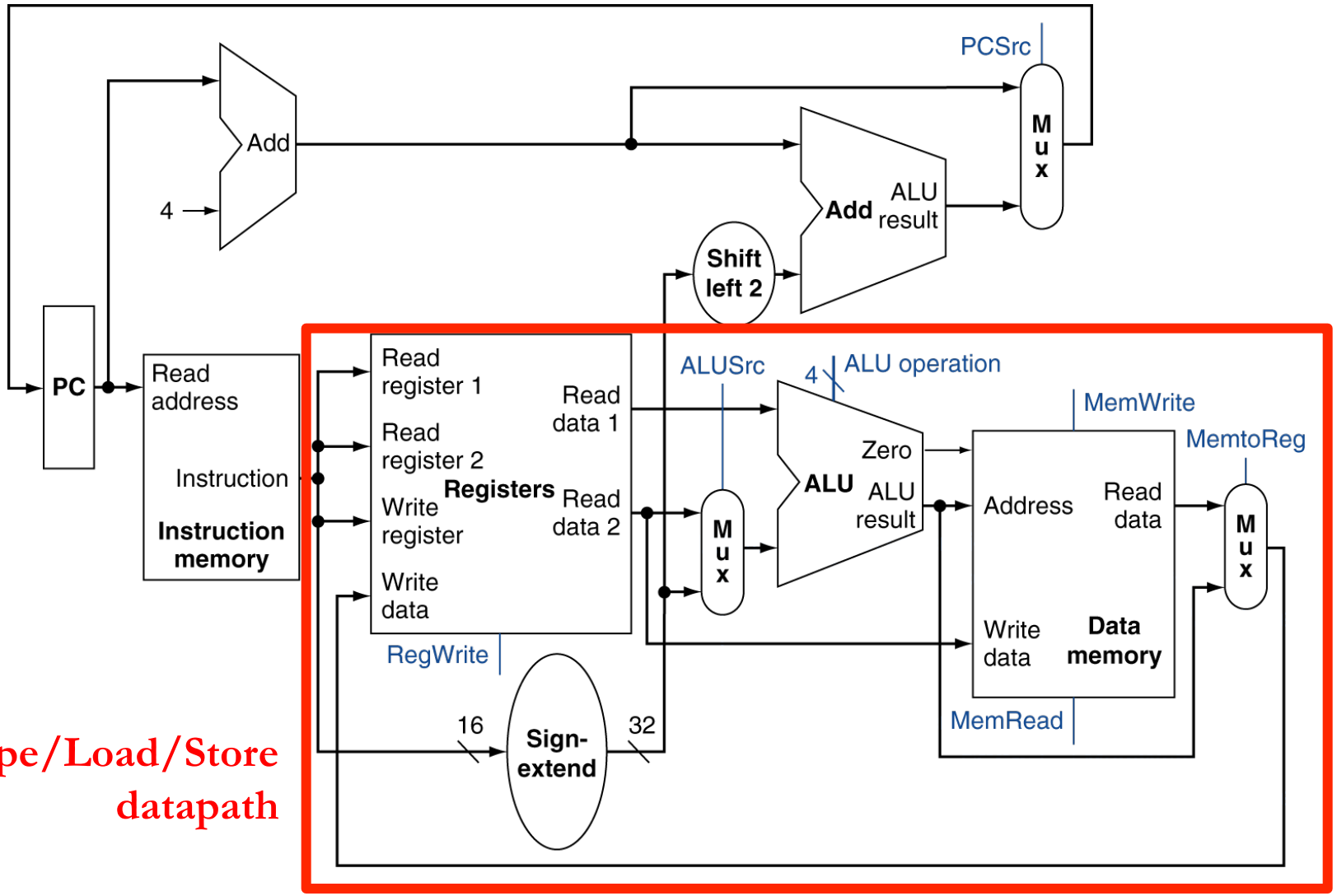
Full Datapath



Full Datapath

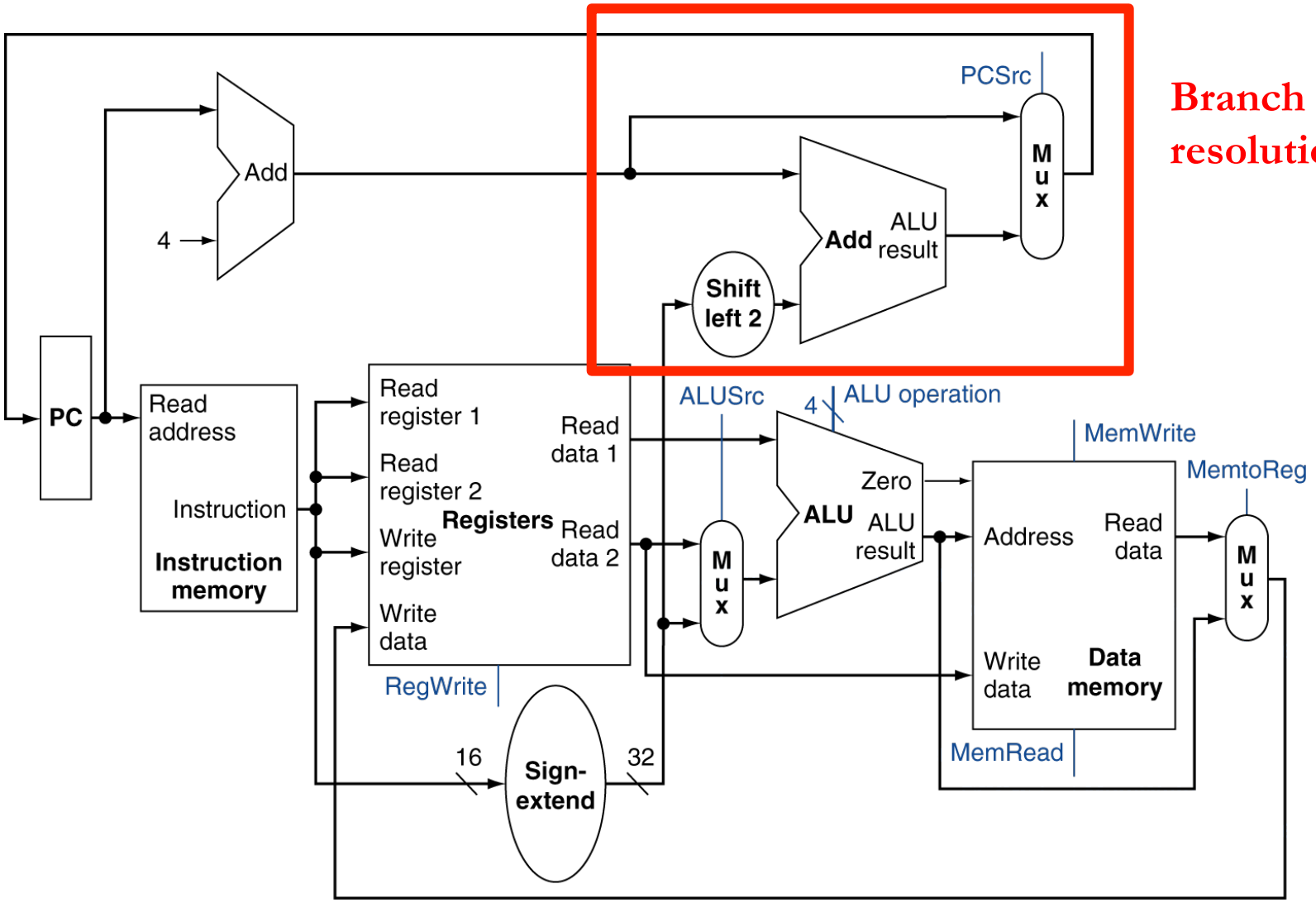


Full Datapath



R-type/Load/Store datapath

Full Datapath

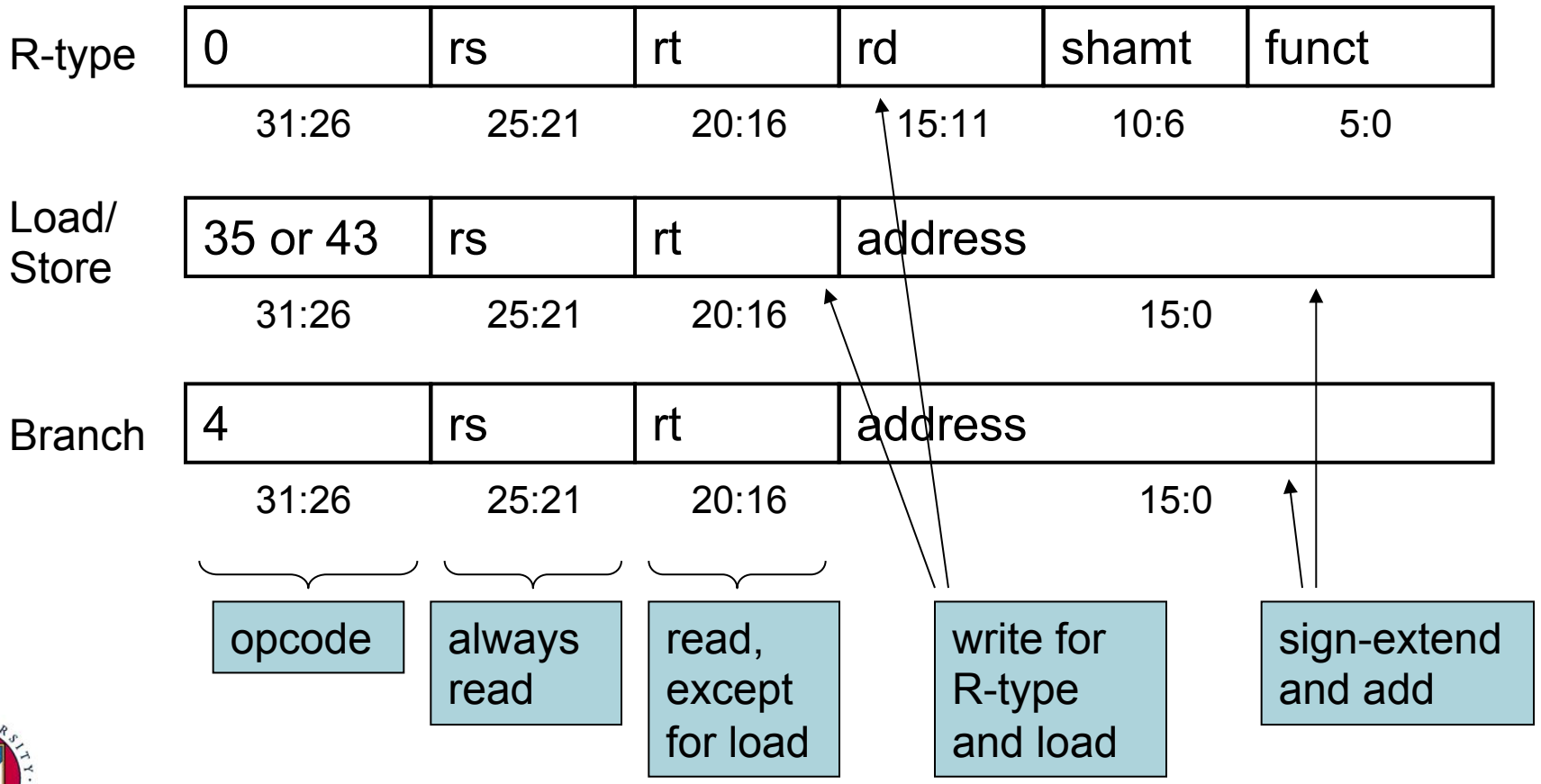


How to design the control part

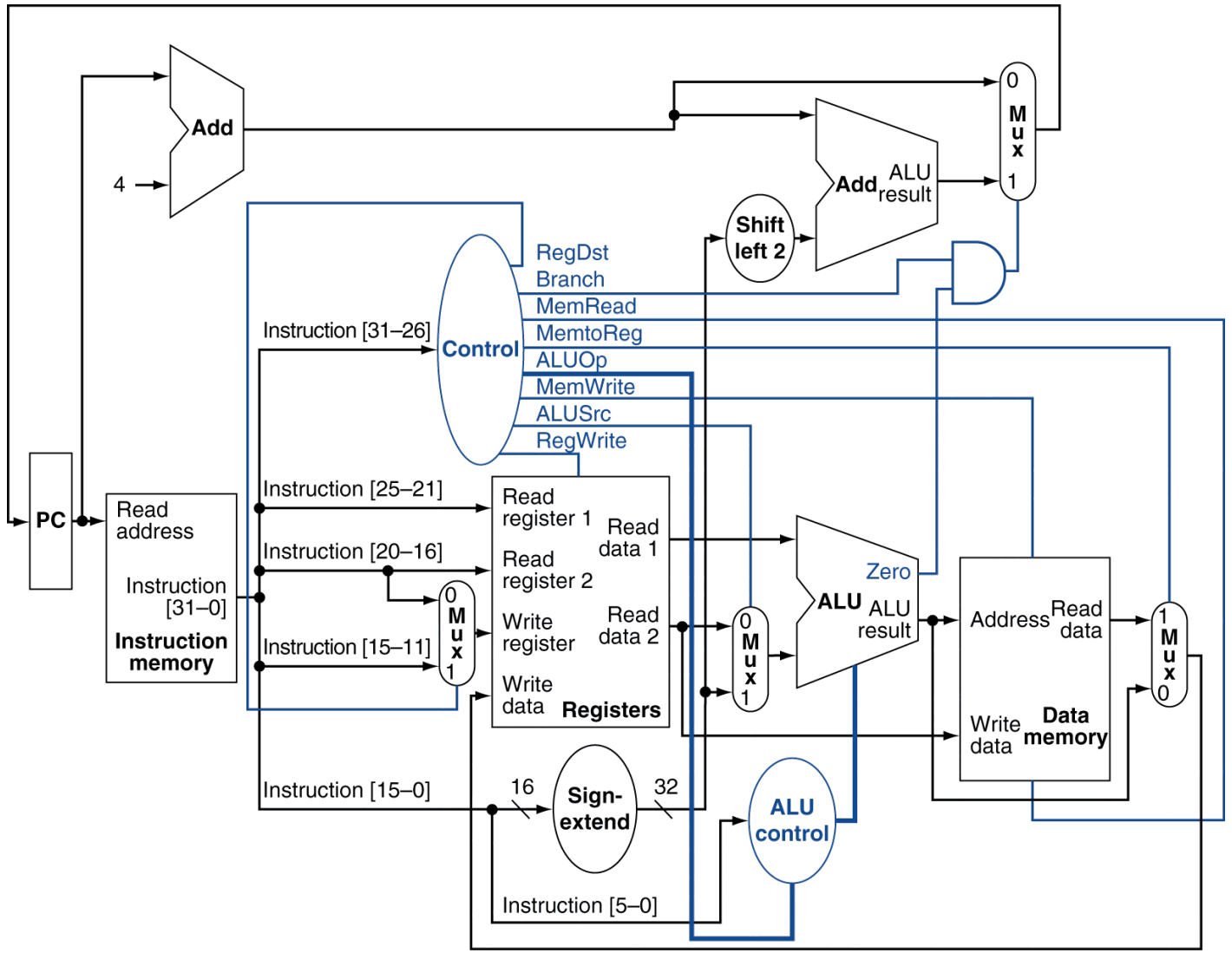
- For all control signals determine which value selects what operation, input, etc.
- Make truth table of control signal values for each instruction, or instruction group
- Convert table to combinational circuit

Designing the Main Control Unit

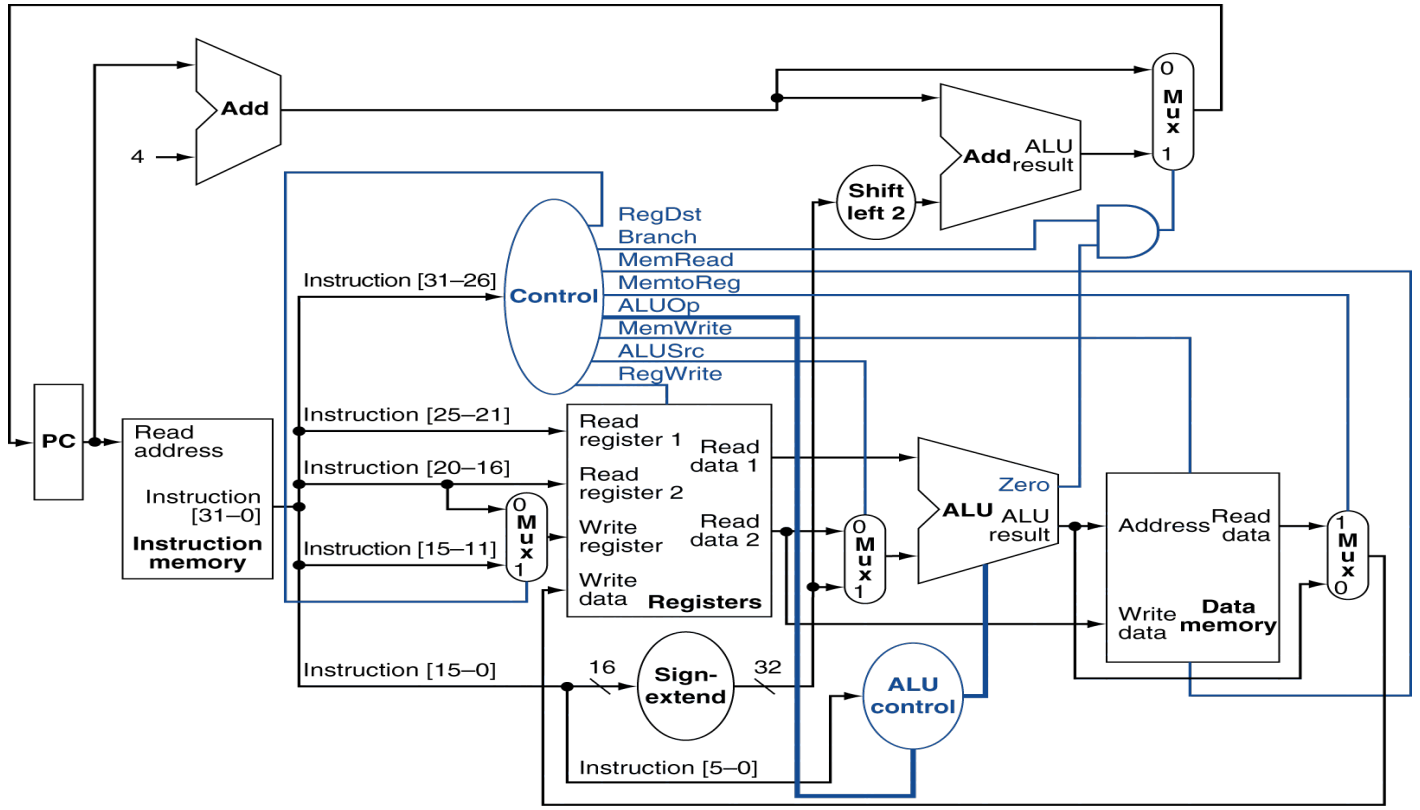
Control signals derived from instruction fields



Datapath with Control



Datapath and control truth table



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
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

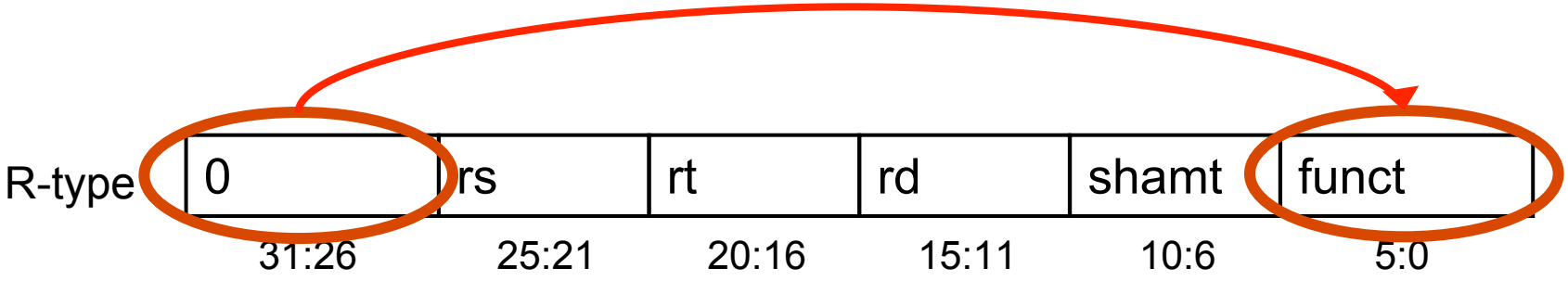
Don't Care

ALU control

ALU operation:

- Data transfers (ld/st) – add
- Branches – sub
- All other – determined by funct field, I[5:0]

derived from opcode



ALU control

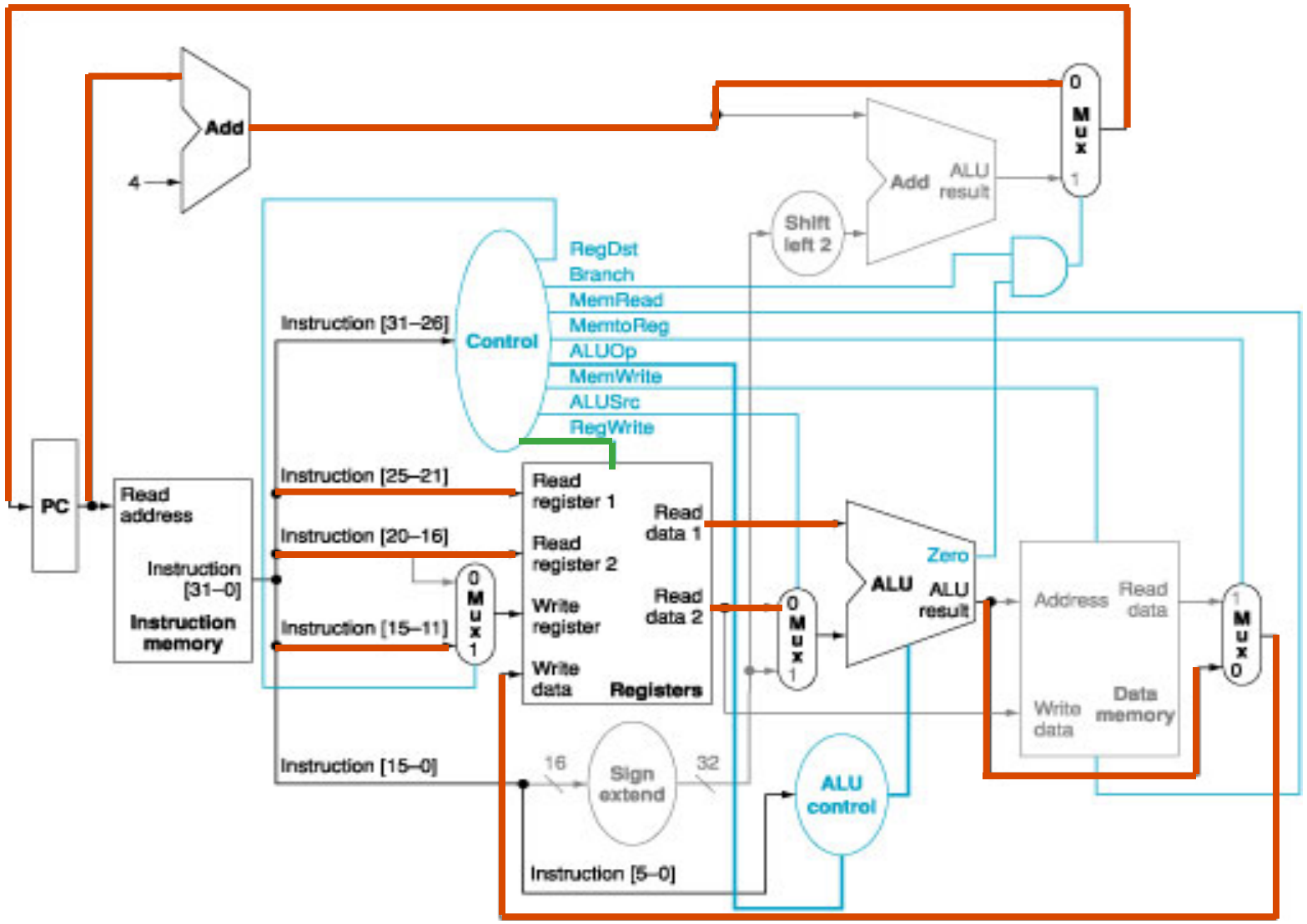
ALU control is hierarchical:

- Main control specifies which of the 3 op types
  - Add, Sub, or based on Funct bits
- Second level provides actual ALU control signal

