

Computer Architecture – An Introduction



CS2053 Computer Architecture

Computer Science & Engineering

University of Moratuwa

by

Dr. Sulochana Sooriyaarachchi &

Dr. Chathuranga Hettiarachchi

Acknowledgement: Dr. Dilum Bandara

Notice

- Lecture time: Friday 8.15 -10.15 am
- Lab time: Mon 1.15 – 3.15 pm
- Lab classes
 - Student grouping - 40 per group
 - Access
- Instructors:
 - Batch 19

From Outside



Source: techwench.com

From Outside (Cont.)



Source: Amazon.com

Touch pad
Touch screen
Wireless
Weight

Screen size
Battery capacity
Weight
Camera
Sensors



Source: Daniel Zanetti, wikimedia.org

From Inside

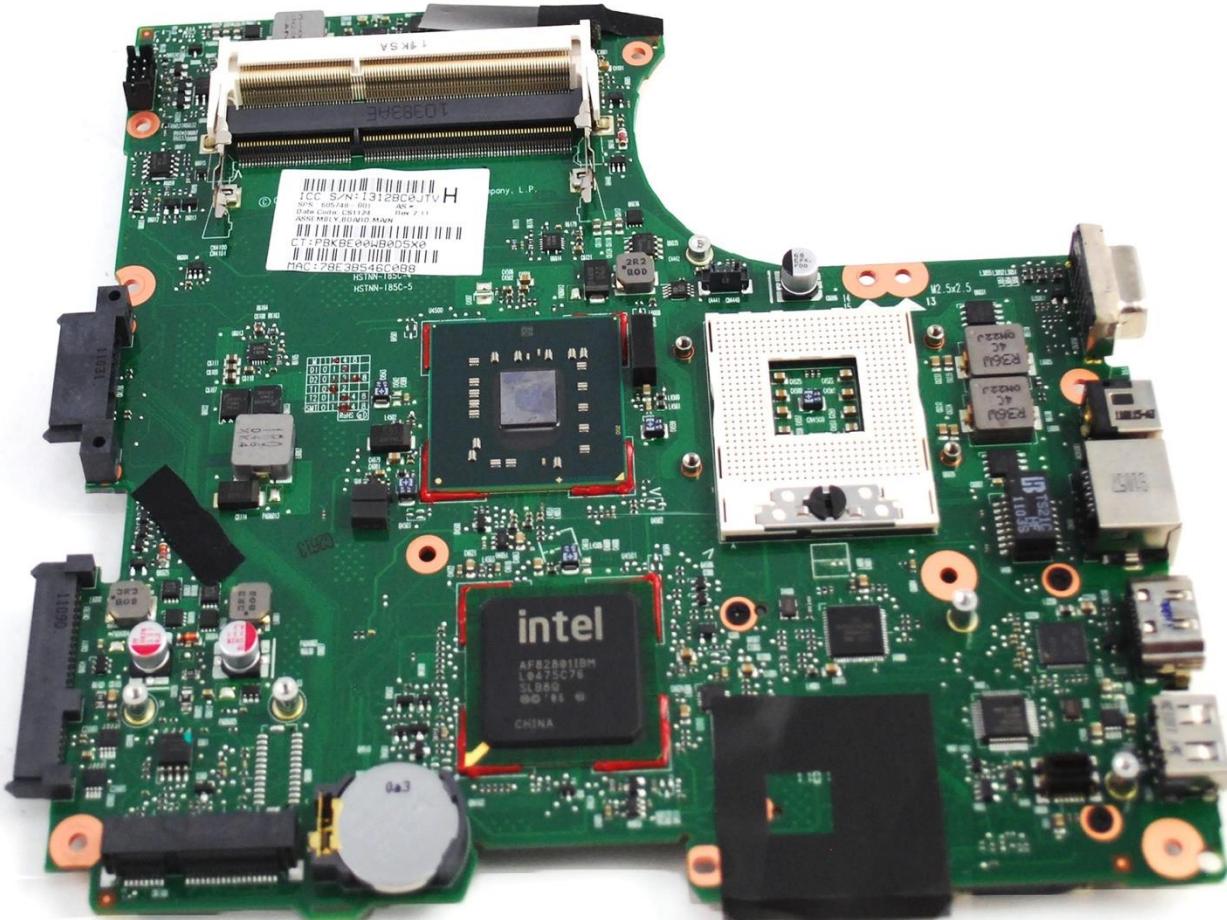


Source: <http://technologyuk.net>



Source: <http://rays-place.net>

From Inside (Cont.)

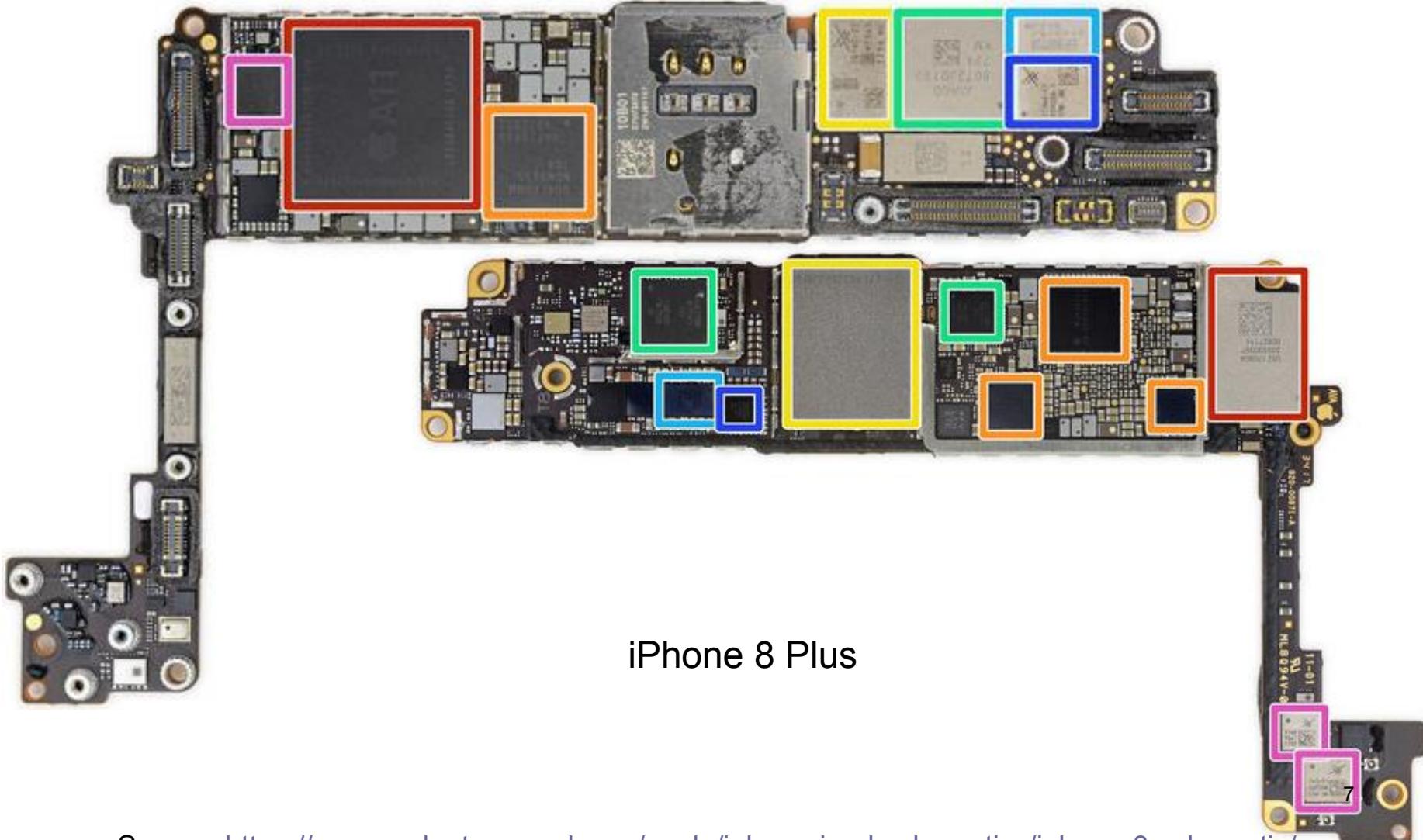


Source: www.laptopaid.com



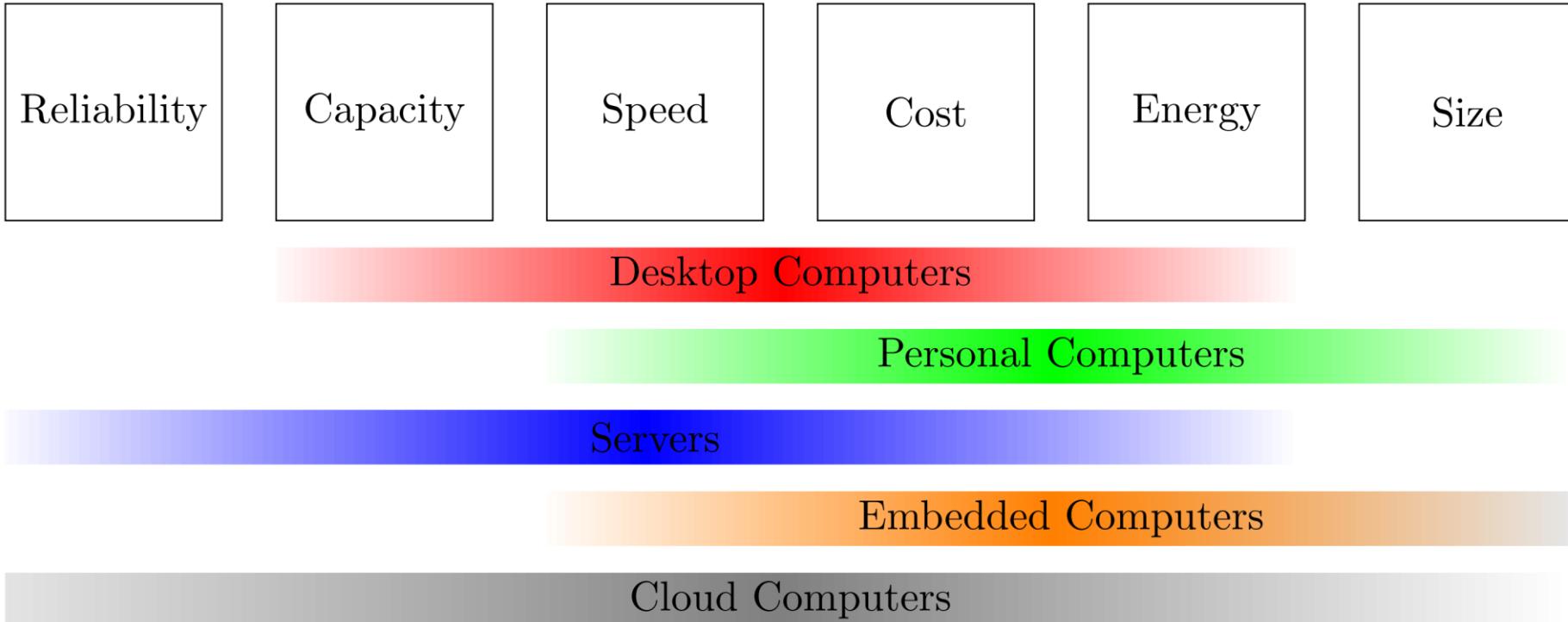
Source: <http://techgoesboom.com> 6

From Inside (Cont.)



Source: <https://www.gadget-manual.com/apple/iphone-ipad-schematics/iphone-8-schematic/>

Classes of Computers & Performance Metrics



Want to achieve these performance metrics?
Then you need to understand & design based on
principles of computer architecture

What We Are Going To Study?

- How these internal components look like?
 - Top-down approach with schematics
- How do they fit together?
- How to program them?
- How to benefit from performance enhancement options?
 - Focus on abstract views using schematic diagrams
 - Not on how those are built using semiconductors

Terminology

Computer Architecture

Blueprint/plan that is visible to programmer
Key functional units, their interconnection, & instruction to program
Instruction Set Architecture (ISA)
e.g., x86 vs. ARM

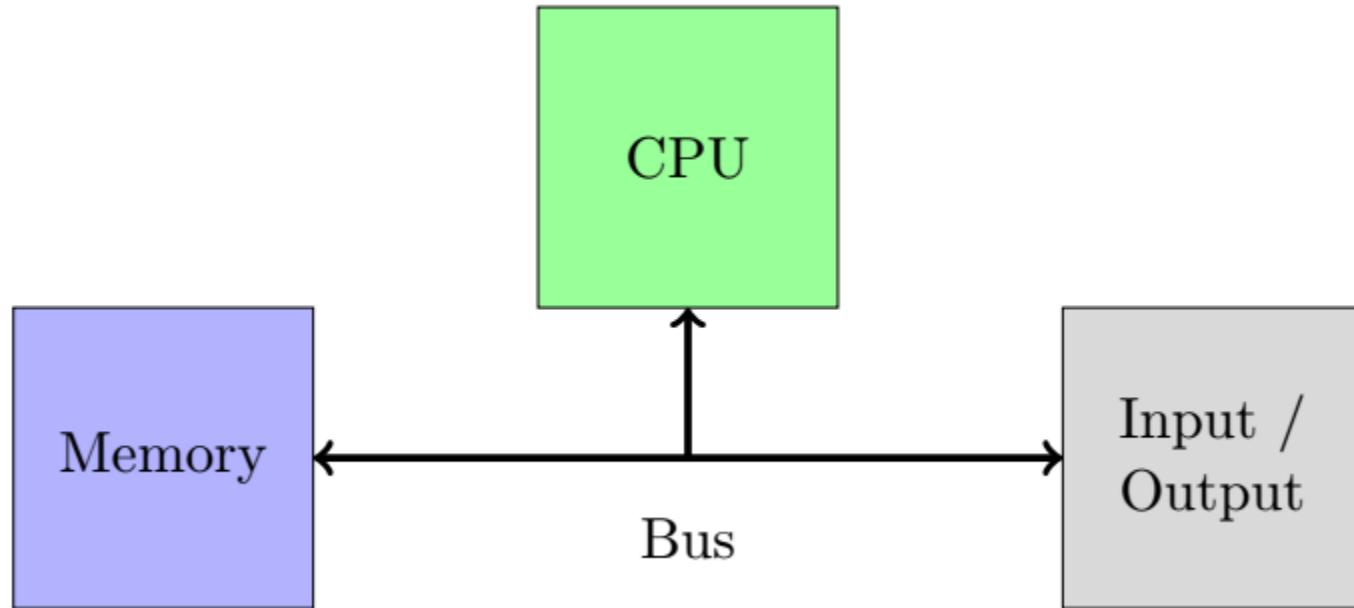
Computer Organization

Internal details of operational units, their interconnection, & control
View of a computer designer
How to support multiplication – multiply circuit or repeated addition
e.g., Intel & AMD both support x86 with different organizations

Computer Design

Maps a given organization to a logic design, logic design to a Silicon layout, & chip packaging
View of hardware designer
Design decisions based on constraints like circuit-level delays, Silicon real estate, heat generation, & cost
e.g., Intel Core i7-6800K vs. Xeon E5-2643 v4

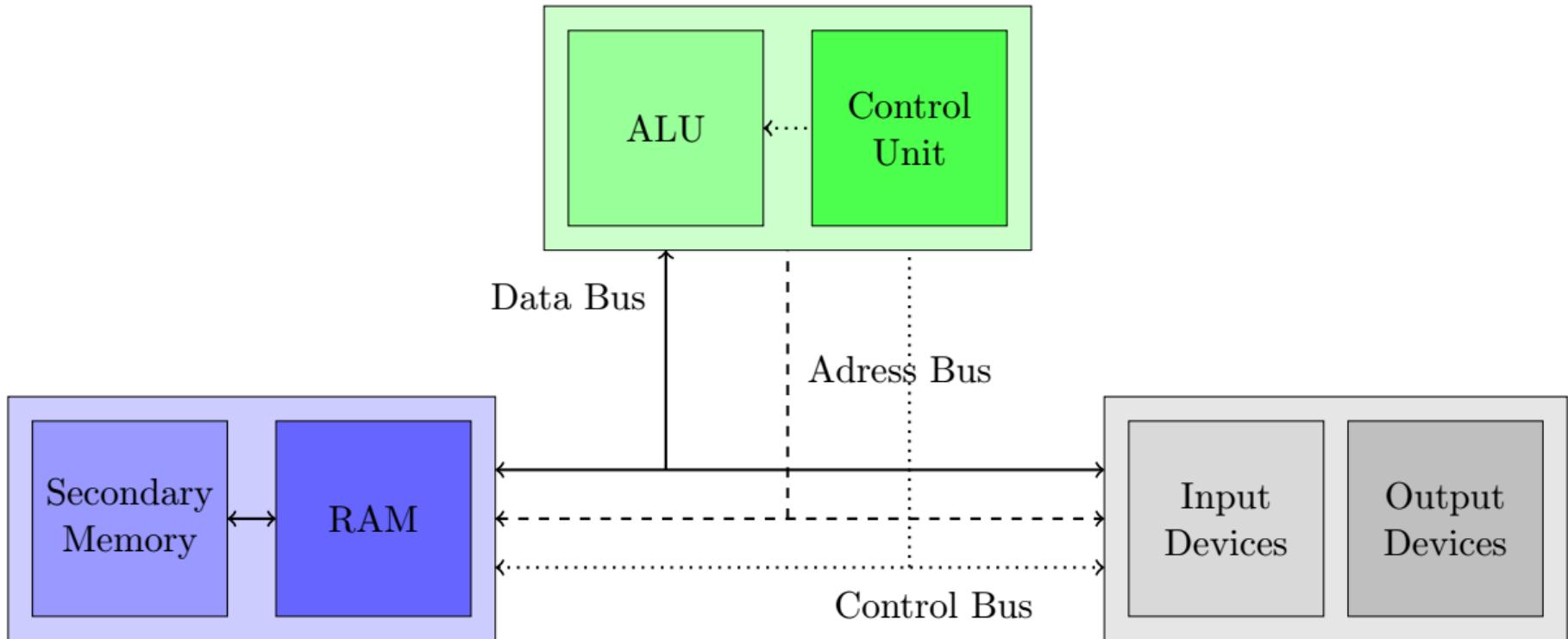
High-Level View of a Computer



- ❑ CPU – execute instructions
- ❑ Memory – store program & data
- ❑ IO devices – receive inputs & produce outputs
- ❑ Bus – interconnects everything by transferring data

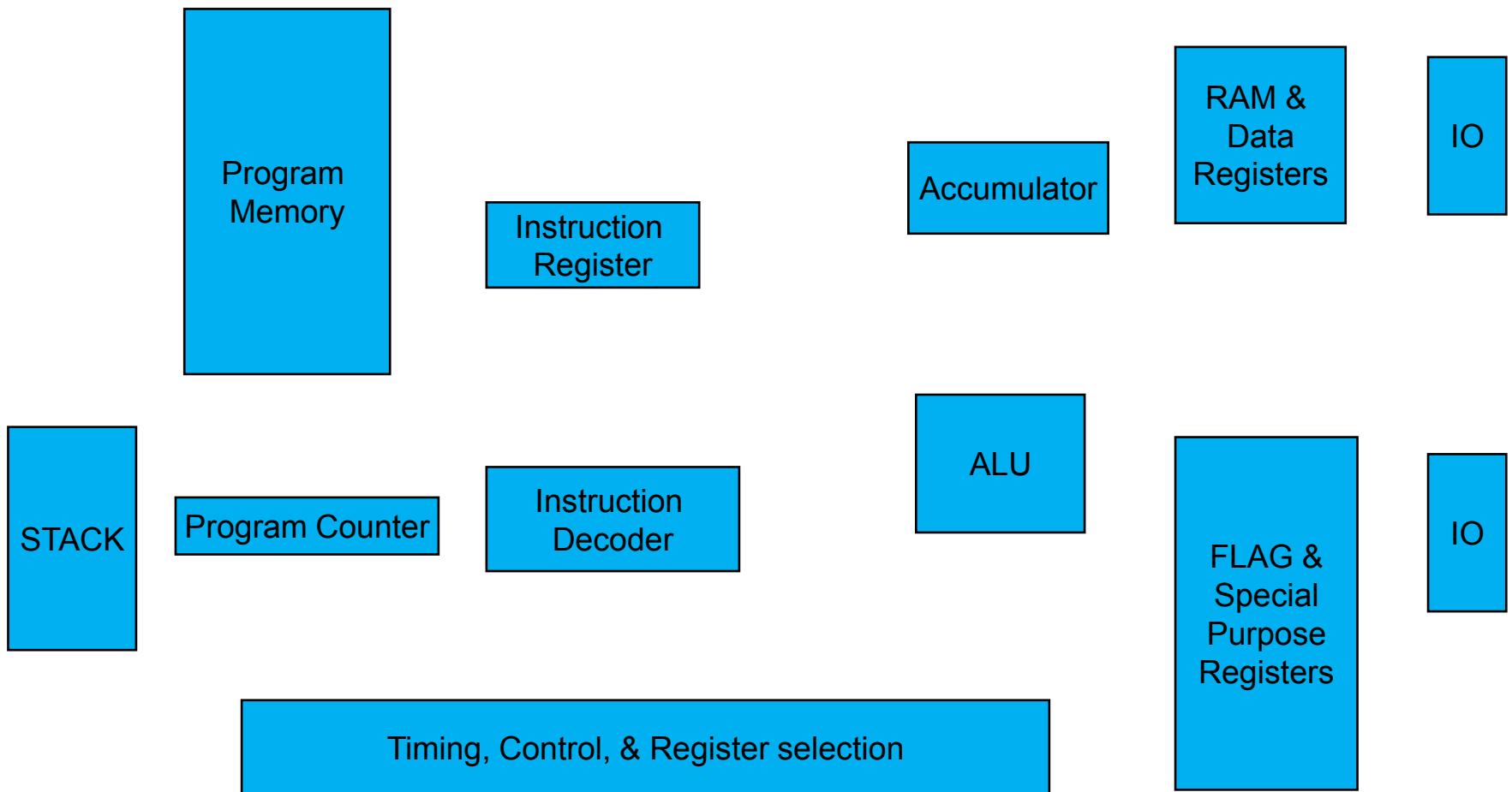


2nd-Level View of a Computer



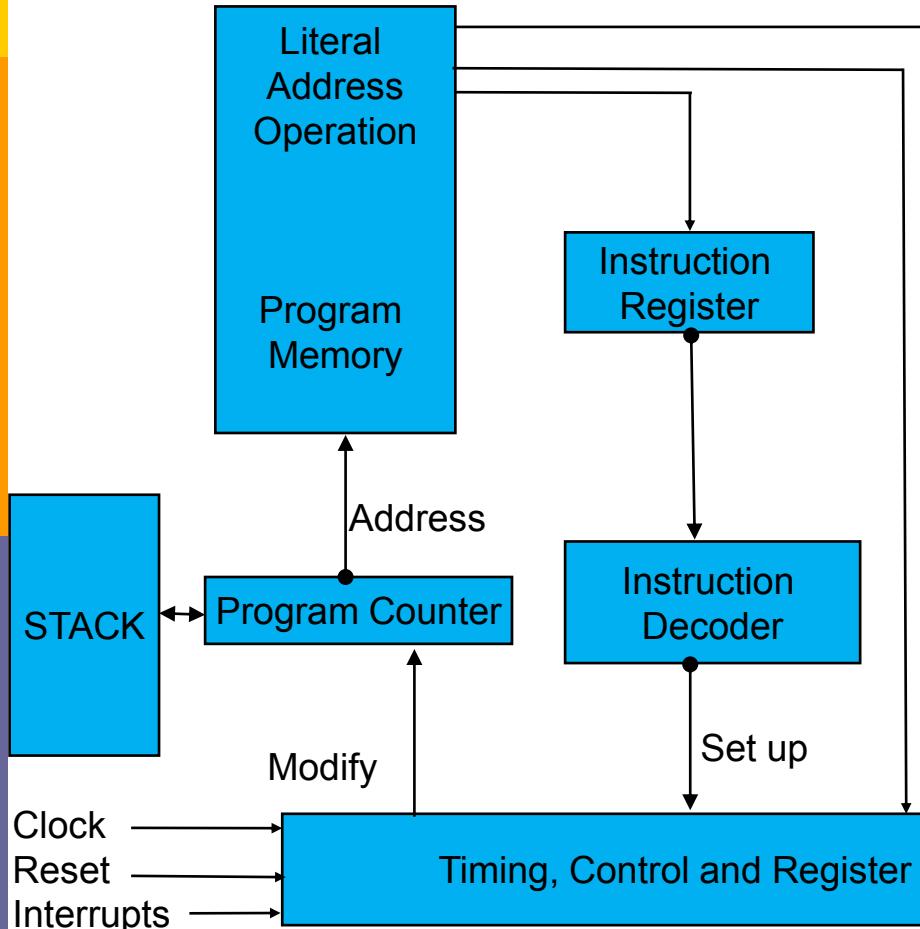


3rd-Level View of a Computer

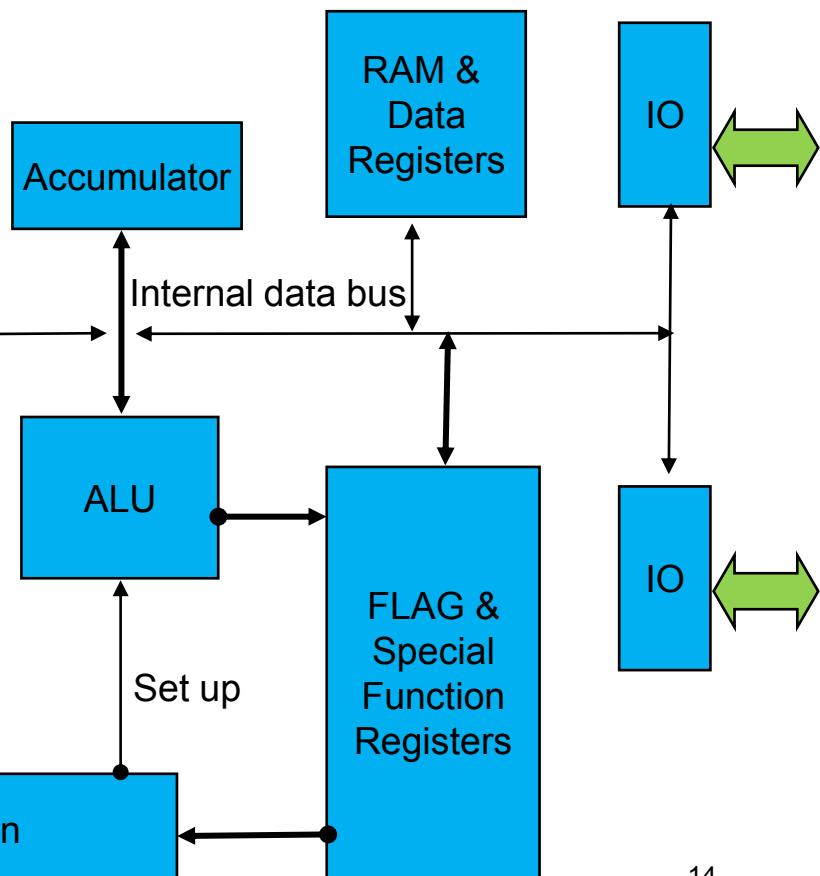


Blocks of a Microprocessor (Cont.)

Program Execution Section



Register Processing Section

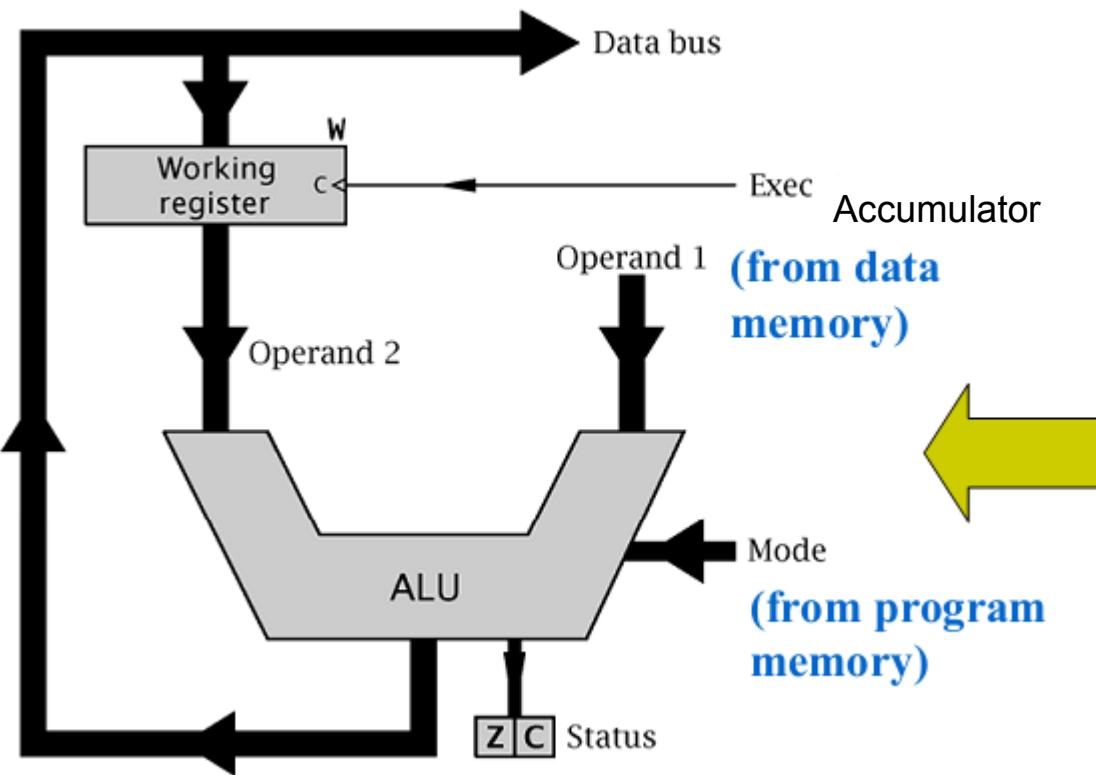


Question(s)

- Which of the following points to the memory address of the next instruction to be executed?
 - a) Program Counter (PC)
 - b) Instruction Register (IR)
 - c) STATUS register
 - d) Accumulator (A)

- _____ interprets an instruction.

Arithmetic & Logic Unit (ALU)



- Data processing unit
- Arithmetic unit
 - Performs arithmetic operations
- Logic unit
 - Performs logic operations

Source: Introduction to PIC Microcontroller – Part 1 by Khan Wahid

Registers

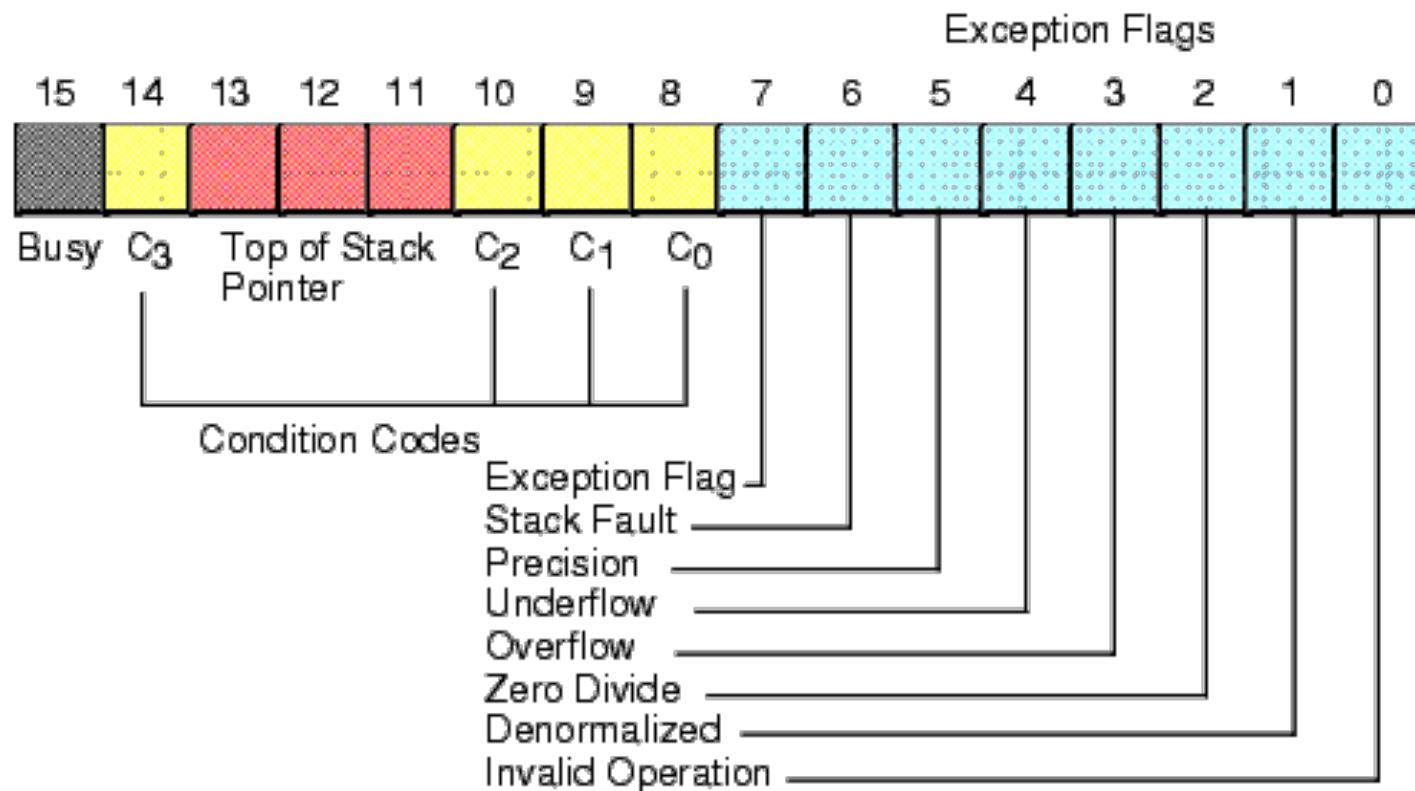
- Type of memory located inside CPU
- Can hold a single piece of data
 - Useful in both data processing & control functionalities
- Special purpose registers
 - Program Counter (PC)
 - Instruction Register (IR)
 - Accumulator or working register
 - Flag/Status register
- General purpose registers
 - Used to store data

Special Purpose Registers

Register	Function
Accumulator (A) / Working Register (W)	<p>Results of arithmetic & logic operations always go to accumulator</p> <p>Connected directly to output of ALU</p>
Program Counter (PC)	<p>Used to keep track of memory address of next instruction to be executed</p> <p>When instructions are <i>fetched</i>, instruction pointed by PC is fetched into CPU</p> <p>Once the instruction is fetched, PC is updated to point to next instruction, i.e.,</p> $PC = PC + d$
Instruction Register (IR)	<p>Once fetched, instructions are stored in IR for execution</p> <p>Located closely to control unit, which decodes the instruction</p>

FLAG/STATUS Register

- Individual bits indicate status of ALU operations



Source: www.plantation-productions.com/Webster/www.artofasm.com/Linux/HTML/RealArithmetic.html

THANK YOU

Computer Architecture – An Introduction



CS2053 Computer Architecture
Computer Science & Engineering
University of Moratuwa

Part II

Computer Architecture

□ How to

- make computers efficient?
- program computers?
- make programs executable across different hardware?
- make programs backward compatible?
- Make computers connected

□ Efficiency

- Pipelining
- Caching
- Etc.

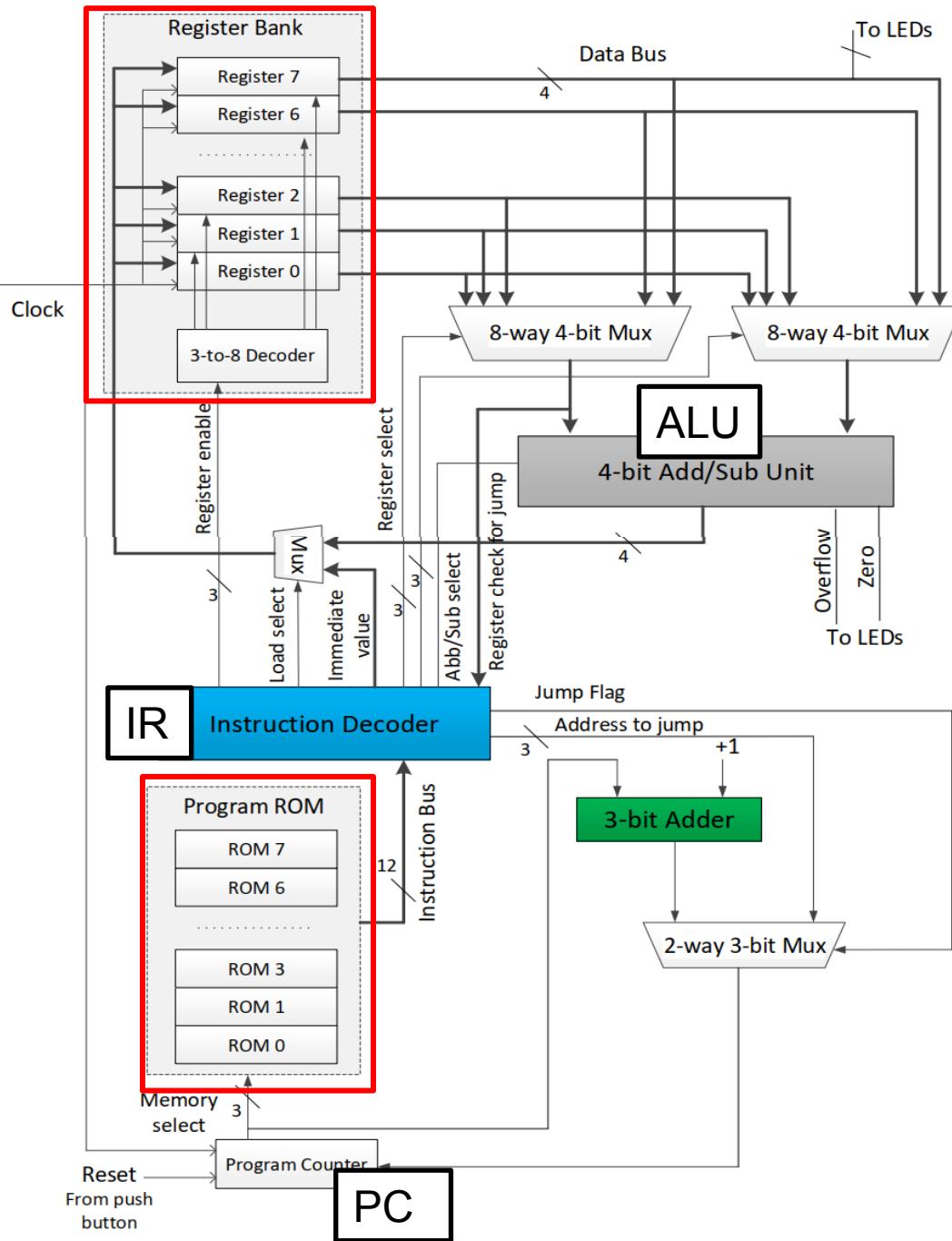
□ Programmability

- ISA (Instruction Set Architecture)

Perspective: Nano-processor

- Can you improve it further?

- Dissecting the process to break-down the different “stages”.



Where is ALU?

Where is IR?

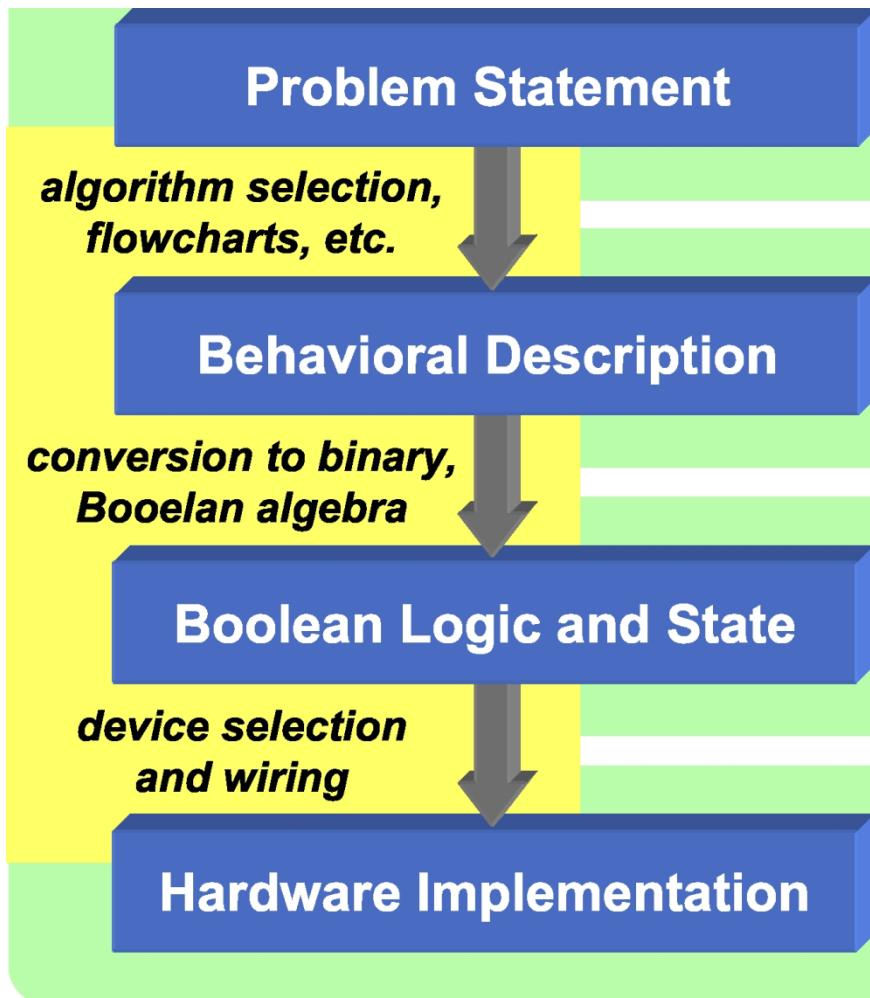
Where is PC?

Where are registers?

Where is the Memory?

*It can be a RAM

Building Digital Solutions to Computational Problems



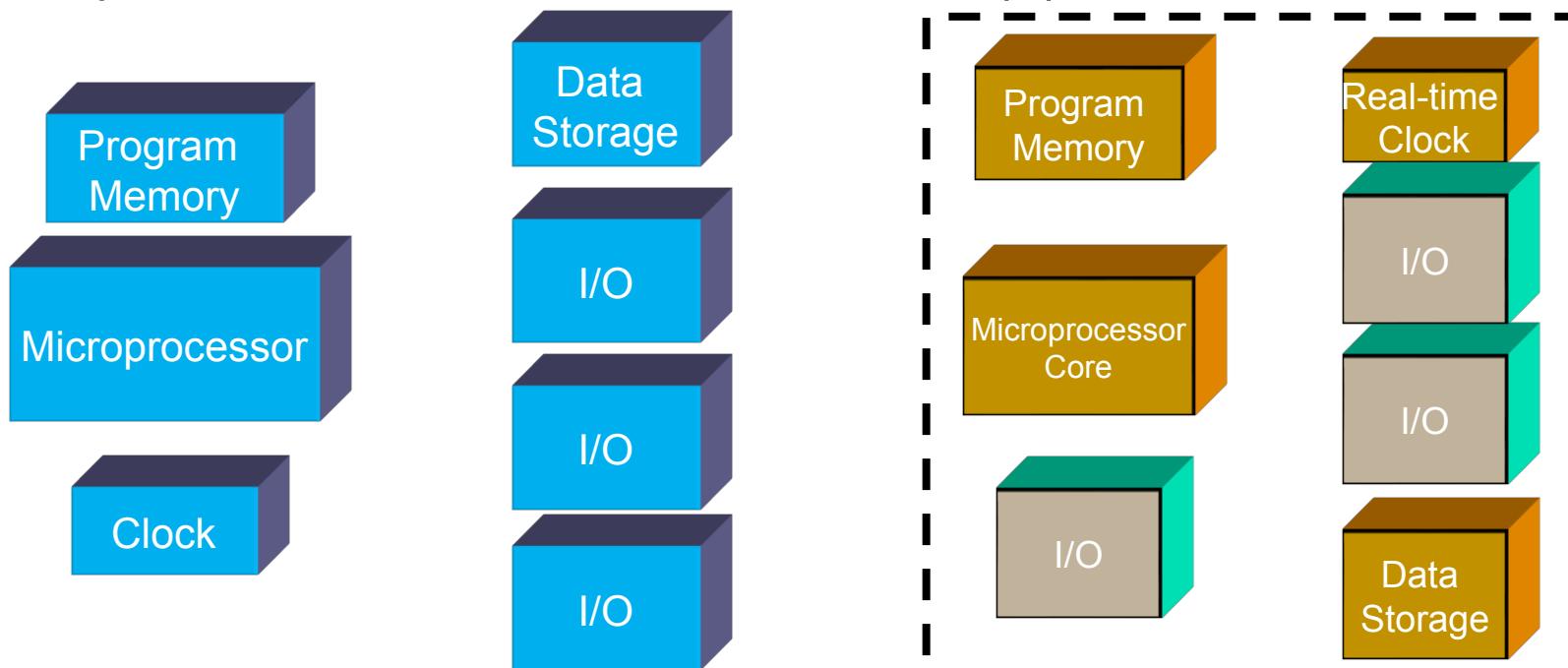
- Algorithms, RTL, etc.
- Flowcharts
- State transition diagrams
- Logic equations
- Circuit schematics
- TTL Gates (AND, OR, XOR ...)
- Programmable Logic
- Custom ASICs
- FPGAs
- MCs, DSPs
- Verilog or VHDL code
- Assembler
- C, C++

Architectural Differences

- Length of microprocessors' data word
 - 4, 8, 16, 32, 64, & 128 bit
- Speed of instruction execution
 - Clock rate → processor speed
- Instruction set – x86, ARM, SPARC, PIC, RISCV
- CPU architecture – RISC vs. CISC
- Size of direct addressable memory
- Number & types of registers
- Support circuits for performance
- Compatibility with existing software & hardware development systems – IBM System/370

Microprocessor vs. Microcontroller

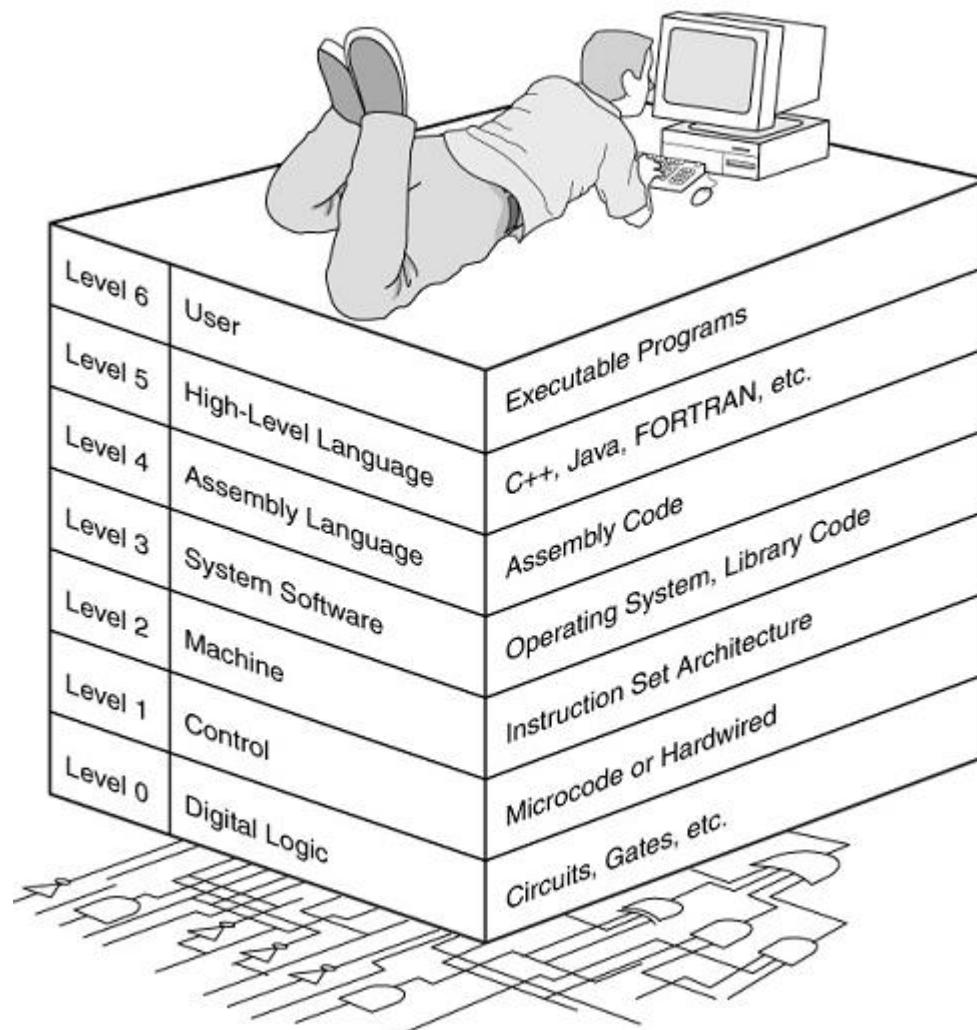
- Microprocessor – CPU & various IO functions are packed as separate ICs
- Microcontroller – Most IO functions are integrated into same package with CPU
- System on Chip SoC – Processor(s) + GPU + FPGA



Processor design to programming

- How to program the computer?
 - Make the computer accessible for programmers.
 - What should a programmer know about the computer, in order to execute a program?
- Can the same program run in different processors?
- Different levels of programming

Programming Hierarchies



Source: Introduction to PIC Microcontroller – Part 1 by Khan Wahid

What is missing in nano-processor?

□ Stages of executing an instruction

- Nano-processor was single cycle
 - Single cycle vs multicycle CPU
- Example
 - Fetch instruction
 - Increment Program Counter
 - Decode instruction
 - Fetch operands from memory
 - Execute instruction
 - Store results back to memory
 - Go to first stage and repeat

□ Pipelining

What is missing in nano-processor?

□ Peripherals

- Memory : RAM (internal, external)
 - **Memory Hierarchy / Cache hierarchy**
- Inputs Outputs
 - Memory mapped inputs/outputs

□ Way to program it

- Abstraction of implementation details from usage
- What do you need to know about the processor in order to program it?
 - Instruction set / Registers / Memory map /Interrupts

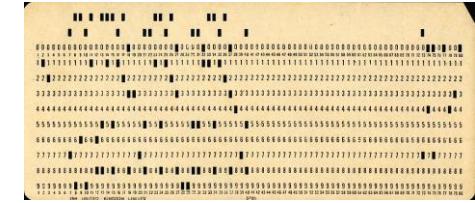
What is missing in nano-processor?

- Efficiency/ Power
 - Features of the processor
 - Floating point/ special co-processors
 - How to speedup?
 - Clock speed
 - Integer operations per second (IOPS) (FLOPS)
 - How to reduce energy consumption ?
 - How to manage heat?

Programming Language Levels

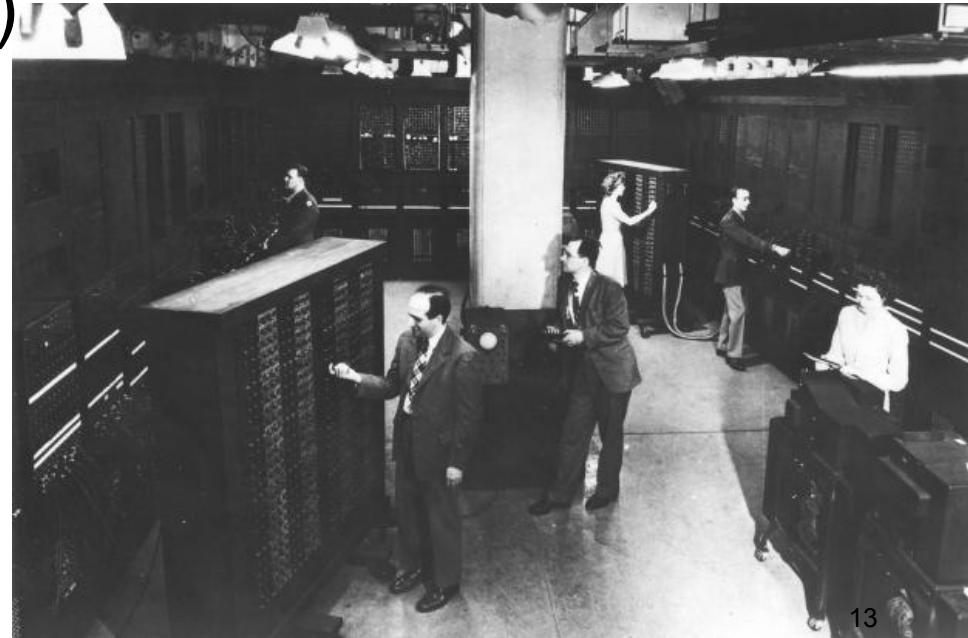
□ Machine code (40s-50s)

- 0001000000111000 0001001000110100
- 0101110000000000
- 0001110000000000 0001001000110101



□ Hex notation (50s-60s)

- 1038 1234
- 5C00
- 1E00 1235



Source: <http://mentalfloss.com/article/53160/meet-refrigerator-ladies-who-programmed-eniac>

Programming Language Levels (Cont.)

□ Assembler

- Machine code (60s-70s)

```
.define const = 6
num1: .byte [1]
num2: .byte [2]
move.b num1,d0
addq.b #const,d0
move.b d0,num2
```

Why Assembly is still useful?

- Can produce code that runs fast
- Better use of CPU resources
- Only way to use some advanced features

□ High-level languages

- C code fragment (70s-80s)

```
#define const 6
int num1, num2;
num2 = num1 + const;
```

Example RISC-V Program

```
.globl main

main:          # Register t1 is also called register 6 (x6)

    li t1, 0x0          # t1 = 0
REPEAT:
    addi t1, t1, 6      # t1 = t1 + 6
    addi t1, t1, -1     # t1 = t1 - 1
    andi t1, t1, 3      # t1 = t1 AND 3
    beq zero, zero, REPEAT # Repeat the loop
    nop
.end
```

Instruction Execution Sequence

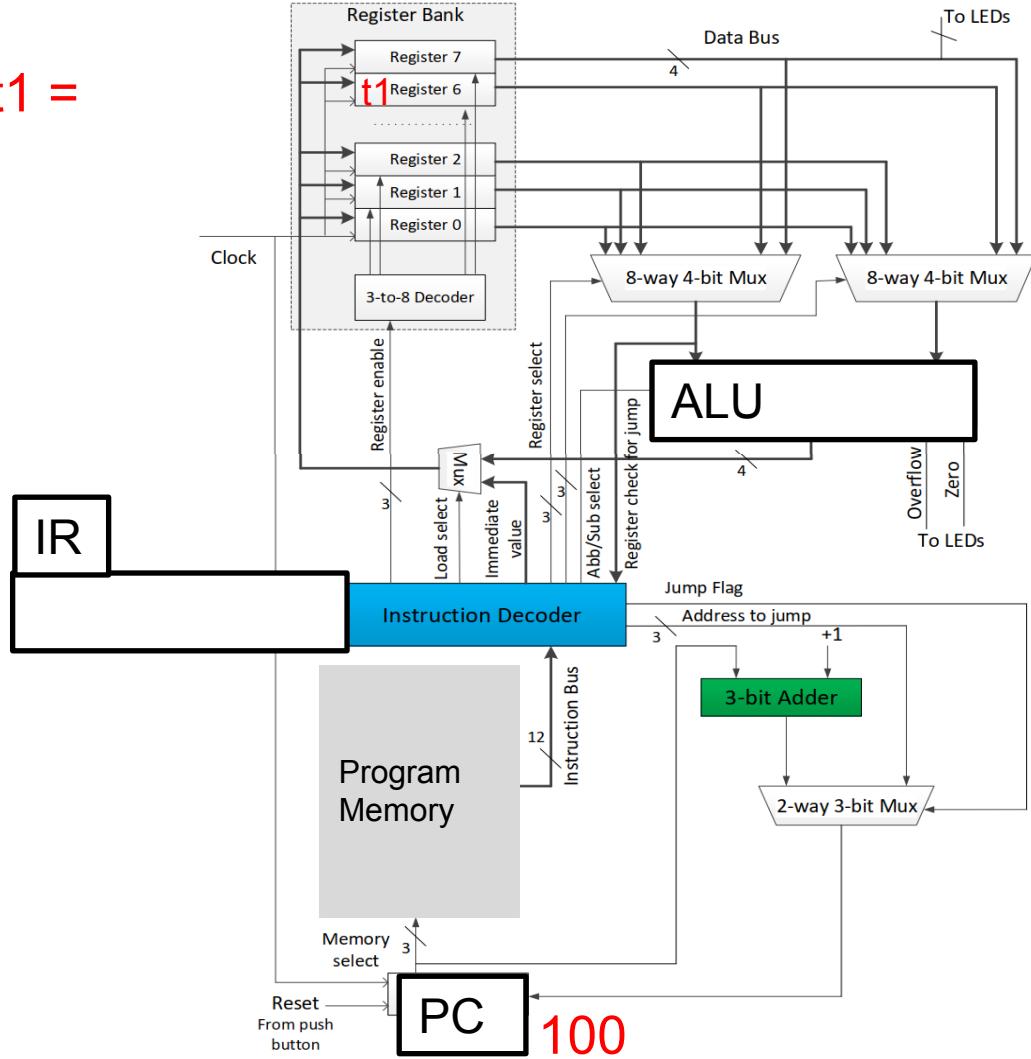
1. Fetch next instruction from memory to IR
2. Change PC to point to next instruction
3. Determine type of instruction just fetched
4. If instruction needs data from memory,
determine where it is
5. Fetch data if needed into register
6. **Execute instruction**
7. Store results back to the memory if needed
8. Go to step 1 & continue with next instruction

Execution of a program

$t1 =$

100 li t1, 0x0
 101 addi t1, t1, 6
 102 addi t1, t1, -1
 103 andi t1, t1, 3
 104
 105

Program memory



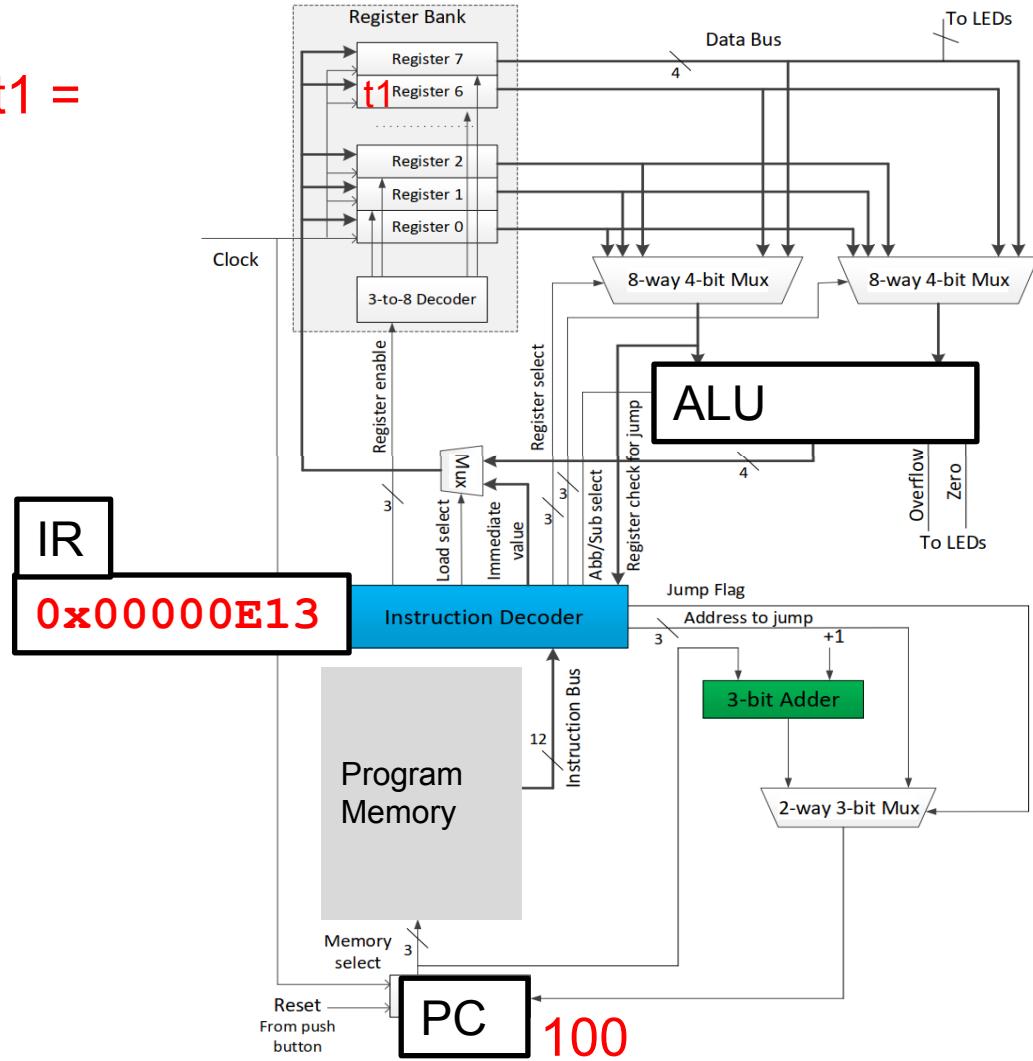
Execution of a program

Fetch Instruction

$t1 =$

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory



Execution of a program

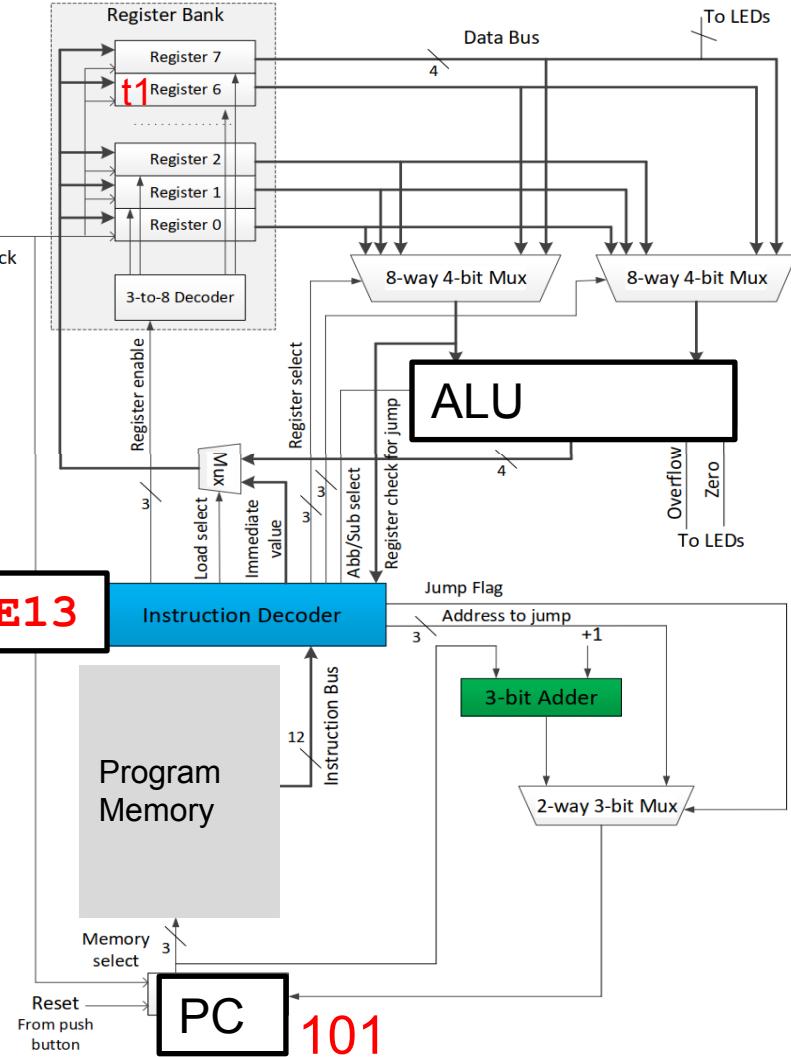
Fetch Instruction
Increment PC

$t1 =$

100 li t1, 0x0
101 addi t1, t1, 6
102 addi t1, t1, -1
103 andi t1, t1, 3
104
105

Program memory

IR
0x00000E13



Execution of a program

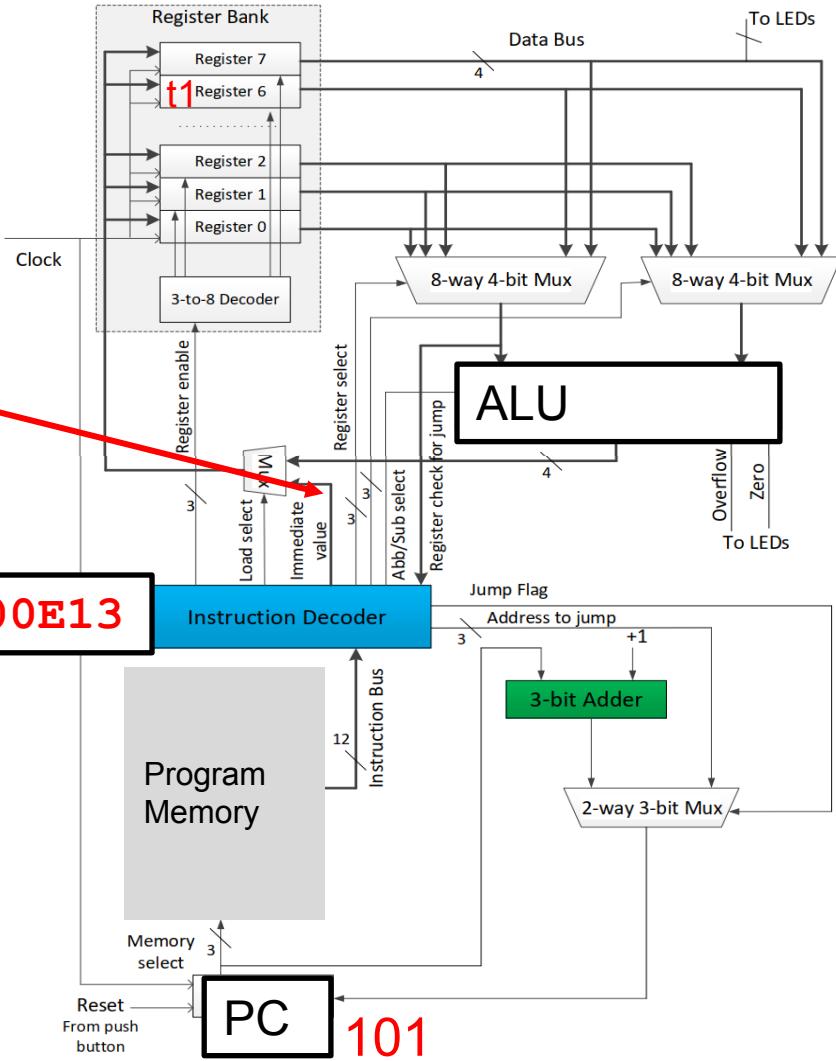
Fetch Instruction
Increment PC
Decode

$t1 =$

The control signals
are enabled

100 li t1, 0x0
101 addi t1, t1, 6
102 addi t1, t1, -1
103 andi t1, t1, 3
104
105

Program memory



Execution of a program

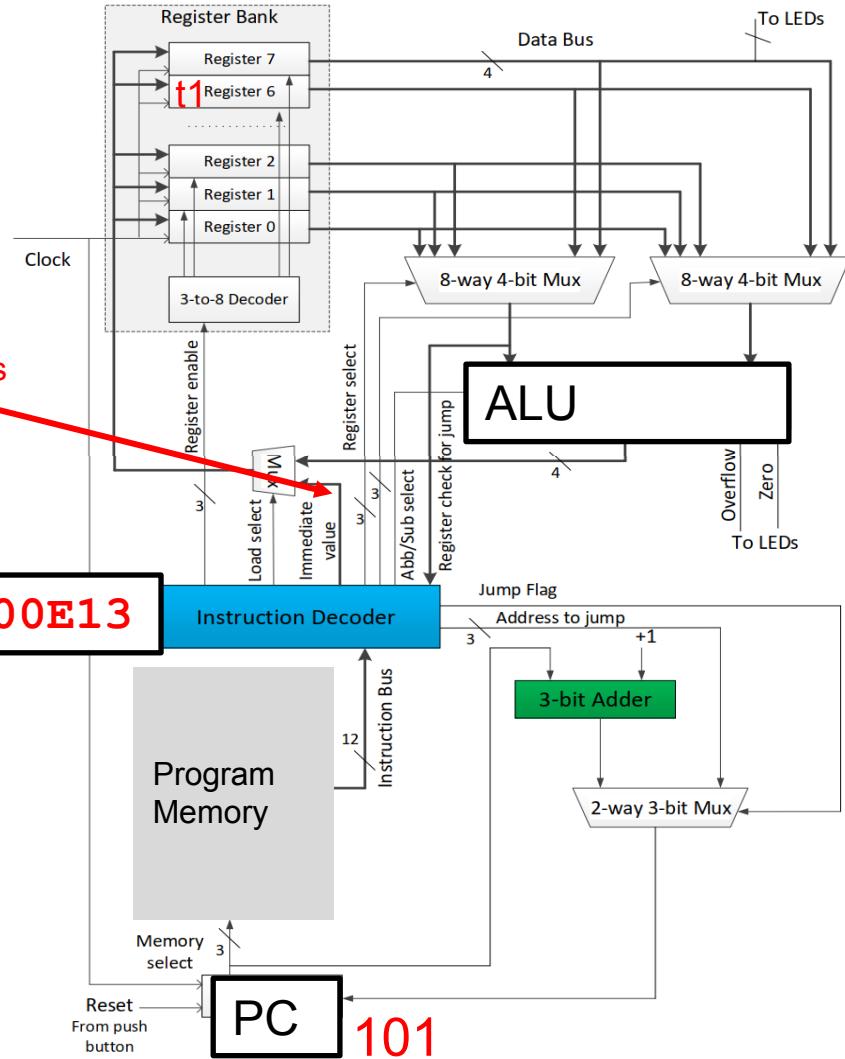
Fetch Instruction
Increment PC
Decode
Execute

$t1 = 0$

The control signals
are enabled

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory



Execution of a program

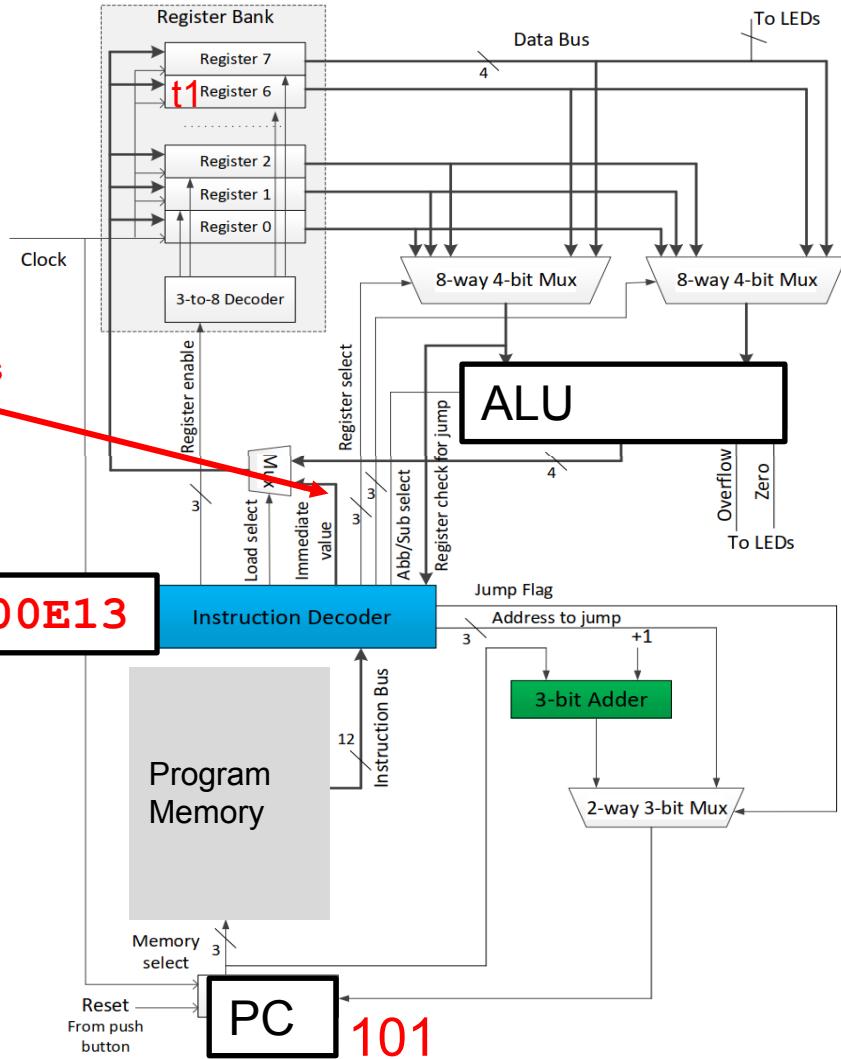
Fetch Instruction
Increment PC
Decode
Execute
Writeback?