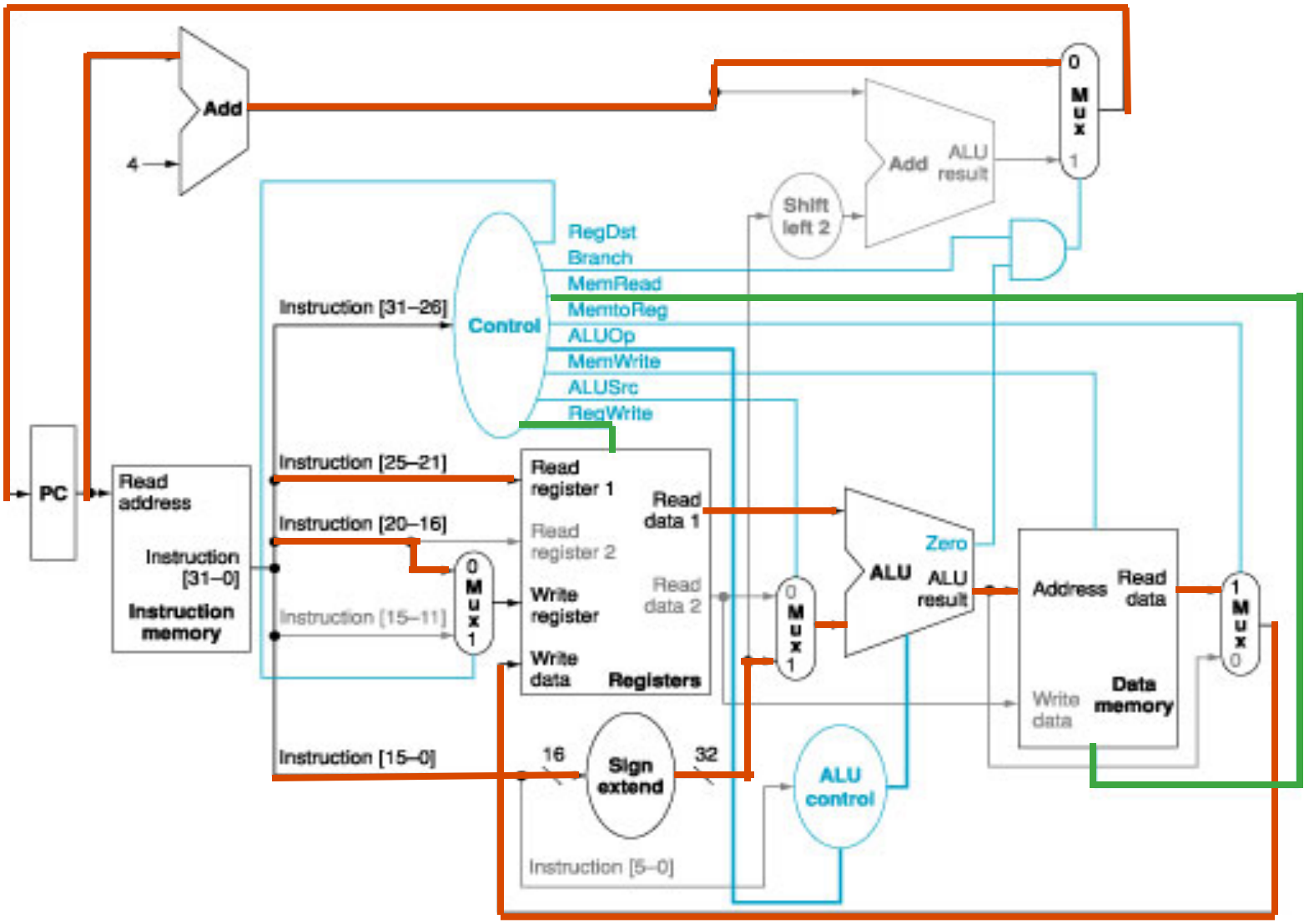
ALU operation	
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less than

actual **ALU Control** signals

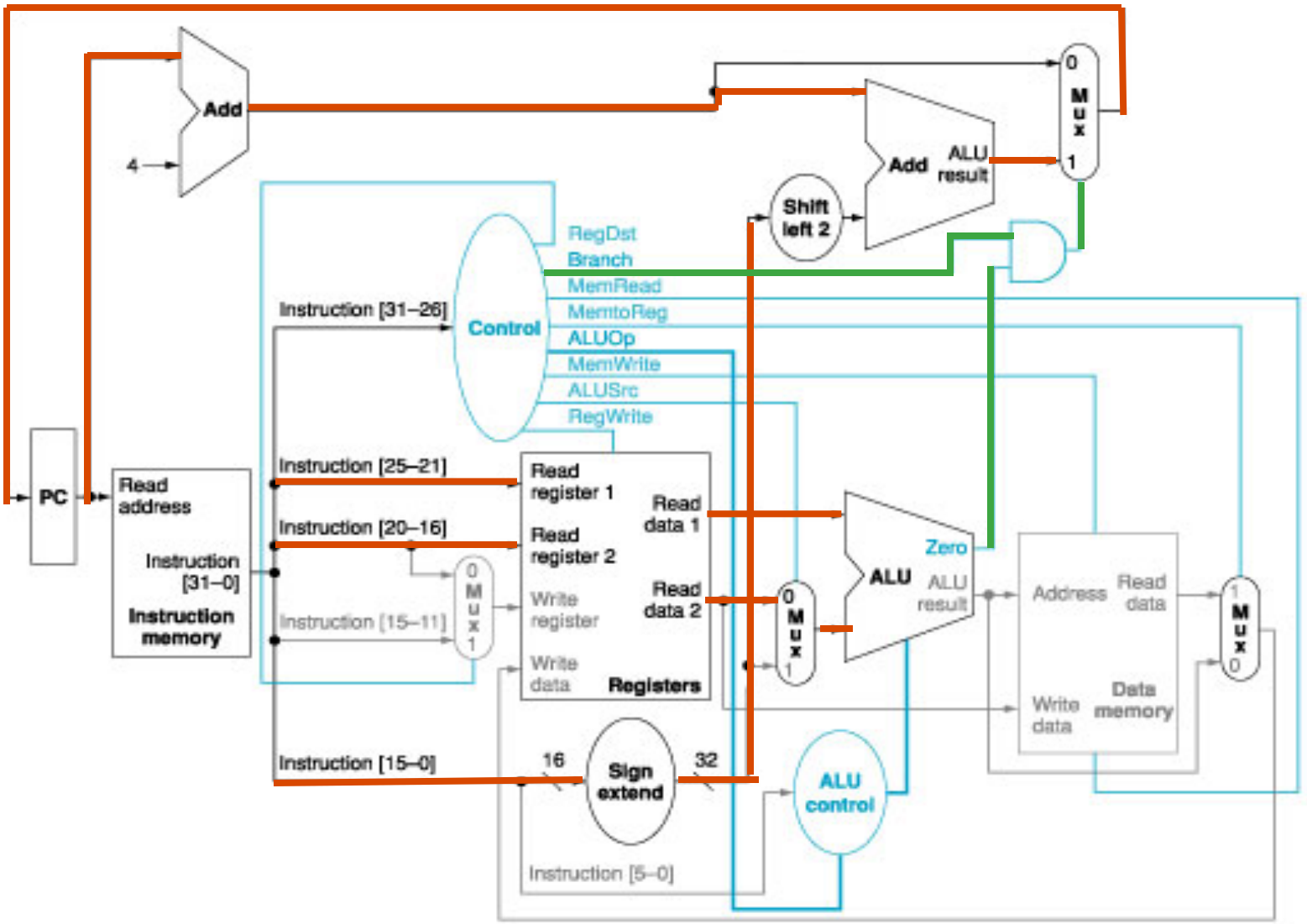
R-type instruction execution



lw execution

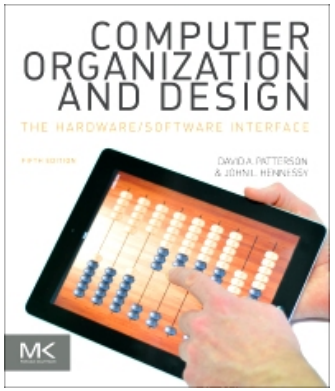


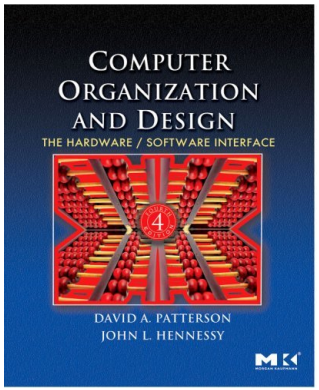
beq (taken) execution

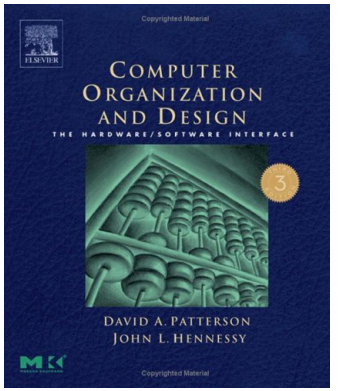


Don't fall behind!

- Read the textbook (Chapter 4 in 4/e & 5/e)







- Multi-cycle datapath (Tues): on Learn
- Tutorials next week
  - Read the "multi-cycle datapath" chapter (on Learn) ahead of time if your section is Mon or Tues

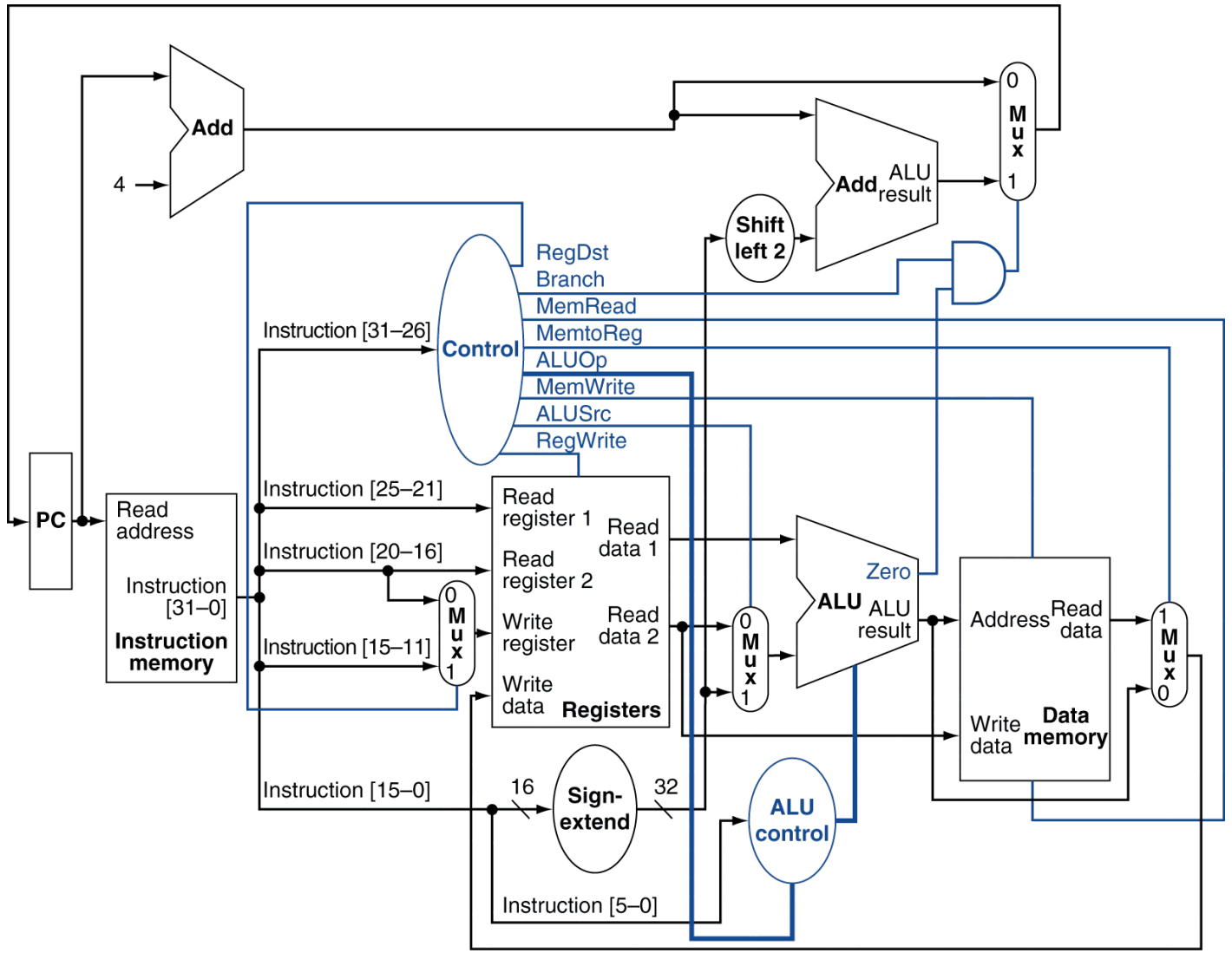
Inf2C - Computer Systems
Lecture 10
Processor Design – Multi-Cycle

Boris Grot

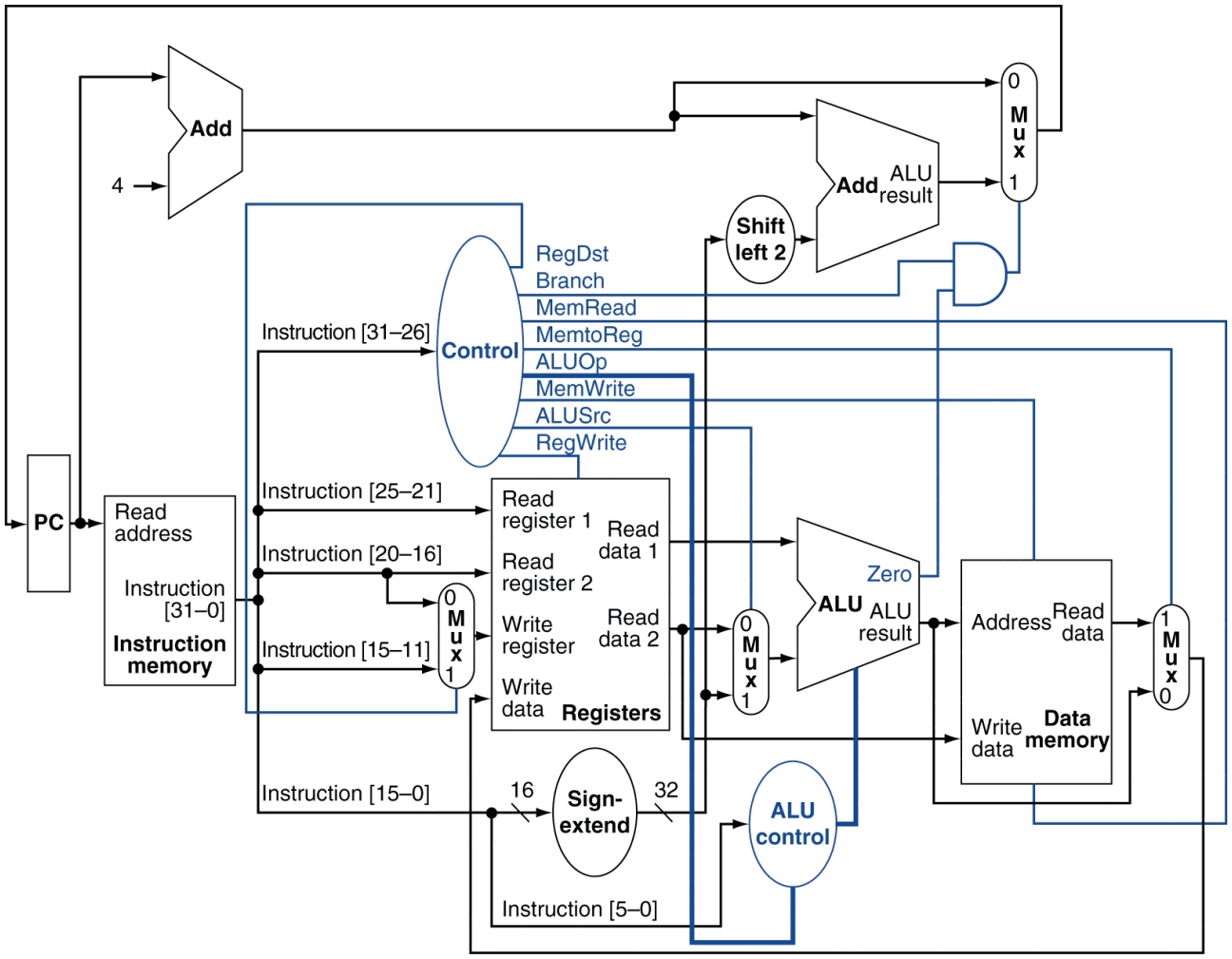
School of Informatics

University of Edinburgh

Previous lecture: single-cycle processor



Question: which instruction takes the most time to complete in a single-cycle processor?



Motivating a multi-cycle processor

Aren't single cycle processors good enough?

No!

- Speed: cycle time must be long enough for the most complex instruction to complete
  - But the average instruction needs less time
- Cost: functional units (e.g. adders) cannot be re-used within one instruction's execution

Multi-cycle processor

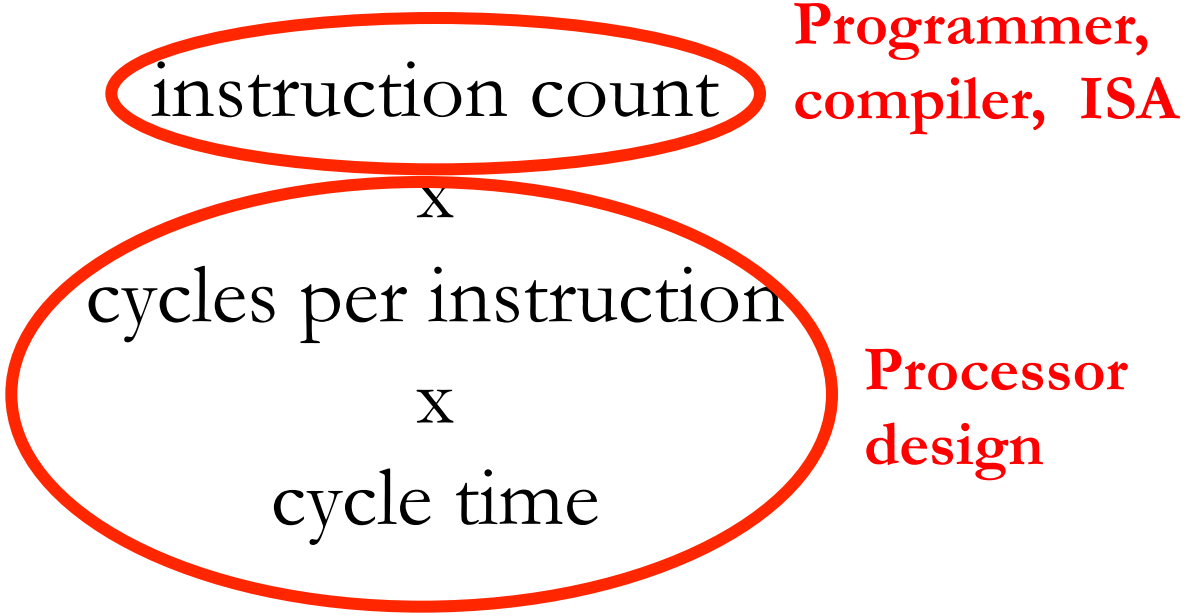
Basic idea:

- Break up the execution of each instruction into multiple cycles
- Reuse a common set of datapath and control components across cycles
- Ensure that the actions performed within each cycle "generic" – i.e., common to many instructions

End result: no instruction takes more time or uses more functional units than required

Measuring processor speed

Execution time is



Multi-Cycle Processor design guidelines

- Cycle time determined by the delay through the slowest functional unit
- At the end of each cycle, data required in subsequent cycles must be stored somewhere
  - Data for other instructions are kept in the memory, register file, or the PC
  - Data for same instruction are kept in new registers not visible to the programmer
- Reuse functional units as much as possible
  - Multiplexors added to select the different inputs

As before

New!

Determine the components

Processor task

- Instruction fetch from memory
- Read registers
- Execution
  - Data processing instructions
  - Data transfer instructions
  - Branch instructions

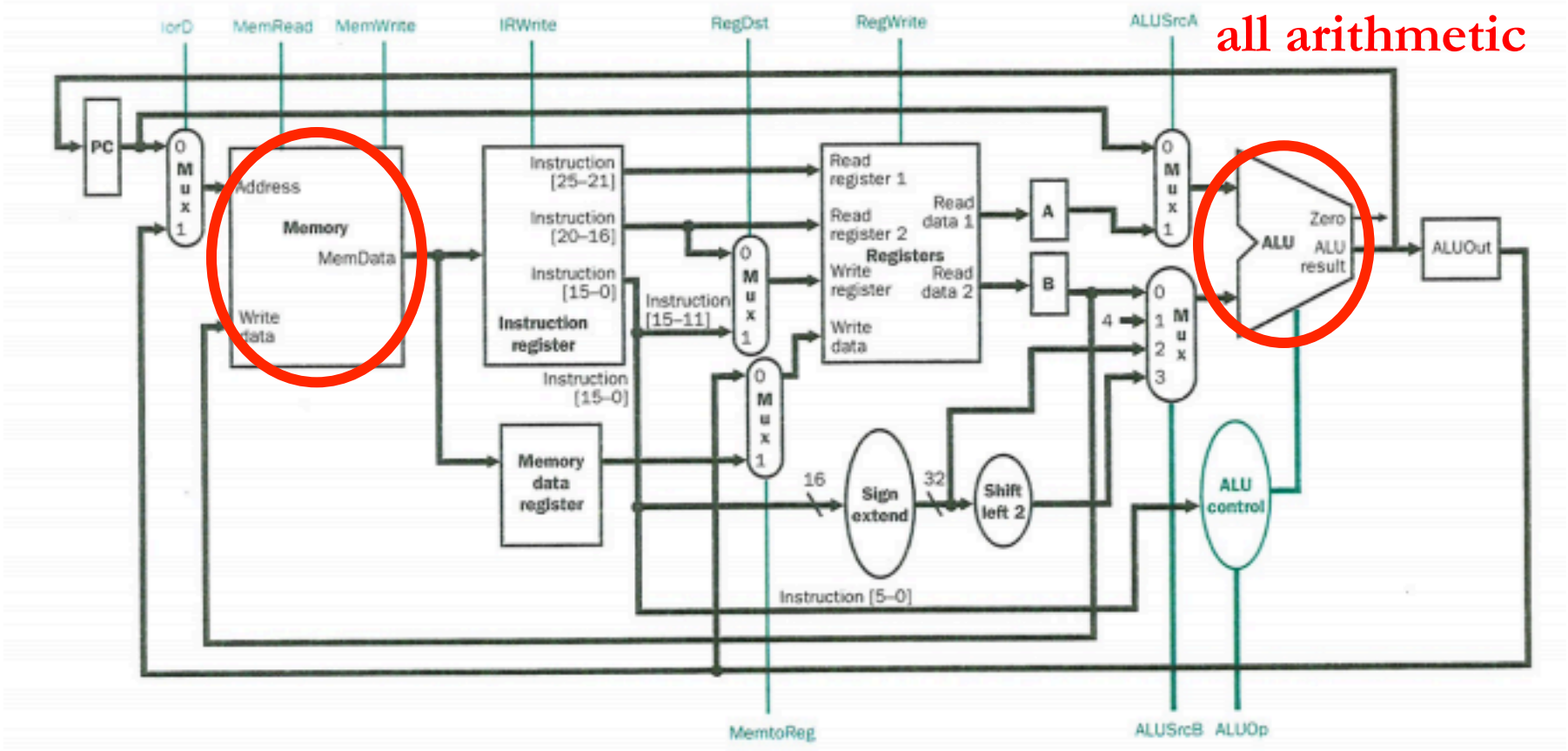
Component list

- PC register
- ~~Memory (instructions)~~
- ~~Adder: PC+4~~
- Register file
- ALU
- Memory (data)
- ~~Adder: branch target~~

Multi-cycle datapath (overview)

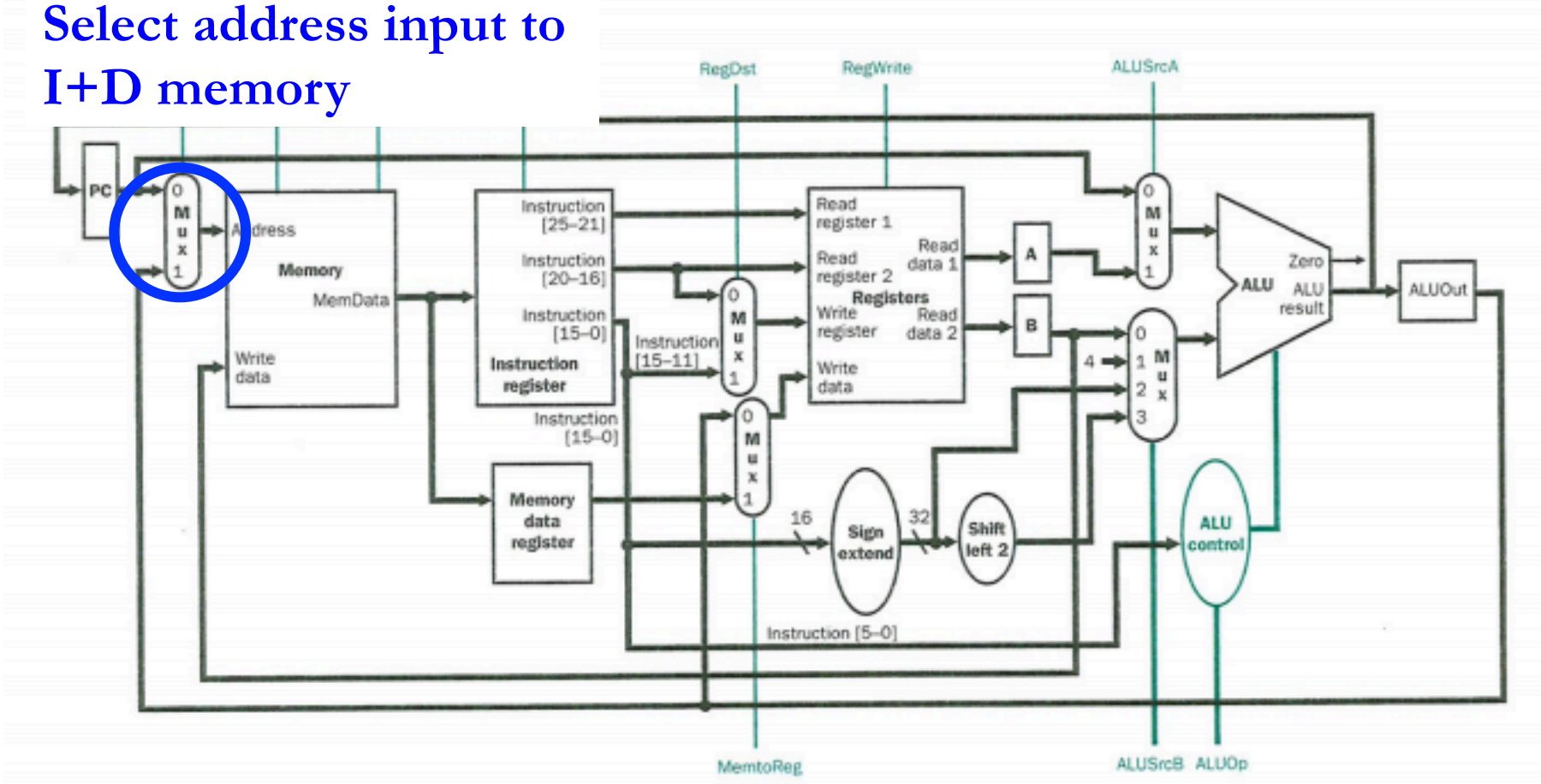
One memory for both Instruction & Data

One ALU for all arithmetic

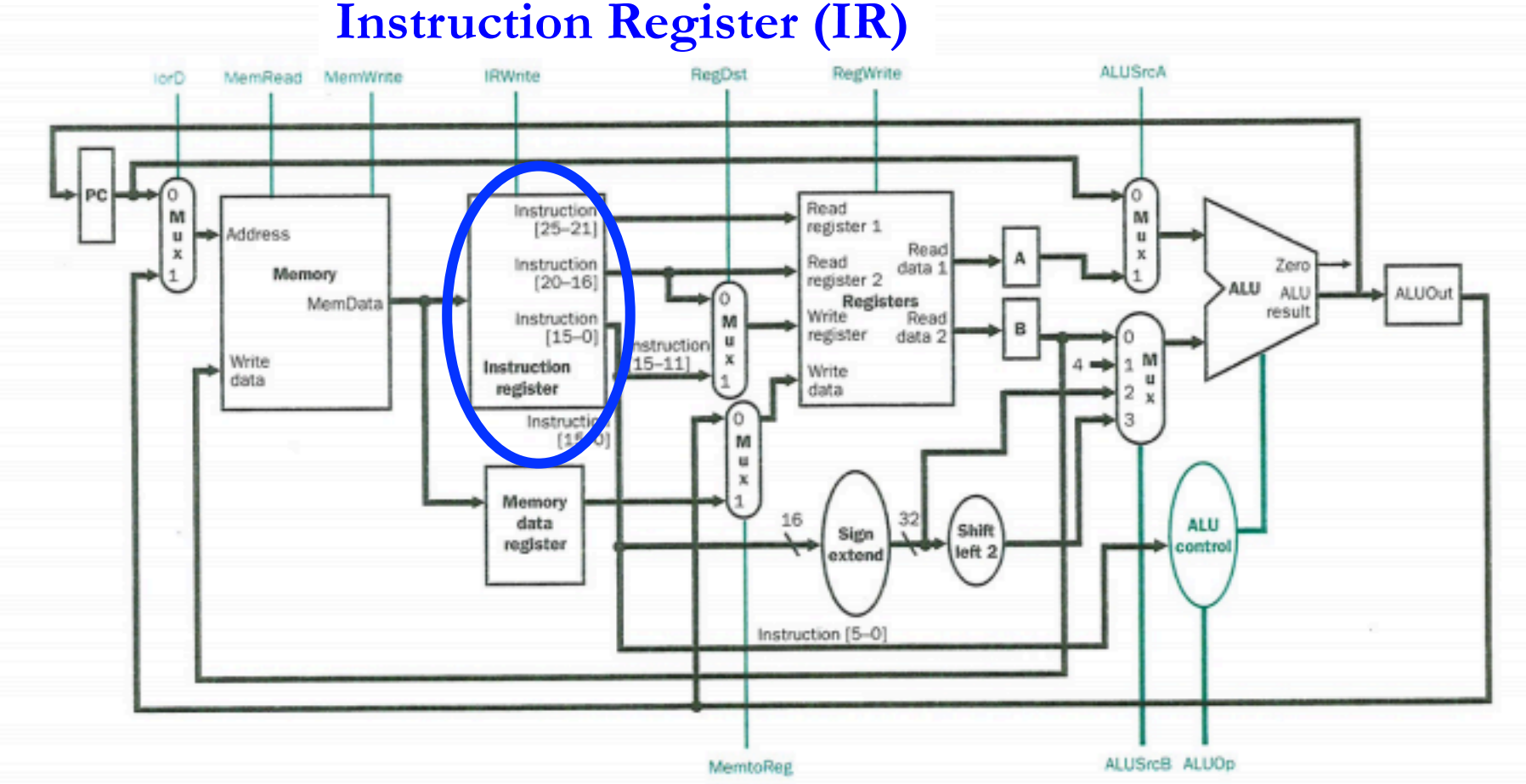


Multi-cycle datapath (overview)