$t1 = 0$

The control signals
are enabled

100 li t1, 0x0
101 addi t1, t1, 6
102 addi t1, t1, -1
103 andi t1, t1, 3
104
105

Program memory



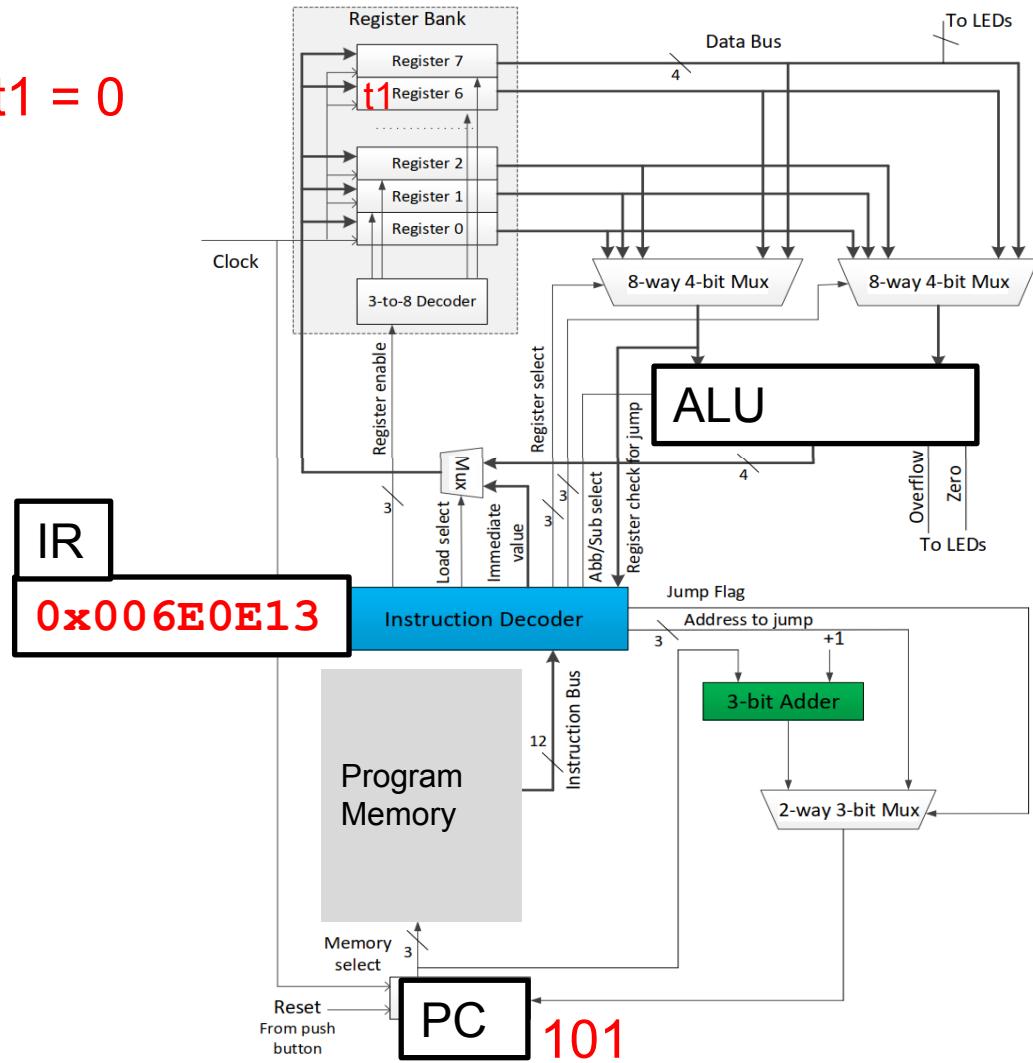
Execution of a program

Fetch Instruction

$t1 = 0$

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory



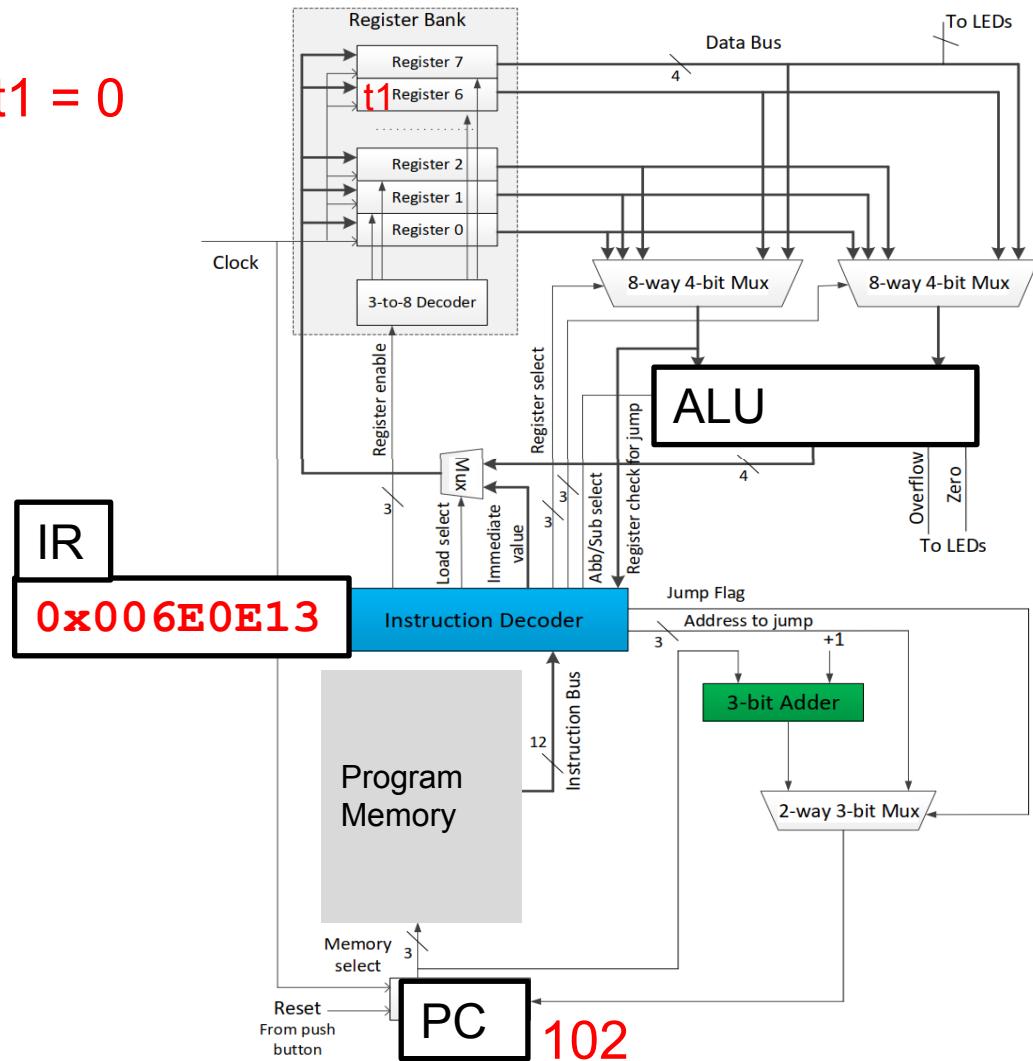
Execution of a program

Fetch Instruction
Increment PC

$t1 = 0$

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory



Execution of a program

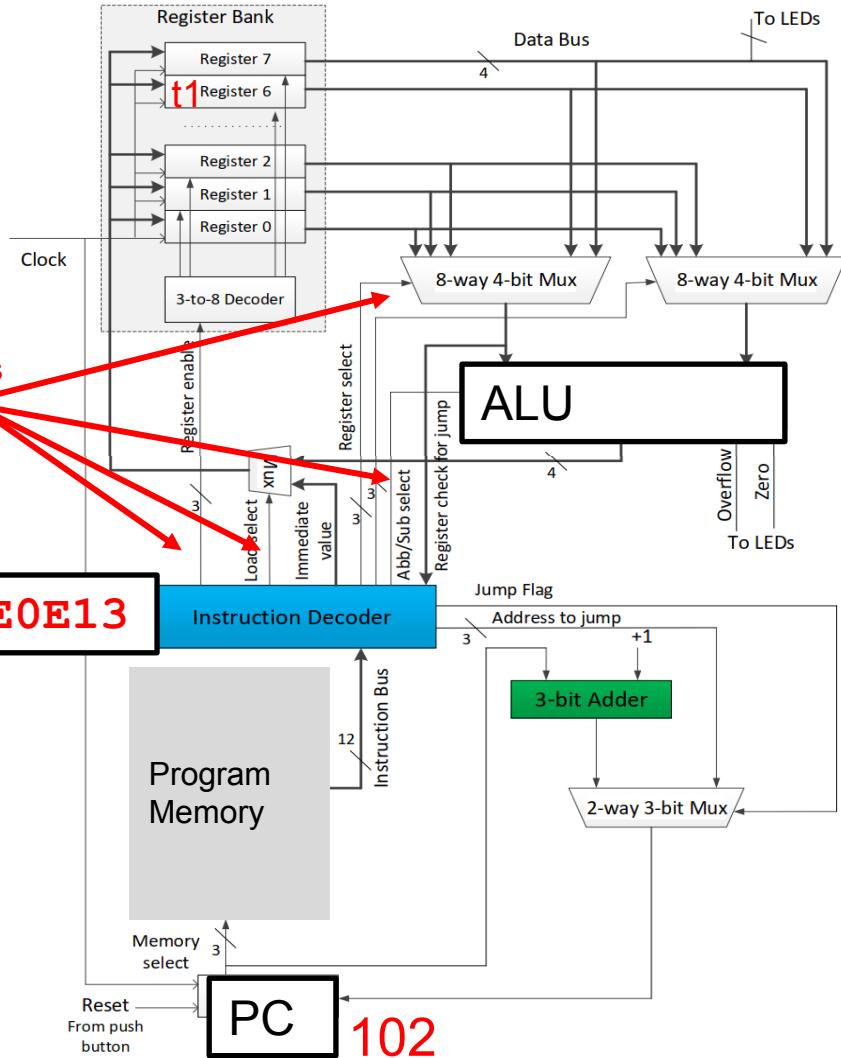
Fetch Instruction
Increment PC
Decode

$t1 = 0$

The control signals
are enabled

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory



Execution of a program

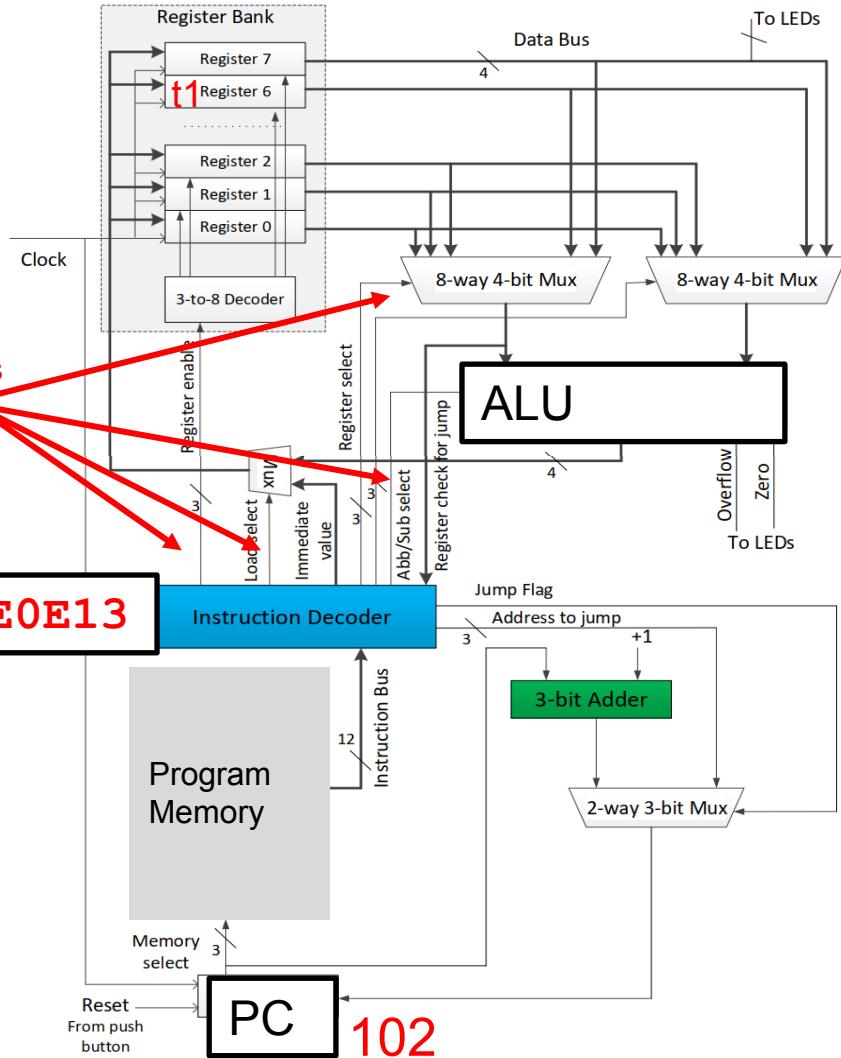
Fetch Instruction
Increment PC
Decode

$t1 = 0$

The control signals
are enabled

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory



Execution of a program

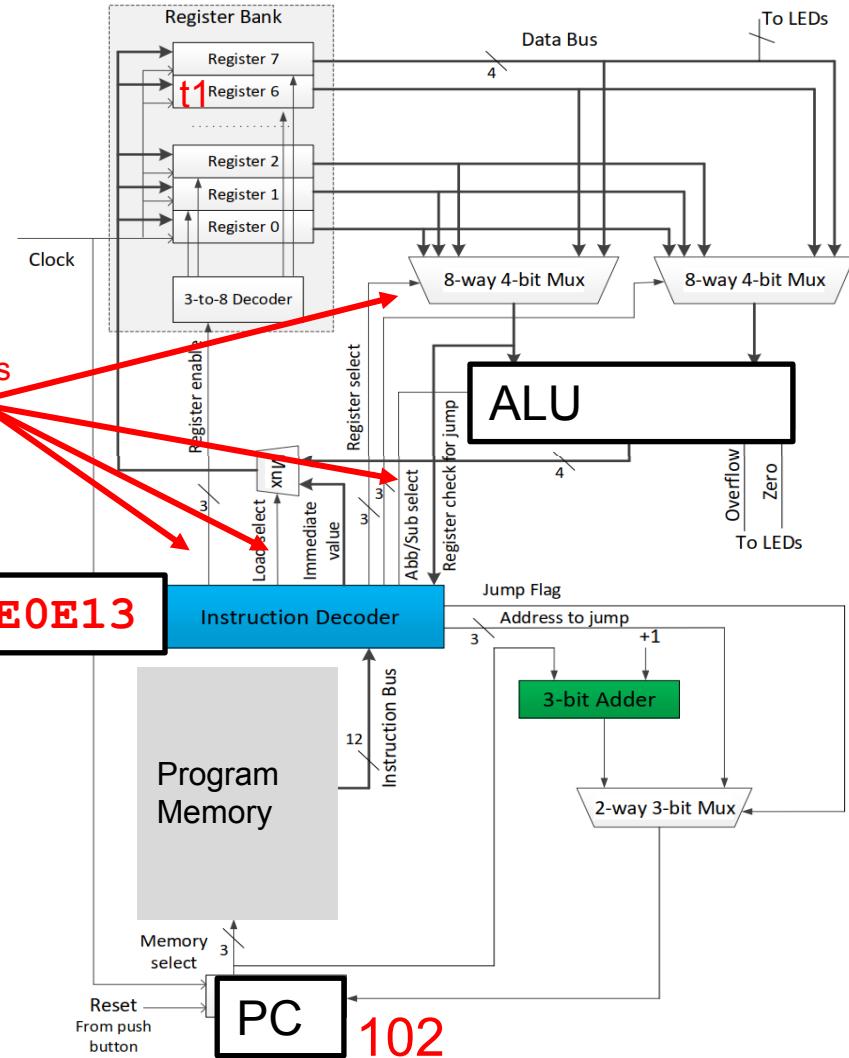
Fetch Instruction
Increment PC
Decode
Execute

$t1 = 6$

The control signals
are enabled

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory



Execution of a program

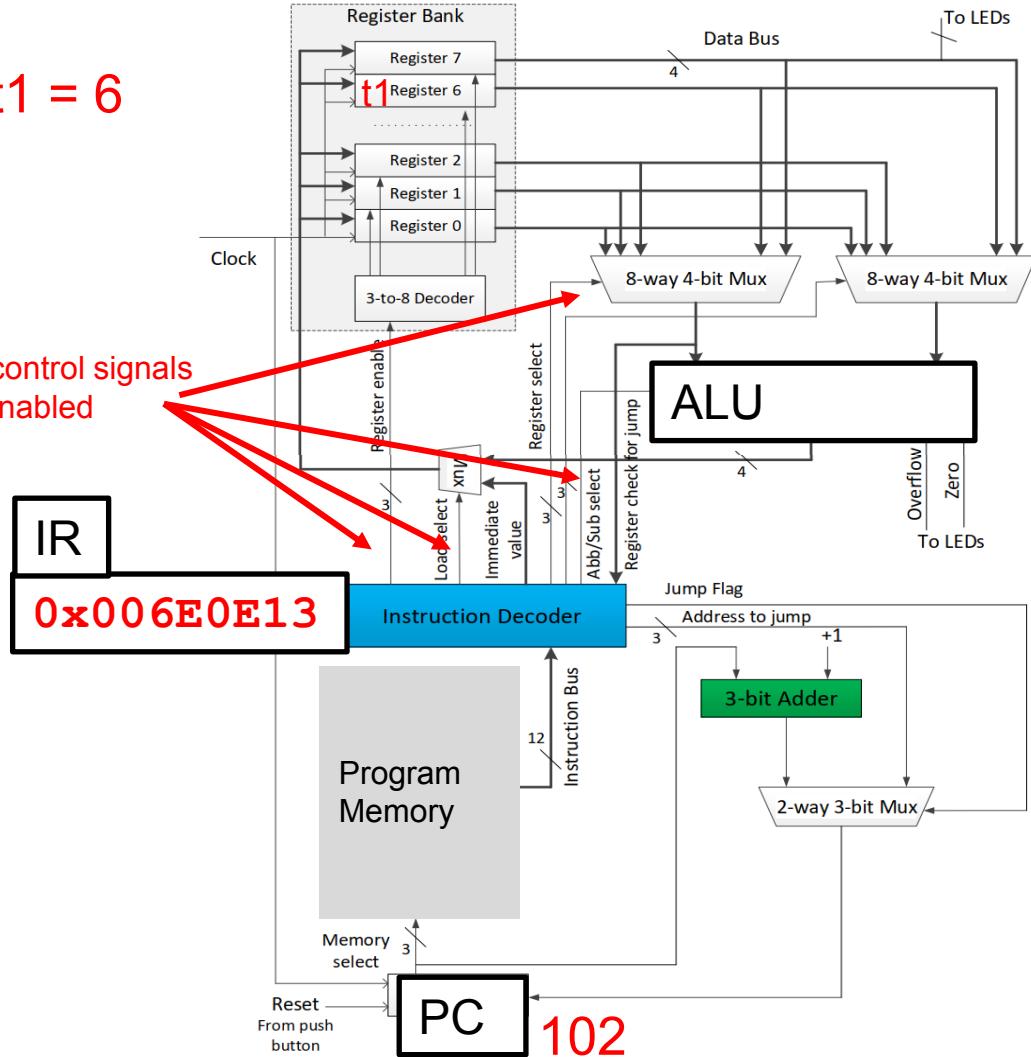
Fetch Instruction
 Increment PC
 Decode
 Execute
 Writeback?
 t1 = 6

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory

t1 = 6

The control signals
are enabled

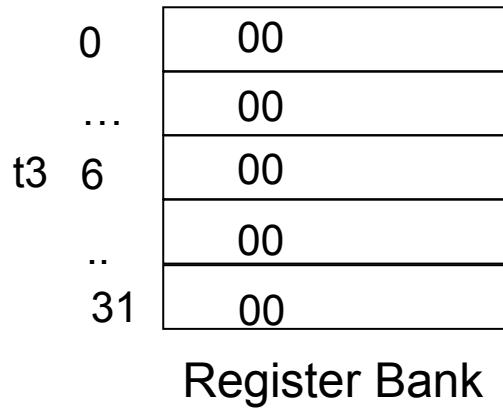


Simple RISC-V Assembly program

```
.globl main

main:
    li t1, 0x0                      # t1 = 0
REPEAT:
    addi t1, t1, 6      # t1 = t1 + 6
    addi t1, t1, -1     # t1 = t1 - 1
    andi t1, t1, 3      # t1 = t1 AND 3

    beq zero, zero, REPEAT
                    # Repeat the loop
    nop
.end
```



100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	beq zero, zero, 101
105	

Program memory

18	00
19	00
20	00
21	00

Data memory

Instruction Execution Sequence

1. Fetch next instruction from memory to IR
2. Change PC to point to next instruction
3. Determine type of instruction just fetched
4. If instruction needs data from memory,
determine where it is
5. Fetch data if needed into register
6. Execute instruction
7. Store results back to the memory if needed
8. Go to step 1 & continue with next instruction

Sample Program

```
.data  
A: .word 10  
.bss  
.text  
.globl main  
  
main:  
    la    a0, A # Address of variable A  
    lw    t0,0(a0)  
    li    t1,15  
    add   t0,t0,t1  
    sw    t0, 0(a0)  
    ret  
.end
```



100	la a0, A
101	lw t0,0(a0)
102	li t1,15
103	add t0,t0,t1
104	sw t0,0(a0)
105	

Program memory



18	00
19	10
20	00
21	00

Data memory

Sample Program

```
.data  
    A: .word 10  
.bss  
.text  
.globl main  
main:  
    la a0, A # Address of variable A  
    lw t0,0(a0)  
    li t1,15  
    add t0,t0,t1  
    sw t0, 0(a0)  
    ret  
.end
```

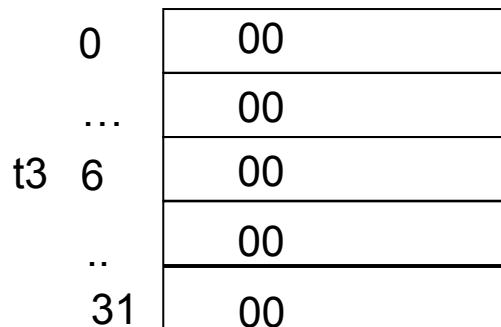
100	la a0, A
101	lw t0,0(a0)
102	li t1,15
103	add t0,t0,t1
104	sw t0,0(a0)
105	

Program memory

18	00
A: 19	10
20	00
21	00

Data memory

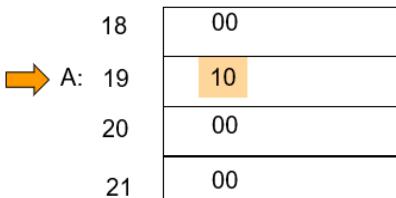
Execution of a program



Register Bank

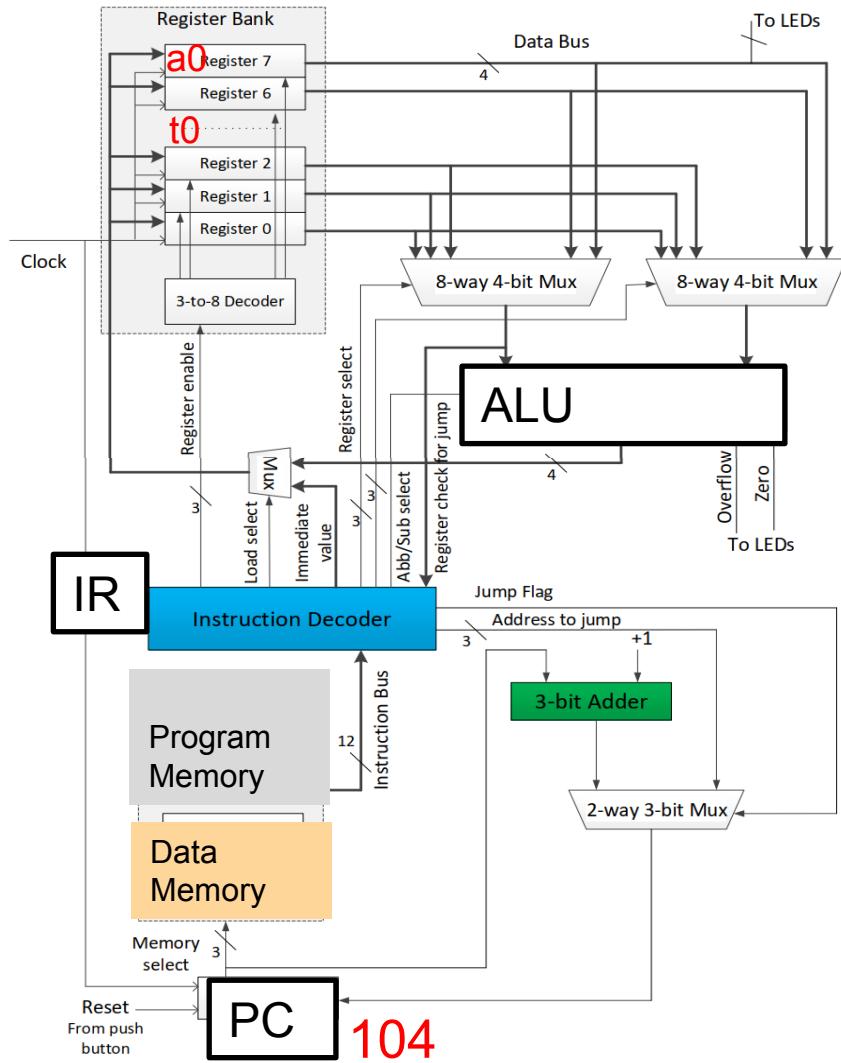


Program memory



Data memory

$$a0 = \\ t0 =$$



Assembling and linking

- Convert written symbols to binary (**Assembling**)

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

Instruction	Format	immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
ld (load doubleword)	I	address	reg	011	reg	0000011

Instruction	Format	immed- iate	rs2	rs1	funct3	immed- iate	opcode
sd (store doubleword)	S	address	reg	reg	011	address	0100011

FIGURE 2.5 RISC-V instruction encoding. In the table above, “reg” means a register number between 0 and 31 and “address” means a 12-bit address or constant. The funct3 and funct7 fields act as additional opcode fields.

R type (Register)
I type (Immediate)
S type (Store)

- Mapping memory to variables and inputs/outputs
- Bringing multiple functions/programs to work together (**linking**)

ISA: Instruction Set Architecture

- What programmer should know to use a processor?
 - Set of registers ?
 - Instruction set ?
 - Memory map ? (RAM/ Inputs / Outputs)
- Compilers make it easy to program at even higher levels

ISA: Instruction Set Architecture
Layer of abstraction between hardware and software

How to design a good ISA?

RISC –V Registers

32 32-bit registers

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary variables
s0/fp	x8	Saved variable / Frame pointer
s1	x9	Saved variable
a0-1	x10-11	Function arguments / Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved variables
t3-6	x28-31	Temporary variables

RISC V Instruction Set

Common RISC-V Assembly Instructions & Pseudoinstructions

RISC-V Assembly	Description	Operation
<code>add s0, s1, s2</code>	Add	$s0 = s1 + s2$
<code>sub s0, s1, s2</code>	Subtract	$s0 = s1 - s2$
<code>addi t3, t1, -10</code>	Add immediate	$t3 = t1 - 10$
<code>mul t0, t2, t3</code>	32-bit multiply	$t0 = t2 * t3$
<code>div s9, t5, t6</code>	Division	$t9 = t5 / t6$
<code>rem s4, s1, s2</code>	Remainder	$s4 = s1 \% s2$
<code>and t0, t1, t2</code>	Bit-wise AND	$t0 = t1 \& t2$
<code>or t0, t1, t5</code>	Bit-wise OR	$t0 = t1 t5$
<code>xor s3, s4, s5</code>	Bit-wise XOR	$s3 = s4 ^ s5$
<code>andi t1, t2, 0xFFB</code>	Bit-wise AND immediate	$t1 = t2 \& 0xFFFFFFFFB$
<code>ori t0, t1, 0x2C</code>	Bit-wise OR immediate	$t0 = t1 0x2C$
<code>xori s3, s4, 0xAB</code>	Bit-wise XOR immediate	$s3 = s4 ^ 0xFFFFAB$
<code>sll t0, t1, t2</code>	Shift left logical	$t0 = t1 << t2$
<code>srl t0, t1, t5</code>	Shift right logical	$t0 = t1 >> t5$
<code>sra s3, s4, s5</code>	Shift right arithmetic	$s3 = s4 >>> s5$
<code>slli t1, t2, 30</code>	Shift left logical immediate	$t1 = t2 << 30$
<code>srlti t0, t1, 5</code>	Shift right logical immediate	$t0 = t1 >> 5$
<code>srai s3, s4, 31</code>	Shift right arithmetic immediate	$s3 = s4 >>> 31$

RISC V Instruction Set

Common RISC-V Assembly Instructions & Pseudoinstructions (continued)

RISC-V Assembly	Description	Operation
<code>lw s7, 0x2C(t1)</code>	Load word	$s7 = \text{memory}[t1+0x2C]$
<code>lh s5, 0x5A(s3)</code>	Load half-word	$s5 = \text{SignExt}(\text{memory}[s3+0x5A]_{15:0})$
<code>lb s1, -3(t4)</code>	Load byte	$s1 = \text{SignExt}(\text{memory}[t4-3]_{7:0})$
<code>sw t2, 0x7C(t1)</code>	Store word	$\text{memory}[t1+0x7C] = t2$
<code>sh t3, 22(s3)</code>	Store half-word	$\text{memory}[s3+22]_{15:0} = t3_{15:0}$
<code>sb t4, 5(s4)</code>	Store byte	$\text{memory}[s4+5]_{7:0} = t4_{7:0}$
<code>beq s1, s2, L1</code>	Branch if equal	$\text{if } (s1 == s2), \text{PC} = L1$
<code>bne t3, t4, Loop</code>	Branch if not equal	$\text{if } (s1 != s2), \text{PC} = \text{Loop}$
<code>blt t4, t5, L3</code>	Branch if less than	$\text{if } (t4 < t5), \text{PC} = L3$
<code>bge s8, s9, Done</code>	Branch if not equal	$\text{if } (s8 >= s9), \text{PC} = \text{Done}$
<code>li s1, 0xABCD12</code>	Load immediate	$s1 = 0xABCD12$
<code>la s1, A</code>	Load address	$s1 = \text{Variable A's memory address (location)}$
<code>nop</code>	Nop	no operation
<code>mv s3, s7</code>	Move	$s3 = s7$
<code>not t1, t2</code>	Not (Invert)	$t1 = \sim t2$
<code>neg s1, s3</code>	Negate	$s1 = -s3$
<code>j Label</code>	Jump	$\text{PC} = \text{Label}$
<code>jal L7</code>	Jump and link	$\text{PC} = L7; ra = \text{PC} + 4$
<code>jr s1</code>	Jump register	$\text{PC} = s1$

Textbook

Instruction Set Architecture (ISA)
Quantitative design and analysis
Memory Hierarchy
Instruction level parallelism
Data level parallelism
Thread-Level Parallelism
Domain specific architectures

John L. Hennessy | David A. Patterson

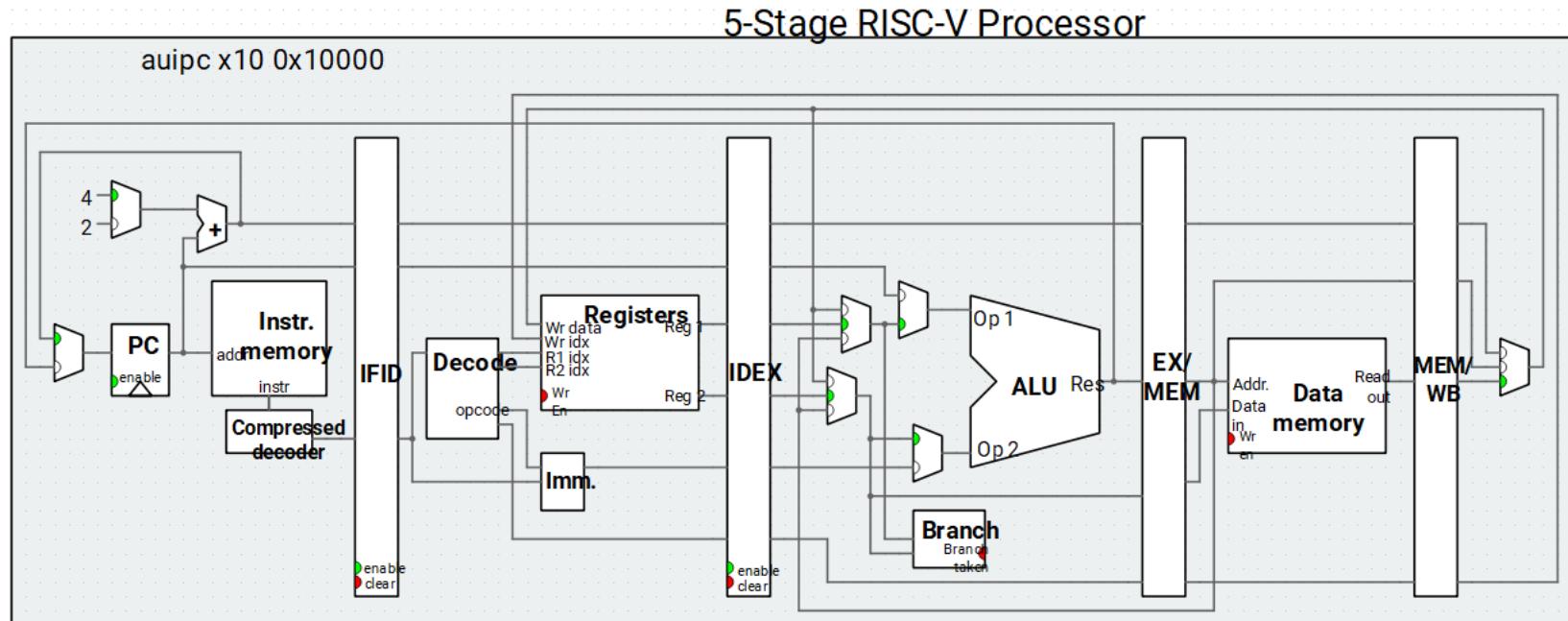
COMPUTER ARCHITECTURE



Sixth Edition

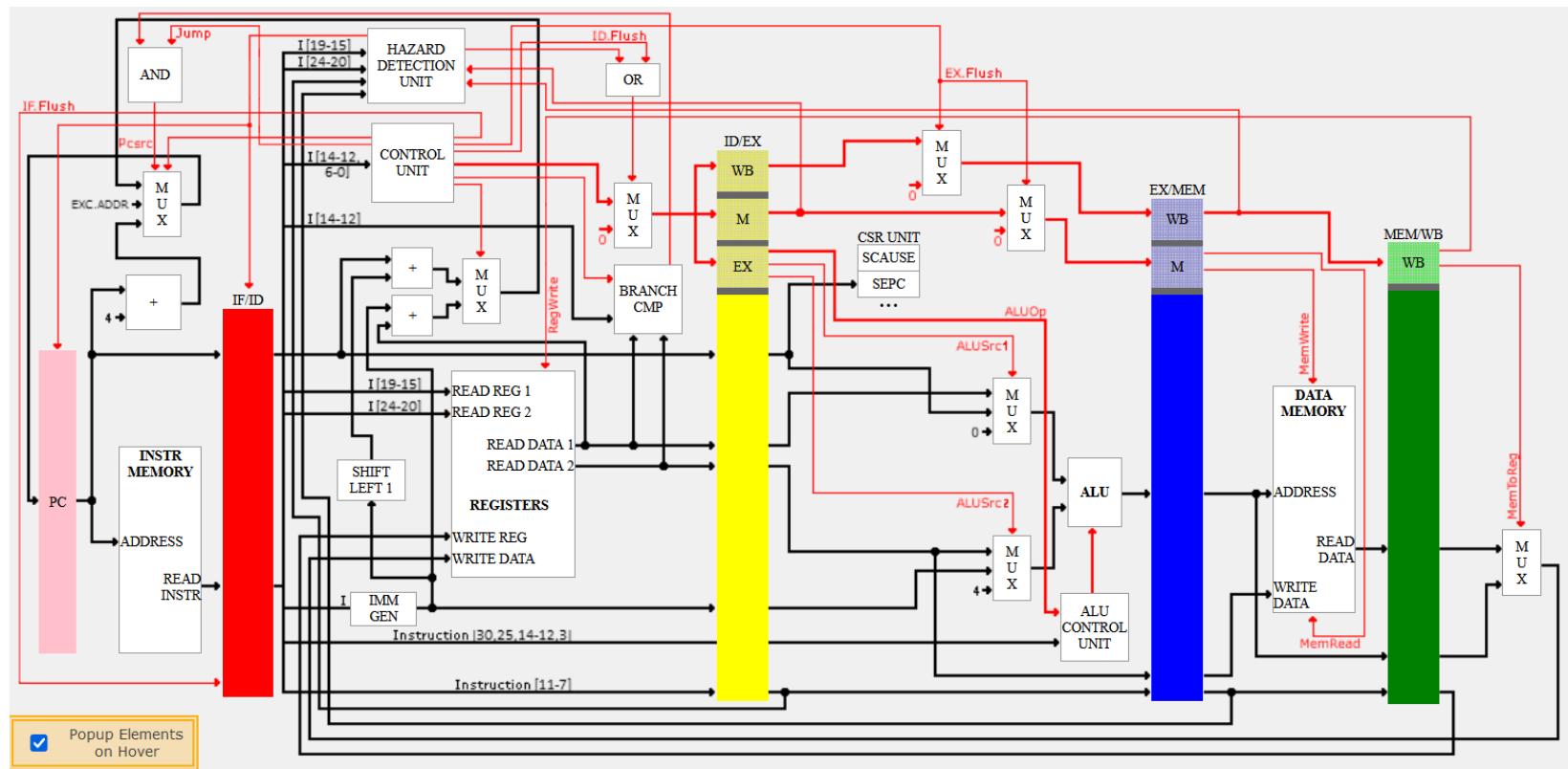
Tools: Simulation

- Learn the concepts on simulation (no hardware)
- Assess various strategies and try-out instructions.
- Example: Ripes simulator (Standalone software)



Tools : Simulation

□ Example: WebRiscV Simulator (Web Based)



<https://webriscv.dii.unisi.it/index.php>

Tools: Optimize Processor Hardware

- Build, and test using Verilog, Try it in FPGAs
- <https://dms.uom.lk/s/PXs7rnCgRL4GgXF>
- Edx Course is available
<https://www.edx.org/course/computer-architecture-with-an-industrial-risc-v-core>

Tools: Build and Optimize

- Hardware
 - Verilog
- Test software

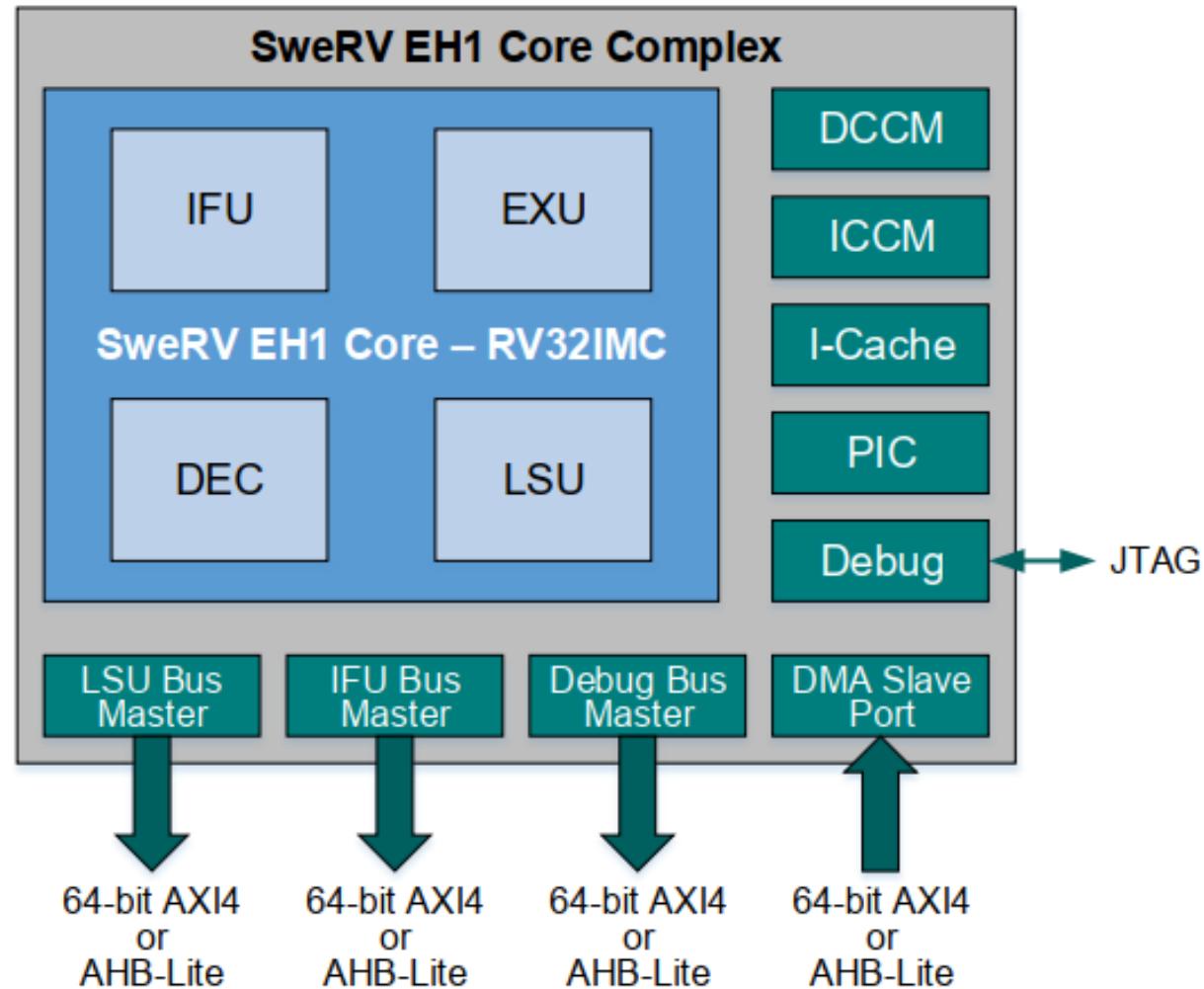


Figure 1-1 SweRV EH1 Core Complex

Stage

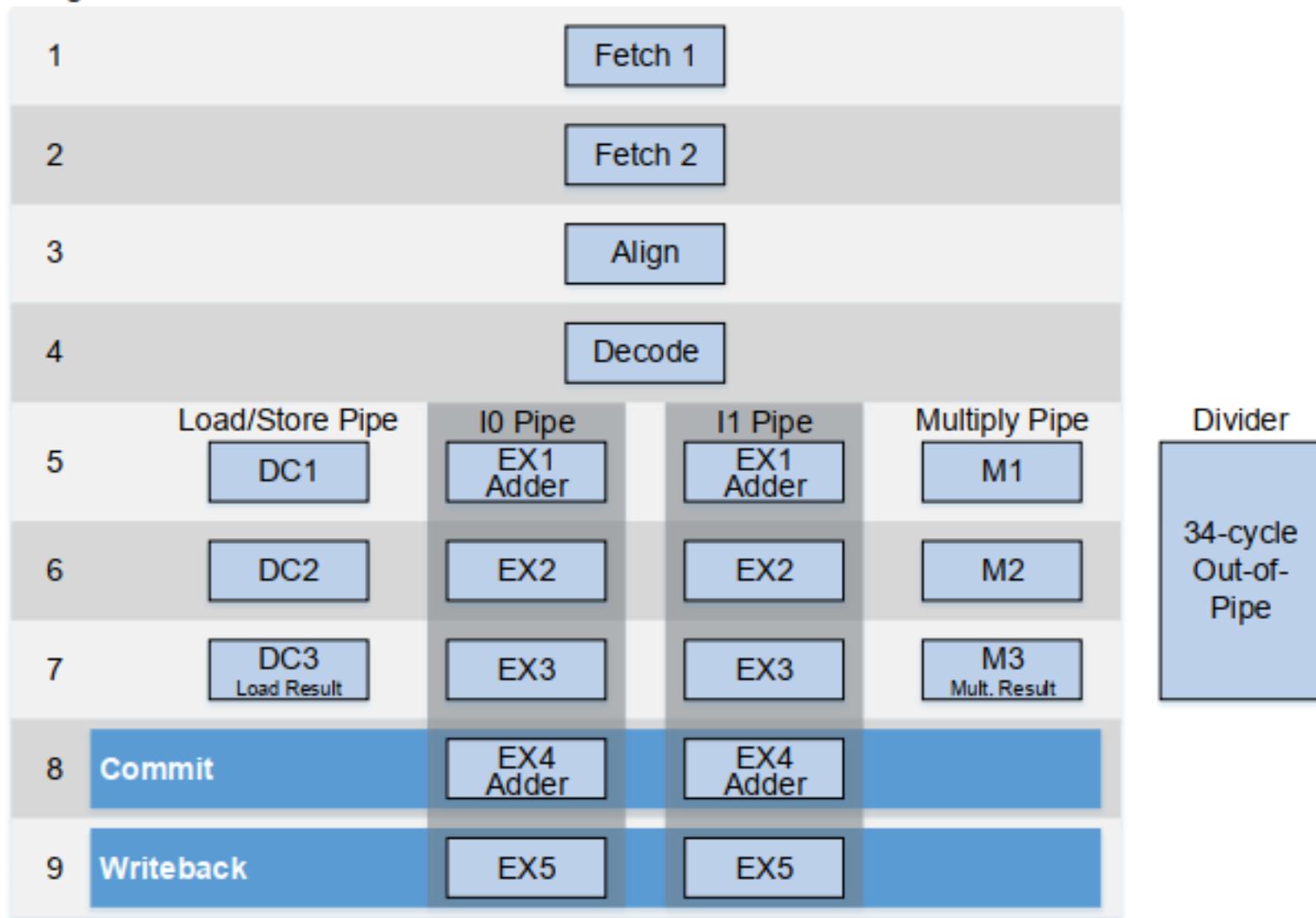


Figure 1-2 SweRV EH1 Core Pipeline

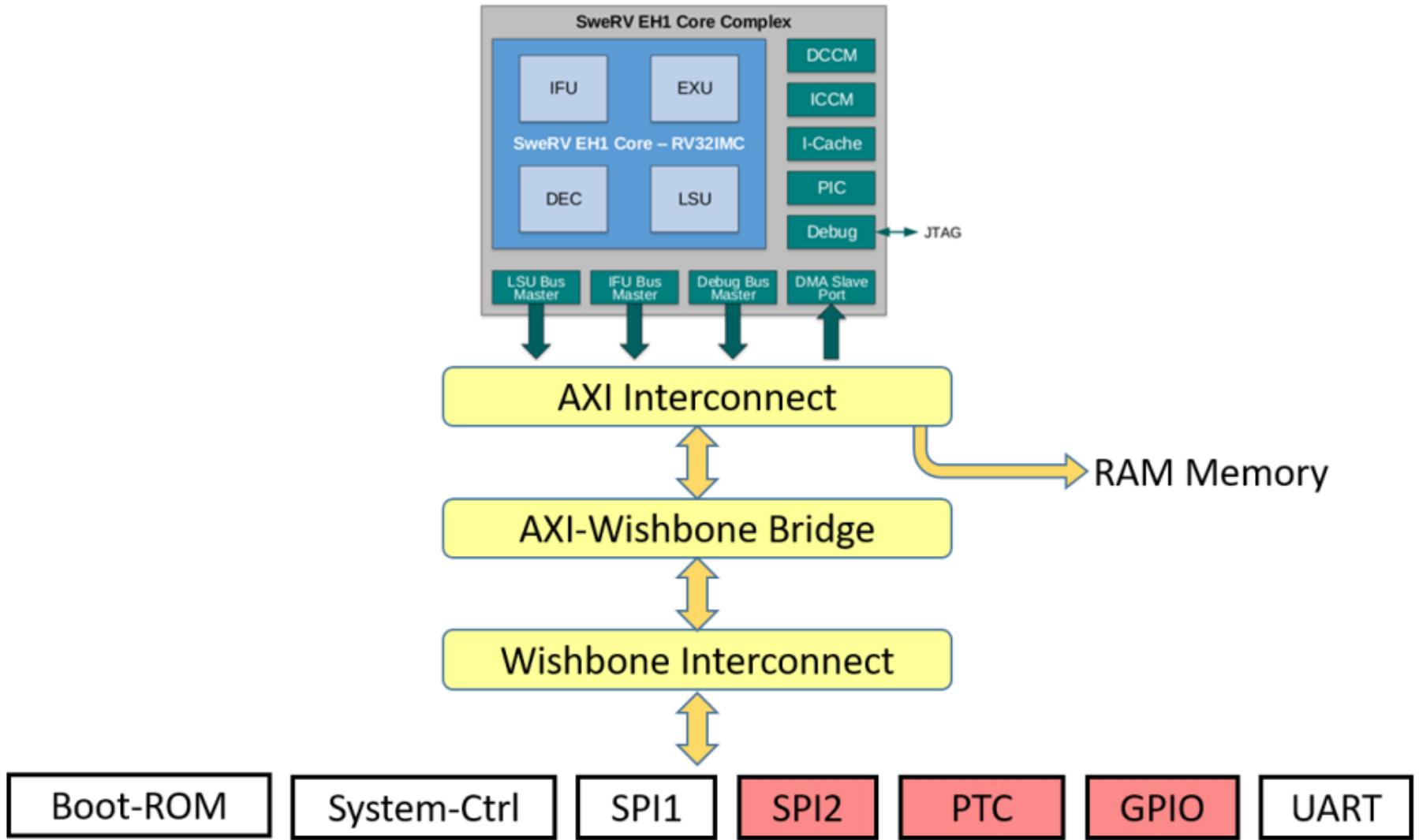


Figure 21. SweRVolfX (SweRVolf eXtended with 4 new peripherals) System on Chip

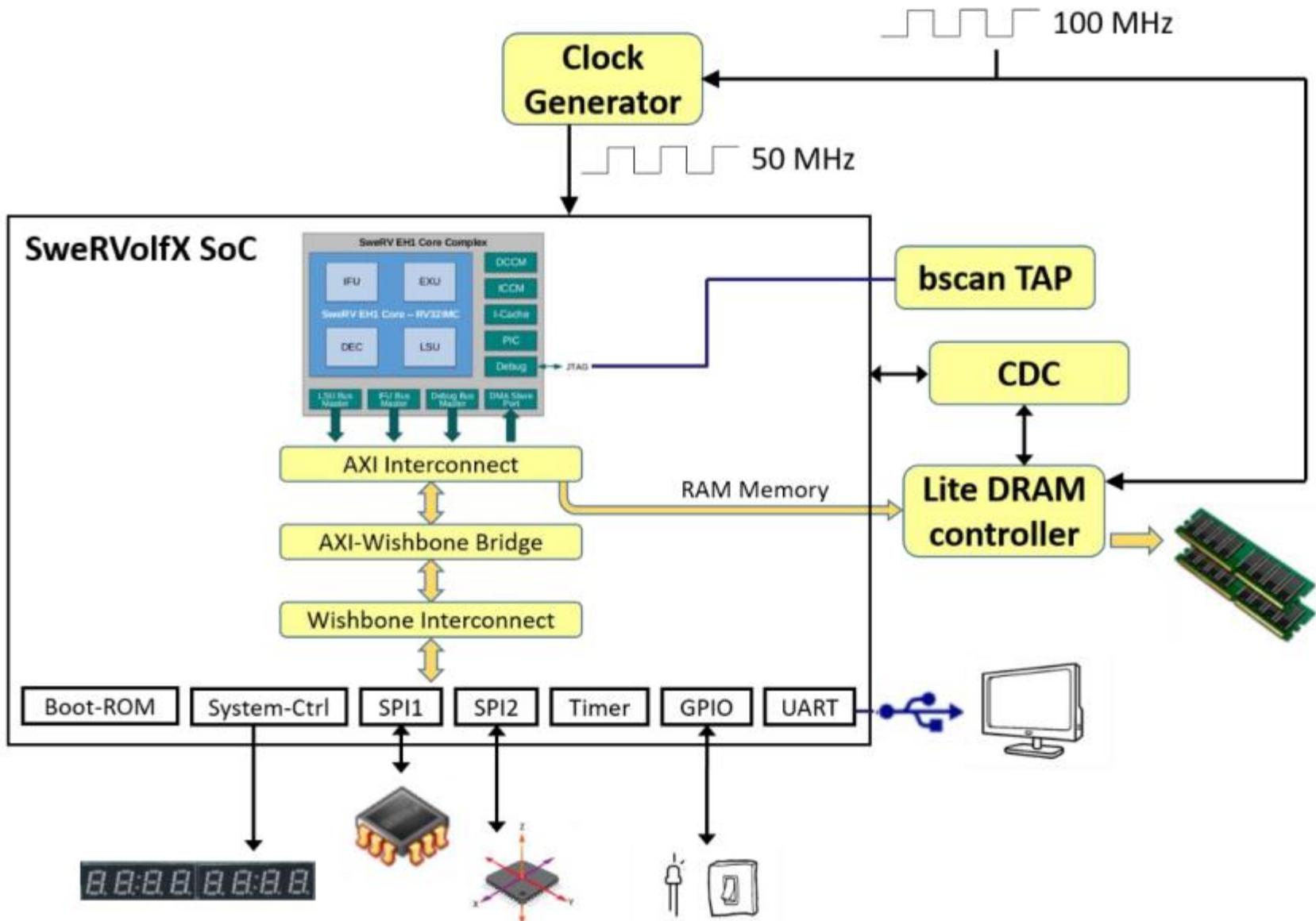


Figure 25. RVfpgaNexys

Programming Microprocessor – Instruction Set Architecture I



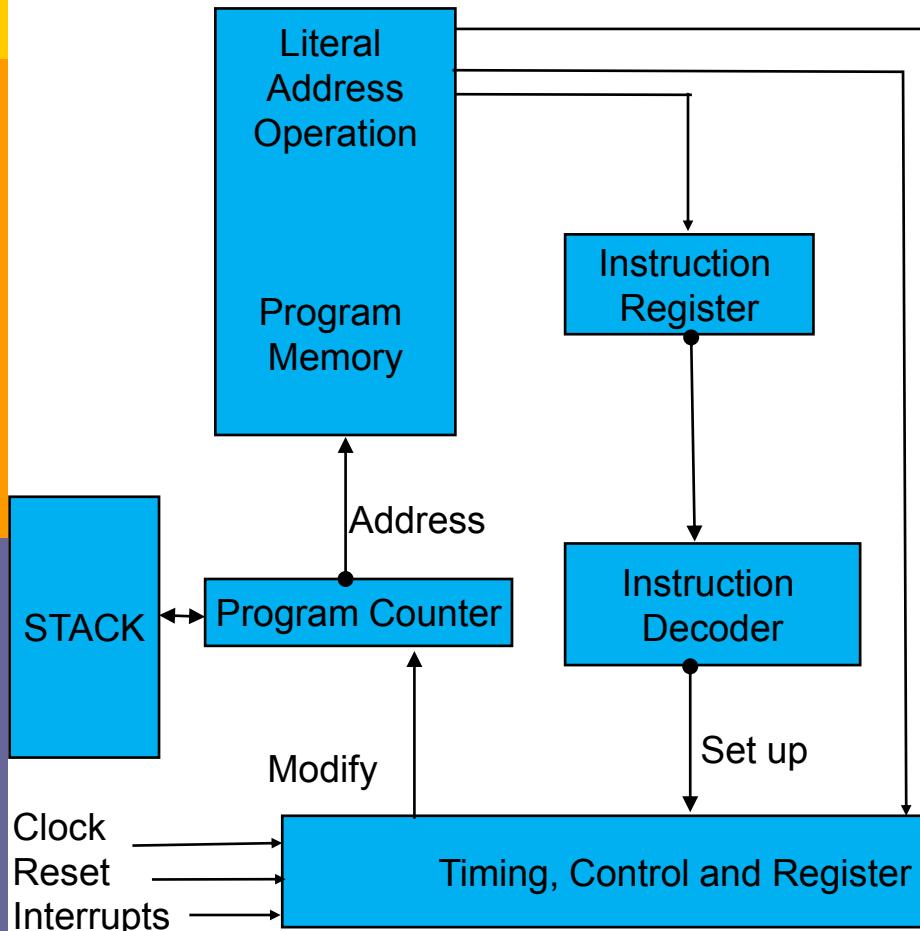
CS2053 Computer Architecture
Computer Science & Engineering
University of Moratuwa

Sulochana Sooriyaarachchi
Chathuranga Hettiarachchi

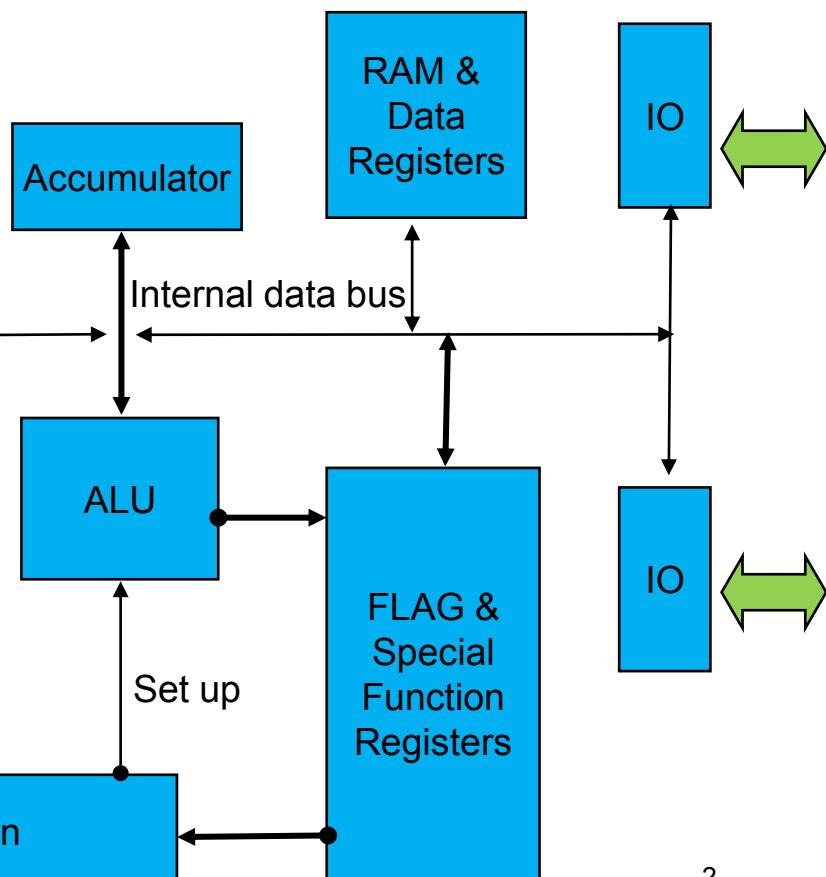
Slides adopted from Dr.Dilum Bandara

Blocks of a Microprocessor

Program Execution Section



Register Processing Section



So many types of devices!



NVIDIA GEFORCE RTX



Different demands

- ❖ Can you run all your desktop/laptop computer in the mobile phone as well? If not, why?
- ❖ Which battery lasts longer? Laptop battery or the mobile phone battery? Why?
- ❖ What computing devices are specialized and what are meant for general purpose tasks?

Major concerns of processor design

Available memory
Power consumption
Heat dissipation
Fabrication technology
Hardware complexity/ dedicated hardware
Connectivity
How to enable software development?

Levels of abstraction

What do you mean by x86 and x64?

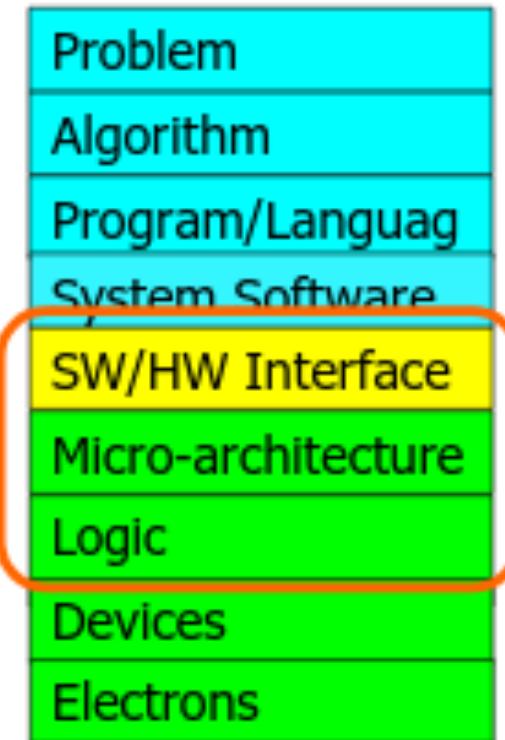
What are the other alternatives you know of?

We need programmable computers

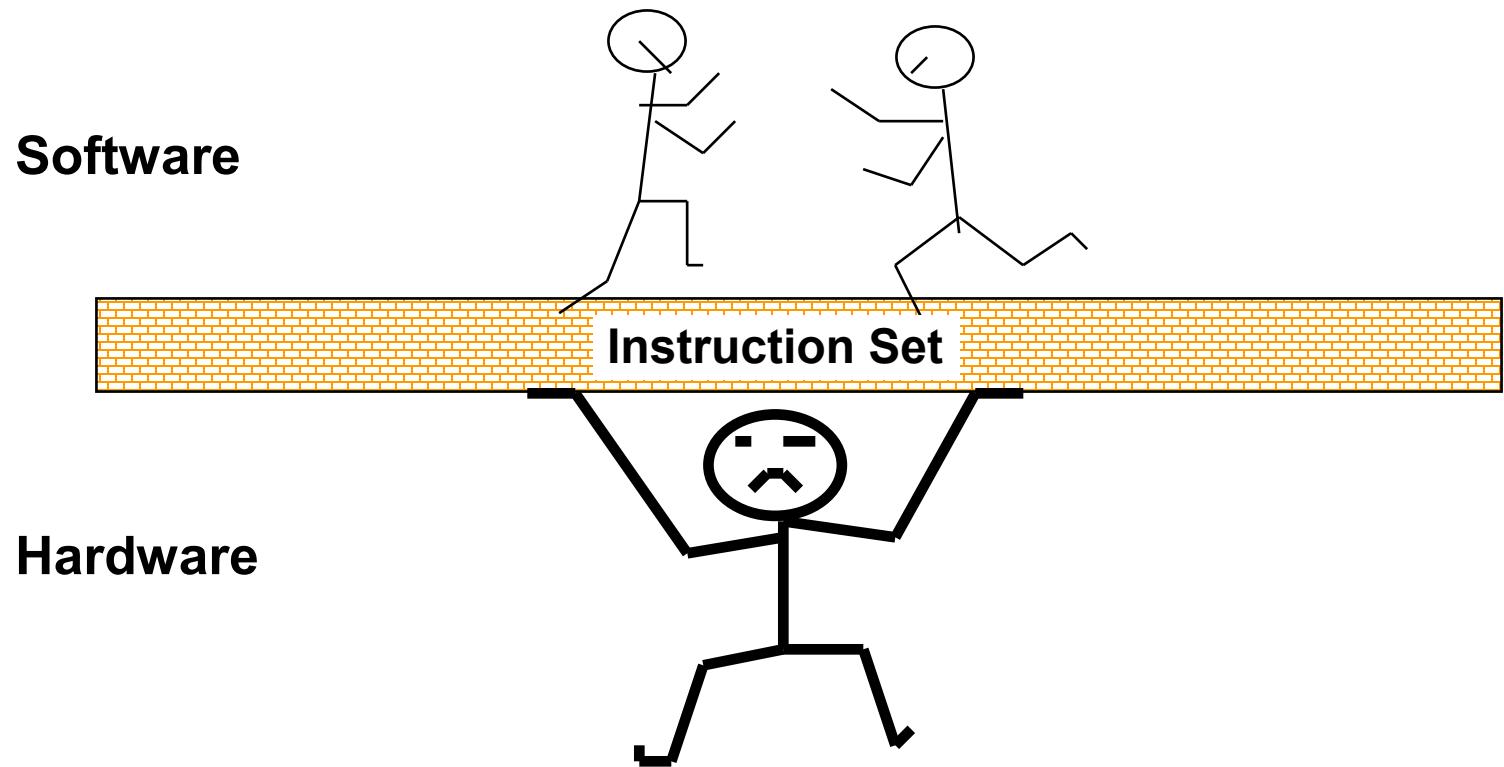
Computer is just a collection of hardware modules without a program.

How to make all the hardware features available to programmers?

- ❖ More dedicated hardware modules => More instructions
- ❖ Less hardware modules => Less instructions



Instruction Set Architecture (ISA)



Source: Computer Architecture: A Quantitative Approach, J. L. Hennessy & D. A. Patterson, 3rd Edition.

ISA (Cont.)

- Part of computer architecture related to programming
- Include native data types, instructions, registers, addressing modes, memory architecture, interrupt & exception handling, & external I/O
 - e.g., Registers x0 – x31 , PC, Control/Status registers
 - e.g., Instructions: li, addi, andi, beq
- ISA specifies the set of opcodes (i.e., machine language instructions) & native commands implemented by a particular processor

CISC ISA

When there is a large number of dedicated hardware co-processors/ modules, it needs to use more instructions .

- ❖ CISC : Complex Instruction Set Computer
- ❖ Consumes more power, because large number of modules are active
- ❖ Large number of instructions for dedicated hardware operations
- ❖ Difficult to program. Too many ways to choose from!

RISC ISA

When we have a minimal set of hardware we can use a simple set of instructions.

- ❖ RISC : Reduced Instruction Set Computer
- ❖ Consumes less power, hardware is simple and minimal
- ❖ Only a handful of instructions
- ❖ Easy to program

Well Known ISAs

- x86
 - Based on Intel 8086 CPU in 1978
 - Intel family, also followed by AMD
 - X86-64
 - 64-bit extensions
 - Proposed by AMD, also followed by Intel
- ARM
 - 32-bit & 64-bit
 - Initially by Acorn RISC Machine (ARM)
 - ARM Holding
- MIPS
 - 32-bit & 64-bit
 - By Microprocessor without Interlocked Pipeline Stages (MIPS) Technologies

Well Known ISAs (Cont.)

□ SPARC

- 32-bit & 64-bit
- By Sun Microsystems, today Oracle

□ PIC

- 8-bit to 32-bit
- By Microchip

□ Z80

- 8-bit
- By Zilog in 1976

□ Many extensions

- Intel – MMX, SSE, SSE2, AVX
- AMD – 3D Now!

What is Instruction Set Architecture (ISA)?

A layer of abstraction between hardware implementations and the what the processor can do.

Processor developer may choose to keep the hardware implementation details a secret.

Argue:

List of available assembly instructions and register/ memory map in a processor provides 'everything' necessary to get work done using a processor.

(OR) Programmer need to know some hardware aspects as well

A Good ISA

- Lasts through many implementations
 - Portability, compatibility
- Used in many different ways
 - Servers, desktop, laptop, mobile, tablet
- Provides convenient functions to higher layer
- Permit efficient implementation at lower layer

Introduction to RISC-V

- RISC-V is an open standard, in fact, the specification is public domain, and it has been managed since 2015 by the **RISC-V Foundation**, now called **RISC-V International**, a nonprofit organization promoting the development of hardware and software for RISC-V architectures
- Supported by 200 key players from research, academia, and industry, including Microchip, NXP, Samsung, Qualcomm, Micron, Google, Alibaba, Hitachi, Nvidia, Huawei, Western Digital, ETH Zurich, KU Leuven, UNLV, and UCM
- Modular rather than Incremental
 - x86 started with 80 instructions, and now has over 1300 (with 3600 machine opcodes)

Open Standard Specification

<https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>

RISC-V Base and Extensions

RISC-V Base and Extensions

Mnemonic	Description	Insn. Count
I	Base architecture	51
M	Integer multiply/divide	13
A	Atomic operations	22
F	Single-precision floating point	30
D	Double-precision floating point	32
C	Compressed instructions	36

FIGURE 2.38 The RISC-V instruction set architecture is divided into the base ISA, named I, and five standard extensions, M, A, F, D, and C.

Instruction Variants and Modules

□ Base options/variants

32 General Purpose Registers with ISA word size

RV32I RV64I and RV128I

RV32E (for embedded devices, only 16 registers available)

□ Modular rather than Incremental

- M Multiply/Division
 - A Atomic extensions
 - F Floating point
 - D Double precision Floating Point
- 
- RV32IMAFD or RV32IG
(G - General, same as MAFD)
- B Bit manipulation extension
 - C Compressed extension (More efficient encoding of instructions, reducing code size)
 - N User level interrupts
 - Proprietary extensions can be developed for your own applications.

□ For example, a 64-bit RISC-V implementation, including all four general ISA extensions plus *Bit Manipulation* and *User Level Interrupts*, is referred to as an RV64GBN ISA

- Can find out what modules are available by observing the “misa” status register

Example Implementations

- SweRV EH1 , SweRV EH2 , SweRV EL2
 - Some open-sourced hardware implementations by the company Western Digital
 - Apache 2.0 License : Use as-is or modified for free.
 - Can use in FPGAs since HDL code/IP is available
 - Modify and test
- Many companies have hardware boards with different RISC V processors as well
 - SiFive (Example: Red-V board)
 - Microchip

Encoding Instructions

- Various instruction types
- Limited word size for registers, addresses, and instructions
 - Consider 32bit words in RV32I
 - All the instructions are 32bits
 - Example : If we need to load an immediate value to 32-bit register, how to fit all opcode and operands within 32-bit instruction?
 - Work with small numbers
 - Make compromises

Instruction Formats (6 Types)

- R-Format: instructions using 3 *register* inputs
 - add, xor, mul — arithmetic/logical ops
- I-Format: instructions with *immediates*, loads
 - addi, lw, jalr, slli
- S-Format: **store** instructions: sw, sb
 - SB-Format: **branch** instructions: beq, bge
- U-Format: instructions with *upper* immediates
 - lui, auipc — upper immediate is 20-bits
 - UJ-Format: **jump** instructions: jal

Instruction Formats(Types)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode			R-type	
	imm[11:0]			rs1		funct3		rd		opcode			I-type	
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode			S-type	
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode			B-type	
	imm[31:12]							rd		opcode			U-type	
	imm[20 10:1 11 19:12]							rd		opcode			J-type	

- opcode 7bits
- funct 3 : 3 bit function
- funct 7 : 7 bit function
- rs1, rs2 : two source registers (5 bits each)
- rd : destination register (5 bits)

How many registers can be identified with 5 bits?