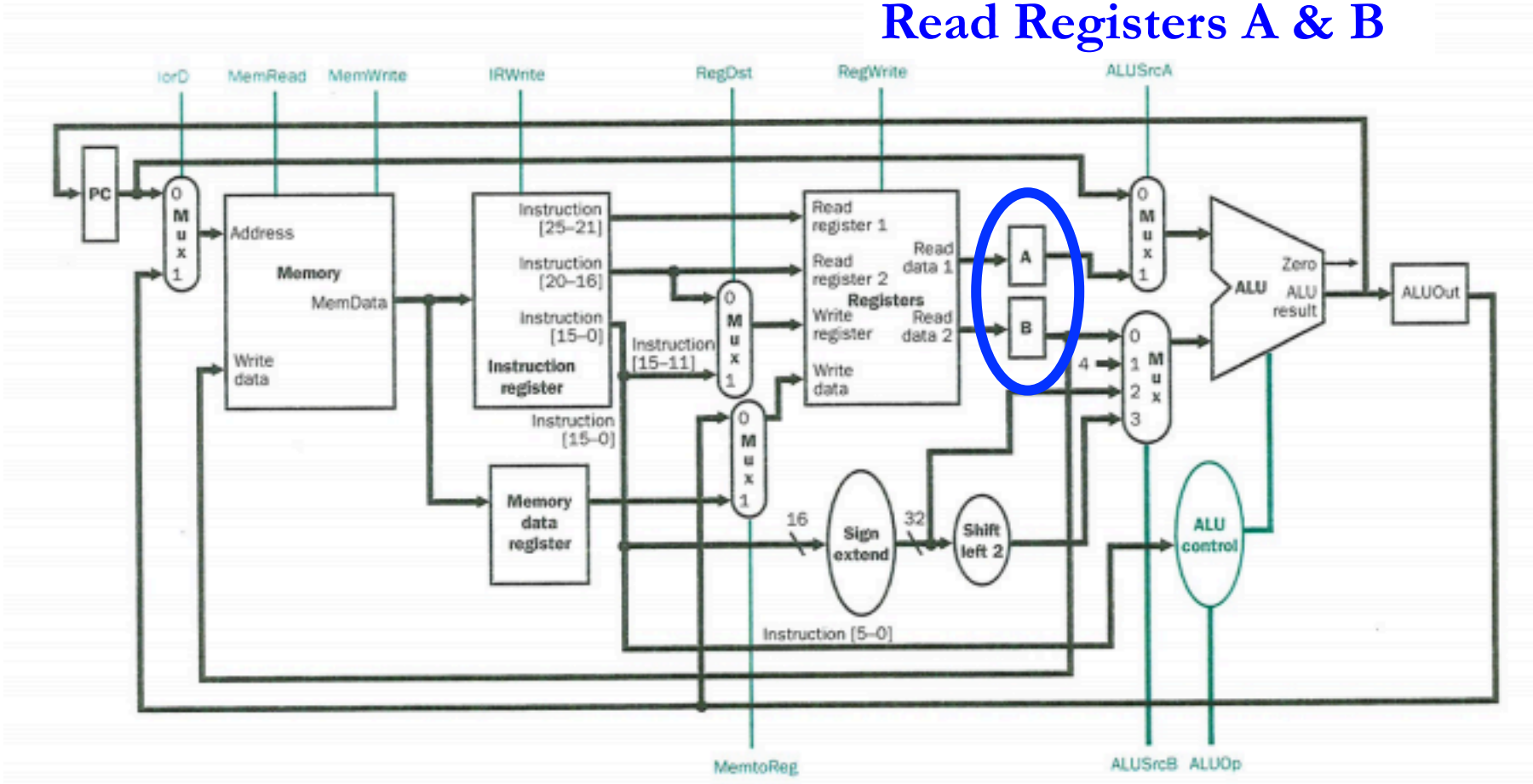
Select address input to I+D memory



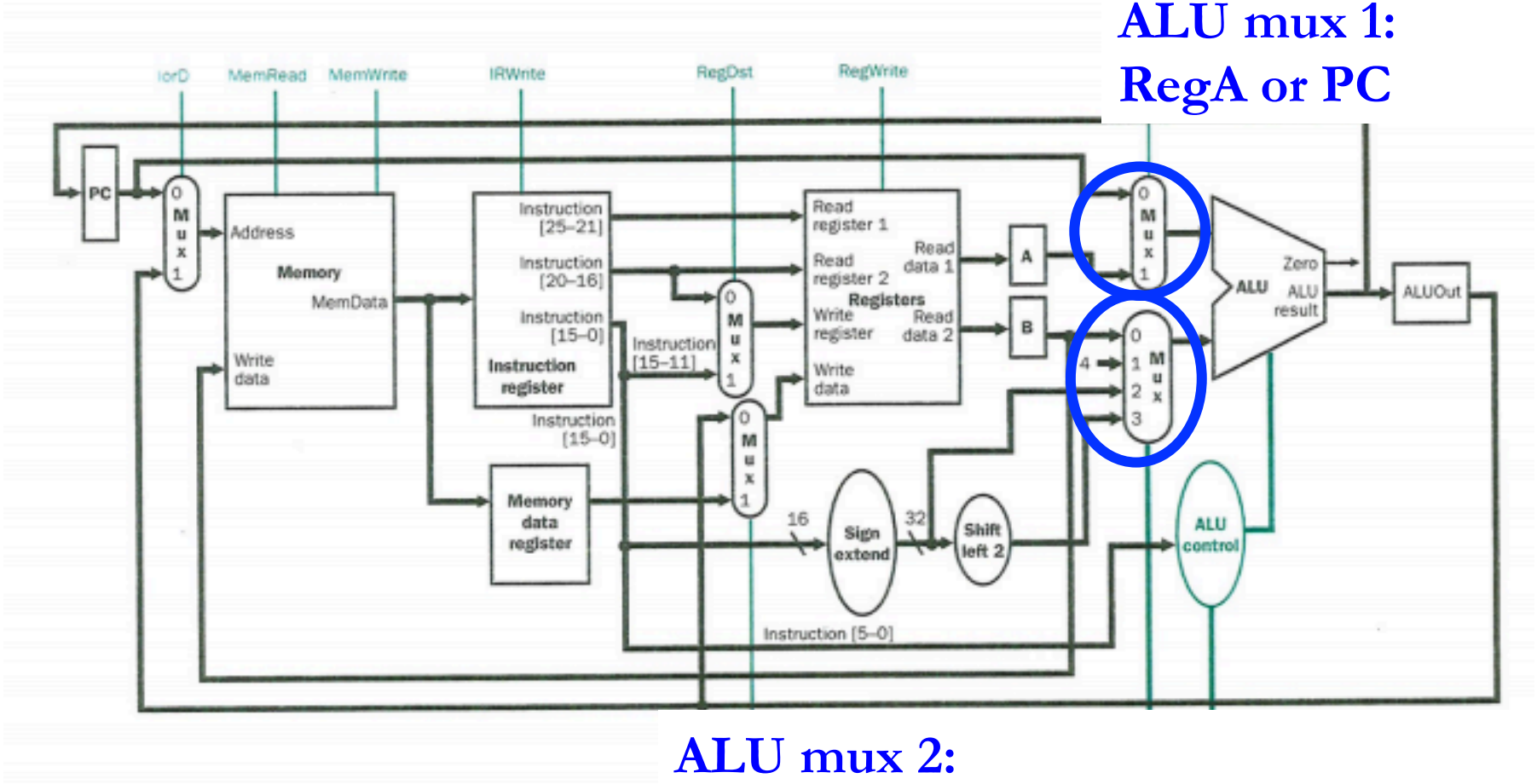
Multi-cycle datapath (overview)



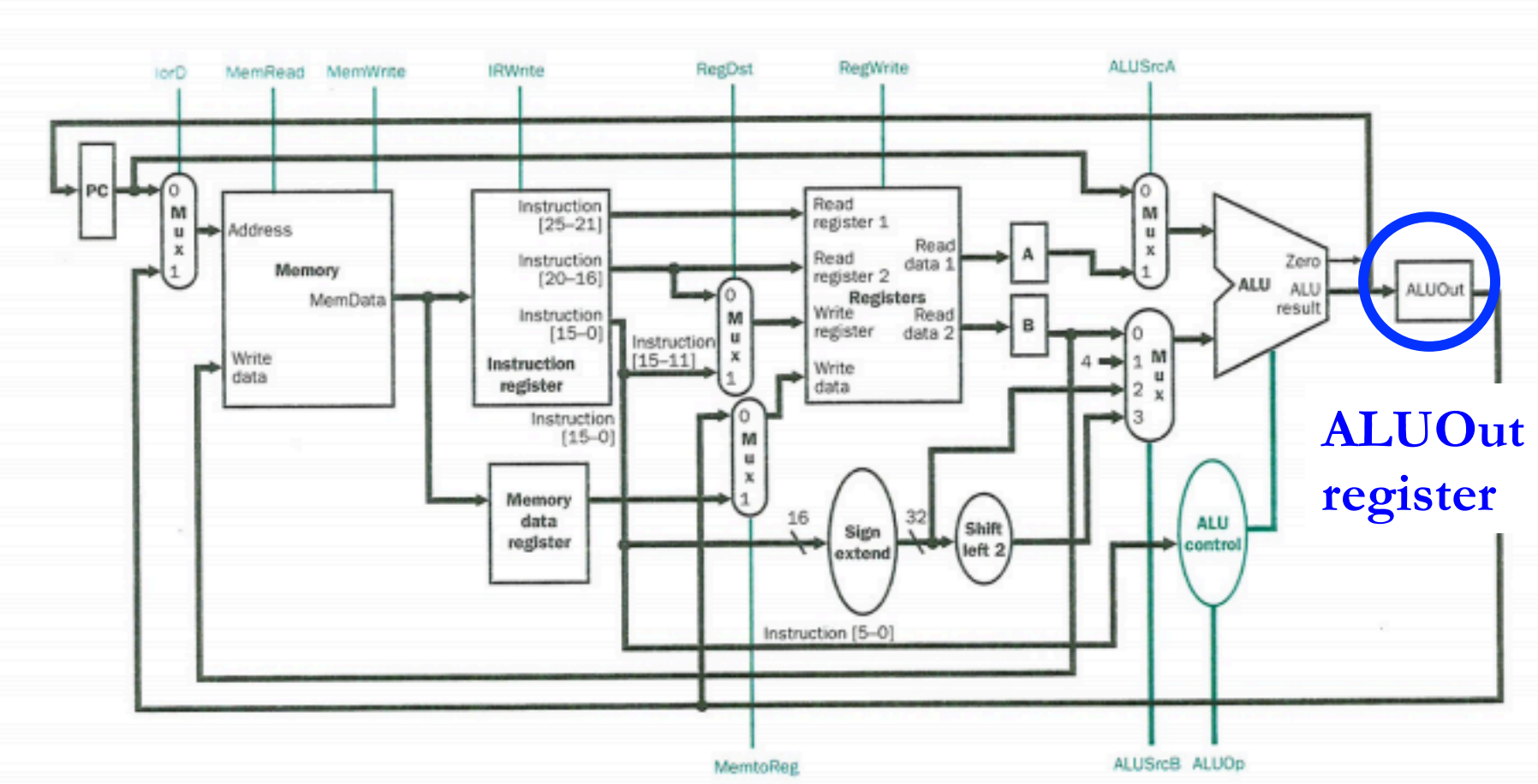
Multi-cycle datapath (overview)



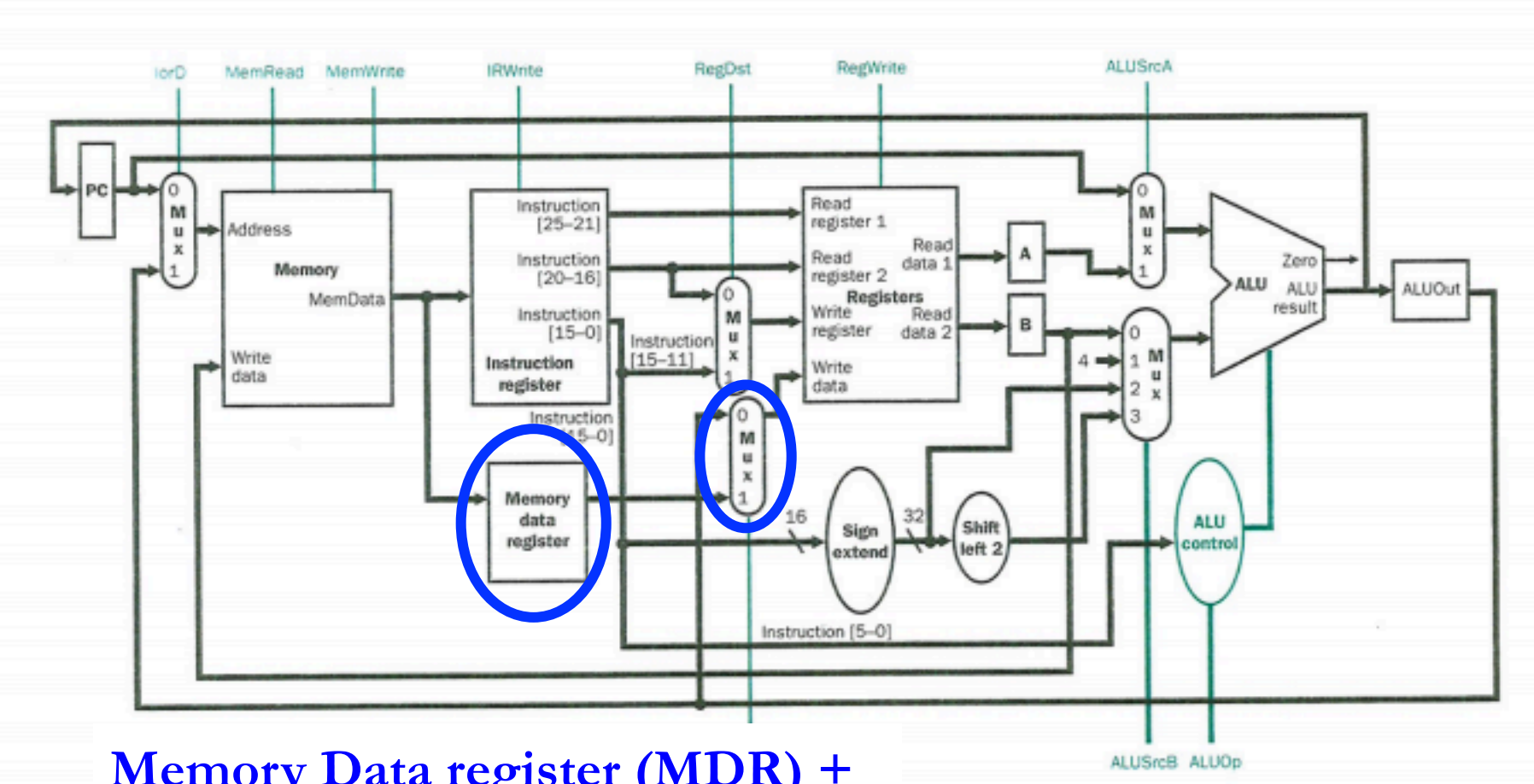
Multi-cycle datapath (overview)



Multi-cycle datapath (overview)

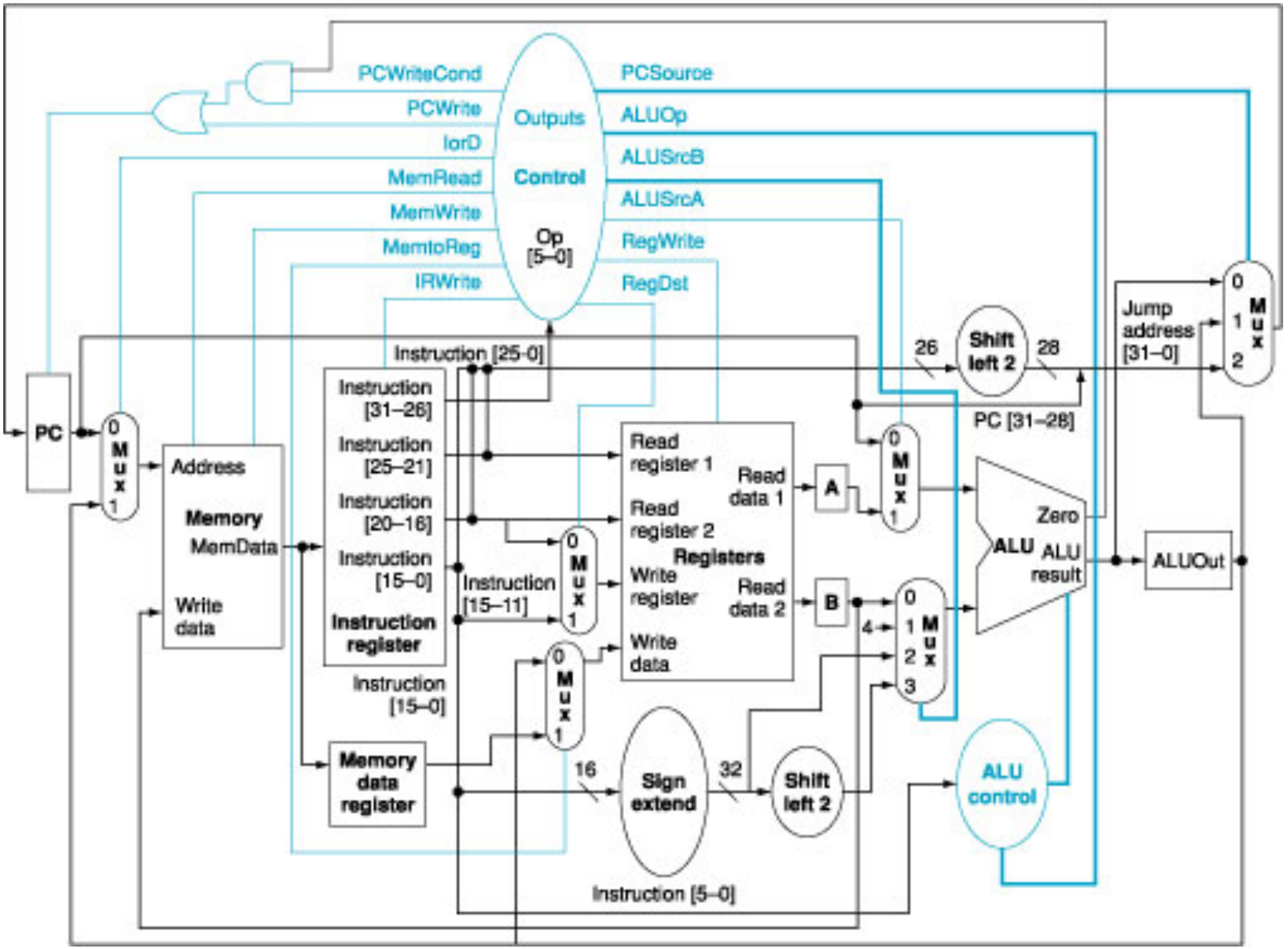


Multi-cycle datapath (overview)



Memory Data register (MDR) + Write Data mux

Multi-cycle datapath (with control)



Multi-cycle processor control signals (1-bit)

Signal name	Function
RegDst	Write register in the register file (*rs* or *rd*)
RegWrite	Write the register selected via RegDst?
ALUSrcA	Select ALU input A (*PC* or *A* register)
MemRead	Read the memory (could be for instruction fetch or data load)
MemWrite	Write the memory
MemtoReg	Select the source of data to be written to the register file (*ALUout* or *MDR*)
IorD	Select the source of the address for the memory unit (*PC* or *ALUout*)
IRWrite	Write the IR with output of the memory unit
PCWrite	Write the PC register (source controlled by *PCSource*)
PCWriteCond	Write the PC if *Zero* output from the ALU is active

Multi-cycle processor control signals (2-bit)

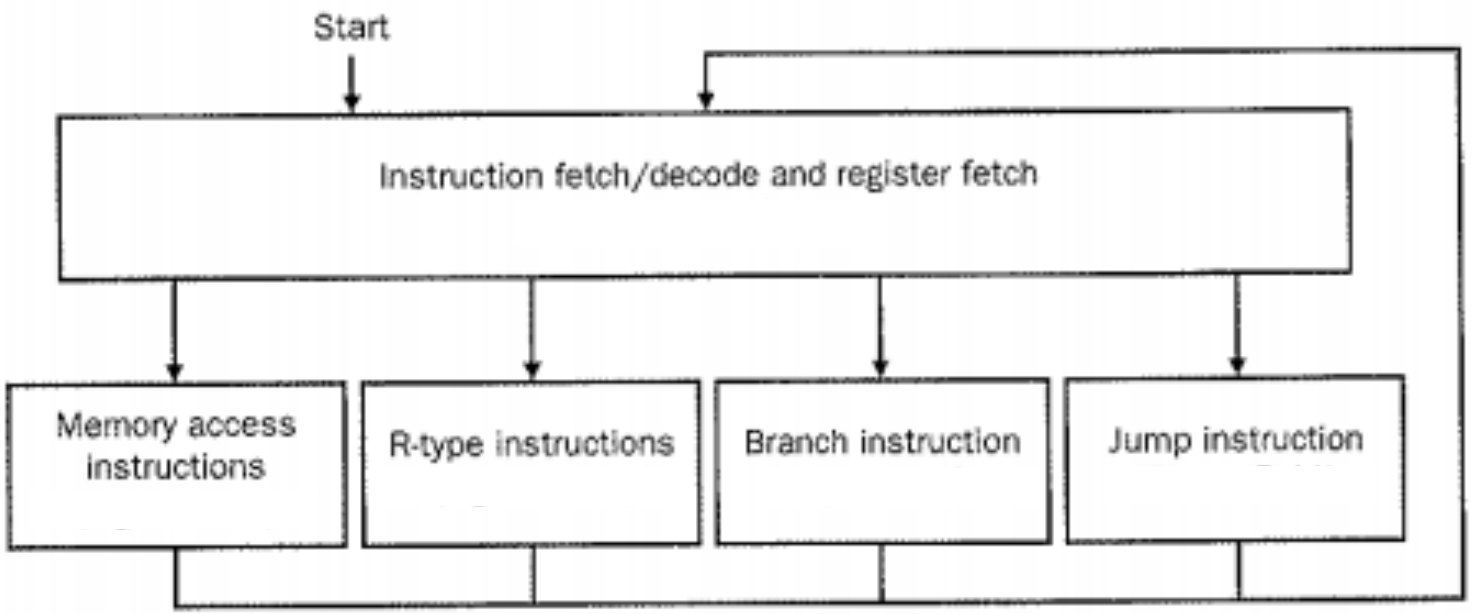
Signal name	Value	Function
ALUSrcB	00	Second ALU input comes from B register
	01	Second ALU input is a constant 4
	10	Second ALU input is sign-extended lower 16 bits of IR
	11	Second ALU input is sign-extended lower 16-bits of IR shifted left by 2
ALUOp	00	ALU performs an Add
	01	ALU performs a Subtract
	10	ALU action determined by FUNCT field
PCSource	00	Output of the ALU (PC+4)
	01	ALUout (branch target address)
	10	Jump target address (IR[25-0] shifted left 2 bits and combined with PC+4[31-28])

How to design the control

- The control unit of a multicycle processor is an FSM

- For a given instruction type, determines the sequence of control signals on a cycle-by-cycle basis
  - On a given cycle: outputs the control signals and next FSM state

Control FSM overview



- Fetch & Decode common for all insts
  - 2 cycles (Fetch, Decode)
- Rest: depends on the inst type
  - 1 to 3 additional cycles

What happens in each cycle – 1 & 2

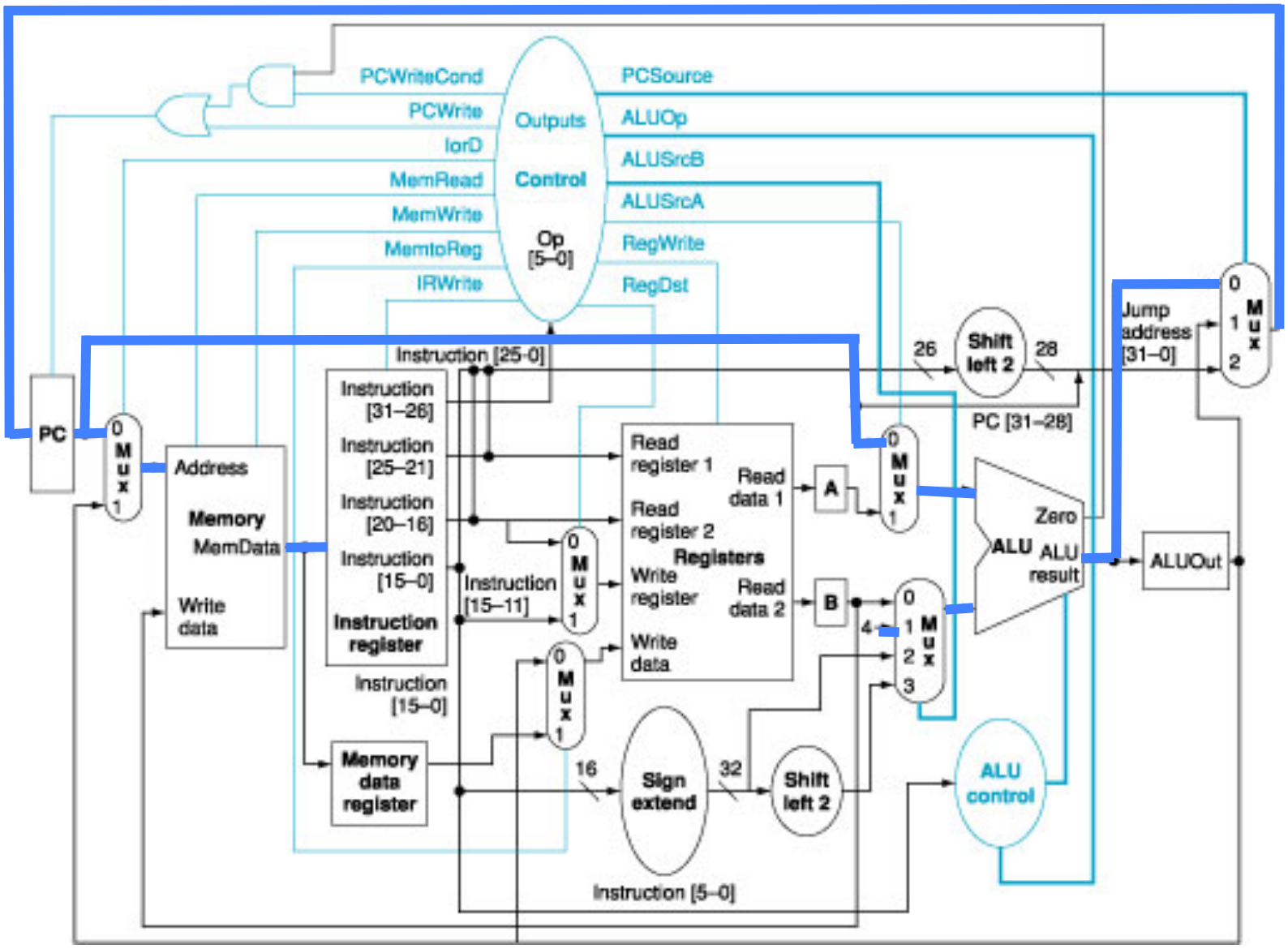
1. Instruction fetch

```
IR <= Mem[PC]
PC <= PC+4
```