Instruction Format-Register Type

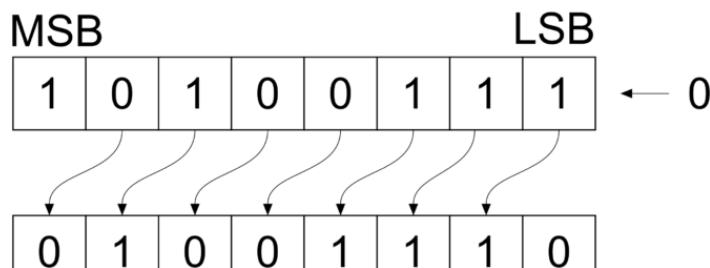
31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7	rs2	rs1	funct3		rd	opcode							R-type	
imm[11:0]	rs1	funct3	rd		opcode								I-type	
imm[11:5]	rs2	rs1	funct3		imm[4:0]	opcode								S-type
imm[12 10:5]	rs2	rs1	funct3		imm[4:1 11]	opcode								B-type
imm[31:12]		rd		opcode										U-type
imm[20 10:1 11 19:12]		rd		opcode										J-type

RV32I Base Integer Instructions (Register Type)

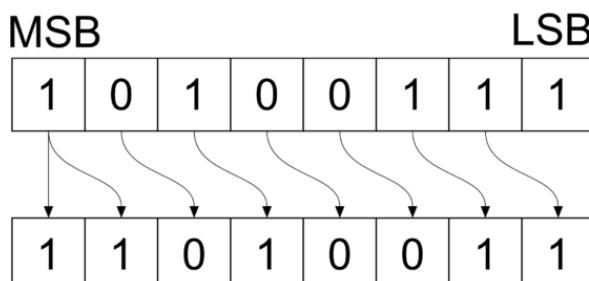
Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

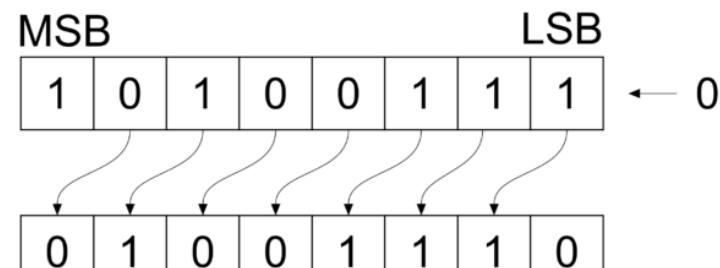
Logical Shift vs Arithmetic Shift



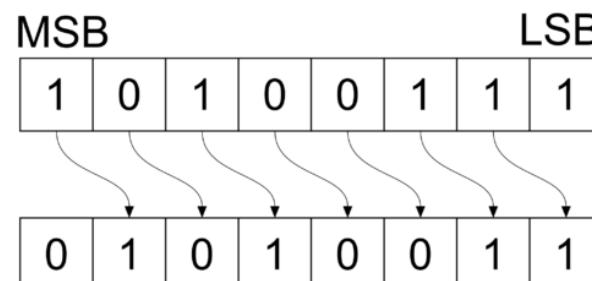
Left Arithmetic Shift



Right Arithmetic Shift



Left Logical Shift

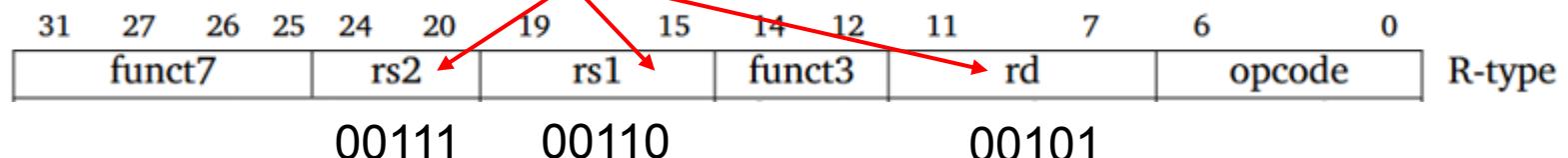


Right Logical Shift

- **Right** Arithmetic shift preserve sign bit, whereas **Right** Logical shift can not preserve sign bit.
 - Arithmetic shift perform multiplication and division operation, whereas Logical shift perform only multiplication operation.
 - Arithmetic shift is used for signed interpretation, whereas Logical shift is used for unsigned interpretation.

Register Type Instruction Example

RISCV Instruction: add x5, x6, x7



funct7	rs2	rs1	funct3	rd	opcode
0 0 0 0 0 0 0			0 0 0		0 1 1 0 0 1 1
	0 0 1 1 1	0 0 1 1 0		0 0 1 0 1	
0 0 0 0 0 0 0 0 1 1 1 0 0 1 1 0 0 0 0 0 0 0 1 0 1 0 1 1 0 0 1 1	0	0	7	3	2 B 3

Machine Instruction: **0x007302B3**

-
- Manipulating values in registers can be done
 - Yet, how to get any value to registers in first place?
 - Immediate values are required.
 - Can we use rs1 or rs2 fields for an immediate value parameter?
 - rs1 and rs2 are just 5 bits each
 - Not enough to hold sufficiently large values
 - Dedicate both funct7 and rs2 for representing bigger immediate values
 - Immediate type instructions

Instruction Format-Immediate Type

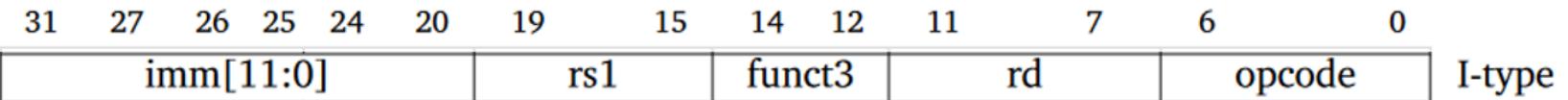
31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode		R-type		
imm[11:0]				rs1		funct3		rd		opcode		I-type		
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type		
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		B-type		
		imm[31:12]						rd		opcode		U-type		
		imm[20 10:1 11 19:12]						rd		opcode		J-type		

RV32I Base Integer Instructions (Immediate Type)

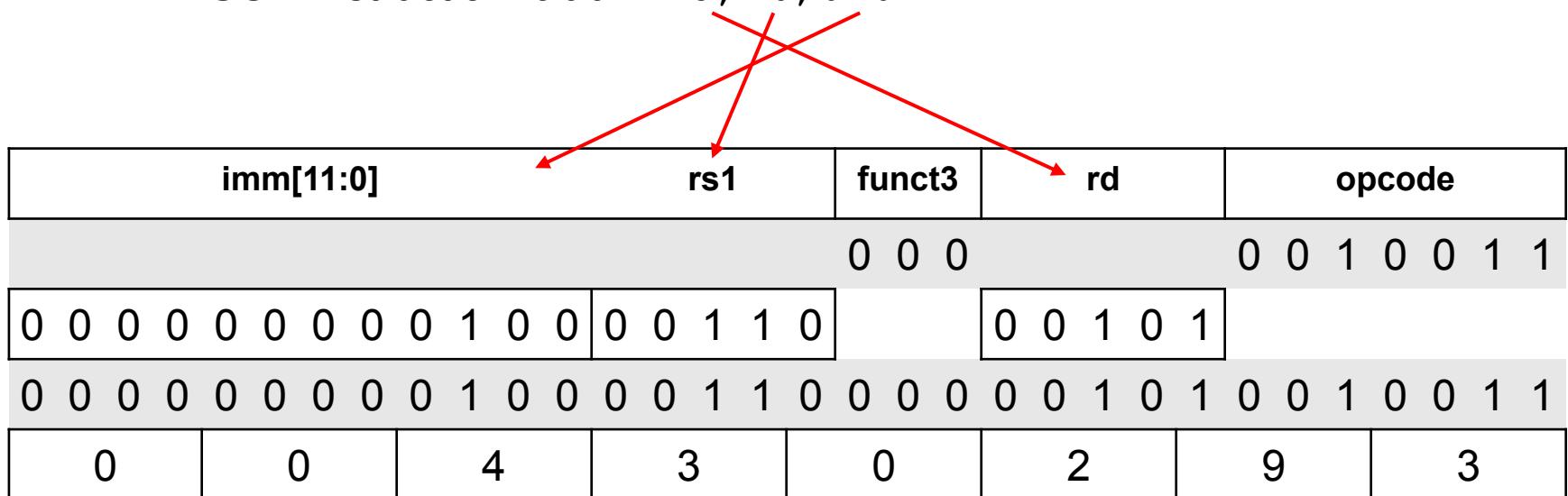
Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends

Instruction	Format	immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
ld (load doubleword)	I	address	reg	011	reg	0000011

Immediate Type Example



RISCV Instruction: addi x5, x6, 0x04



Machine Instruction: **0x00430293**

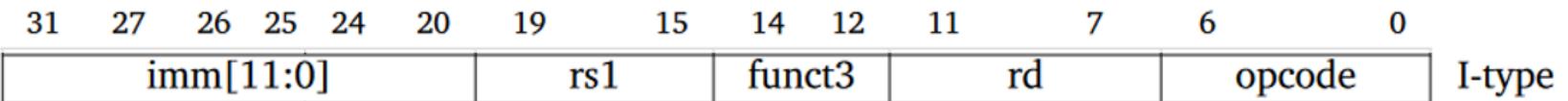
Immediate Values

- immediate(12): 12-bit number
 - All computations done in words, so 12-bit immediate must be extended to 32 bits
 - always sign-extended to 32-bits before use in an arithmetic operation
- Q: What is the largest Immediate value you can use with addi instruction?

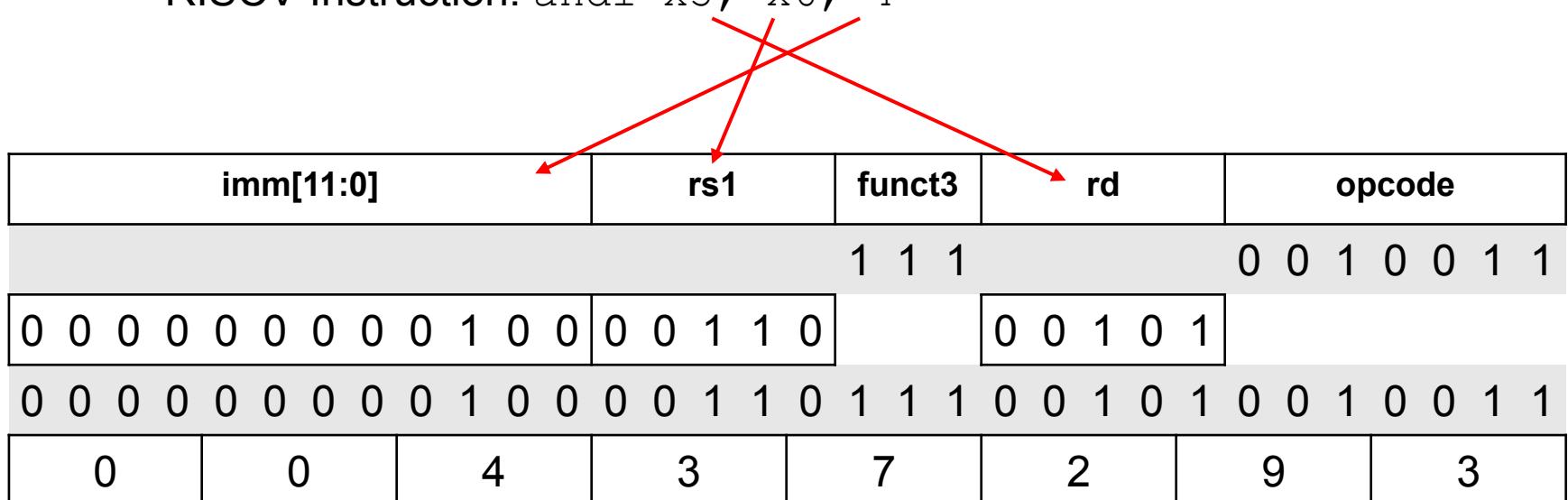
2^{12} Range of Two's complement. $[-2^{11}, +2^{11}]$

Immediate value should be between -2048 and **2047**

Immediate Type Example 2



RISCV Instruction: andi x5, x6, 4



Machine Instruction: **0x00437293**

Exercise in Moodle

- Assume x6 Register holds the value **0xFFFFFFFF**. The computer represent numbers in two's complement format.

- If you execute **andi x5,x6,-4** instruction, what will be the binary format of the following?
 - (a) Immediate value (12bits) encoded into the instruction?
 - (b) Value at x5 at the end of the instruction execution?

RISC –V Registers

32 32-bit registers

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary variables
s0/fp	x8	Saved variable / Frame pointer
s1	x9	Saved variable
a0-1	x10-11	Function arguments / Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved variables
t3-6	x28-31	Temporary variables

RISC V Instruction Set

Common RISC-V Assembly Instructions & Pseudoinstructions

RISC-V Assembly	Description	Operation
<code>add s0, s1, s2</code>	Add	$s0 = s1 + s2$
<code>sub s0, s1, s2</code>	Subtract	$s0 = s1 - s2$
<code>addi t3, t1, -10</code>	Add immediate	$t3 = t1 - 10$
<code>mul t0, t2, t3</code>	32-bit multiply	$t0 = t2 * t3$
<code>div s9, t5, t6</code>	Division	$t9 = t5 / t6$
<code>rem s4, s1, s2</code>	Remainder	$s4 = s1 \% s2$
<code>and t0, t1, t2</code>	Bit-wise AND	$t0 = t1 \& t2$
<code>or t0, t1, t5</code>	Bit-wise OR	$t0 = t1 t5$
<code>xor s3, s4, s5</code>	Bit-wise XOR	$s3 = s4 ^ s5$
<code>andi t1, t2, 0xFFB</code>	Bit-wise AND immediate	$t1 = t2 \& 0xFFFFFFFFB$
<code>ori t0, t1, 0x2C</code>	Bit-wise OR immediate	$t0 = t1 0x2C$
<code>xori s3, s4, 0xAB</code>	Bit-wise XOR immediate	$s3 = s4 ^ 0xFFFFFABC$
<code>sll t0, t1, t2</code>	Shift left logical	$t0 = t1 << t2$
<code>srl t0, t1, t5</code>	Shift right logical	$t0 = t1 >> t5$
<code>sra s3, s4, s5</code>	Shift right arithmetic	$s3 = s4 >>> s5$
<code>slli t1, t2, 30</code>	Shift left logical immediate	$t1 = t2 << 30$
<code>srlti t0, t1, 5</code>	Shift right logical immediate	$t0 = t1 >> 5$
<code>srai s3, s4, 31</code>	Shift right arithmetic immediate	$s3 = s4 >>> 31$

RISC V Instruction Set

Common RISC-V Assembly Instructions & Pseudoinstructions (continued)

RISC-V Assembly	Description	Operation
<code>lw s7, 0x2C(t1)</code>	Load word	$s7 = \text{memory}[t1+0x2C]$
<code>lh s5, 0x5A(s3)</code>	Load half-word	$s5 = \text{SignExt}(\text{memory}[s3+0x5A]_{15:0})$
<code>lb s1, -3(t4)</code>	Load byte	$s1 = \text{SignExt}(\text{memory}[t4-3]_{7:0})$
<code>sw t2, 0x7C(t1)</code>	Store word	$\text{memory}[t1+0x7C] = t2$
<code>sh t3, 22(s3)</code>	Store half-word	$\text{memory}[s3+22]_{15:0} = t3_{15:0}$
<code>sb t4, 5(s4)</code>	Store byte	$\text{memory}[s4+5]_{7:0} = t4_{7:0}$
<code>beq s1, s2, L1</code>	Branch if equal	$\text{if } (s1 == s2), \text{PC} = L1$
<code>bne t3, t4, Loop</code>	Branch if not equal	$\text{if } (s1 != s2), \text{PC} = \text{Loop}$
<code>blt t4, t5, L3</code>	Branch if less than	$\text{if } (t4 < t5), \text{PC} = L3$
<code>bge s8, s9, Done</code>	Branch if not equal	$\text{if } (s8 >= s9), \text{PC} = \text{Done}$
<code>li s1, 0xABCD12</code>	Load immediate	$s1 = 0xABCD12$
<code>la s1, A</code>	Load address	$s1 = \text{Variable A's memory address (location)}$
<code>nop</code>	Nop	no operation
<code>mv s3, s7</code>	Move	$s3 = s7$
<code>not t1, t2</code>	Not (Invert)	$t1 = \sim t2$
<code>neg s1, s3</code>	Negate	$s1 = -s3$
<code>j Label</code>	Jump	$\text{PC} = \text{Label}$
<code>jal L7</code>	Jump and link	$\text{PC} = L7; ra = \text{PC} + 4$
<code>jr s1</code>	Jump register	$\text{PC} = s1$

Next Lecture:

❑ More on ISA

- Store Branch UpperImmediate and Jump instructions.
- Pseudo Instructions
- Compressed Instructions
- (Addressing Modes)

Programming Microprocessor – Instruction Set Architecture II



CS2053 Computer Architecture
Computer Science & Engineering
University of Moratuwa

Sulochana Sooriyaarachchi
Chathuranga Hettiarachchi

Slides adopted from Dr.Dilum Bandara

Encoding Instructions

- Various instruction types
- Limited word size for registers, addresses, and instructions
 - Consider 32bit words in RV32I
 - All the instructions are 32bits
 - Example : If we need to load an immediate value to 32-bit register, how to fit all opcode and operands within 32-bit instruction?
 - Work with small numbers
 - Make compromises

Types of RISC-V Instructions

- Register Type
 - Source Registers
 - Destination Register
- Immediate Type
 - Registers
 - Immediate Value
- Load type
 - Base memory Address
 - Destination Register
 - Offset from base memory address
- Store Type
 - Memory address
 - Value to be stored
- Branching Type
 - Two Registers for comparison
 - Offset from current PC to branching destination
- Upper-Immediate Type
 - Destination register
 - Upper Immediate Value
- Jump Type
 - Register to backup the current PC+4
 - Offset to jump destination

Instruction Formats(Types)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
	funct7		rs2		rs1		funct3		rd		opcode		R-type	
	imm[11:0]				rs1		funct3		rd		opcode		I-type	
	imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type	
	imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		B-type	
			imm[31:12]						rd		opcode		U-type	
			imm[20 10:1 11 19:12]						rd		opcode		J-type	

Fields

- Opcode : 7bits
 - funct 3 : 3 bit function
 - funct 7 : 7 bit function
 - rs1, rs2 : two source registers (5 bits each)
 - rd : destination register (5 bits)

Instruction Formats (6 Types)

- R-Format: instructions using 3 *register* inputs
 - add, xor, mul —arithmetic/logical ops
- I-Format: instructions with *immediates*, loads
 - addi, lw, jalr, slli
- S-Format: *store* instructions: sw, sb
 - SB-Format: *branch* instructions: beq, bge
- U-Format: instructions with *upper* immediates
 - lui, auipc —upper immediate is 20-bits
 - UJ-Format: *jump* instructions: jal

Instruction Format-Register Type

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7	rs2		rs1		funct3		rd		opcode		R-type			
imm[11:0]			rs1		funct3		rd		opcode		I-type			
imm[11:5]	rs2		rs1		funct3		imm[4:0]		opcode		S-type			
imm[12 10:5]	rs2		rs1		funct3		imm[4:1 11]		opcode		B-type			
		imm[31:12]					rd		opcode		U-type			
		imm[20 10:1 11 19:12]					rd		opcode		J-type			

RV32I Base Integer Instructions (Register Type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends

For the complete standard specifications:

<https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>

Instruction Format-Immediate Type

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode		R-type		
imm[11:0]		rs1		funct3		rd		opcode		I-type				
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode								S-type	
imm[12 10:5]	rs2	rs1	funct3	imm[4:1 11]	opcode								B-type	
imm[31:12]						rd		opcode		U-type				
imm[20 10:1 11 19:12]						rd		opcode		J-type				

RV32I Base Integer Instructions(Immediate Type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
addi	ADD Immediate	I	0010011	0x0		$rd = rs1 + imm$	
xori	XOR Immediate	I	0010011	0x4		$rd = rs1 \wedge imm$	
ori	OR Immediate	I	0010011	0x6		$rd = rs1 imm$	
andi	AND Immediate	I	0010011	0x7		$rd = rs1 \& imm$	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	$rd = rs1 << imm[0:4]$	
srlti	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	$rd = rs1 >> imm[0:4]$	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	$rd = rs1 >> imm[0:4]$	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		$rd = (rs1 < imm)?1:0$	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		$rd = (rs1 < imm)?1:0$	zero-extends
lb	Load Byte	I	0000011	0x0		$rd = M[rs1+imm][0:7]$	
lh	Load Half	I	0000011	0x1		$rd = M[rs1+imm][0:15]$	
lw	Load Word	I	0000011	0x2		$rd = M[rs1+imm][0:31]$	
lbu	Load Byte (U)	I	0000011	0x4		$rd = M[rs1+imm][0:7]$	zero-extends
lhu	Load Half (U)	I	0000011	0x5		$rd = M[rs1+imm][0:15]$	zero-extends

Discuss later today

Upper Immediate Instructions

Instruction Format-Upper Immediate

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
	funct7			rs2		rs1		funct3		rd		opcode		R-type
				imm[11:0]			rs1		funct3		rd		opcode	I-type
				imm[11:5]		rs2		rs1		funct3	imm[4:0]		opcode	S-type
				imm[12:10:5]		rs2		rs1		funct3	imm[4:1 11]		opcode	B-type
						imm[31:12]				rd		opcode		U-type
						imm[20:10:1 11:19:12]				rd		opcode		J-type

lui	Load Upper Imm	U	0110111				rd = imm << 12		
auipc	Add Upper Imm to PC	U	0010111				rd = PC + (imm << 12)		

Load Upper Immediate (lui)

lui t1,0x70070

Fill up the upper 20 bits of destination register with immediate value

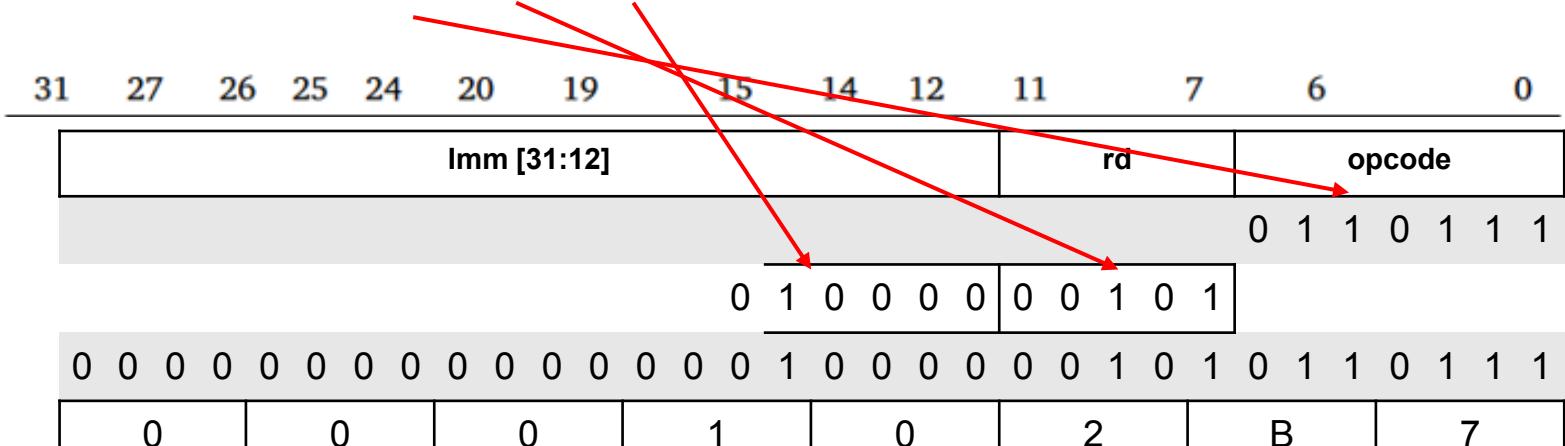
Add Upper Immediate value and Program Counter (auipc)

auipc a0,0x2

Fill the upper 20 bits of destination register with immediate value

U Type Instruction Example

RISCV Instruction: lui x5, 0x10



Machine Instruction: **0x000102B7**

Result ?? Rd = imm << 12

Register x5 (t0) = 0x0010 <<12
= 0x00010000

We can work with 20bit
immediate values!

RISC-V U-type instructions usage

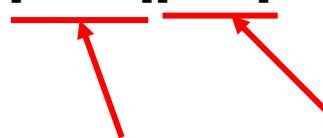
Some instructions are difficult to get done with a single instruction.

Example:

Load immediate **value 0x700707FF** (or any 32 bits long number) to x6 register

Since we must encode the instruction opcode in 32 bits as well, it is impossible to do in a single instruction.

Split the 32bit value to two parts. [31:12][11:0]



Need two instruction to do this:

lui x6, 0x70070 (U type instruction)
addi x6, 0x7FF

Upper Part
(20 bits) Lower Part
(12 bits)

Example 1:Load 0x000707FF to t0

- Split to two numbers 0x00070000 + 0x000007FF

- 0x0070 to t0 upper bits

- lui t0,0x70

0x10 = 0b01110000 << 12

t0 = 0b 0000 0000 0000 0111 0000 0000 0000 0000

0x7FF = 0b 0000 0000 0000 0000 0000 0111 1111 1111

- addi t0,t0, 0x7FF

= 0b 0000 0000 0000 0111 0000 0111 1111 1111

= 0x000707FF

So , we loaded 0x000707FF to t0 32bit register

Example 2: Load 0x0000FFFF to t0

- 0x7FF is the max value for addi instruction
- Split to two numbers 0x00010000 + (-1)
 - 0x0010 to t0 upper bits

□ lui t0,0x10

0x10 = 0b00010000 << 12

t0 = 0b 0000 0000 0000 0001 0000 0000 0000 0000

-1 = 0b 1111 1111 1111 1111 1111 1111 1111 1111

□ addi t0,t0, -1

= 0b 0000 0000 0000 0000 1111 1111 1111 1111

= 0x0000FFFF

So , we loaded 0x0000FFFF to t0 32bit register

How about loading 0x0000F800 ???

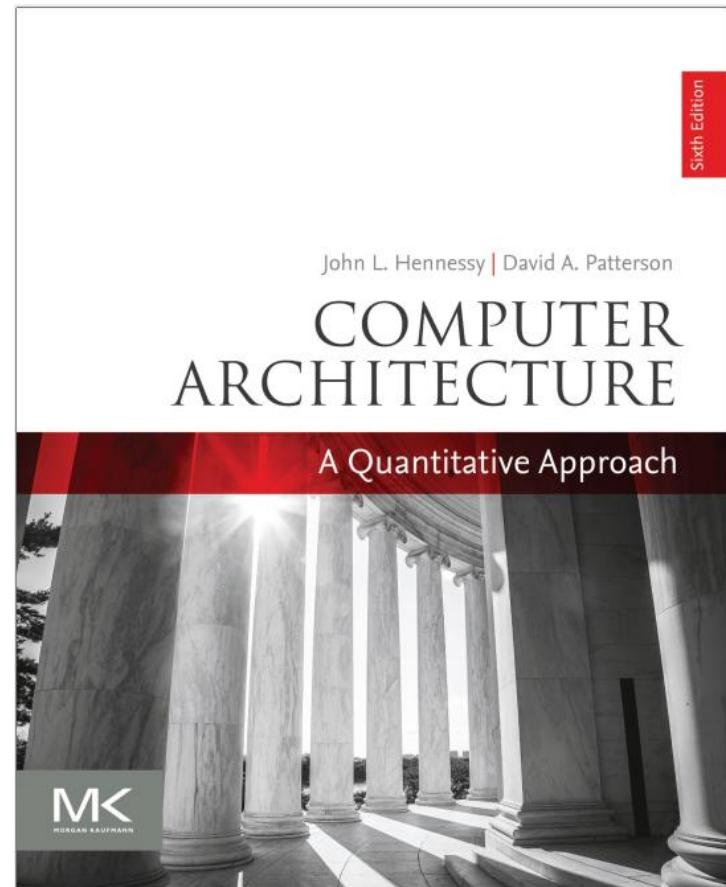
Meh..

- Why so complicated? Difficult to write programs
 - Who designed these??
 - Can't we make it easier???
-
- Early days, the instruction sets used to be relatively easy to understand and write.
 - Microchip PIC instruction set.
 - Z80 instructions.
-
- We need to understand real world demands

How to identify a good ISA?

- Meet functional requirements
- Are the real-world computations cheaper, faster and energy efficient?
- Have less “code density”, so the memory for storing and transferring programs will be less
- The base instruction set is selected based on the real-world performance measures.
 - Run a collection of real-world programs with and without a specific instruction
 - “Benchmark” Example: SPEC benchmark

-
- Good time to start reading the book.
 - Chapter 1



Assembly Programmers View

□ Writing code ?

- Usually, we write codes in C or Python etc.
- Compilers and Assemblers can take care of converting high-level codes to machine codes.

□ Let us define some ‘high-level assembly’ instructions as well to make it easier to write assembly codes

Assembly Programmers View

□ Example:

- Loading 32bit values to registers using base instructions is complicated.
- Let our assembler handle the complexity
- We shall write intuitive instructions

□ Pseudo Instructions

li t1, 0x7FFFFFFF

- Load immediate value to t1 register (assembler will convert this to following two “base instructions”)

80000337 lui t1,0x80000

fff30313 addi t1,t1,-1 # 7fffffff

- List of pseudo instructions can be found in riscv-card.

Some RISC V Pseudo Instructions

nop	addi x0, x0, 0	No operation
li rd, immediate	<i>Myriad sequences</i>	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
negw rd, rs	subw rd, x0, rs	Two's complement word
sext.w rd, rs	addiw rd, rs, 0	Sign extend word
seqz rd, rs	sltiu rd, rs, 1	Set if = zero
snez rd, rs	sltu rd, x0, rs	Set if \neq zero
sltz rd, rs	slt rd, rs, x0	Set if < zero
sgtz rd, rs	slt rd, x0, rs	Set if > zero

Some more are available...

Memory Related Instructions

Registers vs Memory (Norms)

□ General Purpose Registers

- Named (norm)
- Limited
- Located very close to the ALU
- Expensive

- Fast

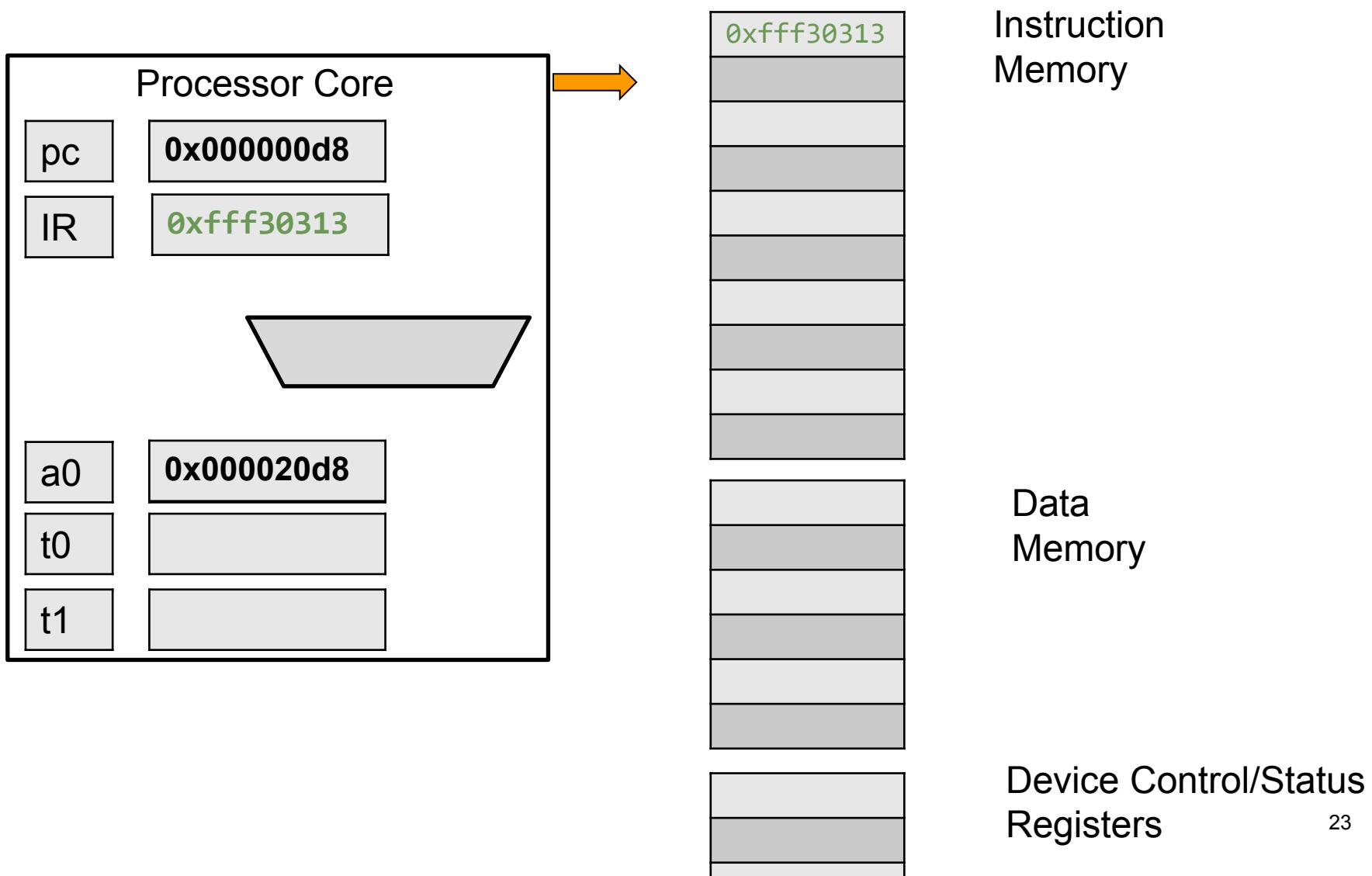
- Locations are not related

□ Memory

- Addressed
- Larger/Extended
- Away from ALU

- Cheaper compared to registers
- Slow compared to registers
- Memory “map”: relative locations (Viewed as an array or a contiguous space)

Registers vs Memory



Instructions for memory access

- Load values from memory to registers
- Store values to memory from registers

- Address? (Byte addresses)

- How many addresses?

- RV32I => Address bus is 32bits

- Maximum Memory Space: 2^{32}

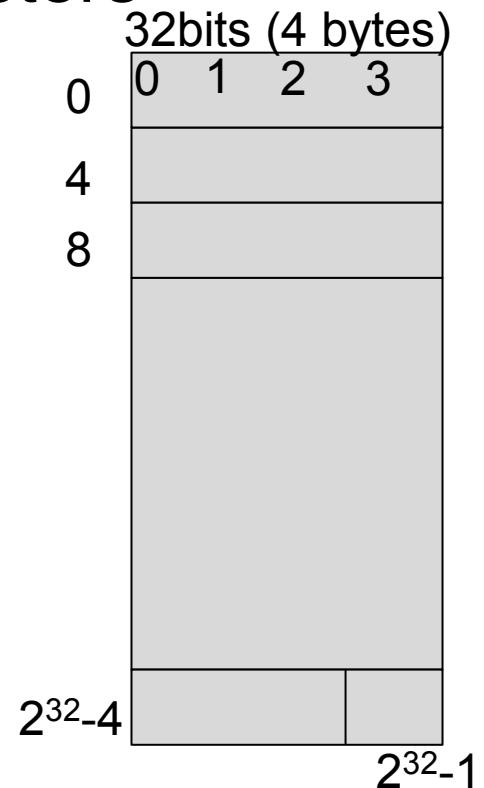
- 4294967296

- $2^{32} / 1024 / 1024 / 1024 = 4GB$

- RV64I => Address bus is 64bits

- Values?

- 32 bits



Load and Store Instructions

31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd		opcode		R-type	
imm[11:0]		rs1		funct3		rd		opcode		I-type			
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type	
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		B-type	
		imm[31:12]						rd		opcode		U-type	
		imm[20 10:1 11 19:12]						rd		opcode		J-type	

Load: copy a value from memory to register *rd* : Immediate type
Source is a memory address

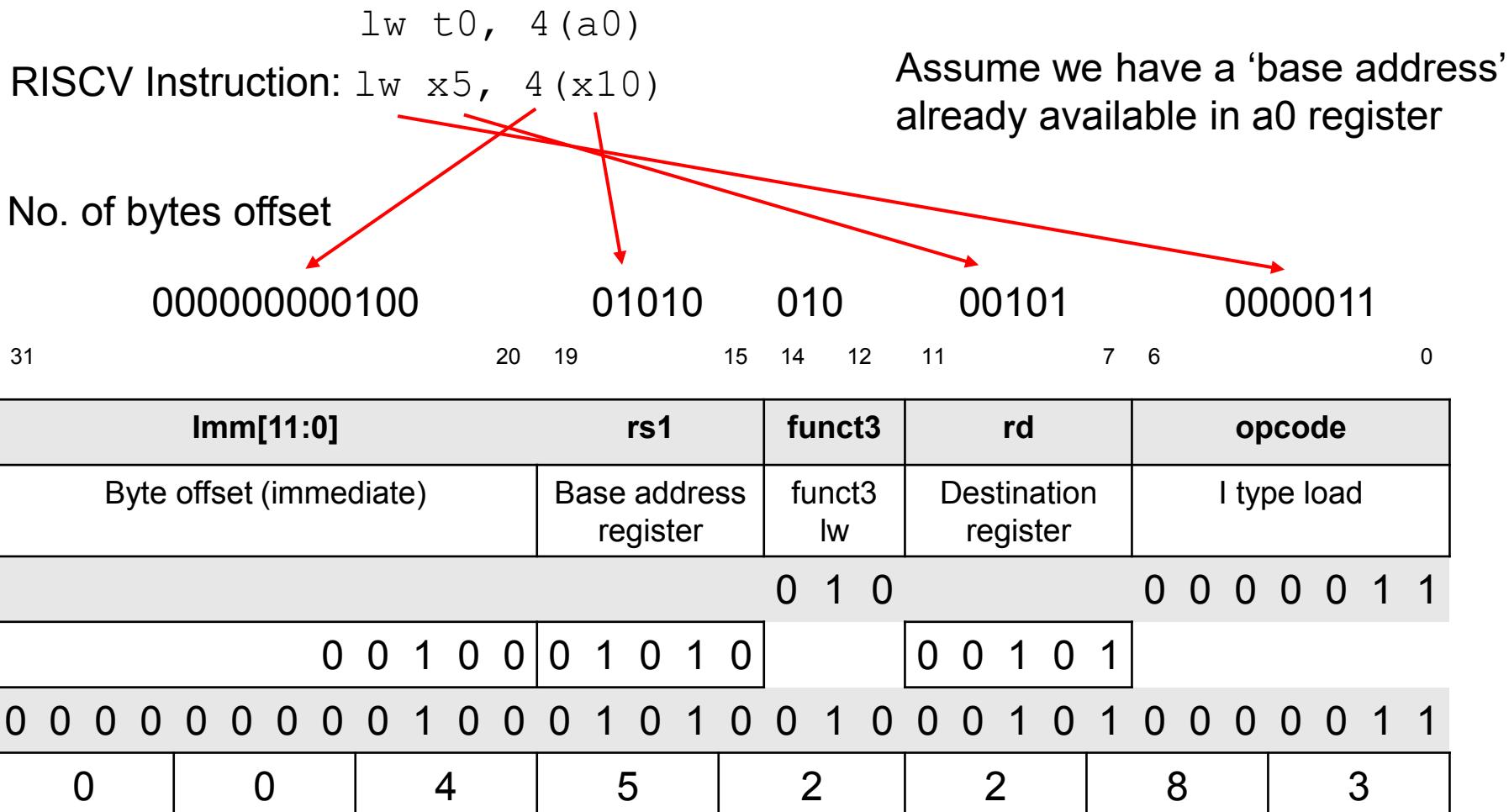
Store: copy the value in register *rs2* to memory : Store type

Need the proper 32bit address

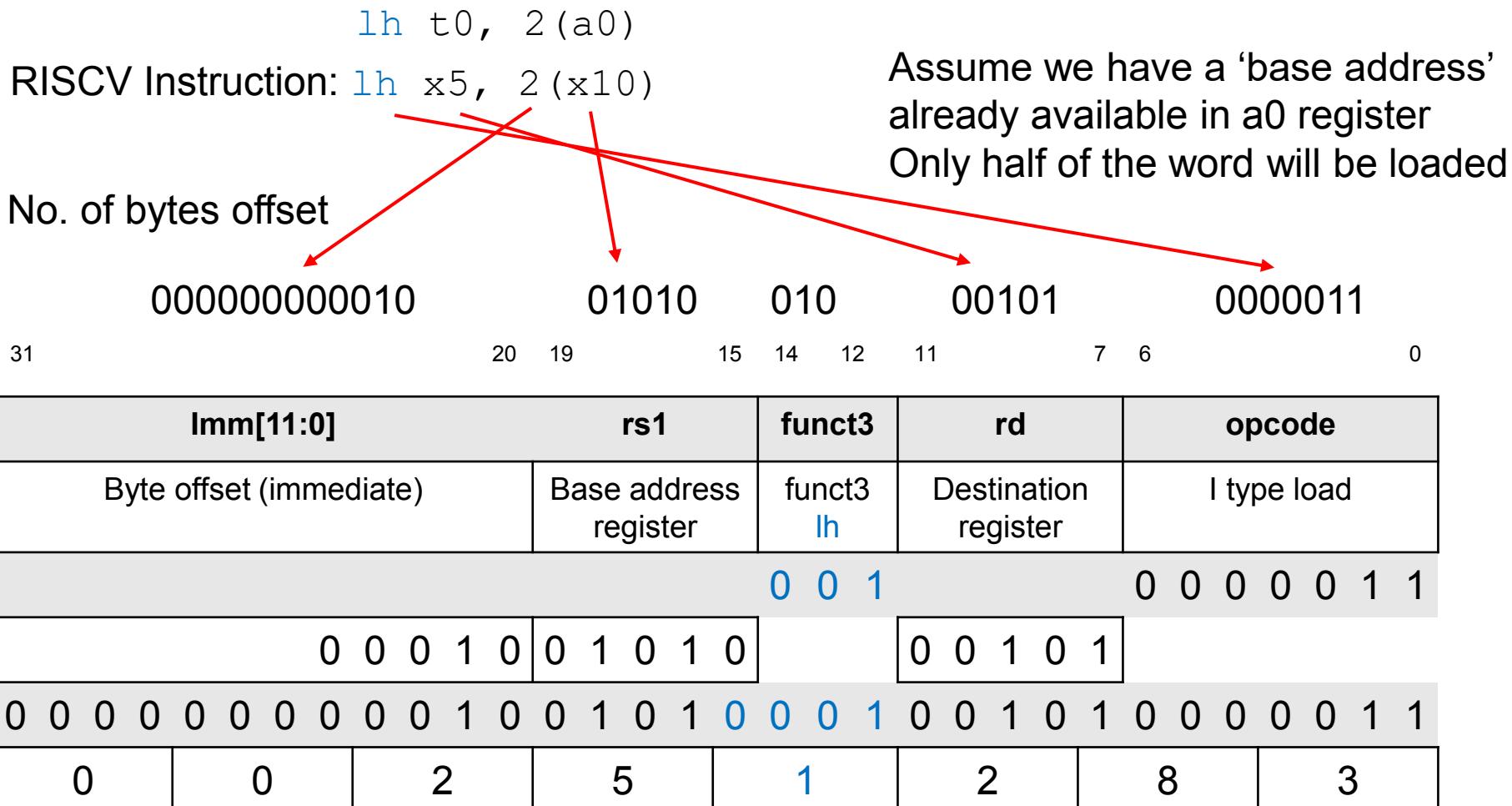
RV32I Base Integer Instructions (Immediate type and Store type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
1b	Load Byte	I	0000011	0x0		$rd = M[rs1+imm][0:7]$	
1h	Load Half	I	0000011	0x1		$rd = M[rs1+imm][0:15]$	
1w	Load Word	I	0000011	0x2		$rd = M[rs1+imm][0:31]$	
1bu	Load Byte (U)	I	0000011	0x4		$rd = M[rs1+imm][0:7]$	zero-extends
1hu	Load Half (U)	I	0000011	0x5		$rd = M[rs1+imm][0:15]$	zero-extends
sb	Store Byte	S	0100011	0x0		$M[rs1+imm][0:7] = rs2[0:7]$	
sh	Store Half	S	0100011	0x1		$M[rs1+imm][0:15] = rs2[0:15]$	
sw	Store Word	S	0100011	0x2		$M[rs1+imm][0:31] = rs2[0:31]$	

Load Word Instruction (I type)



Load Half Instruction (I type)



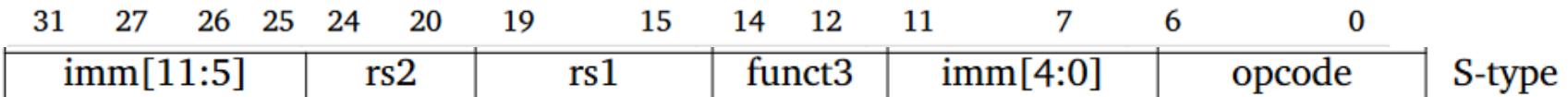
Store Instructions (S type)

- To store a value in memory, you need three things
 - Memory location to save to
 - rs1: base memory address
 - Immediate memory location offset
 - Value to save
 - rs2: contains data to be stored
 - Choice: Move rs2 to different place or split immediate offset
 - Example SW x14, 8(x2)
rs2 register with value Immediate offset rs1 base addr

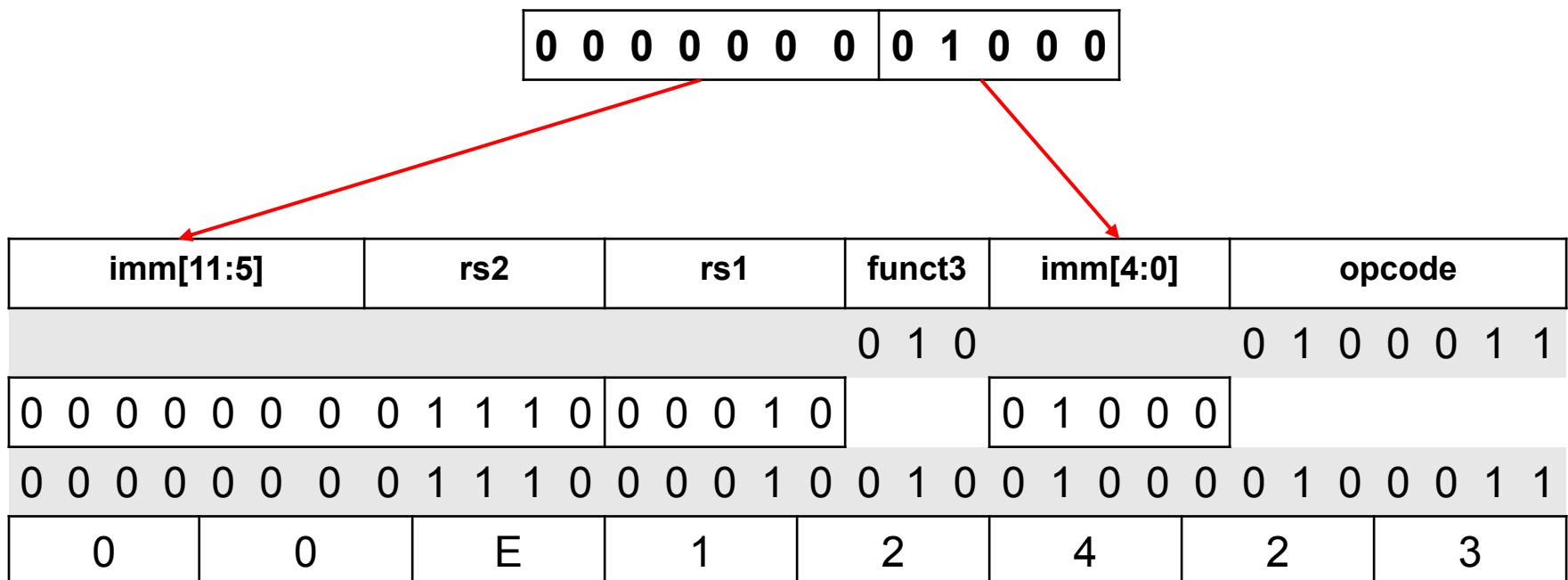
rs2 register with value

Immediate offset

Store Type Example 1



RISCV Instruction: sw x14, 8 (x2)



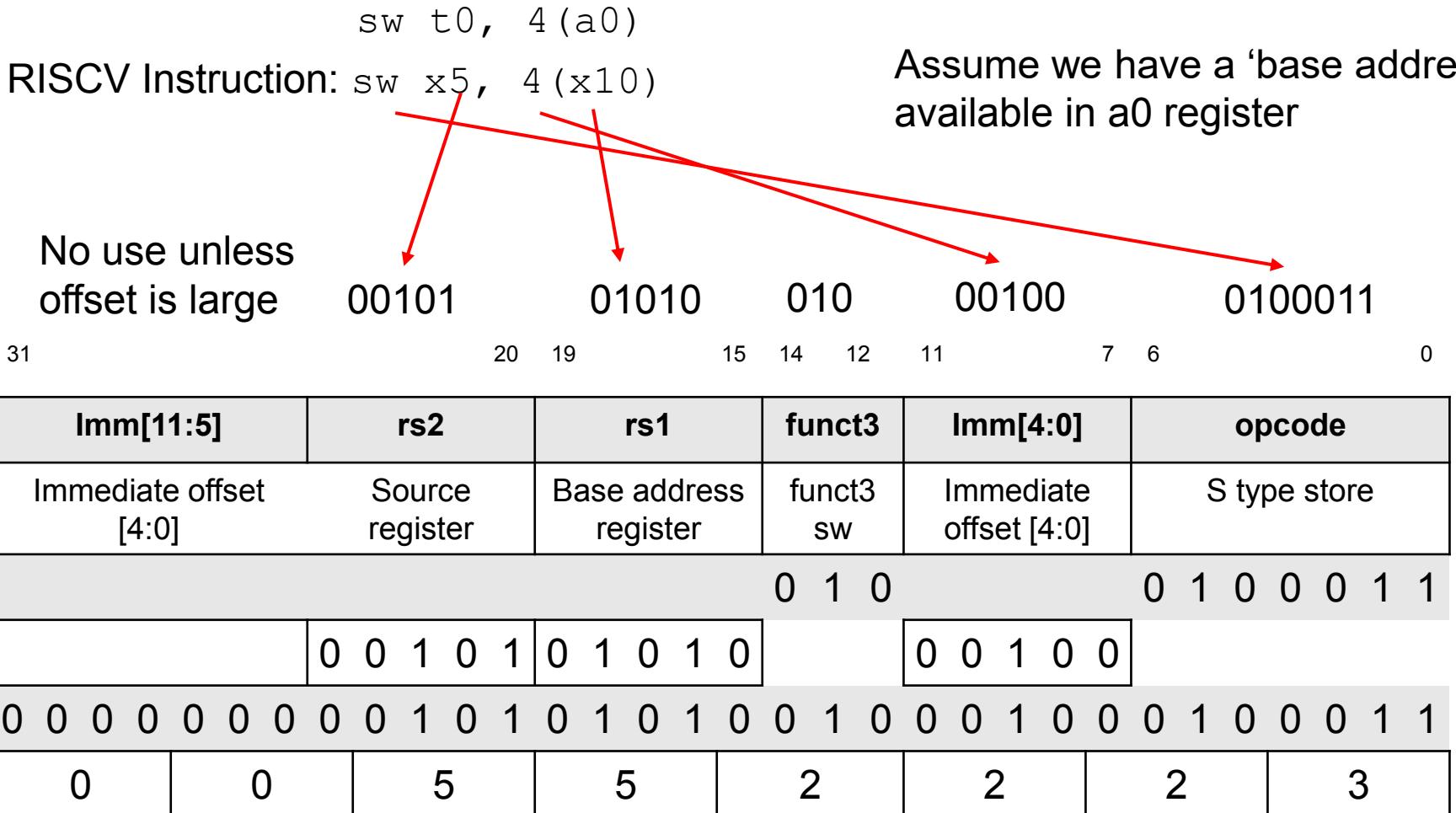
Machine Instruction: **0x00E12423**

Store Instructions (S type)

RISCV Instruction: sw x5, 4(x10)

Assume we have a ‘base address’ available in a0 register

No use unless
offset is large



Machine Instruction: 0x00552223

Programmers Perspective

- We don't know memory addresses.
- Let us write memory **position independent code**
 - Ask the assembler to declare a data section along with the instruction sequence
 - auipc instruction allows to address relative to program counter.

```
.data
    A: .word 0x1F2F3F4F
.text
.globl main

main:
    la a0, A
    # Pseudo instruction la (How it works? will discuss in next slide)
    # Load the address of symbol to the register

    li t1, 0x1E2E3E4E

    lw t0, 0(a0)
    # Get the value in a0 address, byte offset 0 and load it to t0 register

    sw t1, 0(a0)
    # Store value in t1 to A, at byte offset of 0

    ret
.end
```

auipc instruction and la pseudo instruction

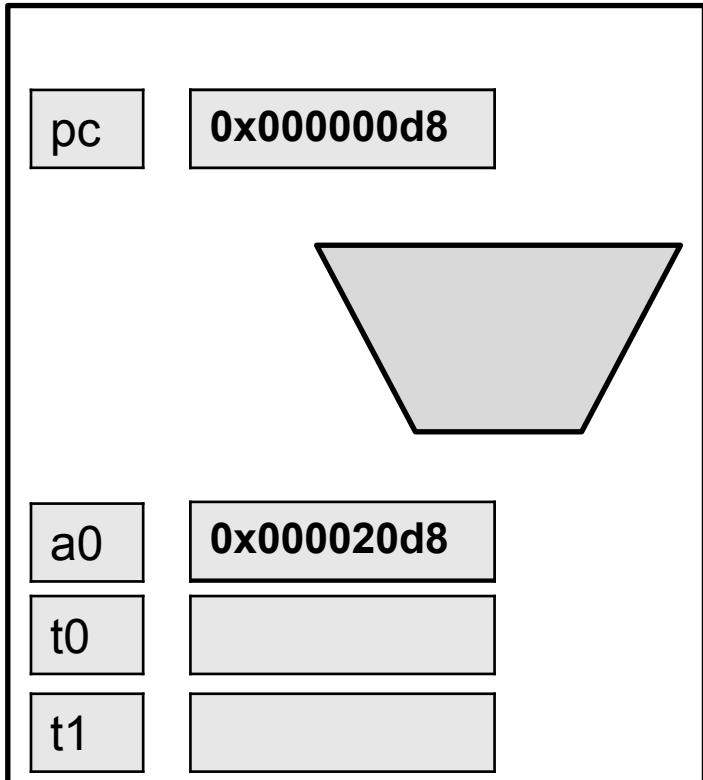
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	

Pseudoinstruction	Base Instruction(s)	Meaning
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load address

```
.data
    A: .word 0x1F2F3F4F
.text
.globl main
main:
    la a0, A
    la a0, A
    la a0, A
    ret
.end
```

```
000000d8 <main>:
d8: 00002517    auipc   a0,0x2
dc: 0b050513    addi    a0,a0,176 # 2188 <A>
e0: 00002517    auipc   a0,0x2
e4: 0a850513    addi    a0,a0,168 # 2188 <A>
e8: 00002517    auipc   a0,0x2
ec: 0a050513    addi    a0,a0,160 # 2188 <A>
f0: 00008067    ret
```

auipc instruction $rd = PC + (imm \ll 12)$

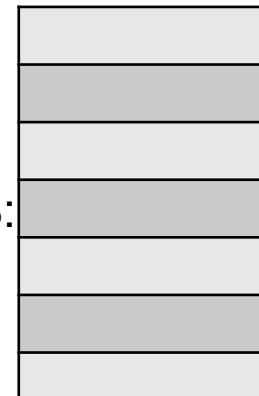


A: 0x00002188:

000000d8 <main>:

d8: 00002517
dc: 0b050513
e0: 00002517
e4: 0a850513
e8: 00002517
ec: 0a050513
f0: 00008067

auipc a0,0x2
addi a0,a0,176 # 2188 <A>
auipc a0,0x2
addi a0,a0,168 # 2188 <A>
auipc a0,0x2
addi a0,a0,160 # 2188 <A>
ret

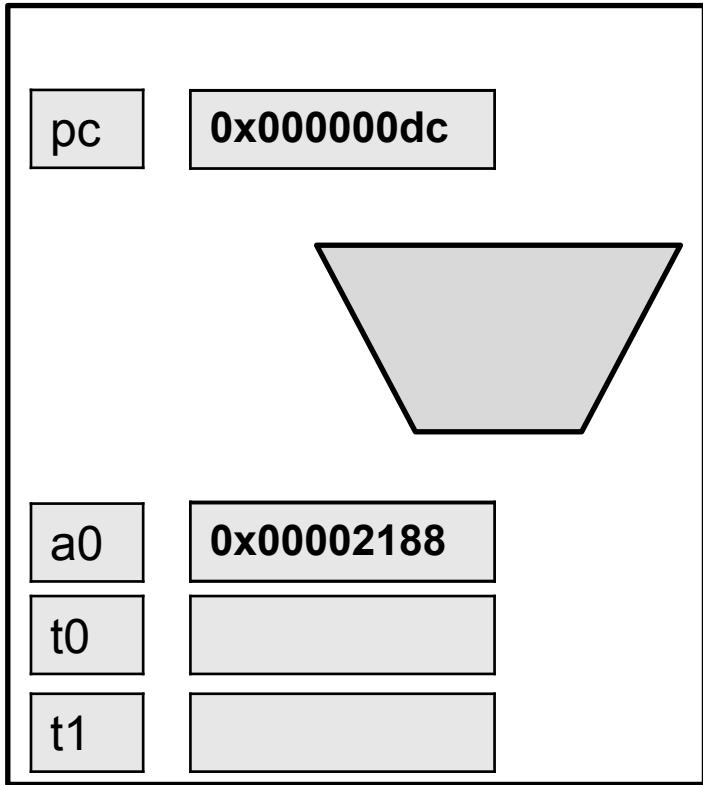


0x00002000

0x000000d8

0x000020d8

auipc instruction $rd = PC + (imm \ll 12)$



A: 0x00002188:

000000d8 <main>:

d8: 00002517
dc: 0b050513
e0: 00002517
e4: 0a850513
e8: 00002517
ec: 0a050513
f0: 00008067

auipc a0,0x2
addi a0,a0,176 # 2188 <A>
auipc a0,0x2
addi a0,a0,168 # 2188 <A>
auipc a0,0x2
addi a0,a0,160 # 2188 <A>
ret

$$176_2 = 0x000000b0$$

$$0x000020dc + 0x000000b0$$

$$\textcolor{red}{0x00002188}$$

la pseudo instruction

- We wrote the same instruction, but it is converted to different immediate values by assembler

Pseudoinstruction	Base Instruction(s)	Meaning
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load address

```
.data
    A: .word 0x1F2F3F4F
.text

.globl main
main:
    la a0, A
    la a0, A
    la a0, A
    ret
.end
```

```
000000d8 <main>:
d8: 00002517    auipc   a0,0x2
dc: 0b050513    addi    a0,a0,176 # 2188 <A>
e0: 00002517    auipc   a0,0x2
e4: 0a850513    addi    a0,a0,168 # 2188 <A>
e8: 00002517    auipc   a0,0x2
ec: 0a050513    addi    a0,a0,160 # 2188 <A>
f0: 00008067    ret
```

Branching Instructions

Branching Instructions

RV32I Base Integer Instructions (Branch type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 ≥ rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 ≥ rs2) PC += imm	zero-extends

□ Change the Program Counter
 ⇔ Change the Program Flow

31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd		opcode			R-type
imm[11:0]				rs1		funct3		rd		opcode			I-type
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode			S-type
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode			B-type
		imm[31:12]						rd		opcode			U-type
		imm[20 10:1 11 19:12]						rd		opcode			J-type

Thank you!

Programming Microprocessor – Instruction Set Architecture III



CS2053 Computer Architecture
Computer Science & Engineering
University of Moratuwa

Sulochana Sooriyaarachchi
Chathuranga Hettiarachchi

Slides adopted from Dr.Dilum Bandara

Encoding Instructions

- Various instruction types
- Limited word size for registers, addresses, and instructions
 - Consider 32bit words in RV32I
 - All the instructions are 32bits
 - Example : If we need to load an immediate value to 32-bit register, how to fit all opcode and operands within 32-bit instruction?
 - Work with small numbers
 - Make compromises

Instruction Formats(Types)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
	funct7		rs2		rs1		funct3		rd		opcode		R-type	
	imm[11:0]				rs1		funct3		rd		opcode		I-type	
	imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type	
	imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		B-type	
			imm[31:12]						rd		opcode		U-type	
			imm[20 10:1 11 19:12]						rd		opcode		J-type	

Fields

- Opcode : 7bits
 - funct 3 : 3 bit function
 - funct 7 : 7 bit function
 - rs1, rs2 : two source registers (5 bits each)
 - rd : destination register (5 bits)

Instruction Formats (6 Types)

- R-Format: instructions using 3 *register* inputs
 - add, xor, mul —arithmetic/logical ops
- I-Format: instructions with *immediates*, loads
 - addi, lw, jalr, slli
- S-Format: *store* instructions: sw, sb
 - SB-Format: *branch* instructions: beq, bge
- U-Format: instructions with *upper* immediates
 - lui, auipc —upper immediate is 20-bits
 - UJ-Format: *jump* instructions: jal

Instruction Format-Register Type

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7	rs2		rs1		funct3		rd		opcode		R-type			
imm[11:0]			rs1		funct3		rd		opcode		I-type			
imm[11:5]	rs2		rs1		funct3		imm[4:0]		opcode		S-type			
imm[12 10:5]	rs2		rs1		funct3		imm[4:1 11]		opcode		B-type			
		imm[31:12]					rd		opcode		U-type			
		imm[20 10:1 11 19:12]					rd		opcode		J-type			

RV32I Base Integer Instructions (Register Type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends

For the complete standard specifications:

<https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>

Instruction Format-Immediate Type

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode		R-type		
imm[11:0]		rs1		funct3		rd		opcode		I-type				
imm[11:5]	rs2	rs1	funct3	imm[4:0]	rd	opcode							S-type	
imm[12 10:5]	rs2	rs1	funct3	imm[4:1 11]	rd	opcode							B-type	
imm[31:12]						rd	opcode						U-type	
imm[20 10:1 11 19:12]						rd	opcode						J-type	

RV32I Base Integer Instructions(Immediate Type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
addi	ADD Immediate	I	0010011	0x0		$rd = rs1 + imm$	
xori	XOR Immediate	I	0010011	0x4		$rd = rs1 \wedge imm$	
ori	OR Immediate	I	0010011	0x6		$rd = rs1 imm$	
andi	AND Immediate	I	0010011	0x7		$rd = rs1 \& imm$	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	$rd = rs1 << imm[0:4]$	
srlti	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	$rd = rs1 >> imm[0:4]$	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	$rd = rs1 >> imm[0:4]$	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		$rd = (rs1 < imm)?1:0$	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		$rd = (rs1 < imm)?1:0$	zero-extends
lb	Load Byte	I	0000011	0x0		$rd = M[rs1+imm][0:7]$	
lh	Load Half	I	0000011	0x1		$rd = M[rs1+imm][0:15]$	
lw	Load Word	I	0000011	0x2		$rd = M[rs1+imm][0:31]$	
lbu	Load Byte (U)	I	0000011	0x4		$rd = M[rs1+imm][0:7]$	zero-extends
lhu	Load Half (U)	I	0000011	0x5		$rd = M[rs1+imm][0:15]$	zero-extends

Instruction Format-Upper Immediate

31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd		opcode		R-type	
imm[11:0]				rs1		funct3		rd		opcode		I-type	
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type	
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		B-type	
imm[31:12]								rd		opcode		U-type	
imm[20 10:1 11 19:12]								rd		opcode		J-type	

<code>lui</code>	Load Upper Imm	U	<code>0110111</code>			<code>rd = imm << 12</code>	
<code>auipc</code>	Add Upper Imm to PC	U	<code>0010111</code>			<code>rd = PC + (imm << 12)</code>	

Load Upper Immediate (lui)

lui **t1,0x70070**

Fill up the upper 20 bits of destination register with immediate value

Add Upper Immediate value and Program Counter (auipc)

auipc a0,0x2

Fill the upper 20 bits of destination register with immediate value

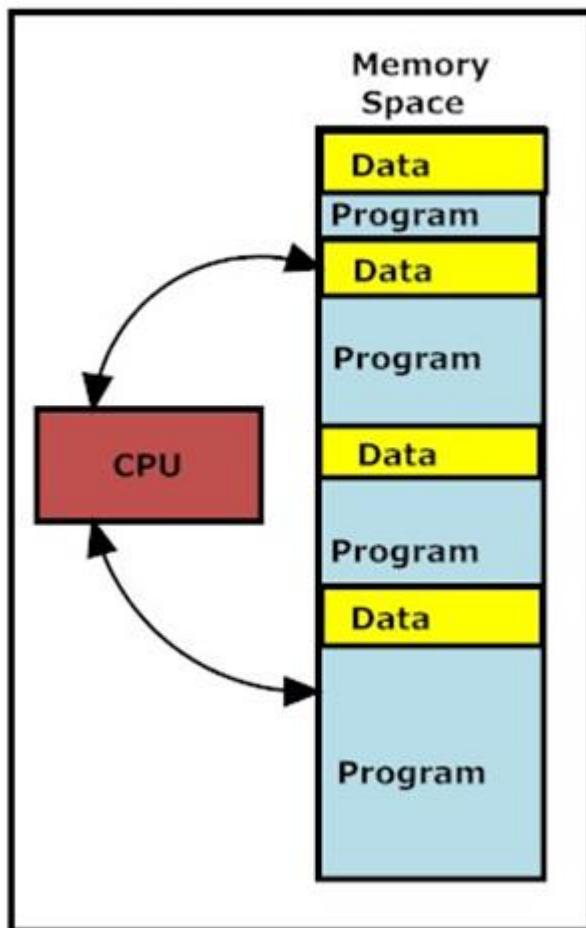
Some RISC V Pseudo Instructions

nop	addi x0, x0, 0	No operation
li rd, immediate	<i>Myriad sequences</i>	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
negw rd, rs	subw rd, x0, rs	Two's complement word
sext.w rd, rs	addiw rd, rs, 0	Sign extend word
seqz rd, rs	sltiu rd, rs, 1	Set if = zero
snez rd, rs	sltu rd, x0, rs	Set if \neq zero
sltz rd, rs	slt rd, rs, x0	Set if < zero
sgtz rd, rs	slt rd, x0, rs	Set if > zero

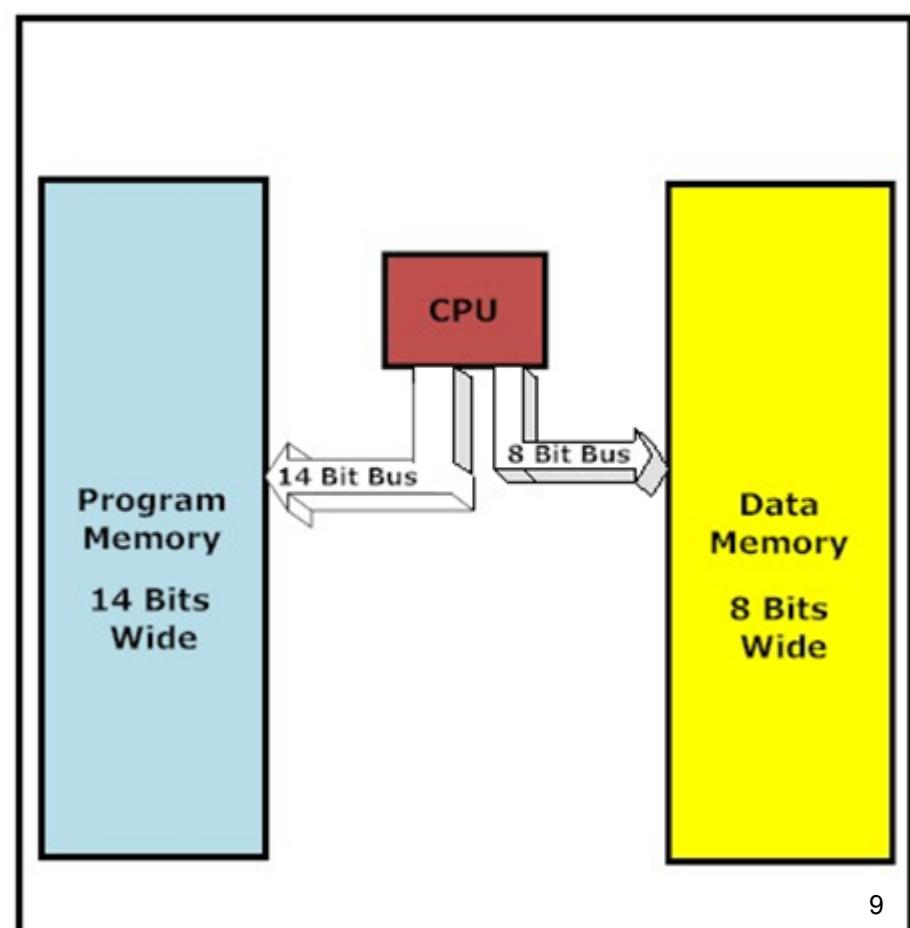
Some more are available...