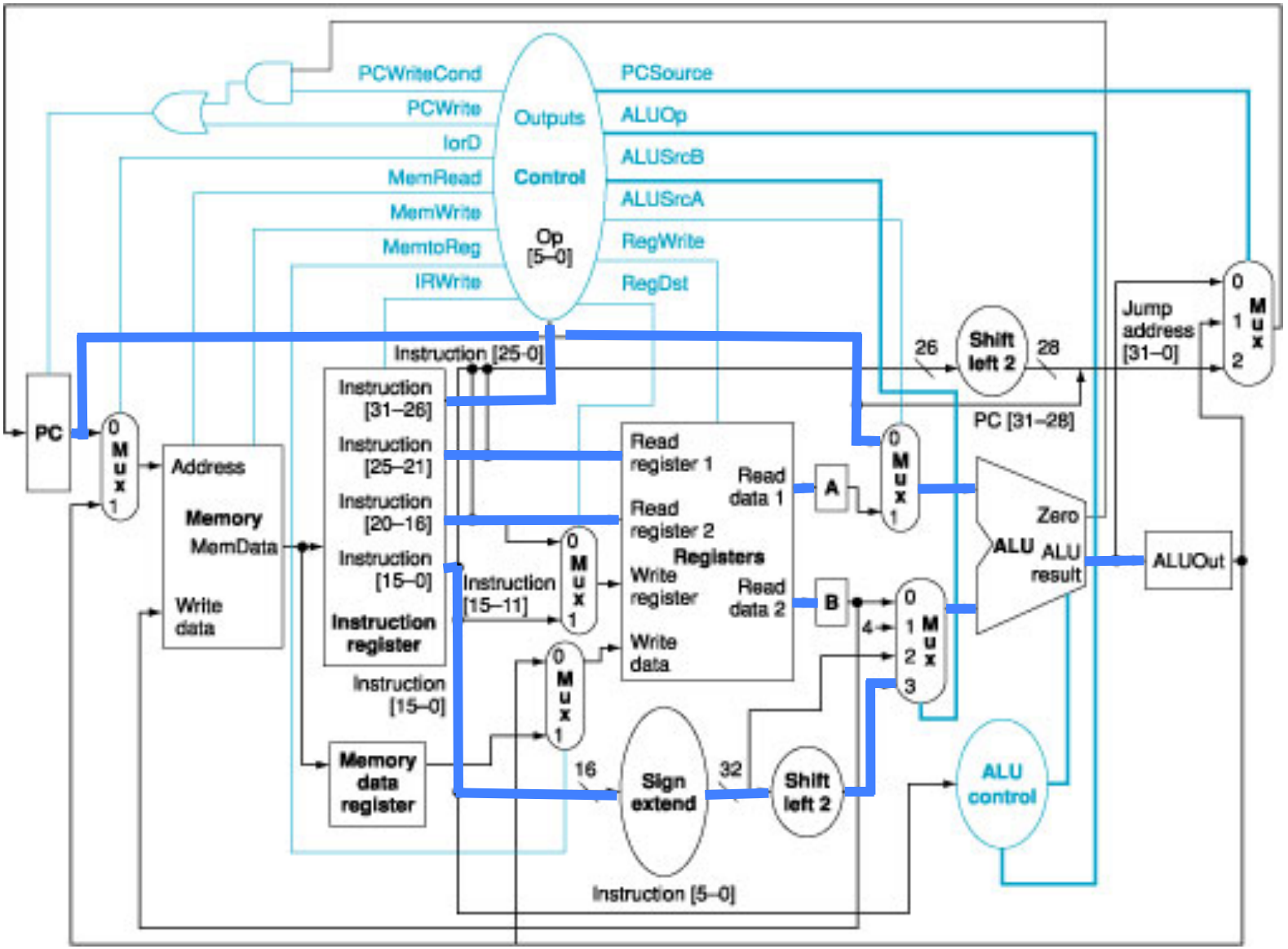
2. Instruction decode and register fetch

```
A <= Reg[IR[25:21]]
B <= Reg[IR[20:16]]
ALUOut <= PC+sgnext(IR[15:0]<<2)
```

Cycle 1 – instruction fetch (all)



Cycle 2 –instr decode & reg read (all)



What happens in each cycle – 3

3a. Memory address generation

```
ALUOut <= A+sgnext(IR[15:0])
```

3b. R-type arithmetic-logical instruction

```
ALUOut <= A op B
```

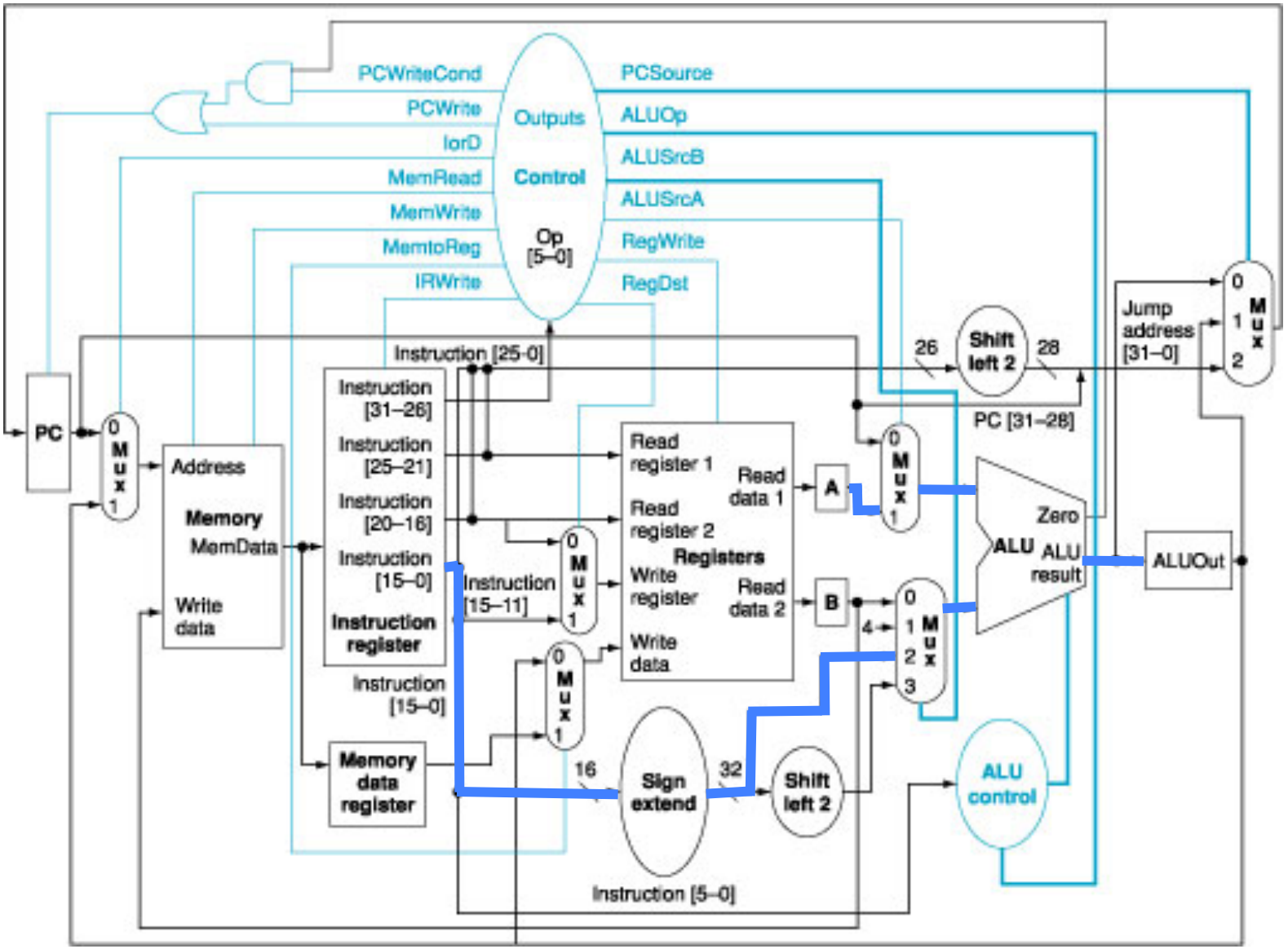
3c. Branch completion

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

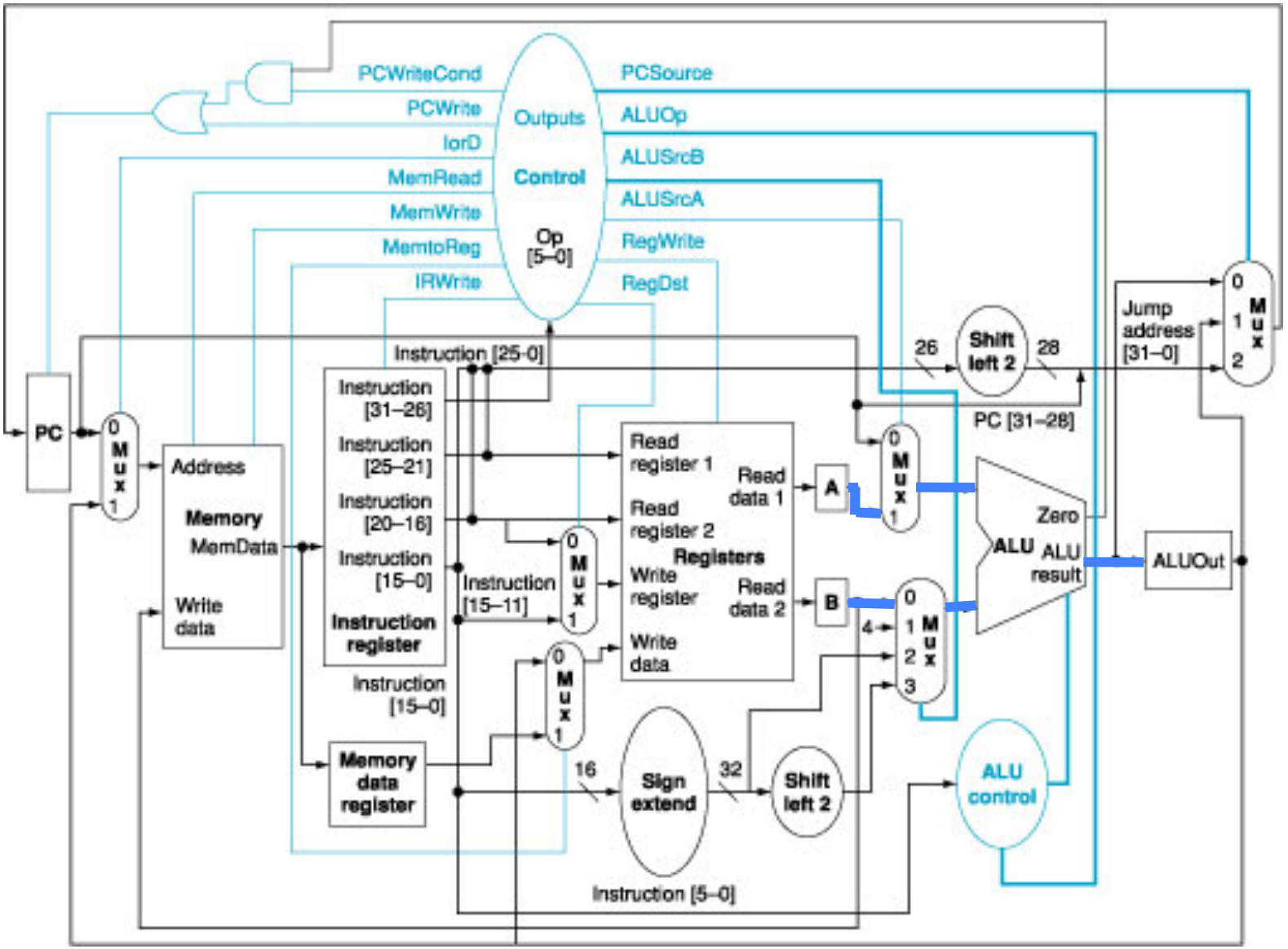
3d. Jump completion

```
PC <= {PC[31:28],IR[25:0],2'b00}
```

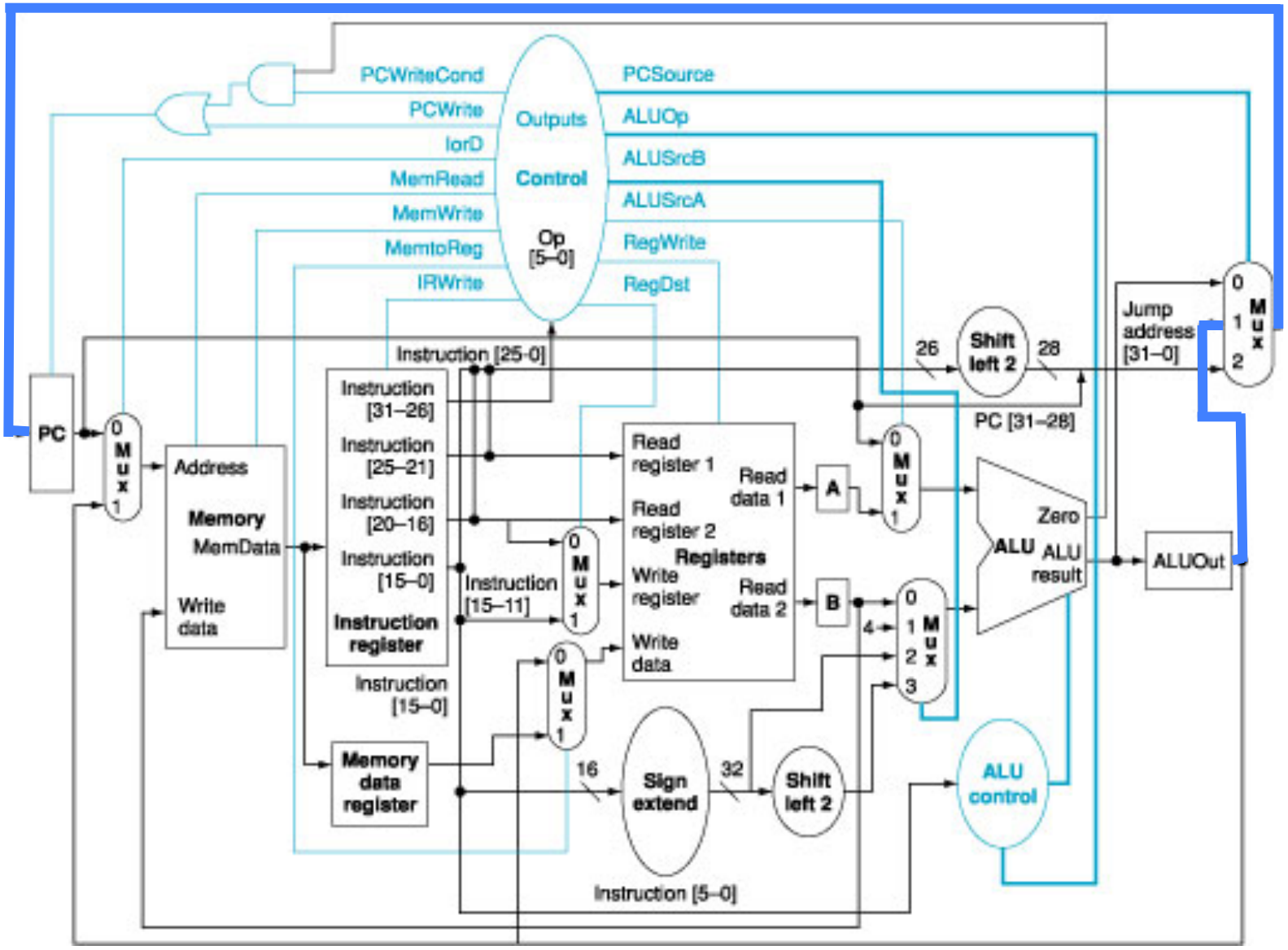
Cycle 3a: add imm arg (addi, lw, sw)



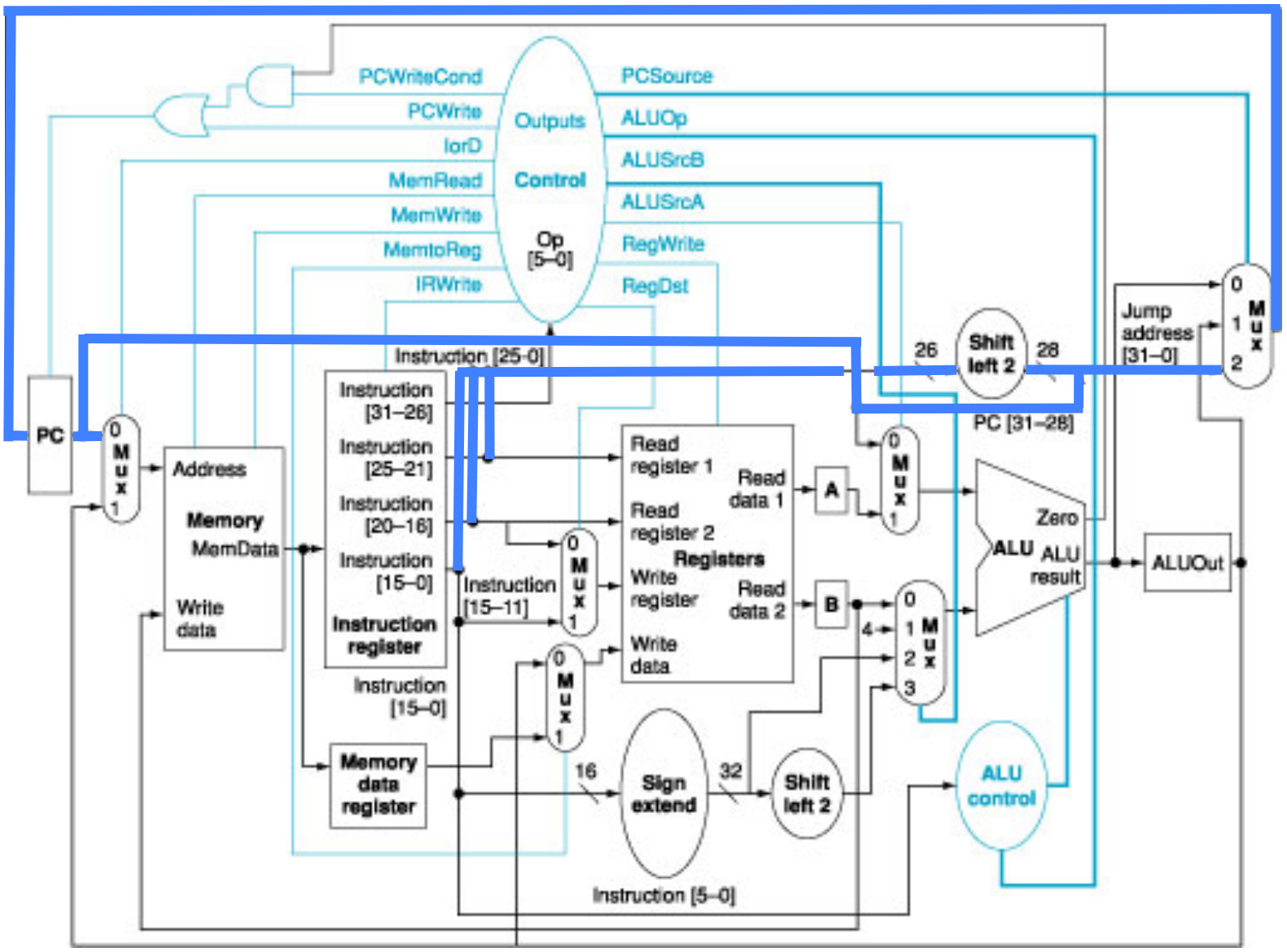
Cycle 3b: R-type ALU op (add, and)



Cycle 3c: Branch taken (beq, bne)



Cycle 3d: jump (j)



What happens in each cycle – 4

4a. Memory access (load)

```
MDR <= Mem[ALUOut]
```

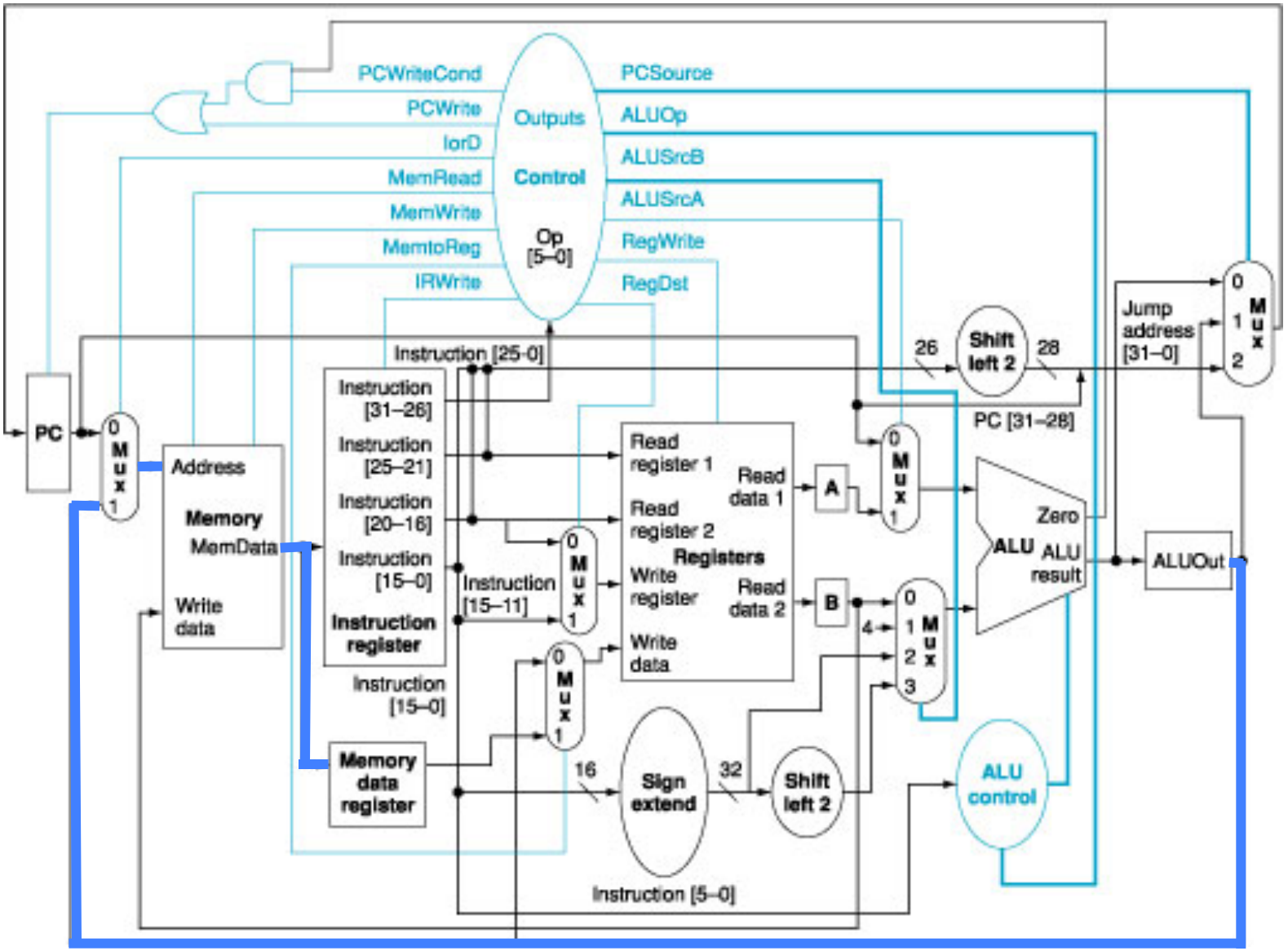
4b. Memory access (store) & completion

```
Mem[ALUOut] <= B
```

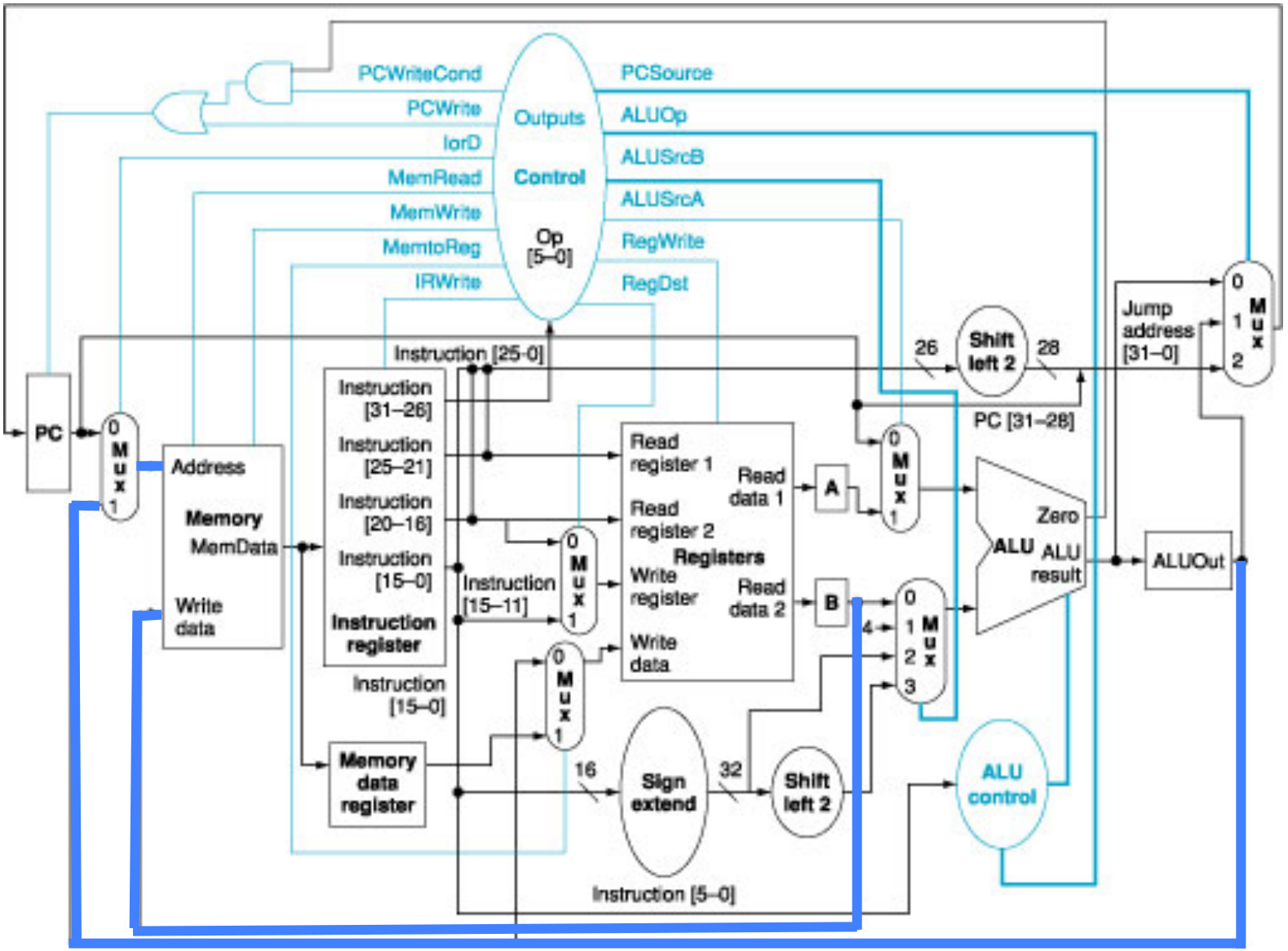
4c. R-type arith-logical instruction completion