Memory Architectures

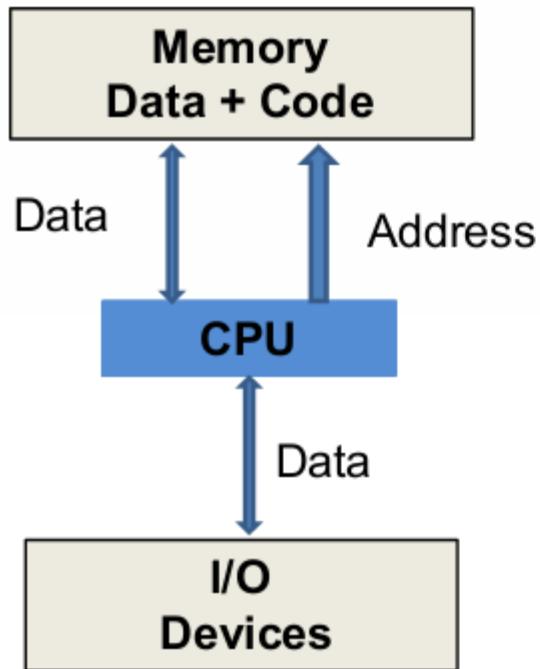
**Von Neumann
Architecture**



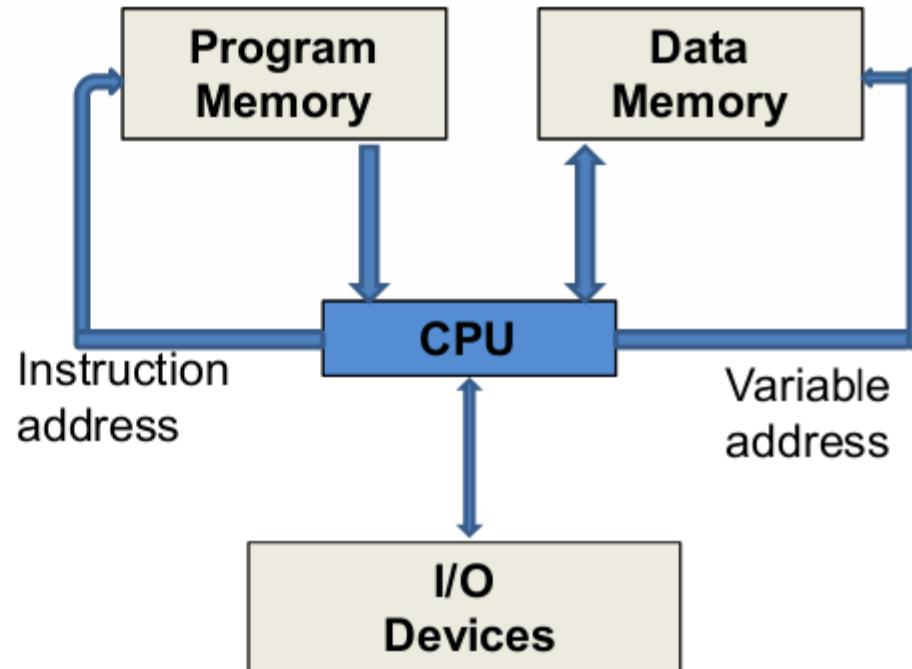
**Harvard
Architecture**



Von Neumann vs. Harvard Architecture

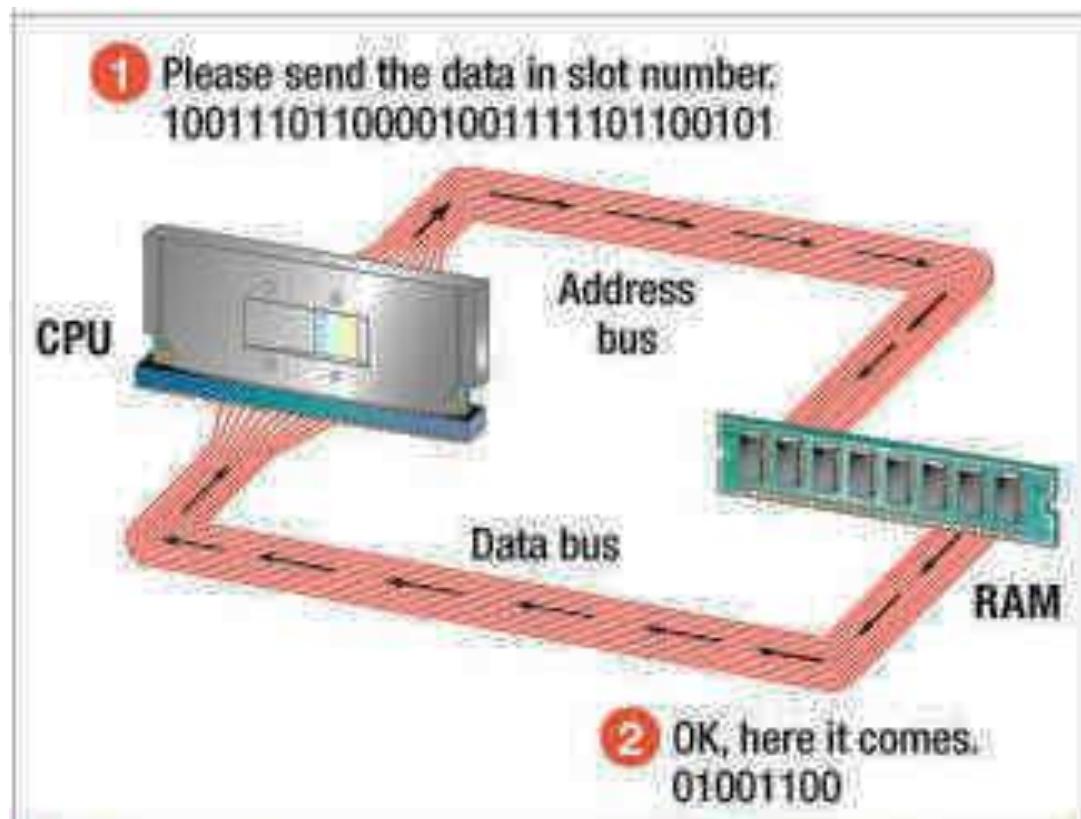


Von Neumann Machine



Harvard Machine

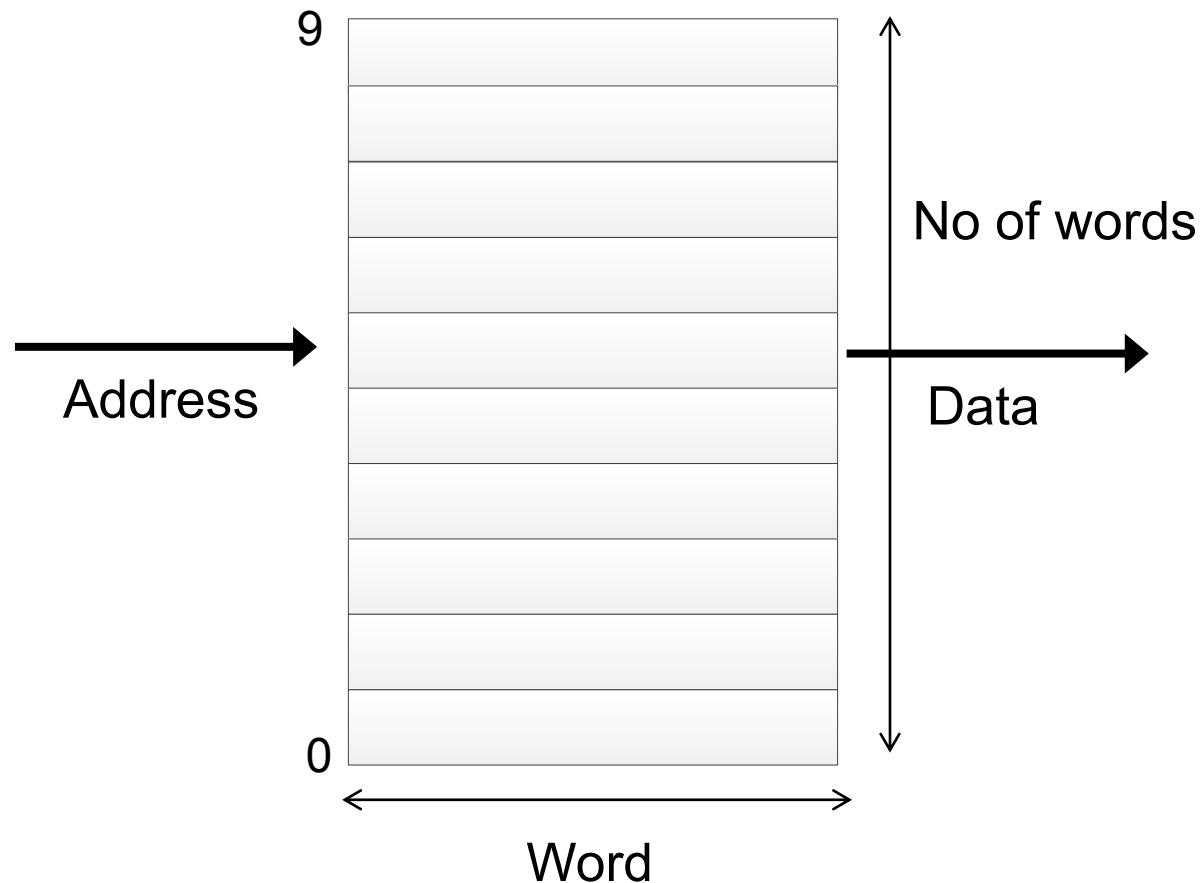
Memory Addressing



- Place an address on address bus
- Read or write operation
- Data placed on data bus

Source: www.alf.sd83.bc.ca/courses/lt12/using_it/processor_speed.htm

Memory Addressing (Cont.)



Addressing Modes

- It is the way microprocessor:
 - Identifies location of data
 - Access data

- Absolute address
 - Actual physical address
 - Direct addressing

- Relative address
 - Address relative to a known reference
 - Indirect addressing

Load and Store Instructions

31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd		opcode		R-type	
imm[11:0]		rs1		funct3		rd		opcode		I-type			
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type	
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		B-type	
		imm[31:12]						rd		opcode		U-type	
		imm[20 10:1 11 19:12]						rd		opcode		J-type	

Load: copy a value from memory to register *rd* : (Immediate type)
Source is a **memory address**

Store: copy the value in register *rs2* to **memory** : (Store type)

Need the proper 32bit address

RV32I Base Integer Instructions (Immediate type and Store type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
1b	Load Byte	I	0000011	0x0		$rd = M[rs1+imm][0:7]$	
1h	Load Half	I	0000011	0x1		$rd = M[rs1+imm][0:15]$	
1w	Load Word	I	0000011	0x2		$rd = M[rs1+imm][0:31]$	
1bu	Load Byte (U)	I	0000011	0x4		$rd = M[rs1+imm][0:7]$	zero-extends
1hu	Load Half (U)	I	0000011	0x5		$rd = M[rs1+imm][0:15]$	zero-extends
sb	Store Byte	S	0100011	0x0		$M[rs1+imm][0:7] = rs2[0:7]$	
sh	Store Half	S	0100011	0x1		$M[rs1+imm][0:15] = rs2[0:15]$	
sw	Store Word	S	0100011	0x2		$M[rs1+imm][0:31] = rs2[0:31]$	

Activity

```
.data
    A: .word 0x1F2F3F4F
.text

.globl main
main:
    la a0, A
    la a0, A
    la a0, A
    ret
.end
```

- Try the following program with RIPES and explain the machine code
- What is the machine code of la a0, A ?
- Try to add a nop before first la and revisit machine code

la (Load Address) pseudo instruction

Pseudoinstruction	Base Instruction(s)	Meaning
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load address

```
.data
    A: .word 0x1F2F3F4F
.text
.globl main
main:
    la a0, A
    la a0, A
    la a0, A
    ret
.end
```

```
000000d8 <main>:
d8: 00002517    auipc   a0,0x2
dc: 0b050513    addi    a0,a0,176 # 2188 <A>
e0: 00002517    auipc   a0,0x2
e4: 0a850513    addi    a0,a0,168 # 2188 <A>
e8: 00002517    auipc   a0,0x2
ec: 0a050513    addi    a0,a0,160 # 2188 <A>
f0: 00008067    ret
```

- We wrote the same instruction, but it is converted to different immediate values by assembler

Branching Instructions

Branching Instructions

RV32I Base Integer Instructions (Branch type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 ≥ rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 ≥ rs2) PC += imm	zero-extends

□ Change the Program Counter
 ⇔ Change the Program Flow

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode				R-type
imm[11:0]				rs1		funct3		rd		opcode				I-type
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode				S-type
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode				B-type
		imm[31:12]						rd		opcode				U-type
		imm[20 10:1 11 19:12]						rd		opcode				J-type

Branching Instructions

31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd		opcode			R-type
imm[11:0]				rs1		funct3		rd		opcode			I-type
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode			S-type
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode			B-type
		imm[31:12]						rd		opcode			U-type
		imm[20 10:1 11 19:12]						rd		opcode			J-type

- Need to specify an address to go to
- Also take two registers to compare
 - Conditional branch
- Doesn't write into a register (similar to stores)
(destination register *rd* is not required)
- How to encode label, i.e., where to branch to?

Branching Instructions -Addressing

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
	funct7		rs2		rs1		funct3		rd		opcode		R-type	
	imm[11:0]				rs1		funct3		rd		opcode		I-type	
	imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type	
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		B-type		
	imm[31:12]								rd		opcode		U-type	
	imm[20 10:1 11 19:12]								rd		opcode		J-type	

□ PC-relative addressing

- Use immediate field **12 bits** as a two's complement offset to PC.
 - Branches generally change the PC only by a small amount. But what is the reach?
 - Can specify $[-2^{11}, 2^{11})$ address offsets from the PC

Branching Instructions -Addressing

- Recall: RISCV uses 32-bit addresses, and memory is **byte-addressed**
- Instructions are “***word-aligned***”: Address is always a multiple of 4 (in bytes)
 - Previous instruction offset : -4 bytes
 - Next instruction offset : 4 bytes
 - Only if compressed instructions (16 bytes) are used, it could be multiple of 2 (in bytes) as well
- PC ALWAYS points to an instruction
 - PC is typed as a pointer to a word
 - can do C-like pointer arithmetic

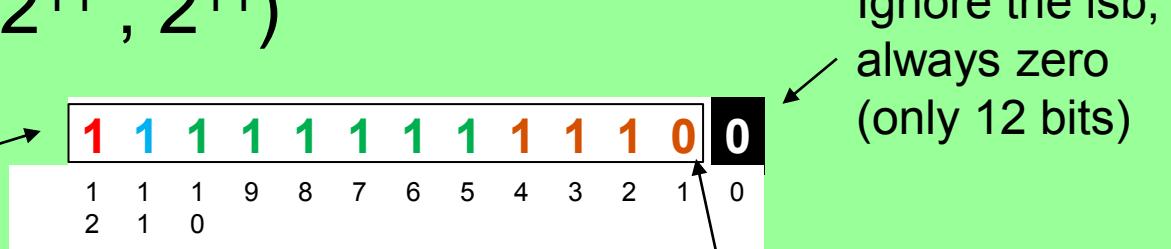
Branching Instructions -Addressing

- Only 12 bits available for immediate offset

- Offset Range $[-2^{11}, 2^{11})$

Offset : -4

12 bit Immediate field
(Two's complement)

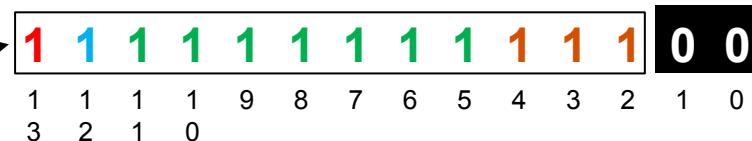


Provision for compressed instructions
(For normal instructions, this also will be always zero)

- Offset Range $[-2^{12}, 2^{12})$

Offset : -4

12 bit Immediate field



We don't do this

If we ignore both lsb's we will have more reach, but compressed instructions cannot be accommodated

Branch Calculation

- If we **don't** take the branch:

- $PC = PC + 4 = \text{next instruction}$

- Assume we have only normal 32bit instructions

- If we **do** take the branch:

- $PC = PC + (\text{immediate field} * 2)$

- If we assume we only have normal 32bit instructions, lsb of immediate field must be zero. If its one, there will be an error, by PC pointing to a middle of the instruction

Branching Instructions (B type)

```
main:
    addi t0,zero,10
    add t1,zero,zero
repeat:
    addi t1,t1,1
    bne t0,t1,repeat
    ret
.end
```

```
bne x5,x6,repeat  
bne t0,t1,repeat
```

Offset : -4 = -000100

$$= 111011 + 1$$

= 111100 (Sign extended)

31 20 19 15 14 12 11 7 6 0

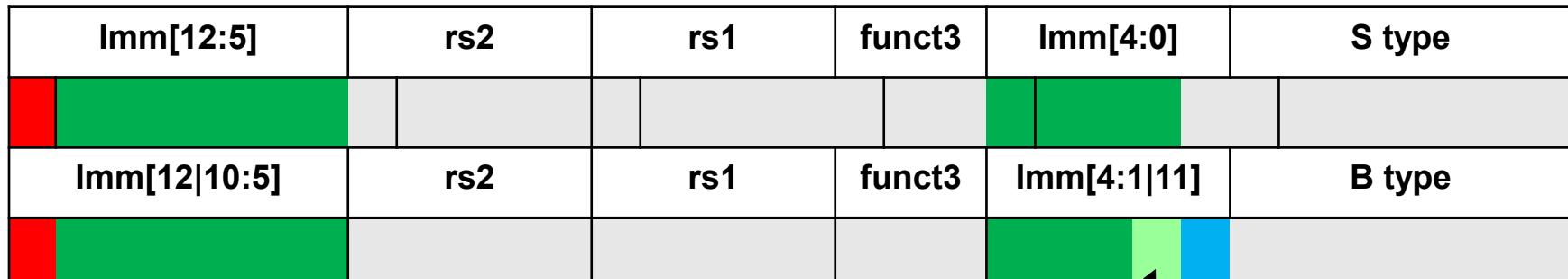
Imm[12 10:5]	rs2	rs1	funct3	Imm[4:1 11]	opcode			
Immediate offset [12 10:5]	rs2	rs1	funct3 sw	Immediate offset [4:1 11]	B type branch			
			0 0 1		1 1 0 0 0 1 1			
1 1 1 1 1 1 1 0 0 1 1 0 0 0 1 0 1				1 1 1 0 1				
1 1 1 1 1 1 1 0 0 1 1 0 0 0 1 0 1 0 0 1 1 1 1 0 1 1 1 0 0 0 1 1	f	e	6	2	9	e	e	3

Machine Instruction: 0xFE629EE3

Branching Instructions -Addressing

31	27	26	25	24	20	19	15	14	12	11	7	6	0					
funct7		rs2		rs1		funct3		rd		opcode		R-type						
imm[11:0]		rs1		funct3		rd		opcode		I-type		opcode						
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type						
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		B-type						
imm[31:12]							rd		opcode		U-type		opcode					
imm[20 10:1 11 19:12]							rd		opcode		J-type		opcode					

- S type and B type immediate value bits are aligning well.



Sign Bit

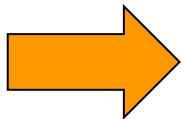
Zero except for branching to a compressed instruction

Extra bit in B type

Branching Instructions -Addressing

□ Pseudo Assembly /Disassembly

```
main:  
    addi t0,zero,10  
    add t1,zero,zero  
repeat:  
    addi t1,t1,1  
    bne t0,t1,repeat  
    ret  
.end
```



PC	Machine Code	Basic Code	Original Code
0x0	0x00A00293	addi x5 x0 10	addi t0,zero,0xa
0x4	0x00000333	add x6 x0 x0	add t1,zero,zero
0x8	0x00130313	addi x6 x6 1	addi t1,t1,0x1
0xc	0xFE629EE3	bne x5 x6 -4	bne t0,t1,repeat
0x10	0x00008067	jalr x0 x1 0	ret


```
000000d8 <main>:  
d8: 00a00293           li  t0,10  
dc: 00000333           add t1,zero,zero  
  
000000e0 <repeat>:  
e0: 00130313           addi t1,t1,1  
e4: fe629ee3           bne t0,t1,e0 <repeat>  
e8: 00008067           ret
```

Remember

bne t0,t1,repeat

Machine Instruction: **0xFE629EE3**

Jump Instructions

Jump Instructions

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode		R-type		
imm[11:0]		rs1		funct3		rd		opcode		I-type				
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type		
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		B-type		
imm[31:12]						rd		opcode		U-type				
imm[20 10:1 11 19:12]				rd		opcode				J-type				

Jump to anywhere in memory

Need the proper 32bit address

Store a return address in a register (rd), so that you can come back to original flow

Known as “Linking”

RV32I Base Integer Instructions (Jump type and Immediate type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	

Jump Instructions

- For branches, we assumed that we won't want to branch too far, so we can specify a **change** in the PC
- For (`jalr`) jumps, we may jump to **anywhere** in code memory
 - Ideally, we would specify a 32-bit memory address to jump to
 - Unfortunately, we can't fit both a 7-bit opcode and a 32-bit address into a single 32-bit word
 - Also, when linking we must write to an `rd` register

jal Instruction and **j** pseudo instruction

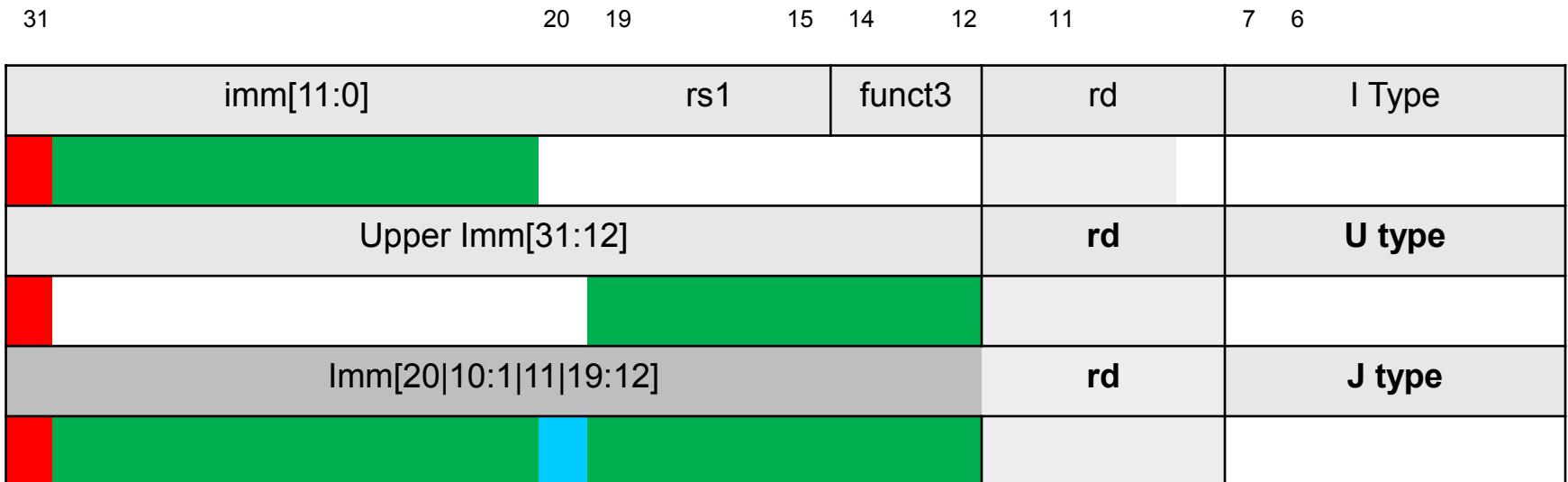
- `jal` saves PC+4 in register `rd` (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
- “j” jump is a pseudo-instruction—the assembler will instead use `jal` but sets `rd=x0` to discard return address
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

jal Instruction Encoding

imm[20 10:1 11 19:12]				rd	opcode	J-type
jal x1,-12				-12	= -001100	
jal ra,myfunction					= 110011 +1	
				= ...	1111 1111 1111 0100	
31	20 19	15 14	12 11	7 6	0	
Imm [20 10:1 11 19:12]				rd	opcode	
Immediate offset [20 10:1 11 19:12]				Register for return address	J type jump jal	
					1 1 0 1 1 1 1	
1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 0 0 0 0 1						
				0 0 0 0 1 1 1 0 1 1 1 1		
F	F	5	F	F	0	E F

Machine Instruction: **0xFF5FF0EF**

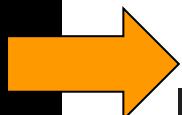
Re-ordering bits in J type



jal Instruction Usage

```
.globl main
myfunc:
    addi t0, zero, 1
    ret

main:
    addi t0, zero, 0
    jal ra,myfunc
    ret
.end
```



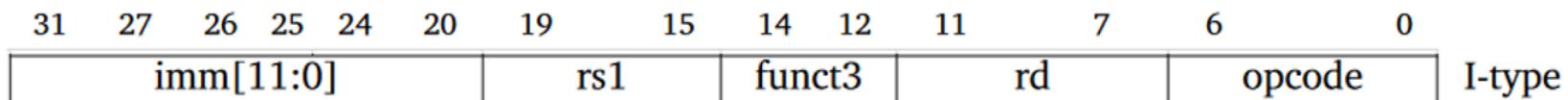
PC	Machine Code	Basic Code	Original Code
0x0	0x00100293	addi x5 x0 1	addi t0, zero, 1
0x4	0x00008067	jalr x0 x1 0	ret
0x8	0x00000293	addi x5 x0 0	addi t0, zero, 0
0xc	0xFF5FF0EF	jal x1 -12	jal ra,myfunc
0x10	0x00008067	jalr x0 x1 0	ret


```
000000d8 <myfunc>:
d8: 00100293          li  t0,1
dc: 00008067          ret

000000e0 <main>:
e0: 00000293          li  t0,0
e4: ff5ff0ef          jal  ra,d8 <myfunc>
e8: 00008067          ret
```

jalr instruction (I type)

- Jump and link register `jalr` to address given in `rs1` register + imm offset



RISCV Instructions:

```
# ret and jr psuedo-instructions  
ret = jr ra = jalr x0, ra, 0
```

```
# Call function at any 32-bit absolute address  
lui x1, <hi 20 bits>  
jalr ra, x1, <lo 12 bits>
```

```
# Jump PC-relative with 32-bit offset  
auipc x1, <hi 20 bits>  
jalr x0, x1, <lo 12 bits>
```

jalr Instruction encoding

```
jalr x1, x1, 0
```

```
jalr ra, ra, 0
```

imm[11:0]	rs1	funct3	rd	Opcode (jalr)
	0 0 0			1 1 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 1		0 0 0 0 1	
0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 1	1 0 0 0	0 0 0 0 1 1 1	0 0 0 1 1 1 1
0	0	0	0	E 7

Machine Instruction: **0x000080E7**

Linking Multiple Functions

```
.globl main
myfunc:
    addi t0, zero, 1
    ret

main:
    addi t0, zero, 0
    la ra,myfunc
    jalr ra,ra,0
    ret
.end
```

PC	Machine Code	Basic Code	Original Code
0x0	0x00100293	addi x5 x0 1	addi t0, zero, 1
0x4	0x00008067	jalr x0 x1 0	ret
0x8	0x00000293	addi x5 x0 0	addi t0, zero, 0
0xc	0x00000097	auipc x1 0	la ra,myfunc
0x10	0xFF408093	addi x1 x1 -12	la ra,myfunc
0x14	0x000080E7	jalr x1 x1 0	jalr ra,ra,0
0x18	0x00008067	jalr x0 x1 0	ret
			
<pre>000000d8 <myfunc>: d8: 00100293 li t0,1 dc: 00008067 ret 000000e0 <main>: e0: 00000293 li t0,0 e4: 00000097 auipc ra,0x0 e8: ff408093 addi ra,ra,-12 # d8 <myfunc> ec: 000080e7 jalr ra f0: 00008067 ret</pre>			

Summary

31	30	25 24	21	20	19	15 14	12 11	8	7	6	0	
												R-type
	funct7			rs2		rs1	funct3		rd		opcode	I-type
		imm[11:0]				rs1	funct3		rd		opcode	S-type
	imm[12]	imm[10:5]		rs2		rs1	funct3	imm[4:1]	imm[11]		opcode	B-type
			imm[31:12]						rd		opcode	U-type
	imm[20]	imm[10:1]	imm[11]		imm[19:12]				rd		opcode	J-type

- The Stored Program concept is very powerful
 - Instructions can be treated and manipulated the same way as data in both hardware and software

Thank you.

Example – Logic Operations

- Write an assembly program to convert a given character from uppercase to lowercase & vice versa
- If we consider ASCII, this can be achieved by changing the 5th bit
 - A = 65 =0x41 = 01000001
 - a = 97 =0x61 = 01100001
- Get XOR with 00100000 = 32 = 0x20

Homework

- Write an assembly program to multiply 3 & 4
- Steps:
 - How many registers?
 - What registers to use?
 - What instructions to use?

Exercise

- Which mask or filter value would you use, and what operation would you perform to make the 2nd & 4th bits one (1) no matter what they were before?

- Which mask or filter value would you use, and what operation would you perform to flip the 2nd & 4th bits no matter what they were before?

Exercise: Convert to Assembly

```
int total = 0;  
for (int i=10 ; i!=0; i--)  
{  
    total += i;  
}
```

Exercise

- Write a program to calculate the total of all integers from 1 to 10
- High-level program

```
int total = 0;  
for (int i=1 ; i<=10; i++)  
{  
    total += i;  
}
```

Exercise (Cont.)

□ Steps

- Are there any conditions/loops?
- How many registers?
- What registers to use?
- What instructions to use?

Summary

- Instruction Set Architecture (ISA) is the layer between hardware & software
- Specific to a given chip unless standardized
 - Defines registers it contain
 - Micro-operations performed on data stored on those registers
- Typical Assembly instructions for
 - Register operations
 - Memory access (Load/Store)
 - Upper immediates and Address calculations
 - Branching
 - Jump and Link
- Assembler translates human readable Assembly code to machine code

Sign extended immediate values

- `addi t1, t1, -0xFFFFFFFF` `# = 1` `# ||`
- `addi t1, t1, -0xFFFFFFF` `# = 2` `# ||`
- `addi t1, t1, -0xFFFFFFF` `# = 3` `# ||`
- `addi t1, t1, -0xFFFFF801` `# = 2047` `# ||`
- `# addi t1, t1, -0xFFFFF800` `# = 2048` `(Out of range)`
- `# addi t1, t1, -0xFFFFF7FF` `# = 2049` `(Out of range)`
- `# addi t1, t1, -0x00000801` `# = -2049` `(Out of range)`
- `addi t1, t1, -0x00000800` `# = -2048` `# ||`
- `addi t1, t1, -0x000007FF` `# = -2047` `# ||`
- `addi t1, t1, -0x00000001` `# = -1` `# ||`
- `addi t1, t1, 0x00000000` `# = 0` `# ||`
- `addi t1, t1, 0x00000001` `# = 1` `# ||`
- `addi t1, t1, 0x000007FF` `# = 2047` `# ||`
- `# addi t1, t1, 0x00000800` `# = 2048` `(Out of range)`
- `# addi t1, t1, 0x00000801` `# = 2049` `(Out of range)`
- `# addi t1, t1, 0xFFFFF7FF` `# = -2049` `(Out of range)`
- `addi t1, t1, 0xFFFFF800` `# = -2048`
- `addi t1, t1, 0xFFFFFFF` `# = -2`
- `addi t1, t1, 0xFFFFFFF` `# = -1`

Thank you!

(Compare with RISC-V Instructions)

TABLE 13-2: PIC16F87X INSTRUCTION SET

Mnemonic, Operands	Description	Cycles	14-Bit Opcode		Status Affected	Notes
			MSb	LSb		
BYTE-ORIENTED FILE REGISTER OPERATIONS						
ADDWF	f, d	Add W and f	1	00 0111 dfff ffff	C,DC,Z	
ANDWF	f, d	AND W with f	1	00 0101 dfff ffff	Z	
CLRF	f	Clear f	1	00 0001 1fff ffff	Z	
CLRW	-	Clear W	1	00 0001 0xxx xxxx	Z	
COMF	f, d	Complement f	1	00 1001 dfff ffff	Z	
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00 1011 dfff ffff	Z	
INCF	f, d	Increment f	1	00 1010 dfff ffff	Z	
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00 1111 dfff ffff	Z	
IORWF	f, d	Inclusive OR W with f	1	00 0100 dfff ffff	Z	
MOVF	f, d	Move f	1	00 1000 dfff ffff	Z	
MOVWF	f	Move W to f	1	00 0000 1fff ffff		
NOP	-	No Operation	1	00 0000 0xx0 0000		
RLF	f, d	Rotate Left f through Carry	1	00 1101 dfff ffff	C	
RRF	f, d	Rotate Right f through Carry	1	00 1100 dfff ffff	C	
SUBWF	f, d	Subtract W from f	1	00 0010 dfff ffff	C,DC,Z	
SWAPF	f, d	Swap nibbles in f	1	00 1110 dfff ffff		
XORWF	f, d	Exclusive OR W with f	1	00 0110 dfff ffff	Z	
BIT-ORIENTED FILE REGISTER OPERATIONS						
BCF	f, b	Bit Clear f	1	01 00bb bfff ffff		1,2
BSF	f, b	Bit Set f	1	01 01bb bfff ffff		1,2
BTFS	f, b	Bit Test f, Skip if Clear	1 (2)	01 10bb bfff ffff		3
BTFS	f, b	Bit Test f, Skip if Set	1 (2)	01 11bb bfff ffff		3
LITERAL AND CONTROL OPERATIONS						
ADDLW	k	Add literal and W	1	11 111x kkkk kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11 1001 kkkk kkkk	Z	
CALL	k	Call subroutine	2	10 0kkk kkkk kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00 0000 0110 0100	TO,PD	
GOTO	k	Go to address	2	10 1kkk kkkk kkkk		
IORLW	k	Inclusive OR literal with W	1	11 1000 kkkk kkkk	Z	
MOVLW	k	Move literal to W	1	11 00xx kkkk kkkk		
RETFIE	-	Return from interrupt	2	00 0000 0000 1001		
RETLW	k	Return with literal in W	2	11 01xx kkkk kkkk		
RETURN	-	Return from Subroutine	2	00 0000 0000 1000	TO,PD	
SLEEP	-	Go into standby mode	1	00 0000 0110 0011		
SUBLW	k	Subtract W from literal	1	11 110x kkkk kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11 1010 kkkk kkkk	Z	

F → file register
 W → working register
 B → bit
 L → literal (number)
 Z → conditional execution
 d → destination bit
 d=0 store in W
 d=1 store in f
 use , w or ,f instead

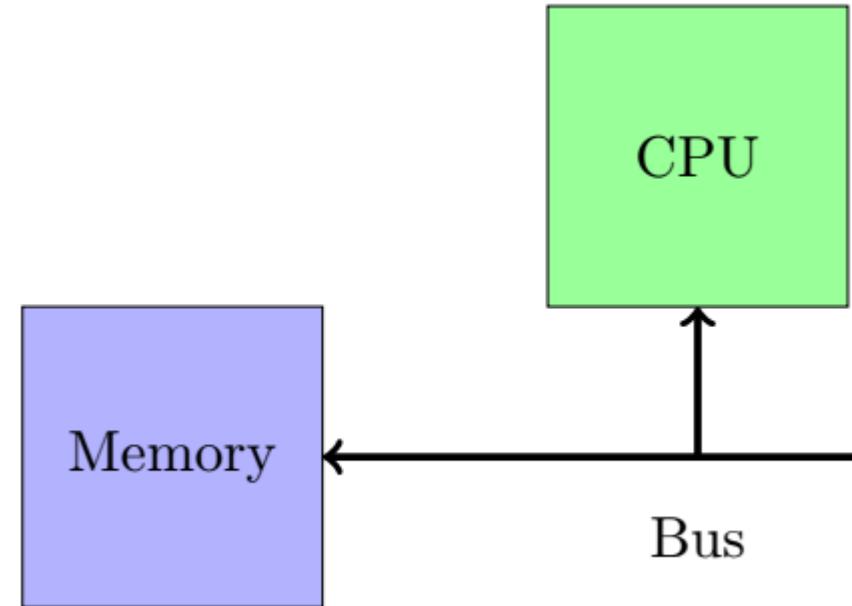
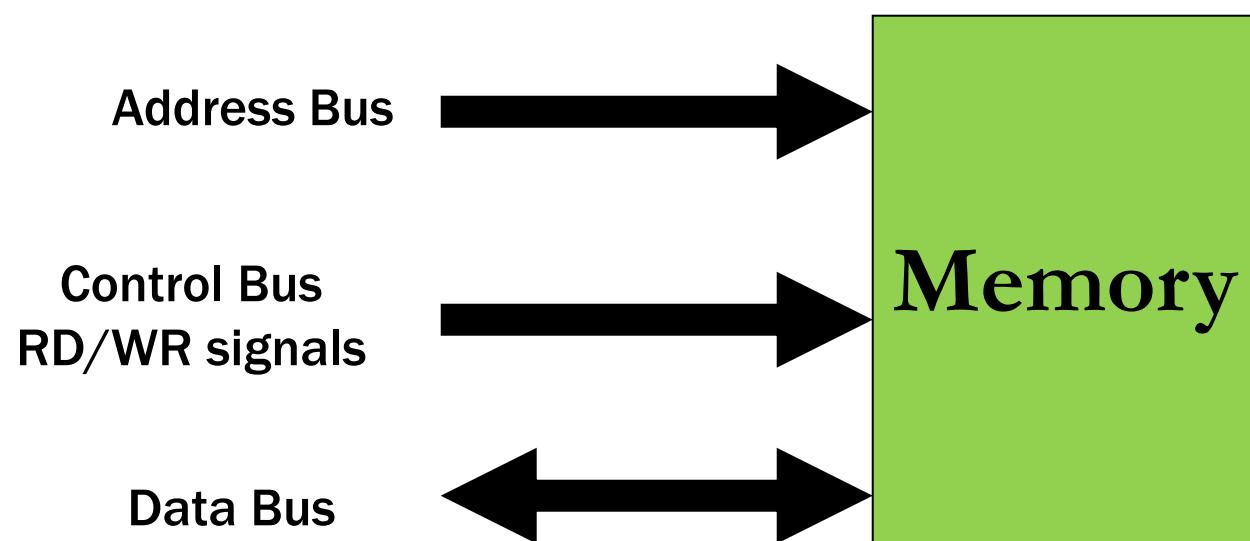
Source: Makis Malliris & Sabir Ghauri, UWE