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

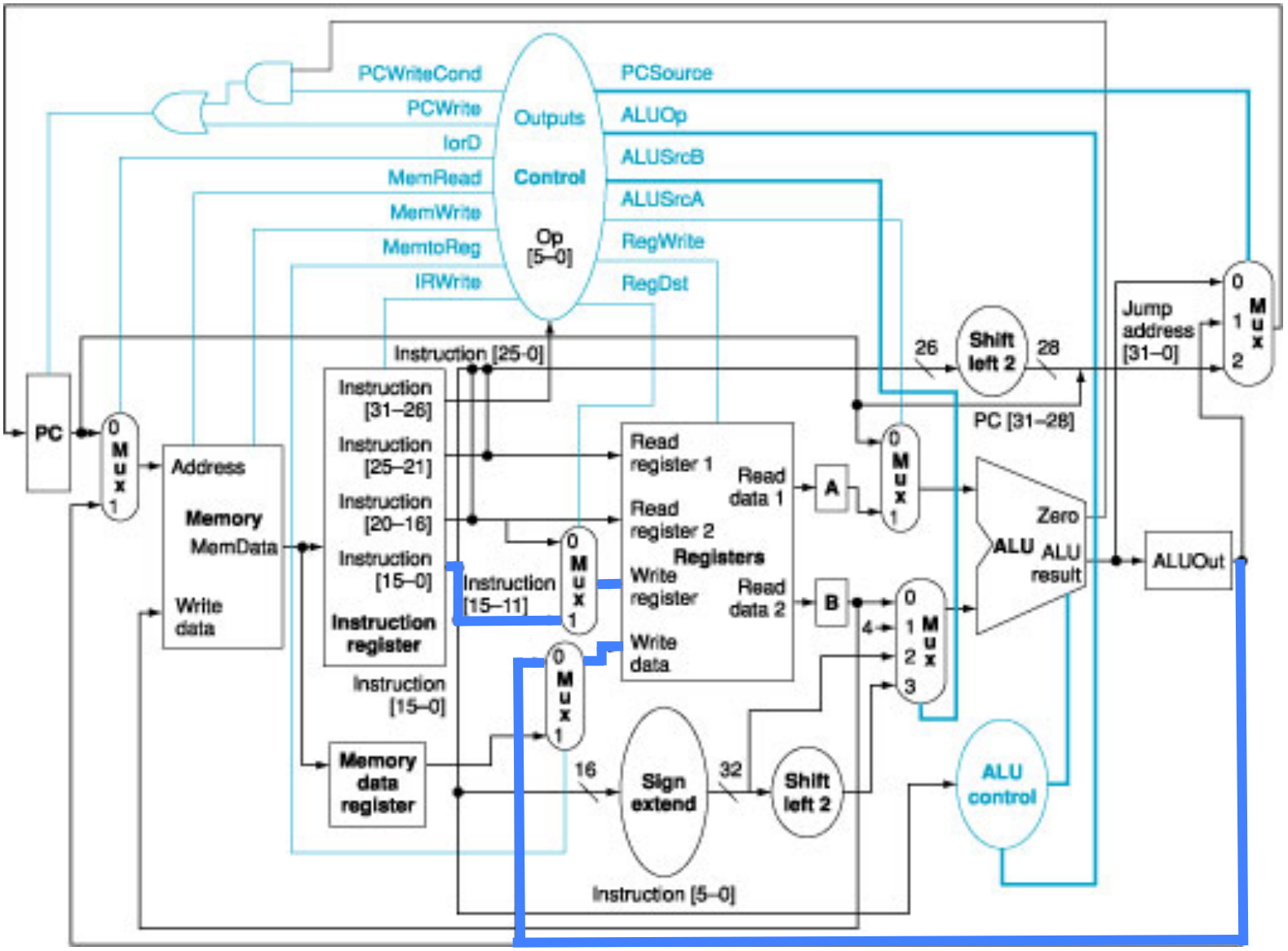
Cycle 4a: Load from mem (lw)



Cycle 4b: Store to mem (sw)



Cycle 4c: R-type result write (add, and)

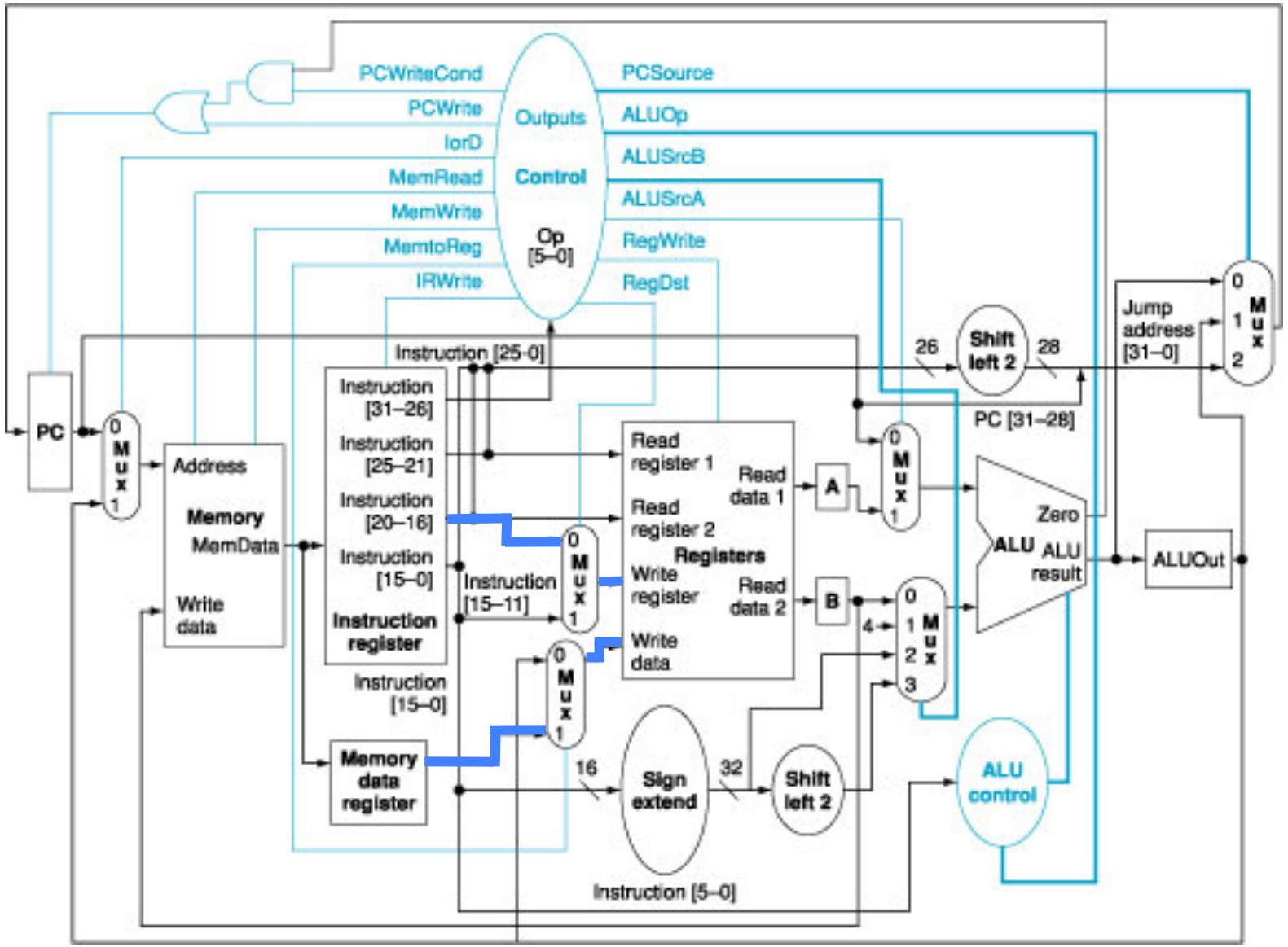


What happens in each cycle – 5

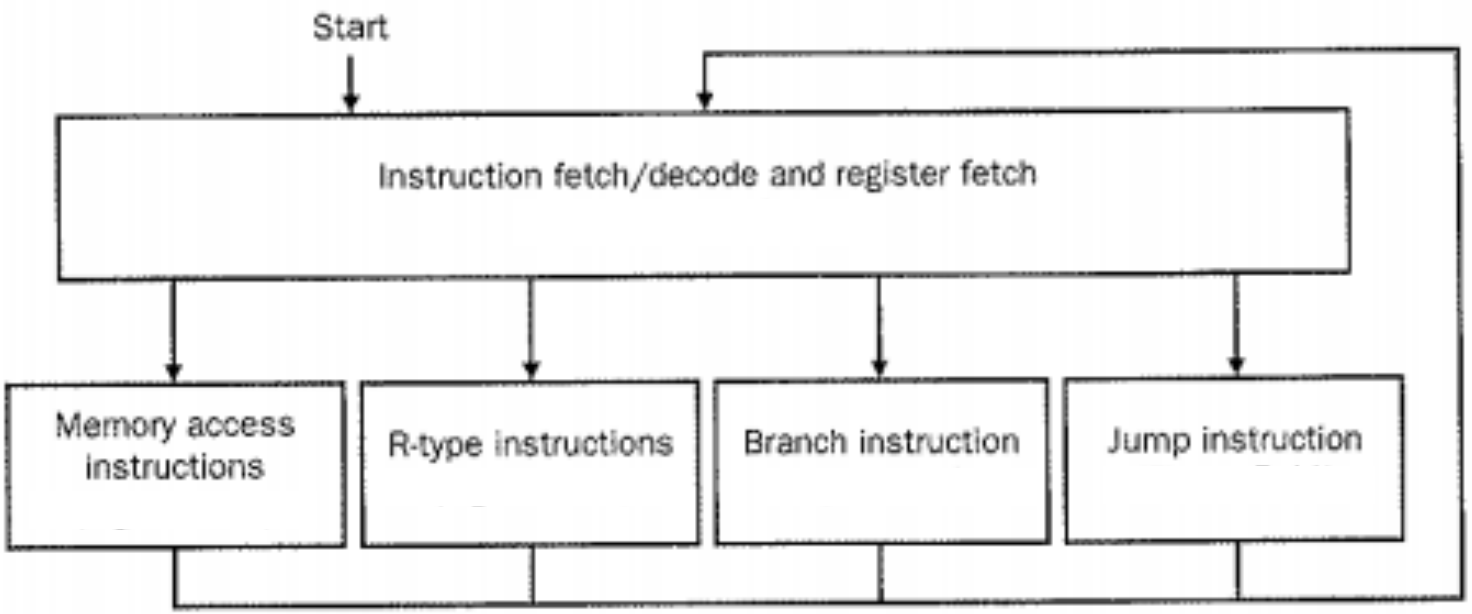
5. Load instruction completion

```
Reg[IR[20:16]] <= MDR
```

Cycle 5: save of loaded value (lw)

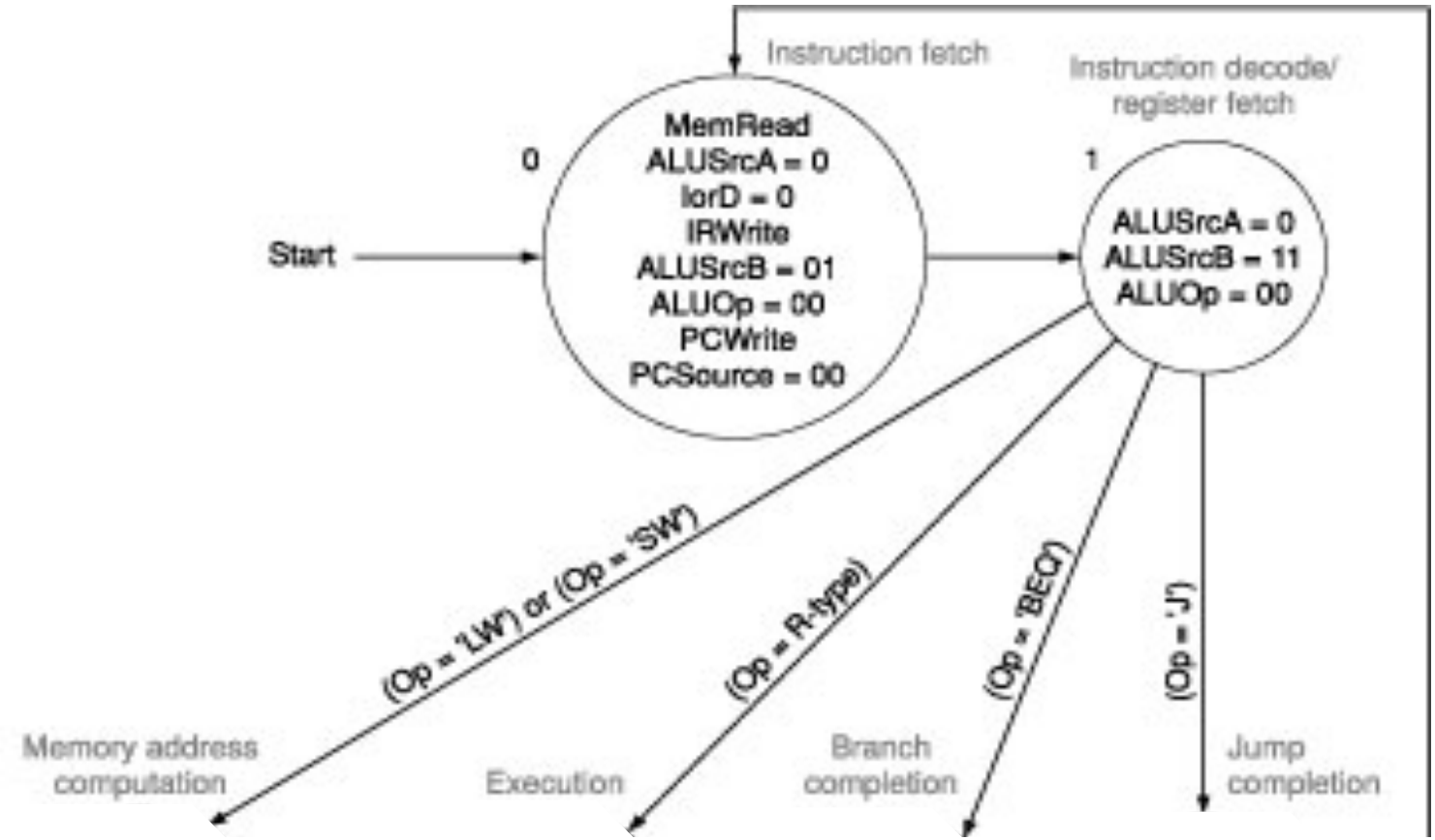


Designing the Control Unit

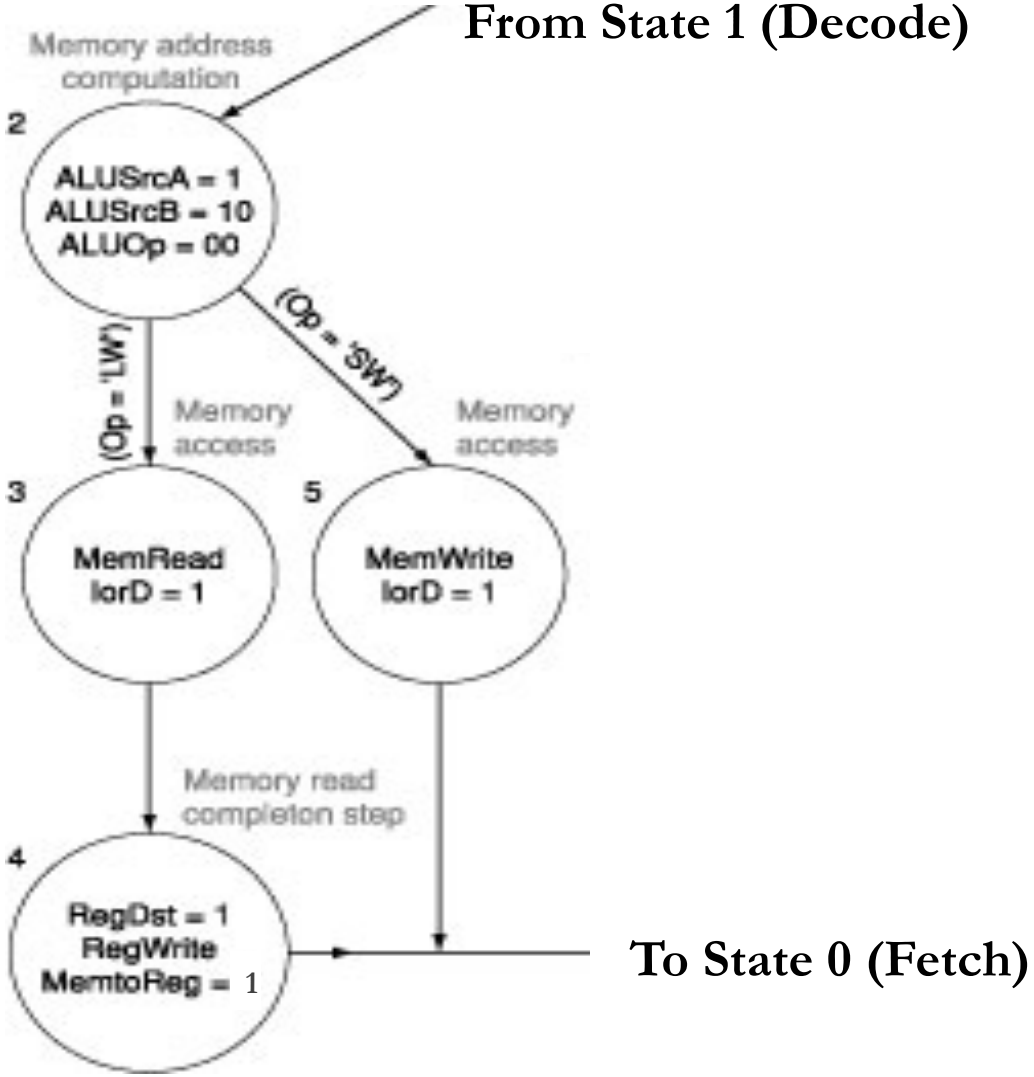


- Fetch & Decode common for all insts
  - 2 cycles total (Fetch, Decode)
- Rest: depends on the inst type
  - 1 to 3 additional cycles

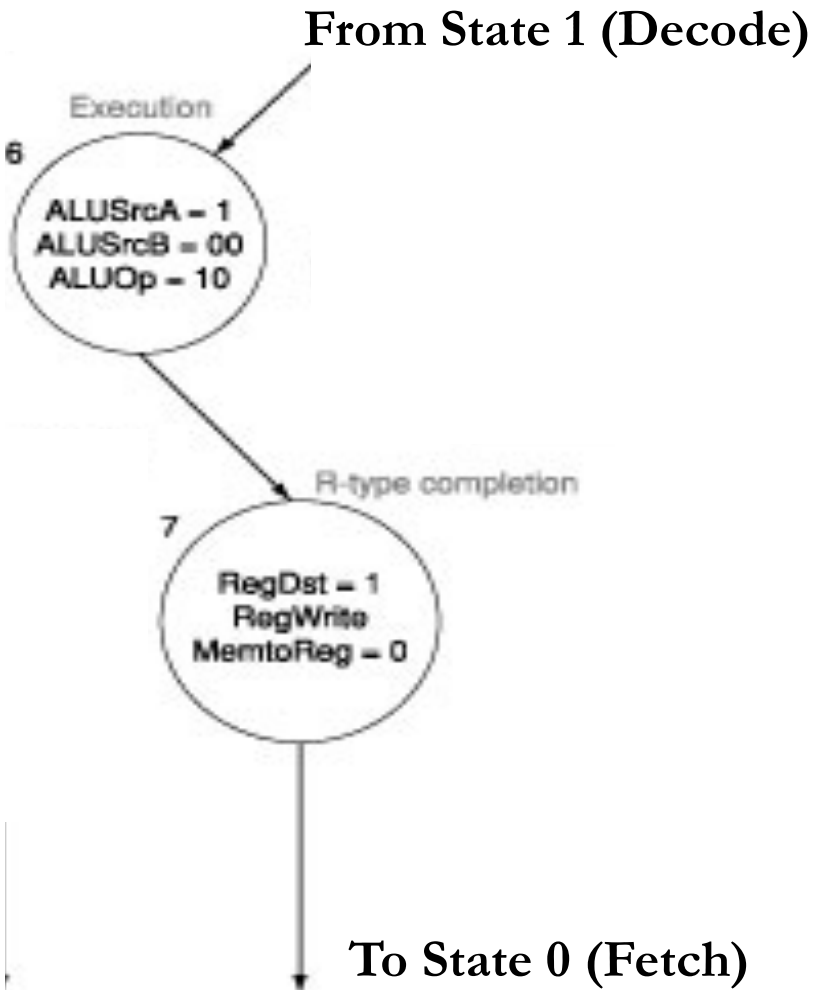
Fetch and Decode States



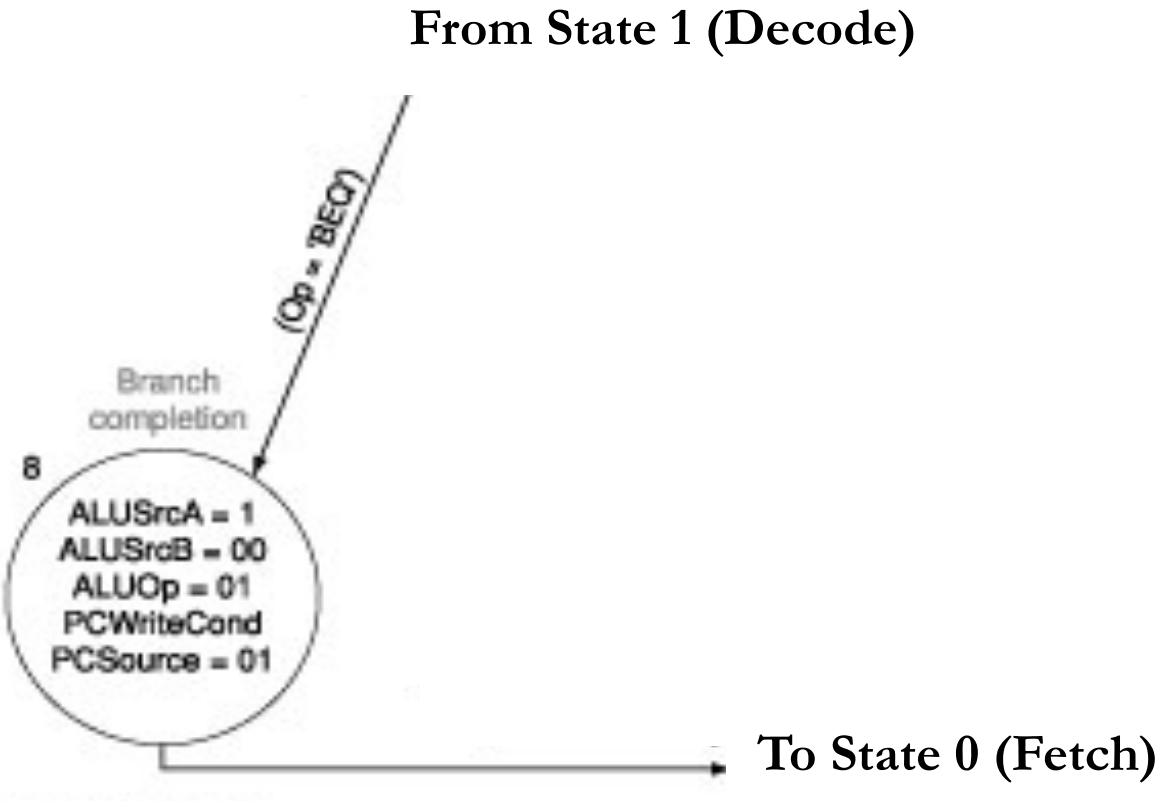
Memory Access States



R-Type Instruction States

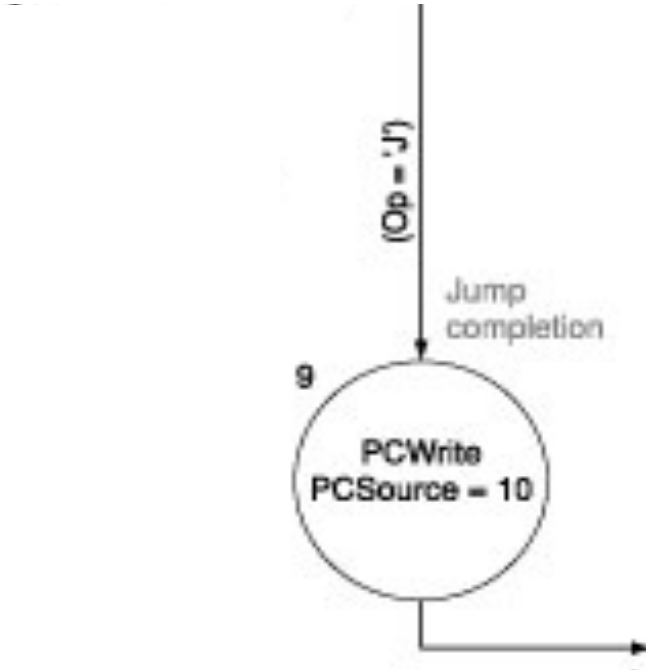


Branch Resolution States

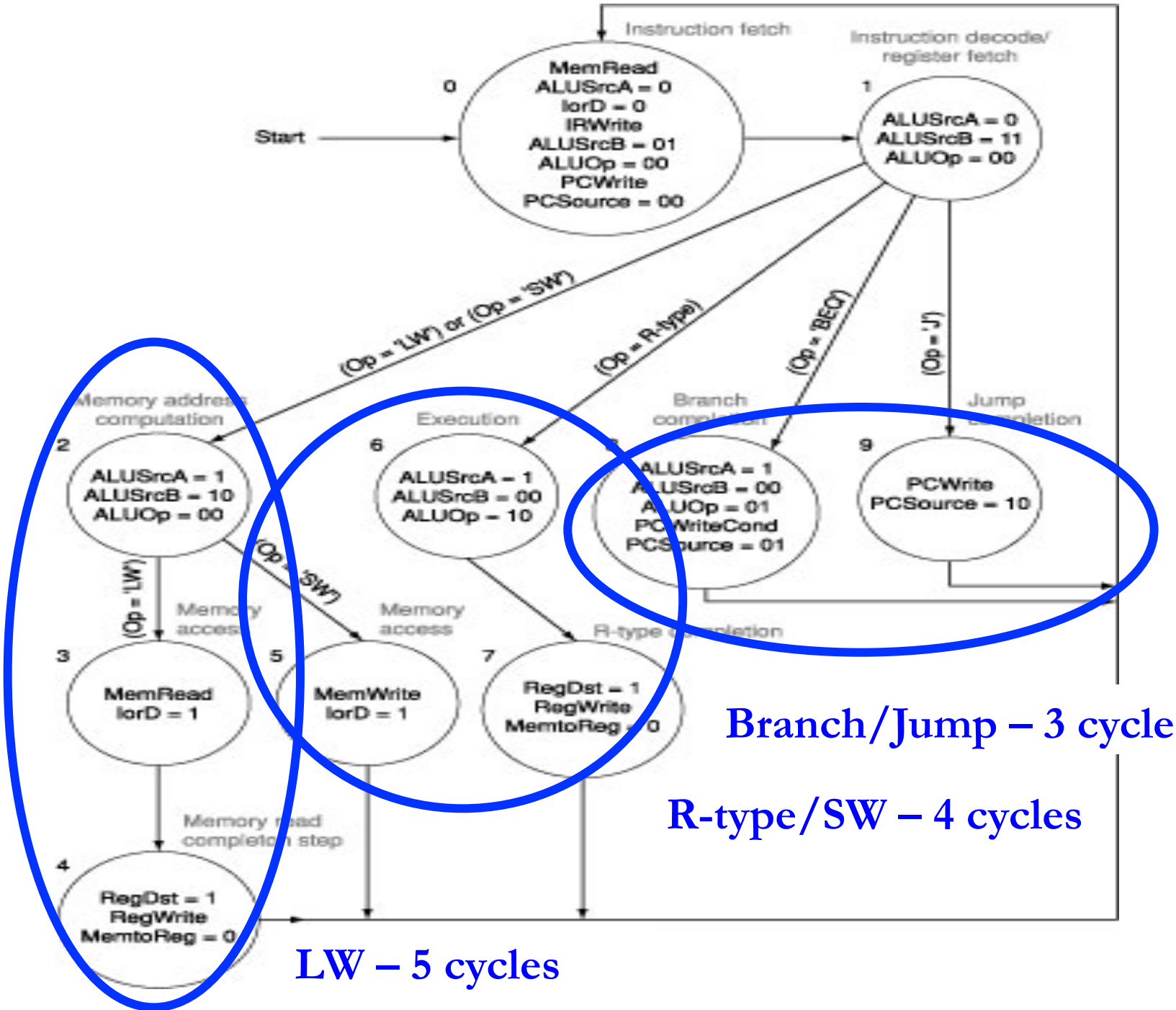


Jump Resolution States

From State 1 (Decode)



To State 0 (Fetch)



Branch/Jump – 3 cycles

R-type/SW – 4 cycles

LW – 5 cycles

Inf2C - Computer Systems
Lecture 11
Memory Hierarchy and Caches

Boris Grot

School of Informatics

University of Edinburgh

Coursework 2

- Will cover caches
- Assigned this Thursday, 12 Nov
  - Due in 2 weeks

Memory examples

Technology	Typical access time	$ per GB
SRAM	1-10 ns	>>$1000
DRAM	~100 ns	$10
Flash SSD	~100 µs	$1
Magnetic disk	~10 ms	$0.1

Which of these is "main memory"? **DRAM**

Memory requirements

- Programmers wish for memory to be
  - Large
  - Fast
  - Random access
- Wish not achievable with 1 kind of memory
  - Issues of cost and technical feasibility
- Idea of a **memory hierarchy**: approximate the "ideal" large+fast memory through a combination of different kinds of memories

Memory hierarchy overview

- Use combination of memory kinds
  - Smaller amounts of expensive but fast memory closer to the processor
  - Larger amounts of cheaper but slower memory farther from the processor
- Idea is not new:

"Ideally one would desire an indefinitely large memory capacity such that any particular … word would be immediately available… we are … forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible."

A. W. Burks, H. H. Goldstine, and J. von Neumann - 1946

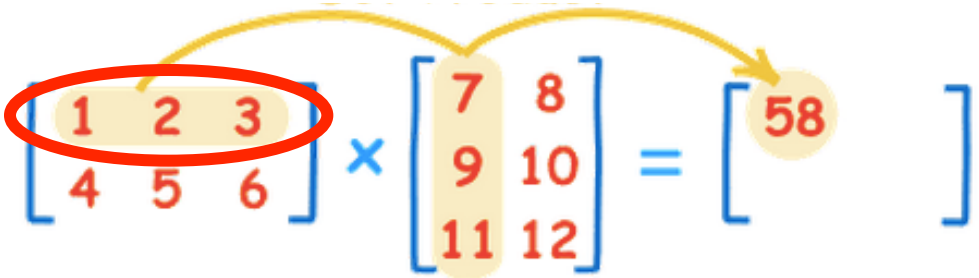
Why is a memory hierarchy effective?

- Temporal Locality:
  - A recently accessed memory location (instruction or data) is likely to be accessed again in the near future
- Spatial Locality:
  - Memory locations (instructions or data) close to a recently accessed location are likely to be accessed in the near future
- Why does locality exist in programs?
  - Instruction reuse: loops, functions
  - Data working sets: arrays, temporary variables, objects

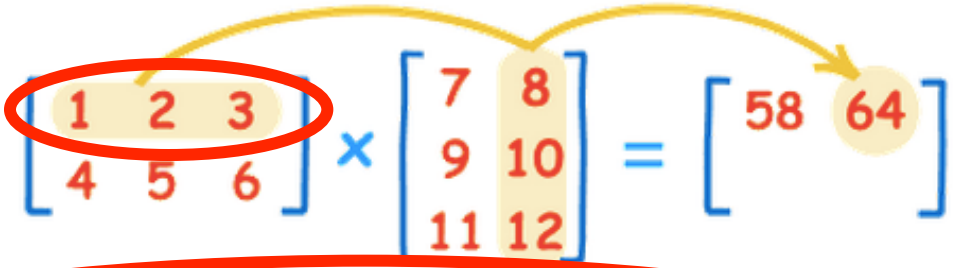
Example of Temporal & Spatial Locality

Matrix – matrix multiplication:

Spatial locality



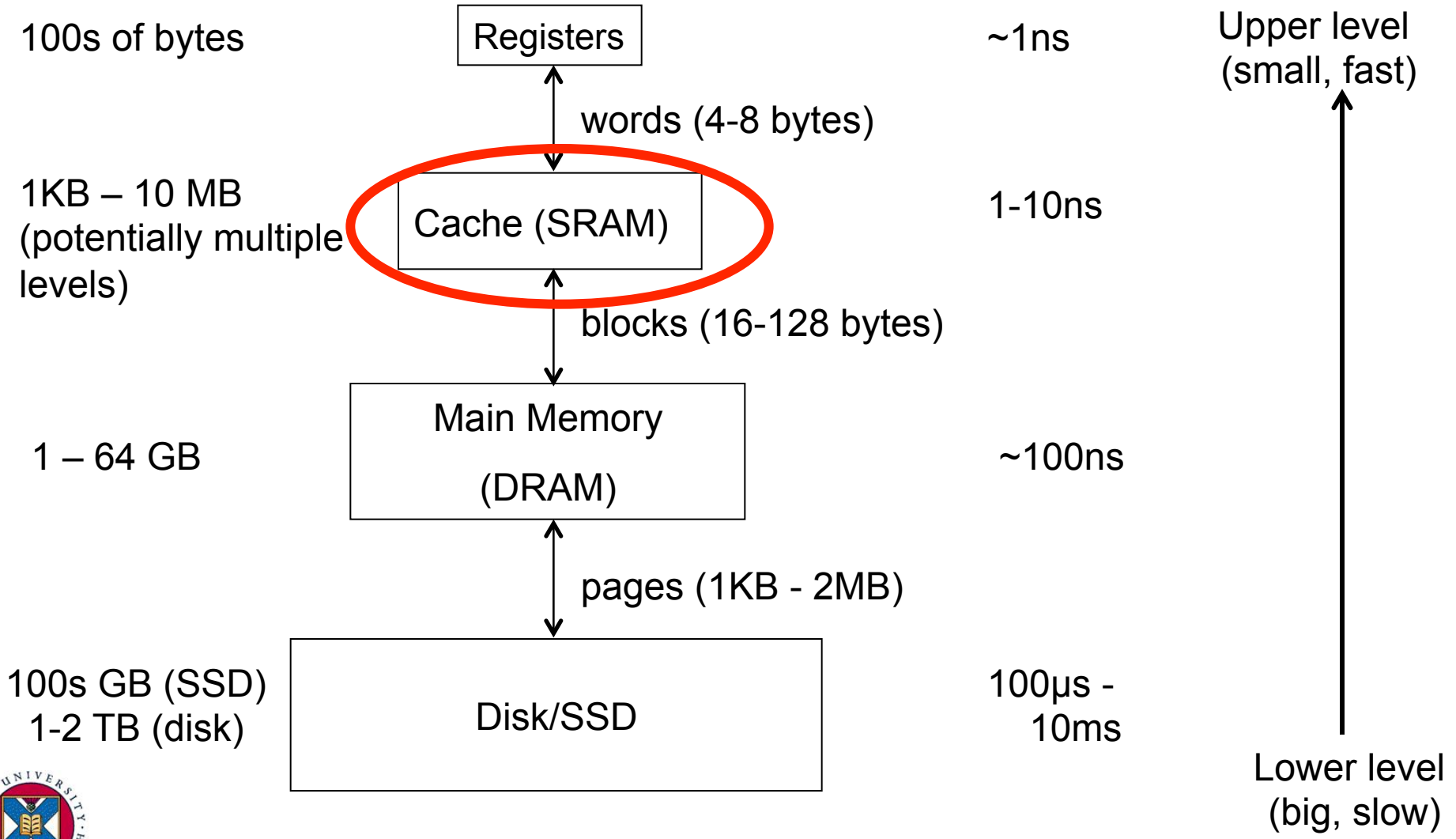
Temporal locality




```
for i = 1 to M
  for j = 1 to N
    for k = 1 to P
      c[i,j] = c[i,j] + a[i,k] * b[k,j]
```

Temporal & spatial locality in the code itself

Levels of the memory hierarchy



Memory hierarchy in a modern processor

- Small, fast cache backed up by larger, slower cache(s) and memories give the impression of a single, large, fast memory
- Take advantage of temporal locality
  - If access data from slower memory, move it to faster memory
  - If data in faster memory unused recently, move it to slower memory
- Take advantage of spatial locality
  - If need to move a word from slower to faster memory, move adjacent words at same time

Control of data transfers in hierarchy

- Q. Should the SW or HW be responsible for moving data between levels of the memory hierarchy?

- A. It depends: there is a trade-off between ease of programming, complexity, and performance.
  - *SW (compiler):* between registers and main memory or cache
  - *HW:* between caches and main memory (SW is usually unaware of caches)
  - *SW (Operating System):* between main memory and disk

Control of data transfers in hierarchy

- Q. Should the programmer explicitly copy data between levels of memory hierarchy?
- A. It depends: there is a trade-off between ease of programming and performance.
  - *Yes:* between registers and caches/main memory
  - *No:* between caches and main memory
  - *Sometimes:* between main memory and disk
    - *No:* when use disk area as virtual memory
    - *Yes:* when read and write files

HW-managed transfers between levels

- Occurs between cache memory and main memory levels
- Programmer & processor both oblivious to where data resides
  - Just issue loads & stores to "memory"
- Cache Hardware manages transfers between levels
  - Data moved or copied between levels automatically in response to the program's memory accesses
  - Memory always has a copy of cached data, but data in the cache may be more recent
    - This creates interesting problems.
      Discussed in Computer Architecture and Parallel Architectures ☺

Memory hierarchy terminology

- Block (or line): the unit of data stored in the cache
  - Typically in the range of 32-128 bytes
- Hit: data is found (this is what we want to happen)
  - Memory access completes quickly
- Miss: data not found
  - Must continue the search at the next level down
  - After data is eventually located, it is copied to the memory level where the miss happened

More memory hierarchy terminology

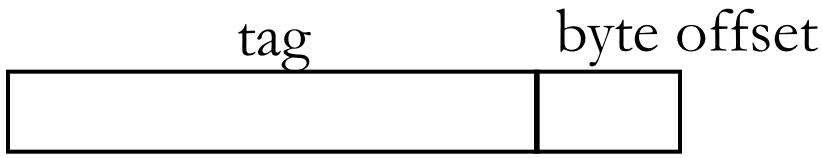
- Hit rate (hit ratio): fraction of accesses that are hits at a given level of the hierarchy
- Hit time: Time required to access a level of the hierarchy, including time to determine whether access is a hit or miss
- Miss rate (miss ratio): fraction of accesses that are misses at a given level (= 1 – hit rate)
- Miss penalty: Extra time required to fetch a block into some level from the next level down

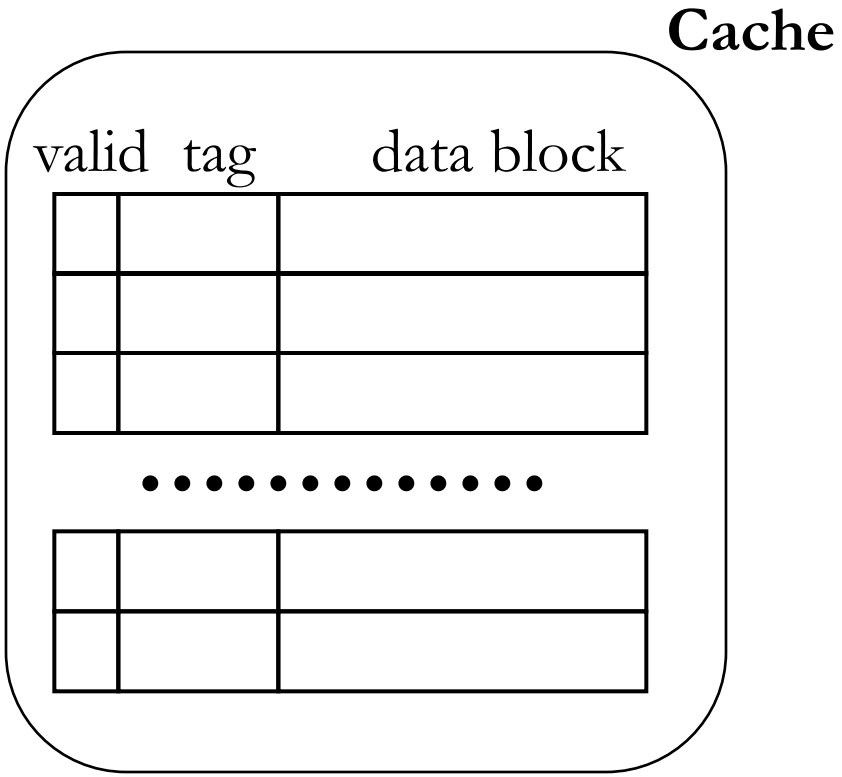
Cache basics

- Data are identified in (main) memory by their full 32-bit address
- Problem: how to map a 32-bit address to a much smaller memory, such as a cache?
- Answer: associate with each data block in cache:
  - a **tag** word, indicating the address of the main memory block it holds
  - a **valid bit**, indicating the block is in use

Fully-associative cache

requested address:





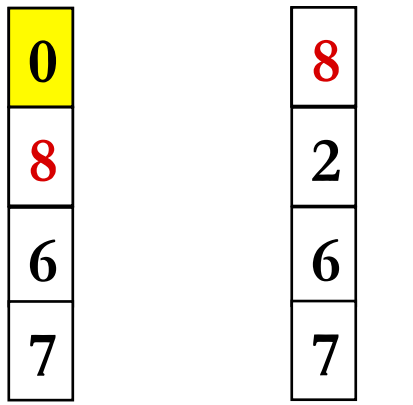
Correct cache block identified by matching tags

Byte offset selects word/byte within block

Address tag can potentially match tag of *any* cache block

Cache Replacement

- Least Recently Used (LRU)
  - Evict the cache block that hasn't been accessed longest
  - Relies on past behaviour as a predictor of the future
- FIFO – replace in same order as filled
  - Simpler to implement
- Example:
  - address references: 0 2 6 0 7 8
  - Cache with 4 blocks

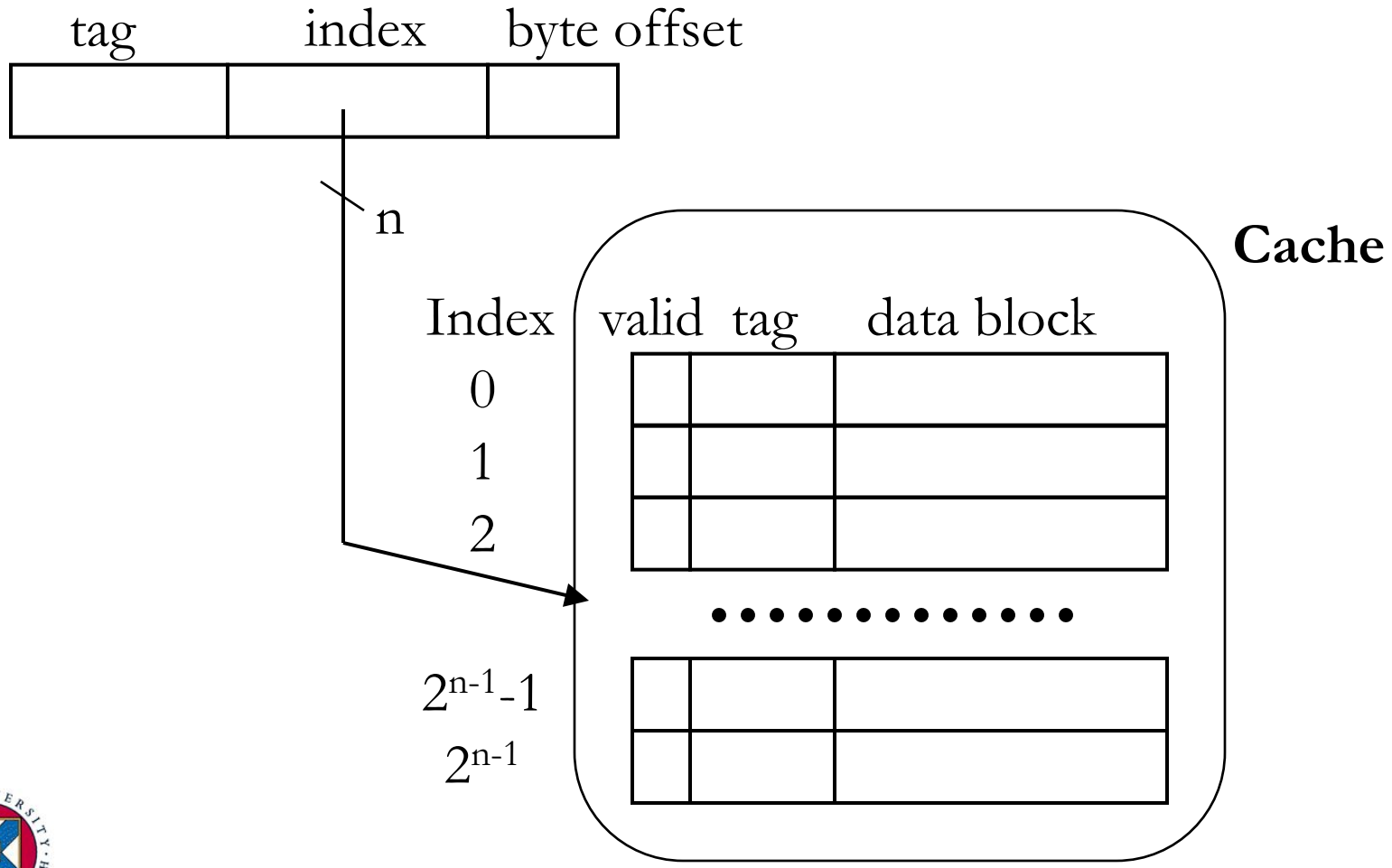


Direct-mapped cache

- In a fully-associative cache, search for matching tags is either very slow, or requires a very expensive memory type called Content Addressable Memory (CAM)
- By restricting the cache location where a data item can be stored, we can simplify the cache
- In a **direct-mapped** cache, a data item can be stored in one location only, determined by its address
  - Use some of the address bits as index to the cache array

Address mapping for direct-mapped cache

requested address:



Example problem

Given a 4 KB direct-mapped cache with 4-byte blocks and 32-bit addresses.

Question: How many tag, index, and offset bits does the address decompose into?

Example problem

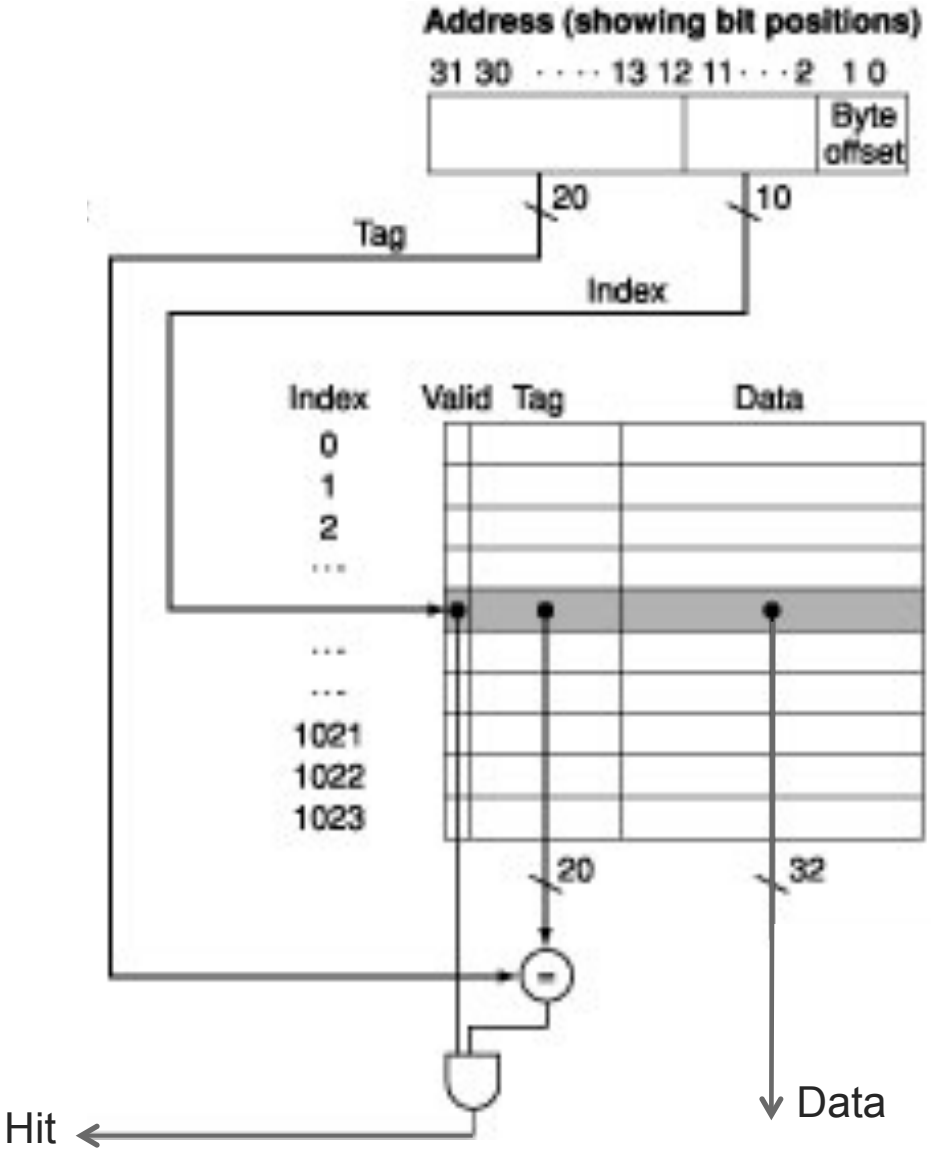
Given a 4 KB direct-mapped cache with 4-byte blocks and 32-bit addresses.

Question: How many tag, index, and offset bits does the address decompose into?

Answer:

- 4 KB / 4 bytes per block = 1K blocks
  - Requires a 10-bit index
- 4-byte block: requires a 2-bit offset
- Tag: 32 – 10 – 2 = 20 bits

Direct-mapped cache in detail



Inf2C - Computer Systems
Lecture 14
Virtual Memory

Boris Grot

School of Informatics

University of Edinburgh

Previous lecture: Memory hierarchy

- Main idea: exploit locality in memory references to create the illusion of a fast & large memory
  - Temporal vs spatial locality
- Memory hierarchy levels: registers, cache (≥1 levels), main memory, disk
- Cache: hardware-managed storage
  - Exploits temporal & spatial locality
  - Fully-associative vs direct mapped

Lecture 14: Virtual memory

- Motivation
- Overview
- Address translation
- Page replacement
- Fast translation – TLB

Motivation

Virtual memory addresses two main problems:

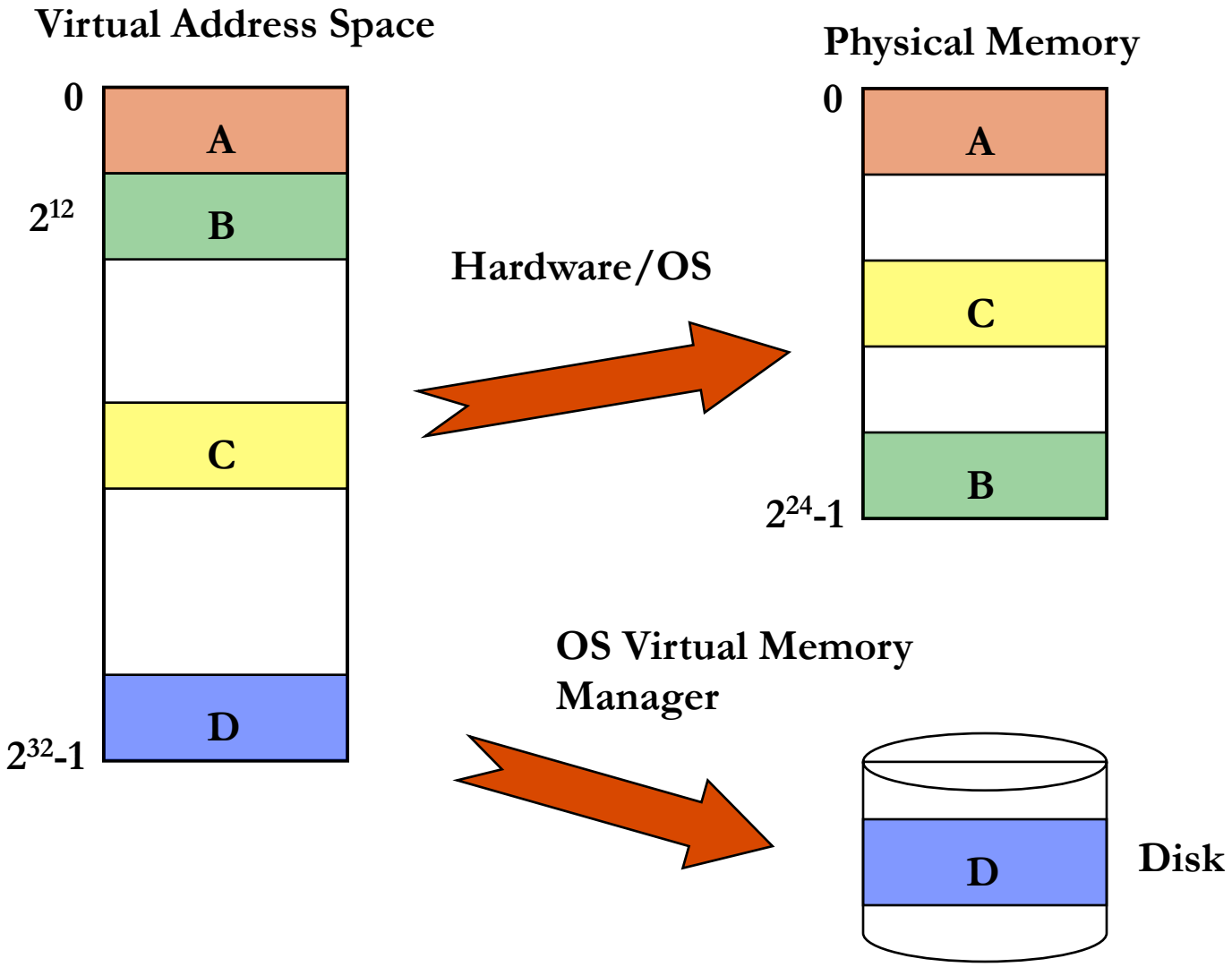
1) Capacity: how do we remove burden of programmers dealing with limited main memory?
   - Want to allow for the physical memory to be smaller than the program's address space (e.g., 32 bits → 4GB)
   - Want to allow multiple programs to share the limited physical memory with no human intervention

2) Safety: how do we allow for safe and efficient sharing of memory among multiple programs?
   - Want to prevent user programs from accessing the memory used by the OS
   - Want strict control of access by each user program to memory of other user programs

Virtual Memory

- Basic idea: each program thinks it owns the entire memory → the virtual address space
  - PC and load/store addresses are virtual addresses
- Actual main memory: physical address space
  - Virtual addresses are translated on-the-fly to physical addresses
  - Parts of virtual address space not recently used are stored on disk
- Dynamic address translation is done by combination of hardware and the OS

Address translation for 1 process



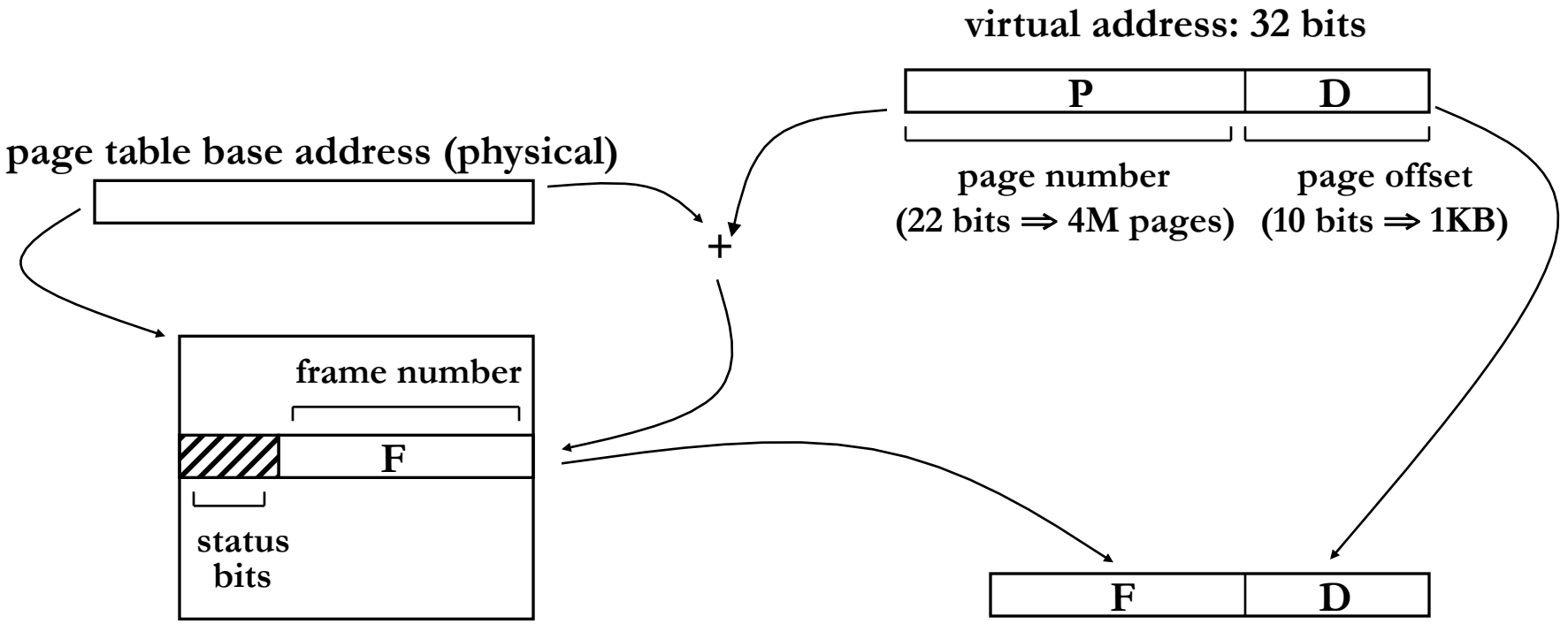
Physical memory as cache for VM

- Virtual memory space can be larger than physical memory
  - Programmer always sees the full address space (MIPS: 2^{32} bytes)
- Physical memory used as a cache for the virtual memory
  - Physical memory holds the currently used portions of a process' code and data areas (exploits locality!)
- Secondary storage (usually disk) "backs" the physical memory
  - OS reserves a portion of the disk for swap space
  - OS swaps portions of each process' code and data areas in and out of physical memory on demand (process called paging)
  - Swapping is transparent to the programmer

Paging

- A "cache line" or "block" of VM is called a page
  - Plain "page" or "virtual page" for virtual memory
  - "Page frame" or "physical page" for physical memory
- Typical sizes are 4-8 KB (MB or GB in servers)
  - Large enough for efficient disk use and to keep translation tables small
- Mapping is done through a per-process page table
  - Allows control of which pages each process can access
  - Different processes can use same virtual addresses

Dynamic Address Translation



page table:
- per process
- one entry per page (e.g. 4M entries)
- located in the system portion of main memory

physical address: 30 bits
(1GB of main memory)

Moving pages to/from memory

- Access to a non-allocated page causes a page-fault which invokes the OS through the interrupt mechanism
  - **R**(esidence) bit in page table status bits is zero
- Pages are allocated on demand
- Pages are replaced and swapped to disk when system runs out of free page frames
  - Aim to replace pages not recently used (principle of locality). **A**(ccess) bit for a page is set whenever page is accessed and is reset periodically
  - If any data in page has been modified, the page must be written back to disk: **M**(odified) bit in status bits is set

Providing Protection

- Each page table entry can have permission bits controlling
  - access allowed or not
  - read & write, read only or execute only access
- This enables per-process memory protection
  - E.g. can set up private and shared areas
- Important that only OS can change page tables
  - One of motivations for having kernel & user modes

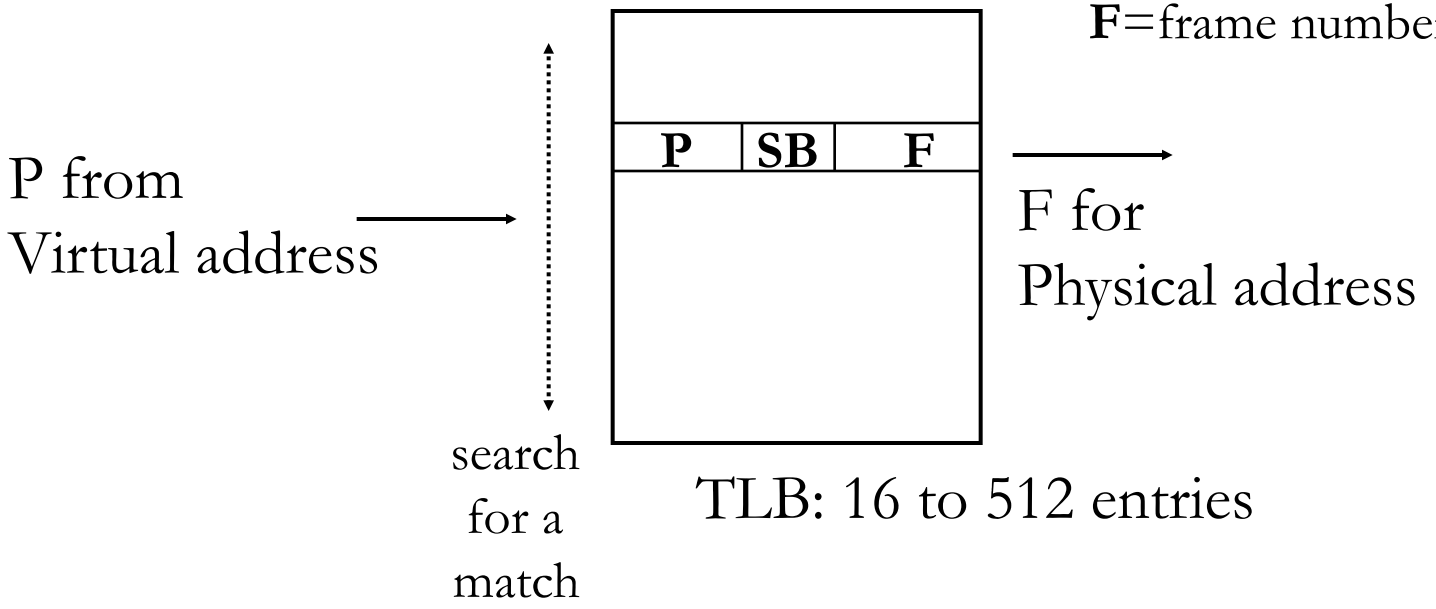
Translation Lookaside Buffer

- Accessing the page table is costly in terms of latency
  - Two memory accesses per load and store (1 to get the page table entry + 1 to get the data)
- Fast address translation: Translation Lookaside Buffer (TLB) contained in the MMU
  - Small and fast table in hardware, located close to processor
  - Is a cache for page table: holds frame addr., not program data
    - Tag: virtual address. Entry: physical frame address
  - Can capture most translations due to principle of locality
  - When page not in TLB: access the page table, and save the translation entry in TLB

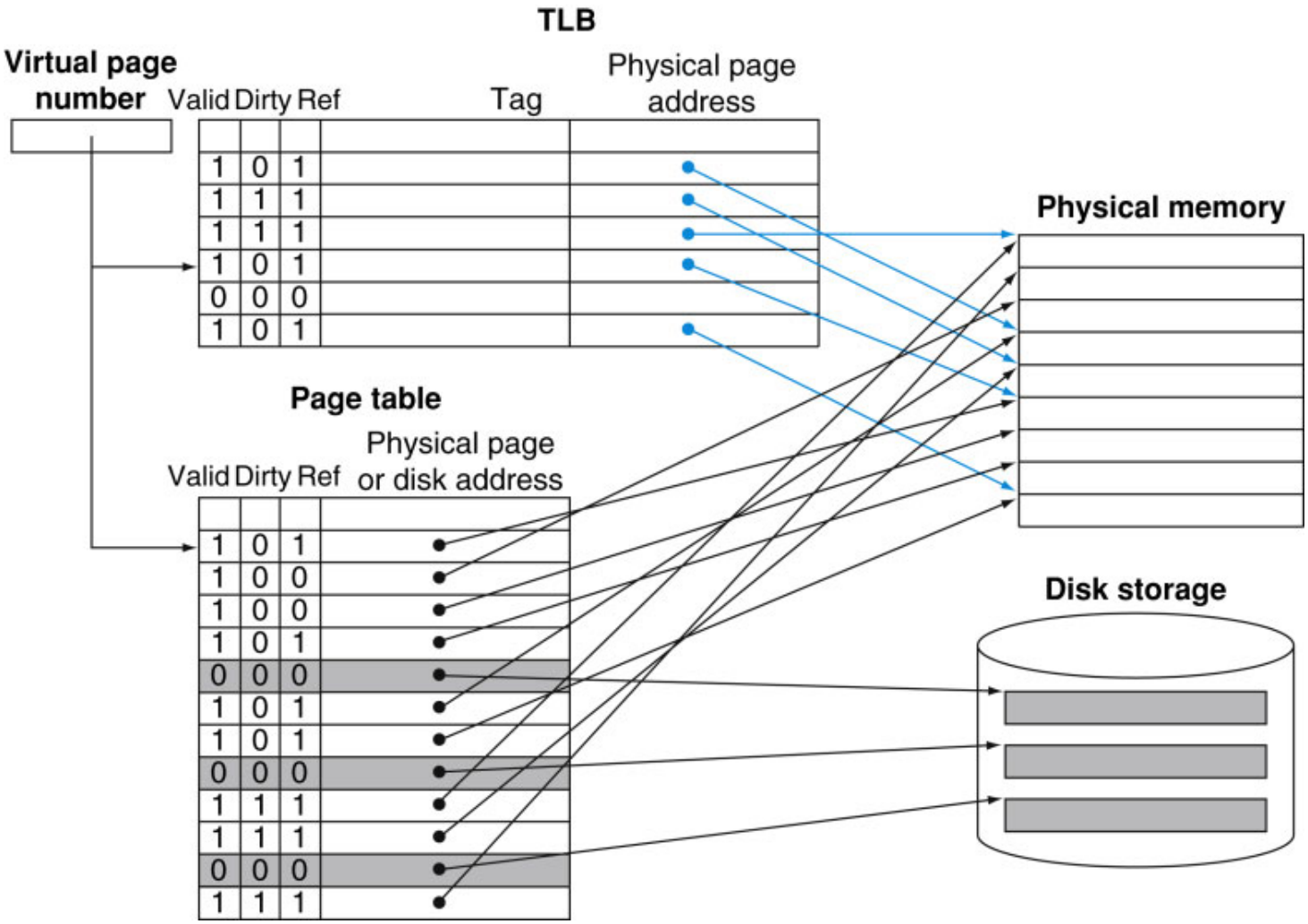
Translation Lookaside Buffer

Fully associative memory:

SB=status bits
P=page number
F=frame number



Virtual Memory: full picture



Inf2C - Computer Systems
Lecture 15
Exceptions and Processor Management

Boris Grot

School of Informatics

University of Edinburgh

Previous lecture: Virtual memory

- Solves two problems:
  - Capacity (physical memory is limited)
  - Safety (physical memory must be shared by multiple programs and the OS)
- Virtual vs physical address space
  - Each program "sees" a full 32-bit address space
  - Actual physical memory managed by the OS
- Address translation
  - Page table – all translations, but slow (in memory)
  - TLB – recent entries only, but fast (cache)

Exceptions – definition

- Exceptional events that interrupt normal program flow and require attention of the CPU
- External ("interrupts")
  - Not caused by program execution
  - E.g. I/O interrupt (e.g., network packet arrived)
- Internal ("traps")
  - Caused by program execution
  - E.g. illegal instruction, arithmetic overflow, TLB miss

Intentional exceptions

- Use exception mechanism to request some OS functions

  e.g., I/O (e.g., print to screen), memory allocation

- User program uses `syscall` instruction
  - Cause register ($v0) is set with a special value to identify the syscall exception
  - OS exception handler invoked when instruction executes
- Parameters are passed to the OS through agreed upon registers (usually $a0, $a1, ..)

Syscall example

The following will print the integer in register $t0 to the screen.

```
li  $v0, 1     # service 1 is "print integer"
add $a0, $t0, $zero  # load integer into $a0
syscall
```

Exception mechanism

- Step 1: Save the address of current instruction
  - into a special register, the exception program counter (EPC)
  - Note: must return to the interrupted instruction (not PC+4)
- Step 2: Transfer control to the OS at a known address (i.e., exception handler PC)
- Step 3: Handle the exception
  - Deal with the cause of the exception
  - All registers must be preserved, similar to a procedure call
- Step 4: Return to user program execution
  - Handler restores user program's registers and jumps back using EPC
  - Relies on special instruction **`eret`**

Finding the exception handler

- Approach 1:
  - Jump to a predefined address (0x800000180)
  - Use the Cause register to then branch to the right handler (e.g., print int, read string, exit program)
  - Works well for syscall – cause register explicitly set
- Approach 2
  - Directly jump to a specific handler depending on the exception (**vectored interrupt**)
  - Eg:

Undefined opcode:	0xC000 0000
Overflow:	0xC000 0020
…:	0xC000 0040

Handling the exception

- Determine action required
  - By inspecting the Cause register or by virtue of being at the right handler (e.g., undefined opcode)
- If restartable:
  - Take corrective action, then use EPC to return to program
- Otherwise:
  - Terminate program and report error using EPC, cause, …
- For a critical time while the interrupt is being handled, other interrupts should not happen
  - Otherwise the EPC, Cause will be overwritten
  - This is forced by masking interrupts → by setting the exception level (EXL) bit in the status register

Protecting system resources

- The OS must guarantee safe and orderly access to critical system resources
  - Hardware (processor, networking, I/O)
  - Program memory (including page tables)
- The OS is the ultimate arbiter of what's allowed
  - TLB miss → OK (but must access page table to service)
  - Write access to a read-only page → not OK (but must access page table to check)
  - Illegal opcode → not OK (kill the program)
- Exceptions are used to hand control over to the OS
  - Need a separate mechanism to limit capabilities of user programs

Kernel vs. User Mode Protection

- Exceptions (including system calls) are handled by the OS
  - CPU has two modes of operation: **user** and **kernel** (OS)
  - Current mode identified by a bit in a special status register
  - Exception mechanism is used to force the mode to change from user to kernel for execution of OS functions
- "Privileged" instructions only executed in kernel mode
  - E.g. accessing I/O devices, handling page tables accesses
- Kernel mode can only be entered through an exception
  - User programs cannot jump to OS instruction space
- `eret` instruction sets mode back to previous mode

Advantages of Dual Mode architecture

- Guarantees that control is transferred to OS when user programs attempt to perform potentially dangerous tasks
- Allows OS to ensure that programs do not interfere with each other
  - e.g., that memory is divided appropriately
- Allows OS to ensure that programs do not have access to resources for which they do not have permission
  - e.g., files
- Ensures that user programs do not have indefinite control of the processor
  - could happen in old-school Windows or ancient DOS

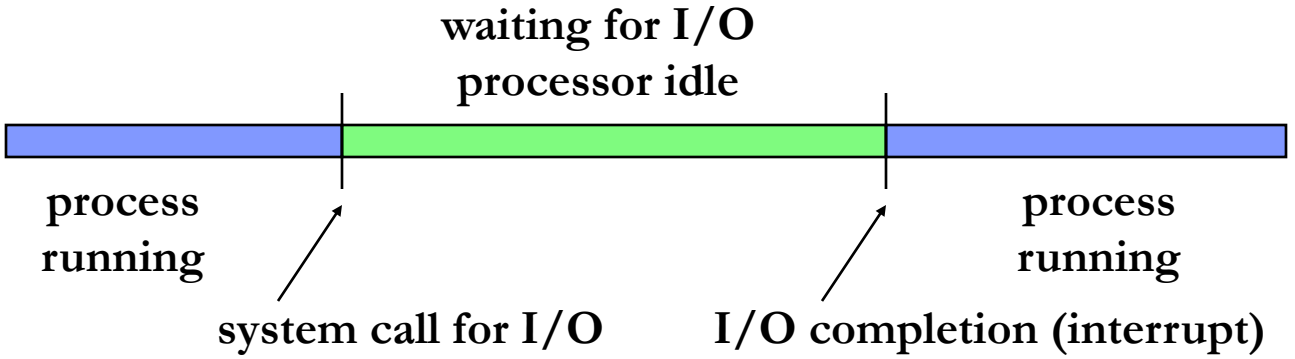
Managing the Processor

- Problem:
  - I/O takes too long → processor idle
  - User programs can crash or monopolize the CPU (either unintentionally or maliciously)
- Solution:
  - Multiplex or time-share the CPU and other resources among several user processes
  - Switch from one process to another when it performs I/O, or when it's time allocation (time slice) expires

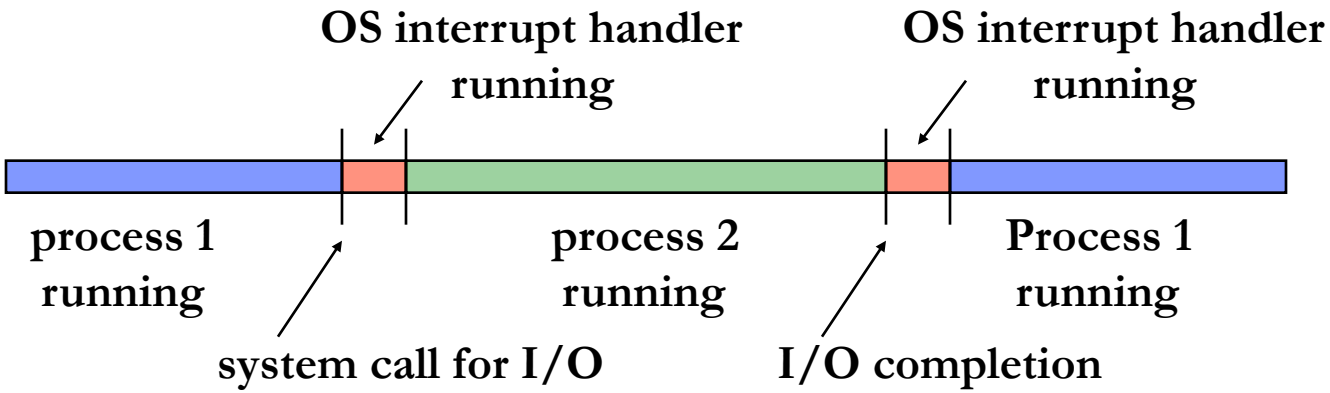
Process: "a program in execution" [Silberschatz, Galvin, Gagne]

Multi-tasking

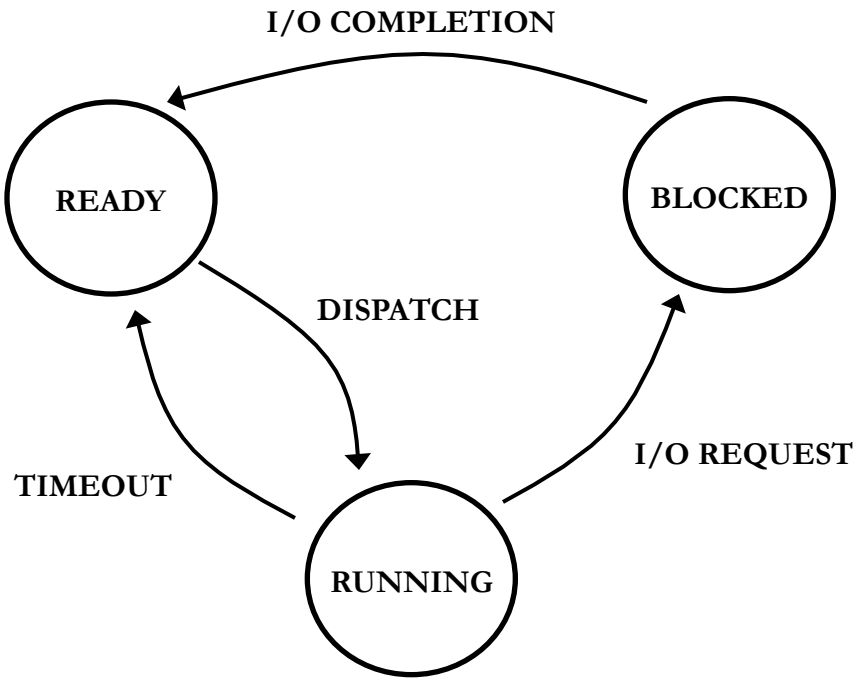
- Single-task system:



- Multi-tasking system:



Process States



States:

RUNNING: process is currently running in the CPU
READY: process is not running, but could run if brought into CPU
BLOCKED: process is not able to run because it is waiting for I/O to finish

Transitions:

I/O REQUEST: process initiates I/O
I/O COMPLETION: I/O finishes
DISPATCH: OS moves process into CPU and it starts executing
TIMEOUT: process's timeslice is over

Process States

- Step 1: process calls (or traps into) the OS, or interrupt occurs (e.g. because of timer)
- Step 2: OS's dispatcher performs context-switch:
  - Process's context is saved (registers, PC, etc) in process control block (PCB)
  - Dispatcher chooses new process to run
  - Processes' states are updated

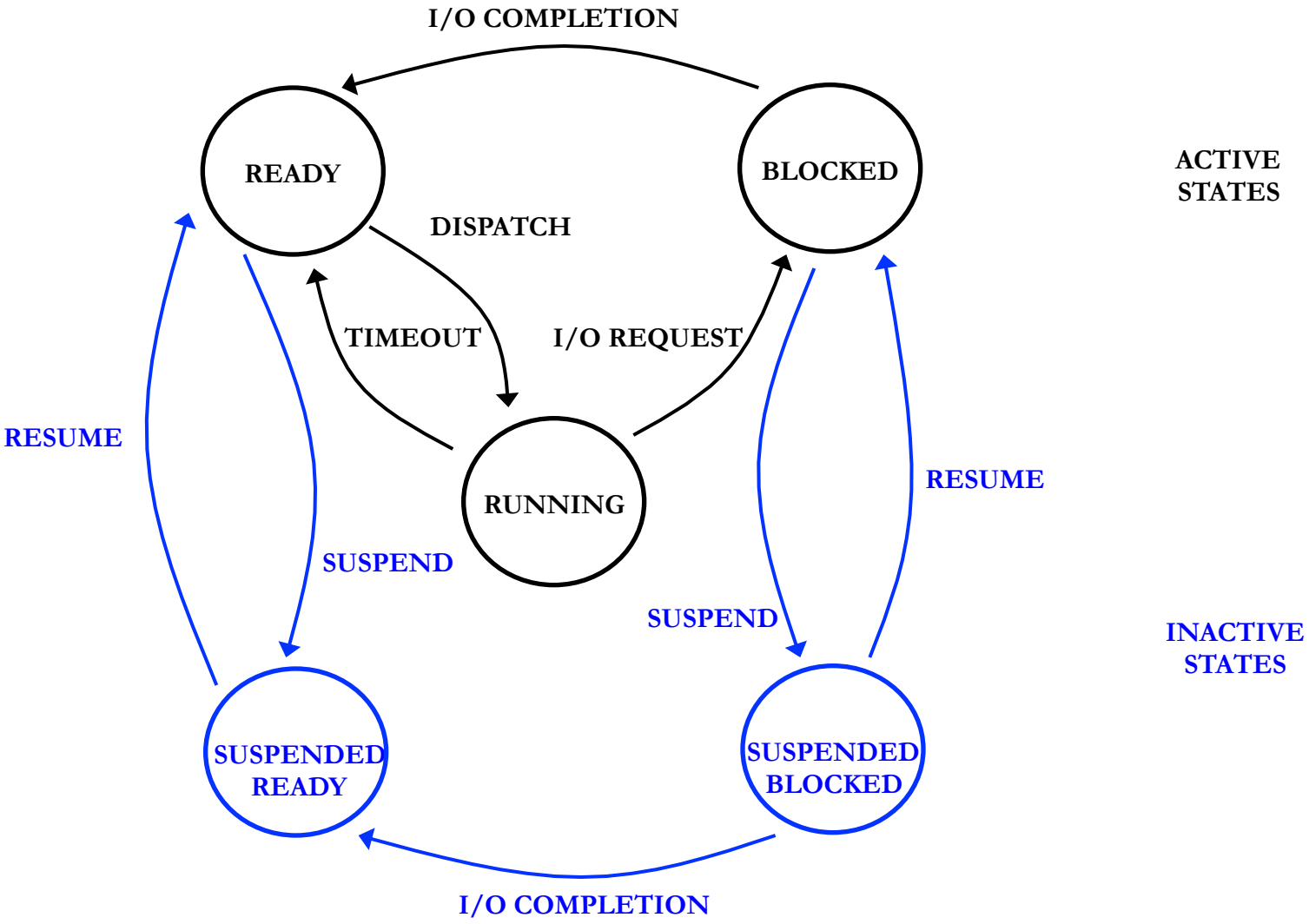
PCB: OS data structure containing each process's information:

- Process id (PID)
- Process state (blocked, running, etc)
- Process priority
- Process permissions
- Etc

Suspending and Resuming Processes

- Problem:
  - Might not have enough physical memory for all processses
  - Some processes have higher priority and must get more processor time (e.g., media playback)
- Solution:
  - Processes can be "swapped out" from memory to disk
  - Such processes are moved into an "inactive" state
    - 2 new process states
  - PCB of inactive processes are still kept in OS memory
  - Inactive processes are resumed by "swapping in" the data from disk back to memory

Suspending and Resuming Processes



Inf2C - Computer Systems
Lecture 14
I/O

Boris Grot

School of Informatics

University of Edinburgh

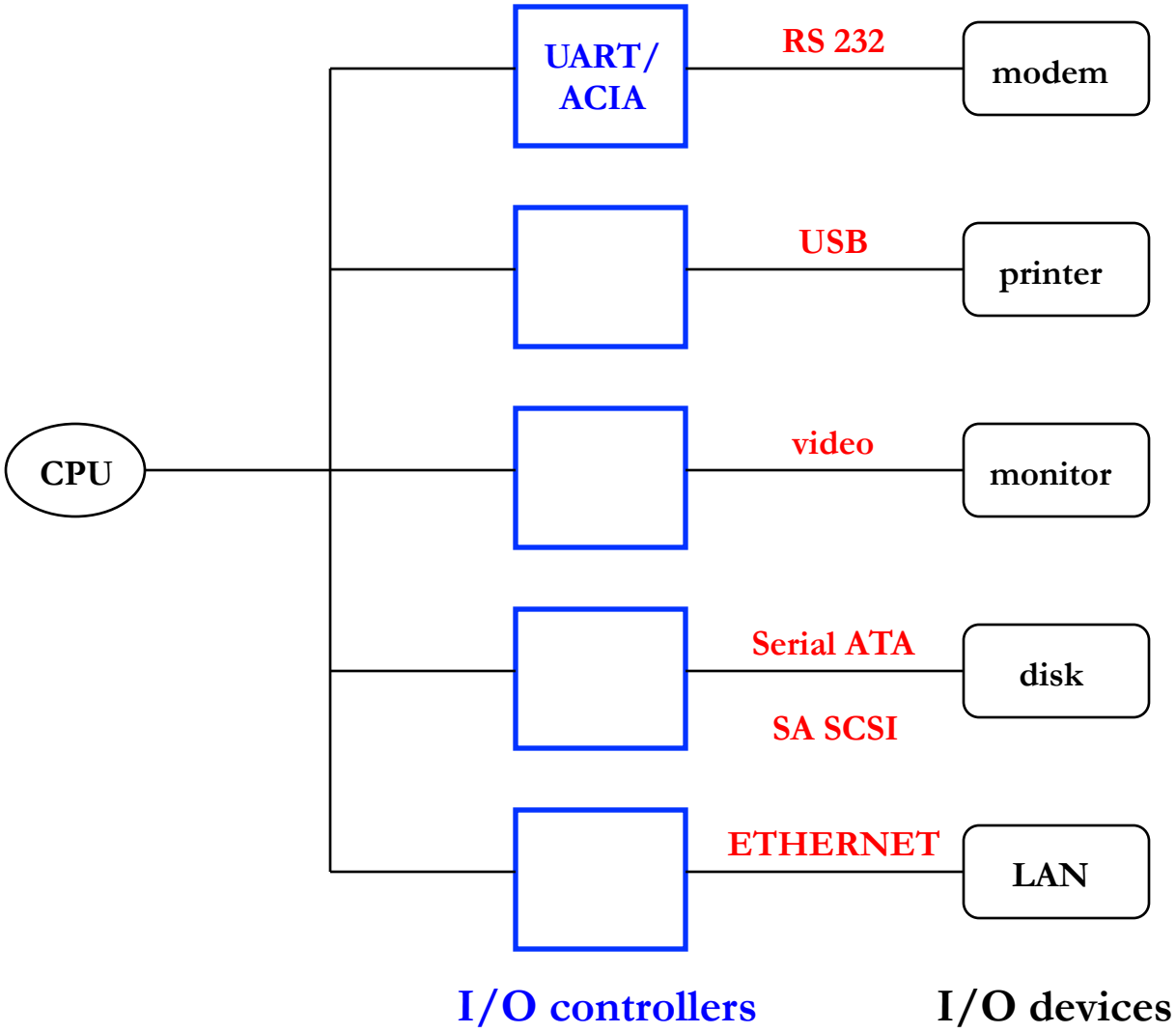
Previous lecture: Exceptions & processor mgmt

- Exceptions: interrupt normal program flow and require servicing by the CPU
  - Internal "traps" (e.g., syscall)
  - external "interrupts" (e.g., keyboard click)
- Exception handling
- Processor protection modes (user vs kernel)
- Processor management via time-sharing

Examples of I/O Devices

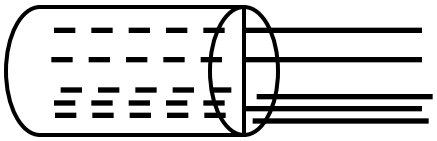
Device	Behaviour	Partner	Data Rate (Mbit/sec)
Keyboard	Input	Human	0.001
Mouse	Input	Human	0.004
Voice input	Input	Human	0.26
Laser printer	Output	Human	3.2
Graphics	Output	Human	800-8000
Magnetic disk	Storage	Machine	800-3000
Network/LAN	Input or output	Machine	100-40,000 (40Gbit/sec)

Lecture 14: I/O Controllers & Devices



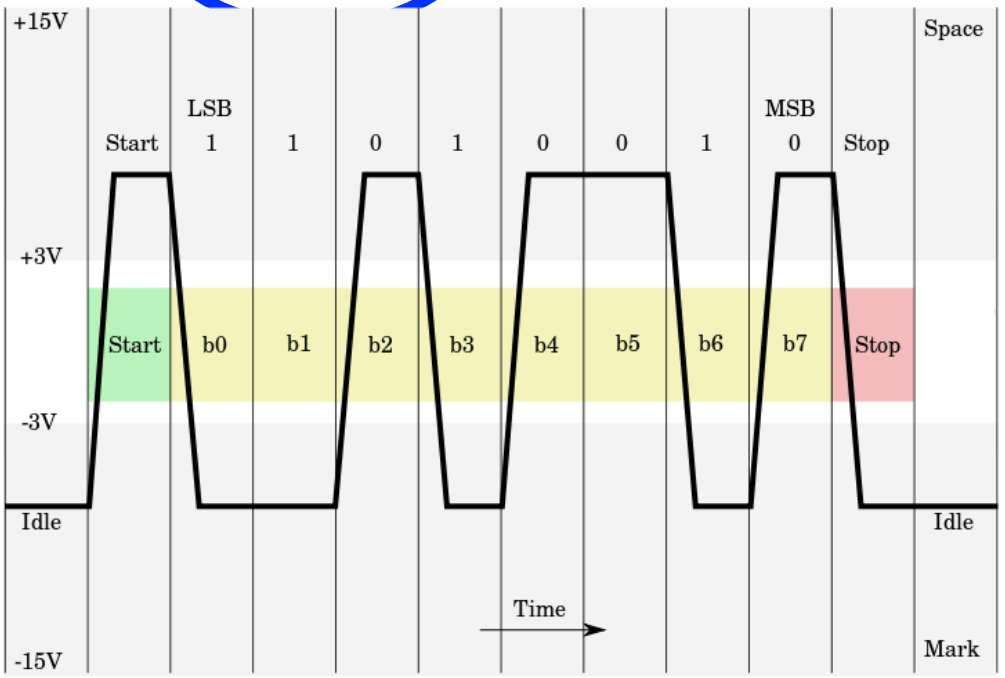
Example: RS232 Serial Interface

- I/O controller: UART (Universal Asynchronous Receiver Transmitter)
  - ACIA (Asynchronous Communications Interface Adapter) also used
- Used for modems and other serial devices
- Physical Implementation:
  - 2 signal wires (one for each direction) + ground reference + status signals



Tx data wire (CPU to I/O device)
Rx data wire (I/O device to CPU)
GROUND wire, status signals

RS232 modem: bits and wires

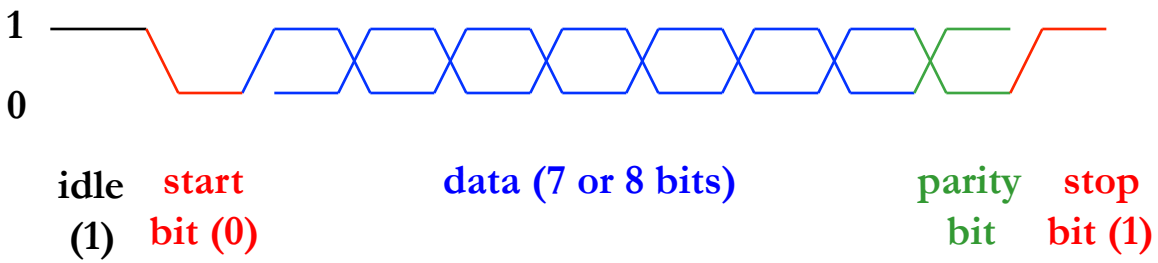


Oscilloscope trace of voltage levels for an ASCII "K" character (0x4B) with 1 start bit, 8 data bits, 1 stop bit.

Source: wikipedia

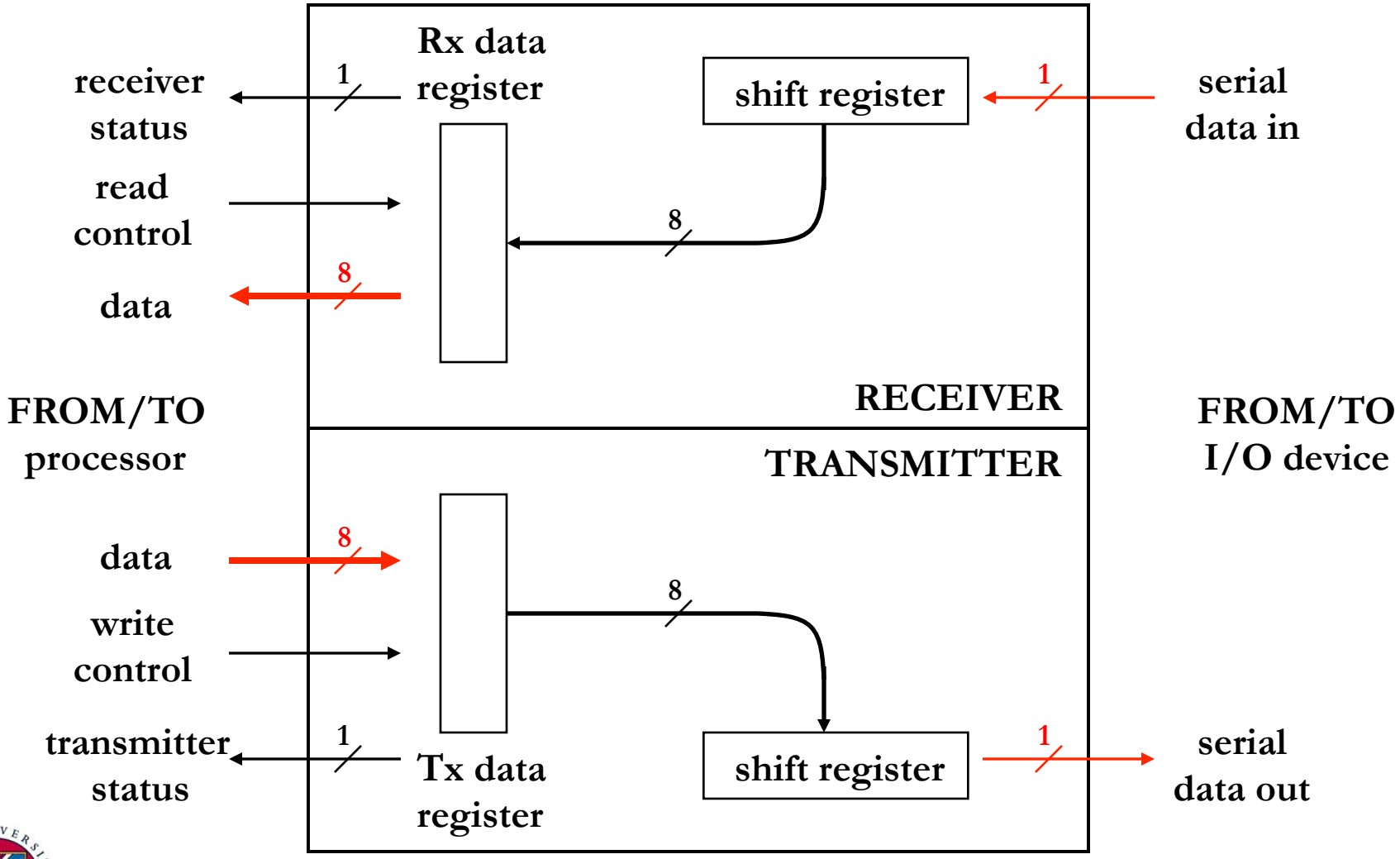
Example: RS232 Serial Interface

- Encoding:
  - 1 character = 10 or 11 bits (including signaling)
  - Idle state is represented by a constant "1"



- Parity: for detection of transmission errors
  - odd → total number of "1"s (including parity bit) is odd
  - even → total number of "1"s is even

UART Controller



Connecting CPU and I/O Controllers

- Option 1: connect the I/O Tx and Rx registers directly into some special CPU I/O registers → not flexible

- Option 2: keep I/O registers in separate I/O controller and connect CPU to I/O controller through special I/O bus → expensive, not flexible

  I/O bus:

  - data lines (8 bits)
  - control lines (READ and WRITE signals),
  - address lines (some few bits) → each I/O controller is assigned a range of addresses for its registers

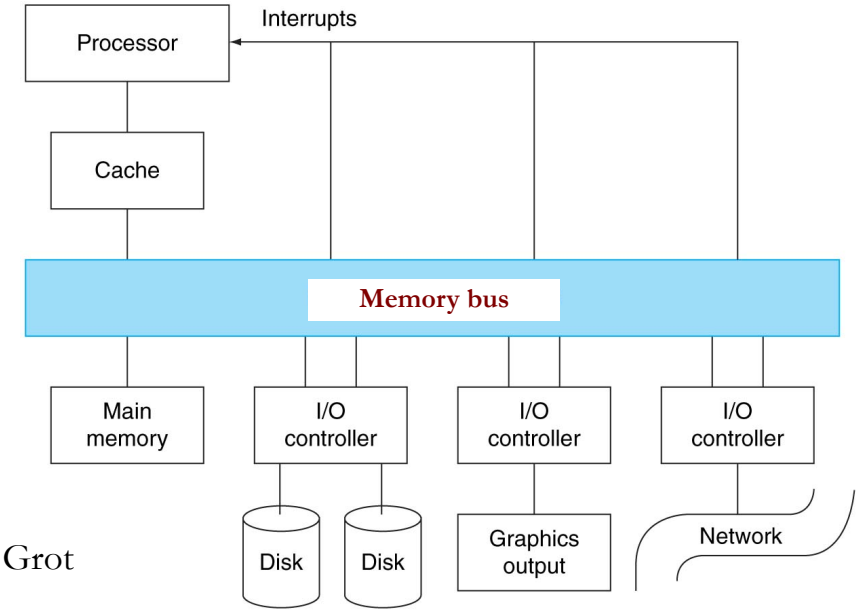
  Data is accessed through special I/O loads and stores

Connecting CPU and I/O Controllers

- Option 3: keep I/O registers in I/O controller and connect the CPU to I/O controller through memory bus

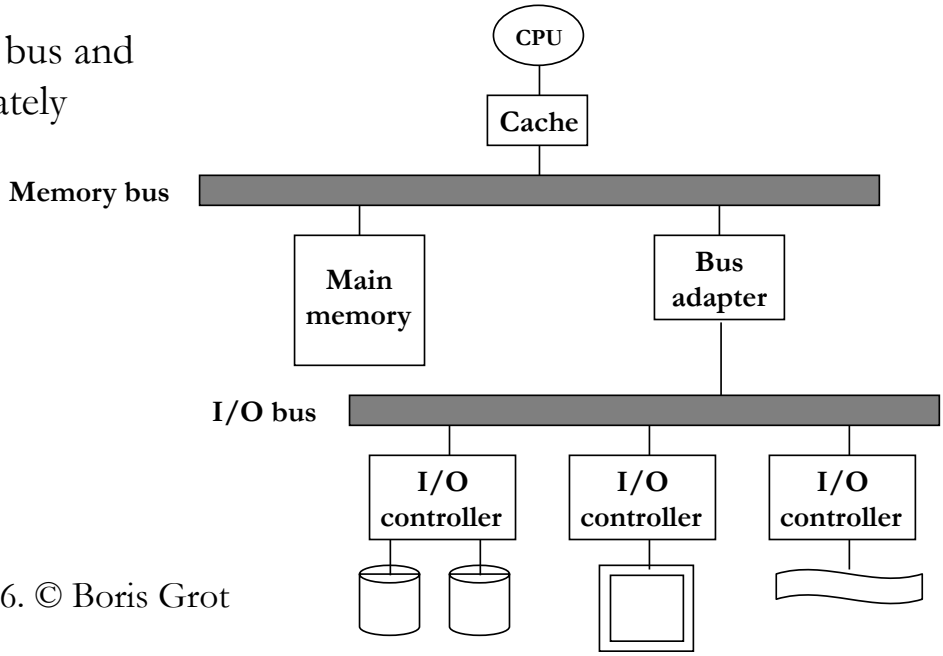
  Memory mapped I/O:

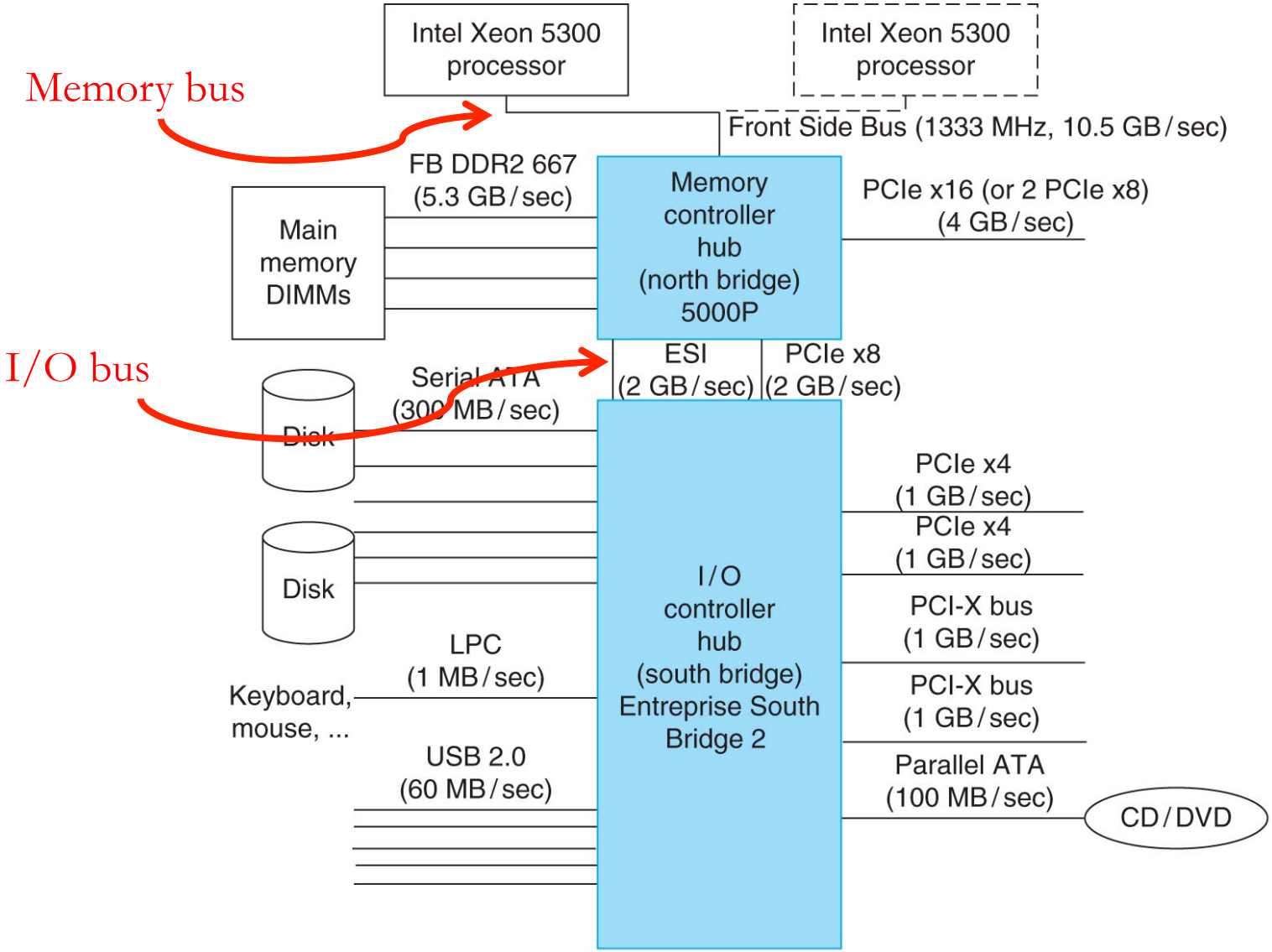
  - I/O controller registers (data and control) are mapped to a dedicated portion of memory
  - Good news: accessed with regular load and store instructions
  - Bad news: Takes bus bandwidth away from CPU-memory



Connecting CPU and I/O Controllers

- Option 4: connect I/O controllers to I/O bus and the I/O bus to the memory bus through a bus adapter
  - Off-load the I/O from the memory bus: multiple I/O devices appear as a single device to the memory bus)
  - Pros:
    - better performance: slow I/O bus doesn't clog up the fast memory bus
    - higher flexibility: add/remove devices without impacting the high-performance processor-memory interface
    - Modularity and extensibility: memory bus and I/O bus technology can evolve separately





Organization of the Memory & I/O system on an Intel server using the Intel 5000P chip set.

Fig. 6.9. Computer Organization & Design, 4th Edition.

Polling and interrupt-based I/O

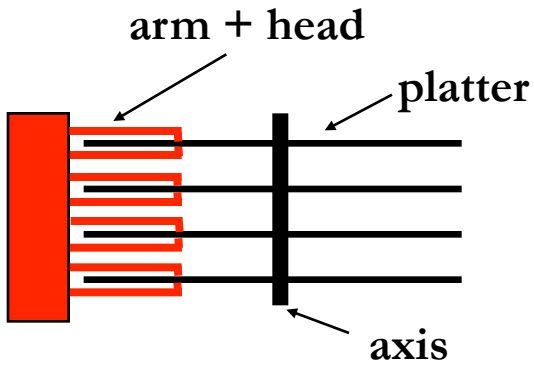
How do we check I/O status (e.g., key clicked)?

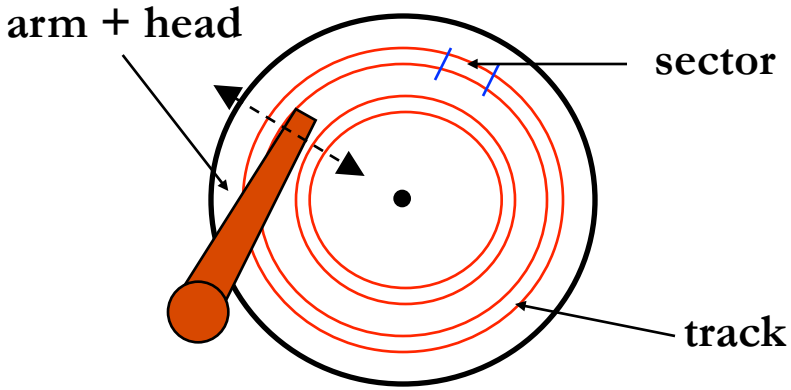
- Option 1: Polling
  - User process calls OS at regular intervals to check status of I/O operation
  - Wasteful for events that happen very infrequently (e.g., keyboard clicks)
- Option 2: Interrupt
  - I/O controller interrupts user process to signal an I/O event
  - Heavy-weight (break out of user code into kernel mode)

USB example to bring things together

- USB devices don't generate interrupts
  - Keeps cost low
- Computer can't afford to keep polling the multitude of USB ports
  - Would lose too much performance for nothing
  - Not polling often enough not an option: will lose data
- USB controller on the CPU side does the polling (simple FSM) and generates an interrupt to inform the CPU
  - This functionality is in the Southbridge in the picture on slide 13

Hard Disks





- 1-4 platters per drive
  2 surfaces per platter
  10-50k tracks per surface
  100-500 sectors per track
  512B – 4KB per sector
- Spinning speed:
  - 5400-15000 rpm
    (90-250 revs per sec)

Disk Performance

- Total time of a disk operation is divided in two parts:

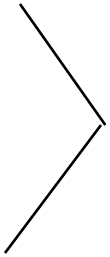
1. **Access time**: time to get head into position to read/write data
   access time = seek time + rotational latency
   - **Seek time**: time to move head to appropriate track (< 10ms)
   - **Rotational latency**: time to wait for appropriate sector to arrive underneath the head (< 10ms)
     Dependent on spinning speed

2. **Transfer time**: time to move data to/from disk
   transfer time = time to transfer 1 byte * number of bytes of data
   - Dependent on both spinning speed and recording density
   - 75-125 MB/s (changing very slowly – limited by mechanics)

Disk Controllers

- Disk controller inside disk unit → responsible for all mechanical operation of disk + interface with CPU
- I/O registers:
  - Exchange data and control between CPU and disk controller
  - Command register → tells disk controller what to do next

e.g. Seek n
Read Sector m
Write Sector m
Format Track



High-level commands

Using a Disk Controller

- Step 1: user program requests data from a file
- Step 2: OS file system determines sector(s) to be accessed
- Step 3: OS disk handler issues Seek command and CPU goes to work on some other process (multi-tasking)
- Step 4: I/O controller interrupts CPU to signal completion of seek
- Step 5: OS disk handler issues Read Sector command and CPU goes to work on some other process
- Step 6: I/O controller interrupts CPU to signal data ready
- Step 7: OS disk handler transfers data to/from disk
- Step 8: go to step 3 or 5 and repeat until all data transferred

Interrupt Approach

- CPU (through OS) has to issue individual commands to read every sector from disk
- Data transfer is very slow (~100μs for a 512 byte sector)
- Frequent interrupts detrimental to processor performance
  - Switch to OS, then switch back to the application (Lecture 13)

- Solution: Direct Memory Access (DMA)

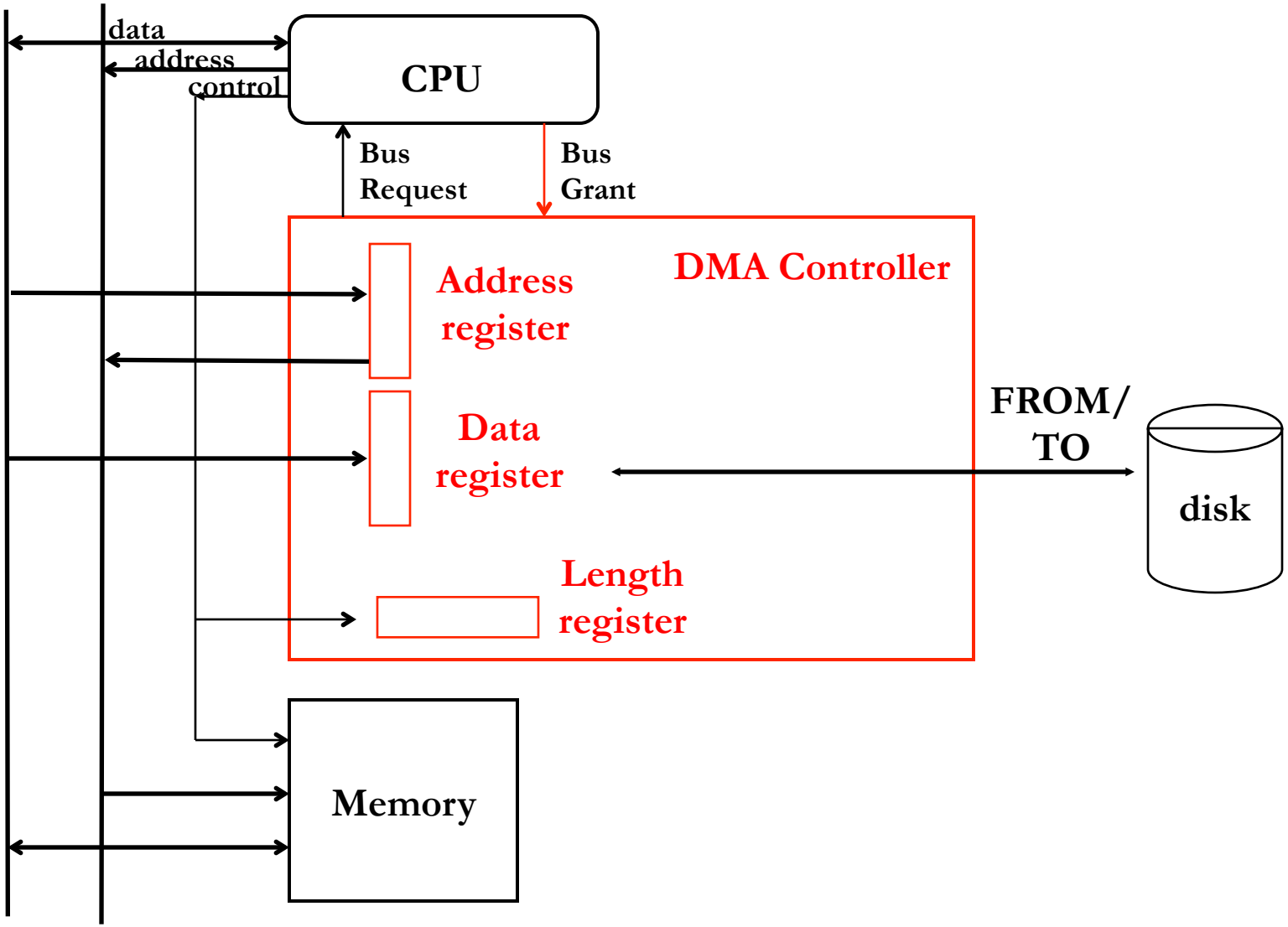
Direct Memory Access (DMA)

- DMA controller: stateful device that sits on the memory bus and can independently transfer data between memory and disk
  - Setup by the processor for each transfer using memory-mapped registers inside the DMA controller

- DMA registers:
  - Address register → position in memory of next data to be read/written
  - Data register → temporary storage for data to be transferred
  - Length register → number of bytes remaining to be transferred

- Technical write-up of how this works for x86: http://www.brokenthorn.com/Resources/OSDev21.html

DMA Organization



DMA Operation

- Step 1, 2: user program requests data, OS determines location of data on disk
- Step 3: OS disk handler issues Seek command and sets up DMA registers (address, length); CPU goes to work on another process
- Step 4, 5: I/O controller interrupts CPU, OS disk handler issues Read Sector command
- Step 6: I/O controller informs DMA controller that data is ready (no need to interrupt CPU)
- Step 7: DMA controller transfers data into memory; length register is decremented until all data is moved (advanced DMA controllers can access multiple tracks with a single operation)
- Step 8: DMA controller interrupts CPU to inform completion of DMA operation

Bus Arbitration

- DMA and CPU connect to memory bus → access must be somehow arbitrated to avoid conflicts
- Solution: additional logic (bus arbiter)
  - Authorizes CPU or DMA controller to access memory at any given time
- Two new wires on the memory bus:
  - Bus Request → asserted by the DMA controller when it requires the bus
  - Bus Grant → asserted by the CPU when it is not using the bus and thus the DMA controller can use it
  - In case of conflicting requests in the same cycle, CPU usually has priority