CS2053 Computer Architecture

Memory Hierarchy

Dr. Sulochana Sooriyaarachchi

Accessing Memory

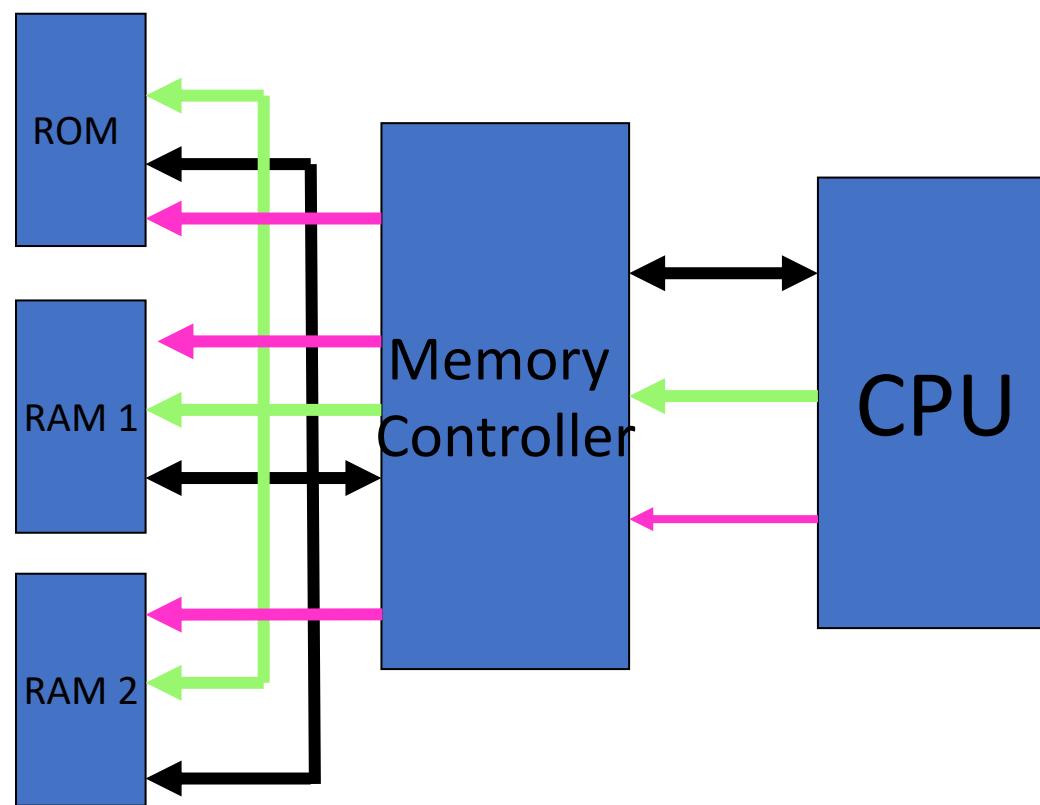


Reading From & Writing to Memory

- Reading from memory
 - **Address** of memory location to read is placed on Address Bus
 - **Read Signal (RD)** in control bus is activated
 - **Data** is fetched (read) from Data Bus
- Writing to memory
 - **Address** of memory location to write is placed on Address Bus
 - **Data** is placed on Data Bus
 - **Write Signal (WR)** in control bus is activated

Connecting Memory & CPU

Details of handling
different memory modules
& memory hierarchy is off
load to **memory controller**



Addressing Modes

- How architectures specify the address of an object they access
- Specifies
 - Constants
 - Registers
 - Memory locations
- In addressing memory locations
 - Actual memory address is called *effective address*

Example Addressing Modes

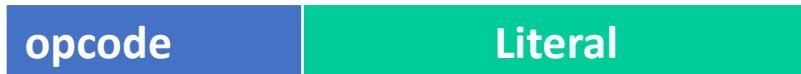
Addressing mode	Example instruction	Meaning	When used
Register	Add R4, R3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Regs}[R3]$	When a value is in a register
Immediate	Add R4, 3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + 3$	For constants
Displacement	Add R4, 100(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[100 + \text{Regs}[R1]]$	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	Add R4, (R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[\text{Regs}[R1]]$	Accessing using a pointer or a computed address
Indexed	Add R3, (R1+R2)	$\text{Regs}[R3] \leftarrow \text{Regs}[R3] + \text{Mem}[\text{Regs}[R1] + \text{Regs}[R2]]$	Sometimes useful in array addressing: R1 = base of array; R2 = index amount
Direct or absolute	Add R1, (1001)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[1001]$	Sometimes useful for accessing static data; address constant may need to be large
Memory indirect	Add R1, @(R3)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Mem}[\text{Regs}[R3]]]$	If R3 is the address of a pointer p , then mode yields $*p$
Autoincrement	Add R1, (R2)+	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$ $\text{Regs}[R2] \leftarrow \text{Regs}[R2] + d$	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d
Autodecrement	Add R1, -(R2)	$\text{Regs}[R2] \leftarrow \text{Regs}[R2] - d$ $\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	Add R1, 100(R2)[R3]	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[100 + \text{Regs}[R2] + \text{Regs}[R3] * d]$	Used to index arrays. May be applied to any indexed addressing mode in some computers

Addressing modes illustrations

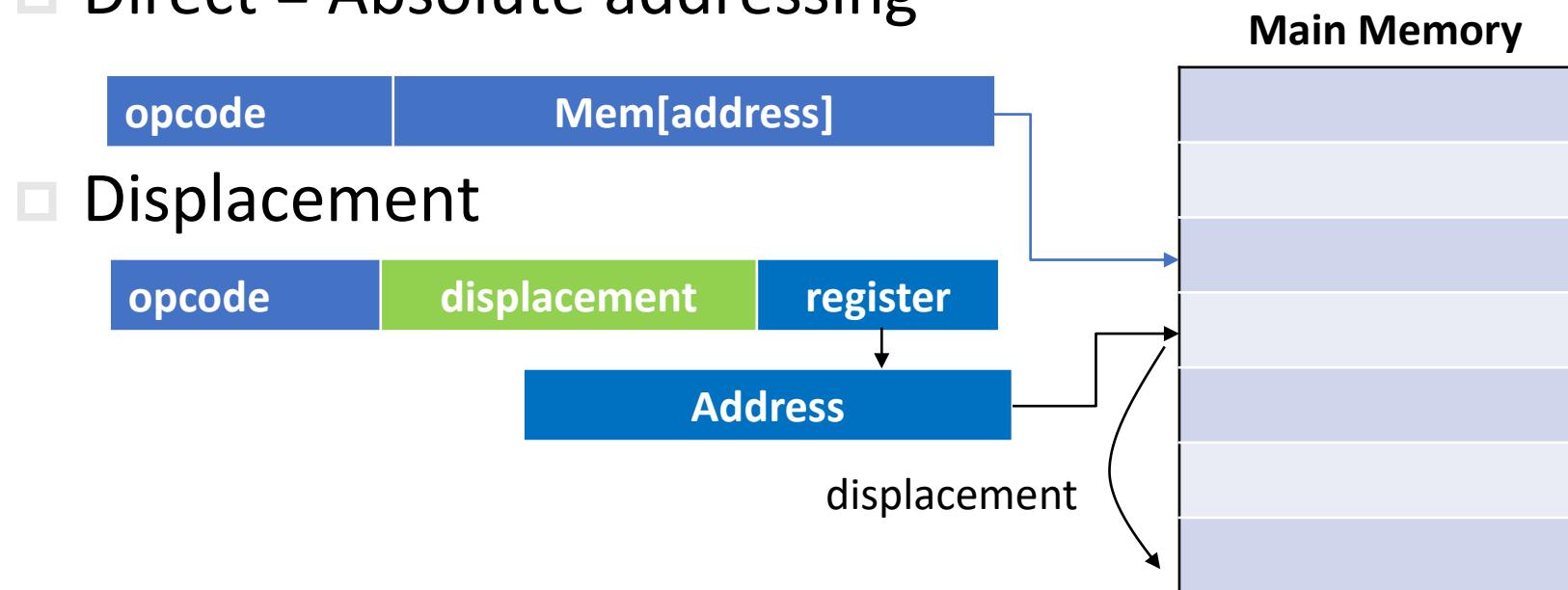
- # Register opcode



- ## Immediate value



- Direct = Absolute addressing

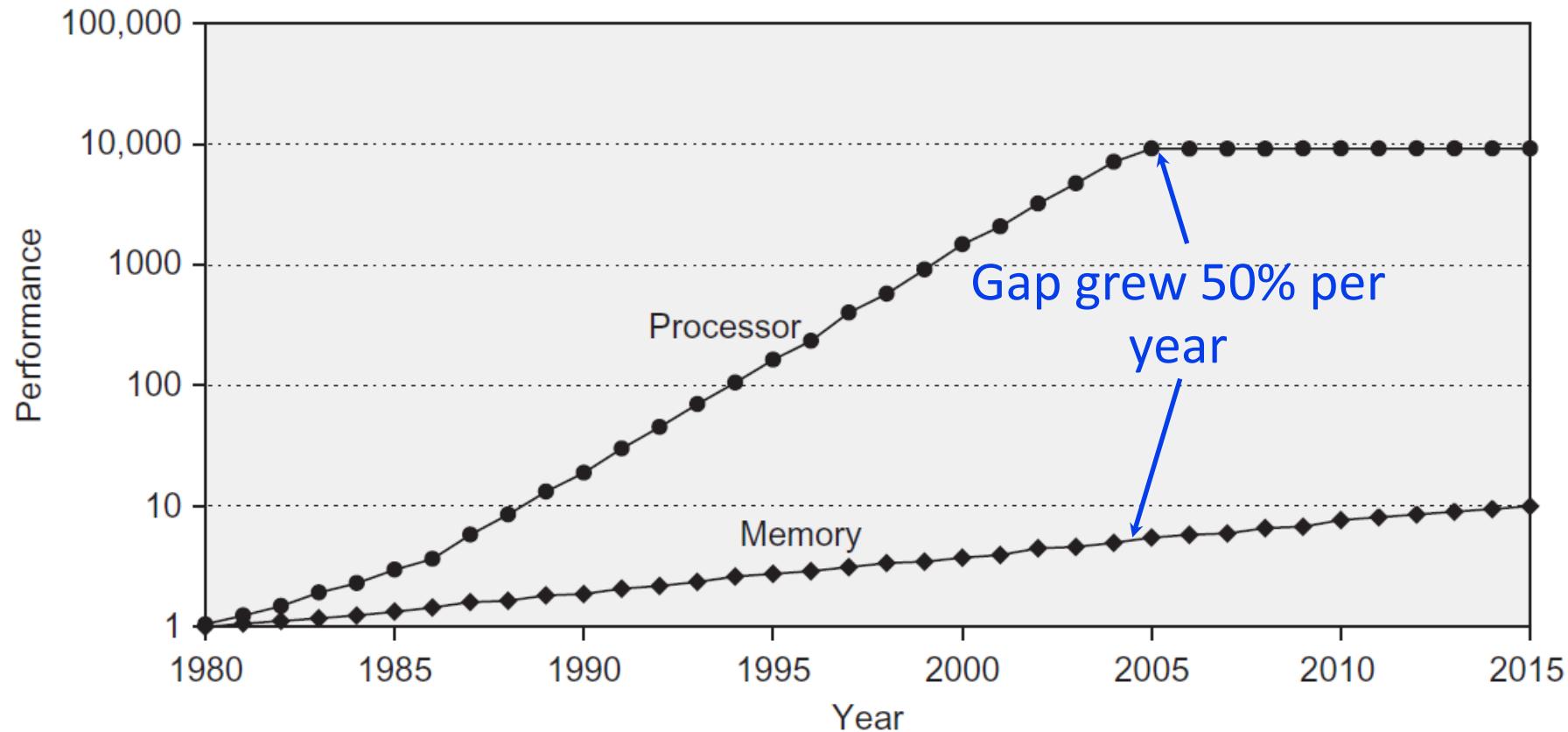


Take home assignment 1

- Study the addressing modes in the processor architecture in your Lab Series.
 - PC-relative: via *auipc*, *jal* and *br** instructions
 - Register-offset: via the *jalr*, *addi* and all memory instructions.
 - Absolute: via the *lui* instruction
- Read relevant sections in: RISCV ISA SpecificationsURL
 - <https://online.uom.lk/mod/url/view.php?id=343622>
 - Volume 1, Unprivileged Specification version 20191213 [\[PDF\]](#)

Submit a short note for each of above instructions with an illustration of addressing involved

Processor Memory Gap



Memory Hierarchy

- Memory has to be organized in such a way that its slowness doesn't reduce performance of overall system
- Some memory types are fast but expensive
 - Registers, Static RAM
- Some other types are cheap but slow
 - Dynamic RAM
- Solution
 - Hierarchical memory system
 - Limited capacity of fast but expensive memory types
 - Larger capacity of slow but cheap memory types

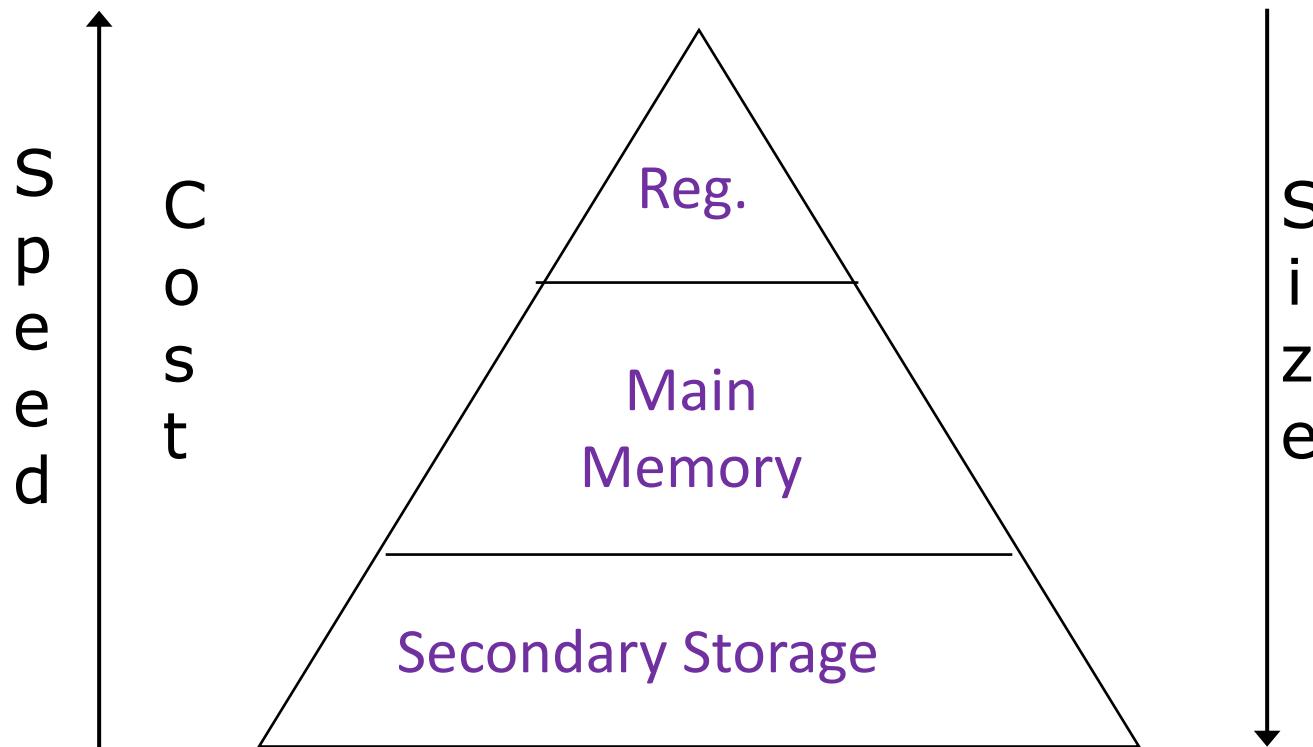
Memory Hierarchy (Cont.)

- Objective is to have a memory system
 - with sufficient speed
 - with sufficient capacity
 - as cheap as possible

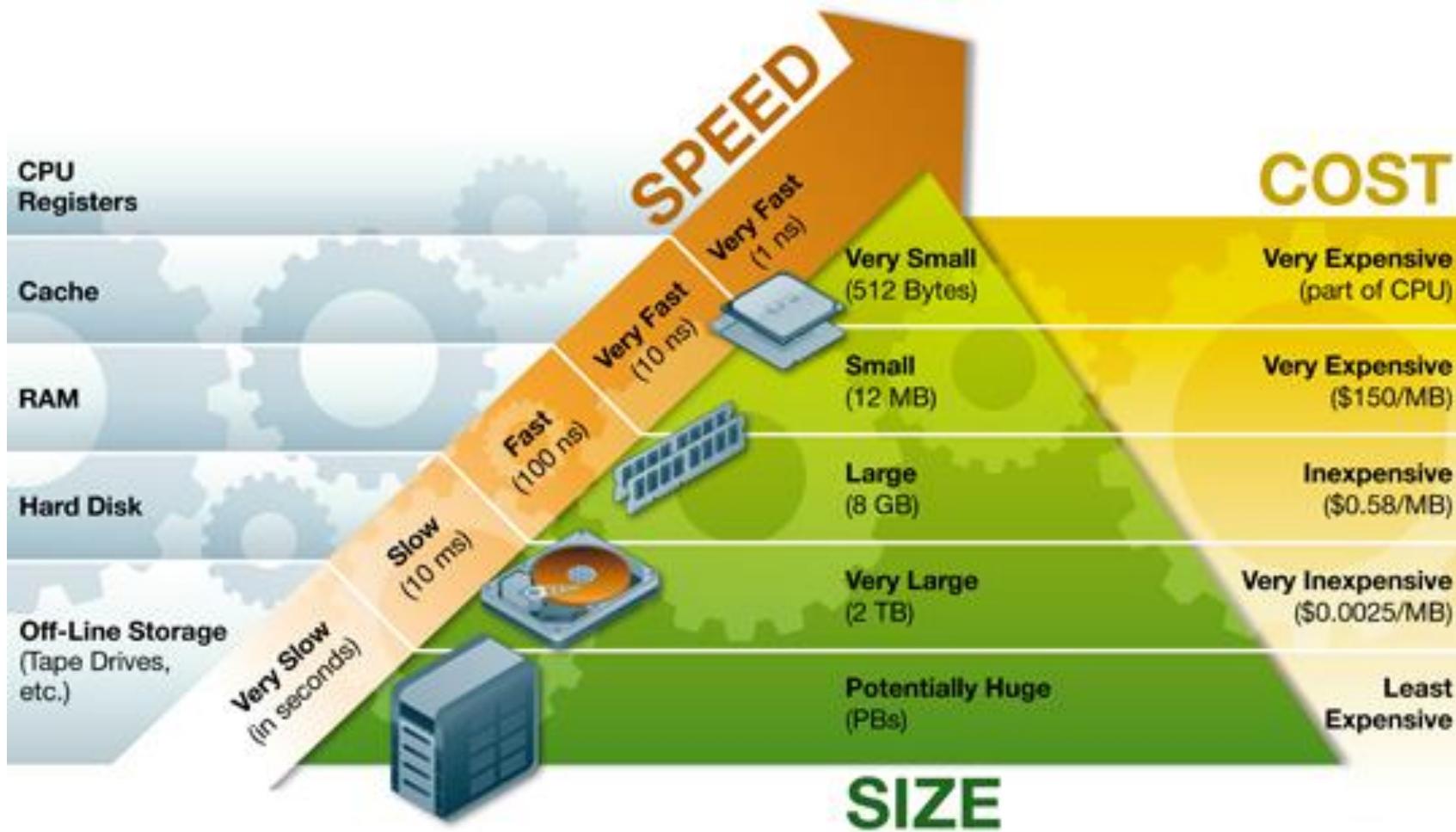
Ideally one would desire an indefinitely large memory capacity such that any particular... word would be immediately available... We are... forced to recognize the possibility of constructing a hierarchy of memories each of which has greater capacity than the preceding but which is less quickly accessible.

A. W. Burks, H. H. Goldstine,
and J. von Neumann,
*Preliminary Discussion of the
Logical Design of an Electronic
Computing Instrument* (1946).

Traditional Memory Hierarchy

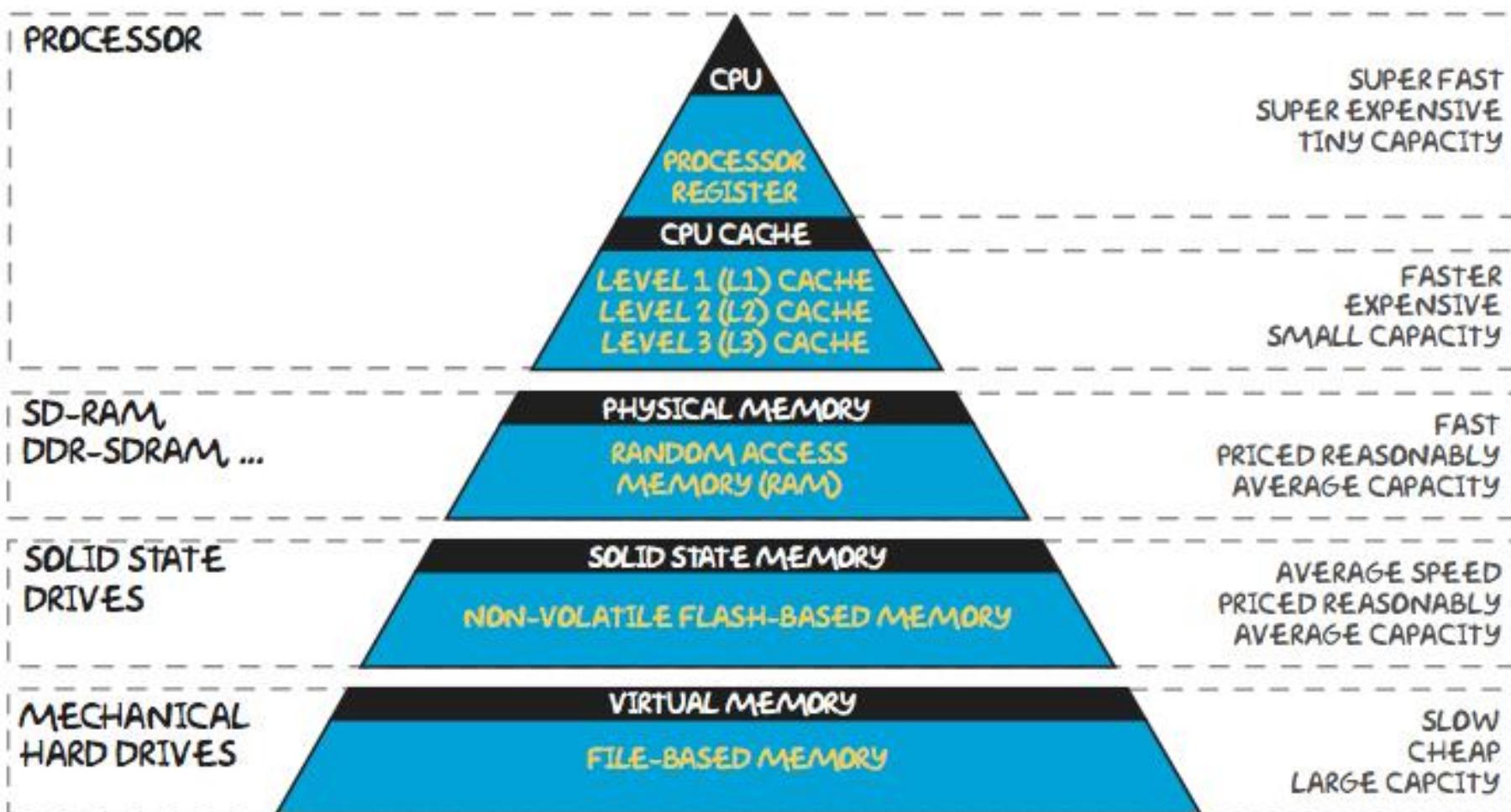


Extended Memory Hierarchy



Source: http://www.ts.avnet.com/uk/products_and_solutions/storage/hierarchy.html

Modern Memory Hierarchy



Principle of Locality

- Programs tend to reuse data & instructions that are close to each other or they have used recently
- Temporal locality
 - Recently referenced items are likely to be referenced in the near future
 - A block tends to be accessed again & again
- Spatial locality
 - Items with nearby addresses tend to be referenced close together in time
 - Near by blocks tend to be accessed

Locality – Example

```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += a[i];  
return sum;
```



- Data
 - Access array elements in succession – Spatial locality
 - Reference `sum` each iteration – Temporal locality
- Instructions
 - Reference instructions in sequence – Spatial locality
 - Cycle through loop repeatedly – Temporal locality

Locality examples

- Loops
- Sequential instructions
- Array elements

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Prefetching – Example

- Which of the following code is faster?

```
sum = 0;  
for (i = 0; i < n; i++)  
    for (j = 0; j < m; j++)  
        sum += a[i][j];  
return sum;
```

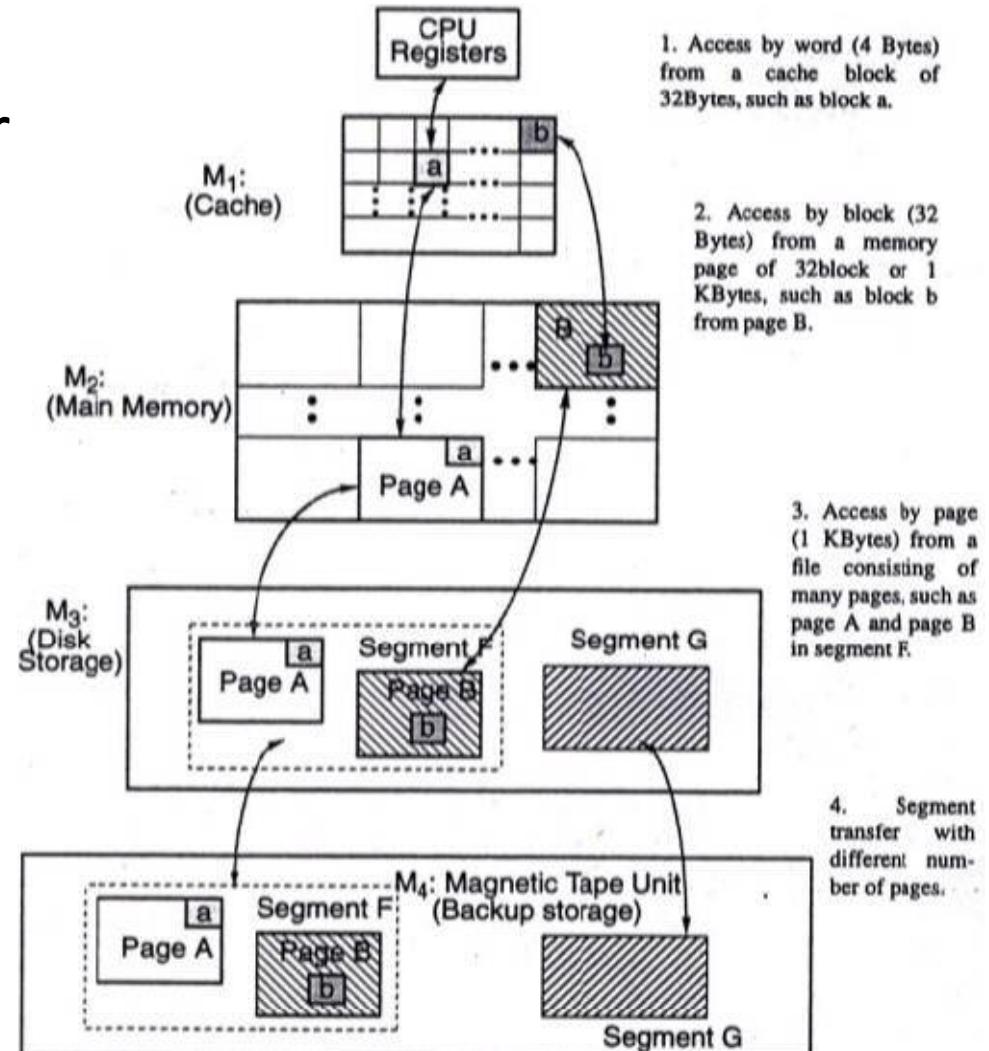
```
sum = 0;  
for (j = 0; j < m; j++)  
    for (i = 0; i < n; i++)  
        sum += a[i][j];  
return sum;
```

Programmer's view of matrix

	Col 1	Col 2	Col 3	Col 4
Row 1	1, 1	1, 2	1, 3	1, 4
Row 2	2, 1	2, 2	2, 3	2, 4
Row 3	3, 1	3, 2	3, 3	3, 4
	1,1	1,2	1,3	1,4
	2,1	2,2	2,3	2,4
	3,1	...		

Inclusion Property

- Often data in lower levels of hierarchy are a **superset** of higher level memories
- Terminology
 - Block (=line)
 - Hit, Miss
 - Hit rate, Miss rate
 - Hit time, Miss penalty

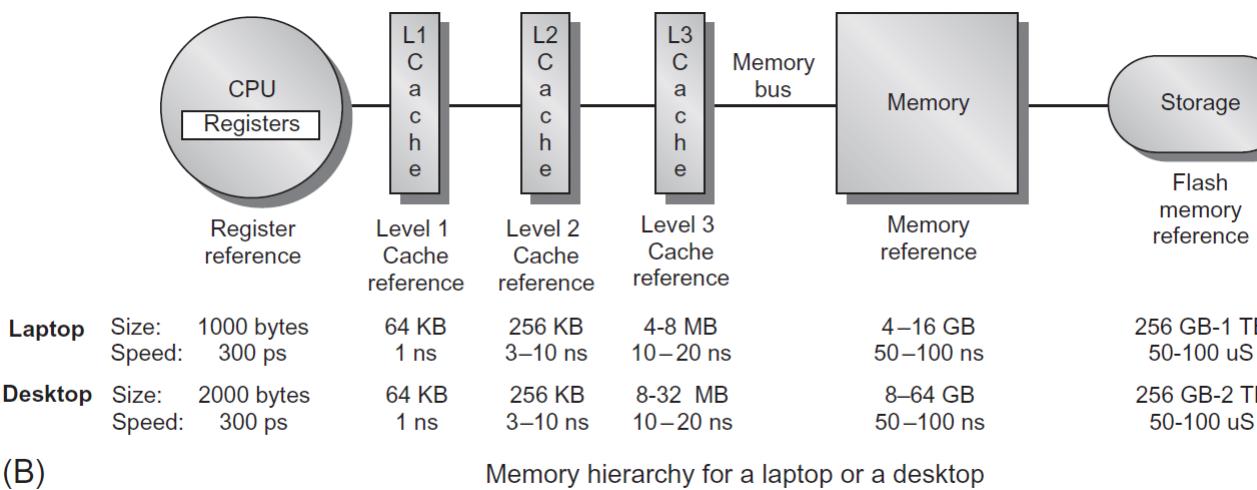
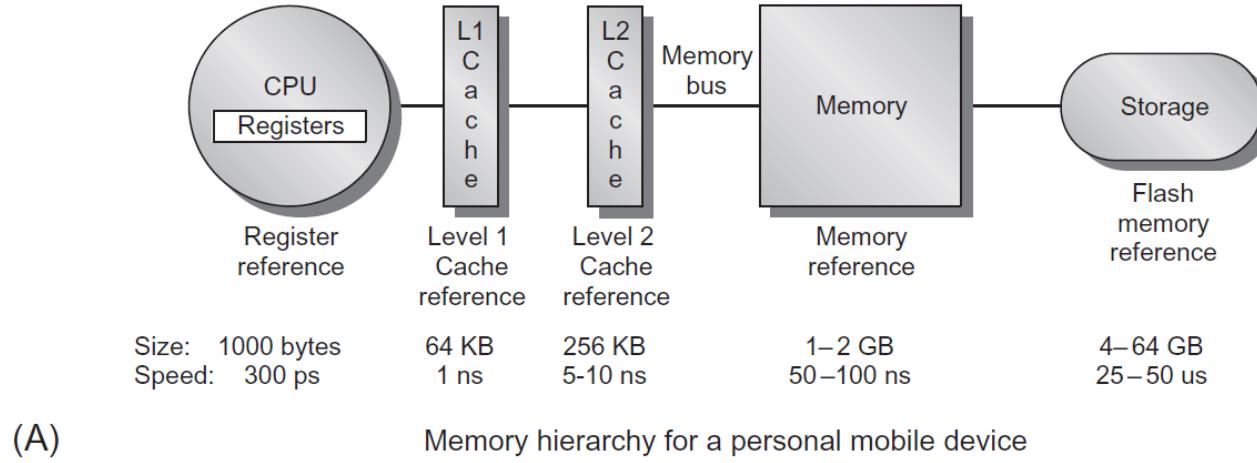


The inclusion property and data transfers between adjacent levels

Impact of Hierarchical Memory

- Memory is no more a flat structure
- Affect many aspects of a computer
 - How OS manages memory & IO
 - How compilers generate codes
 - How applications use the computers
- Programs spend more time on memory access
 - Processors *stall* waiting for data from memory
- Need to know hierarchy for optimizing

Memory Hierarchy in Operation



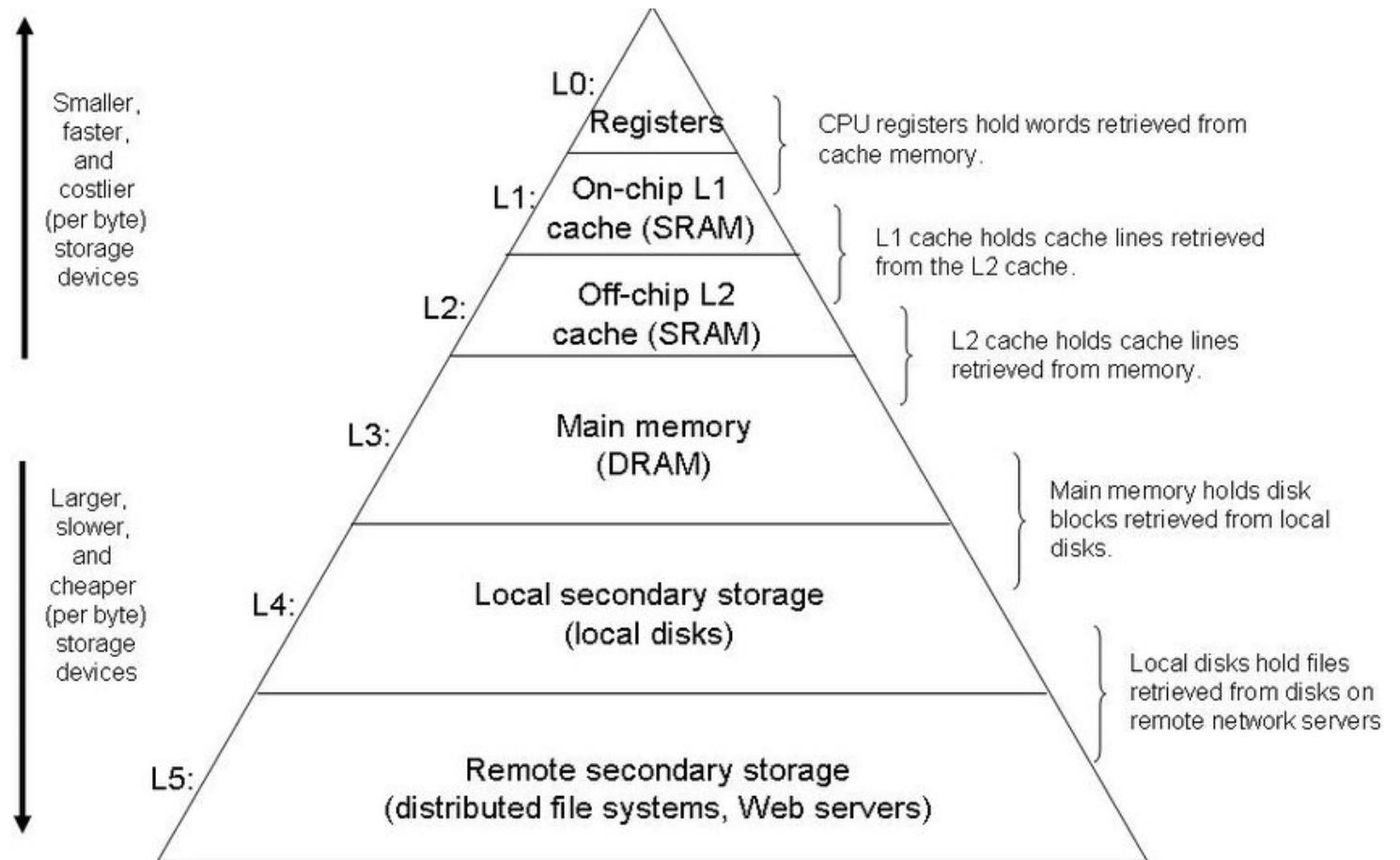
Summary

- **Memory hierarchy** - Illusion of a large amount of fast memory
- All data and instructions in memory are **not accessed at once with equal probability**
- **Principle of Locality** – accessing small portion of address space at any instance of time
 - **Temporal locality** – refer the same item again and again
 - **Spatial locality** – refer the items stored close to each other

Hierarchical Organization of Memory

- Hardware:
 - Small
 - Fast
 - Expensive
- Data:

Data in upper level
 \subseteq data in lower level

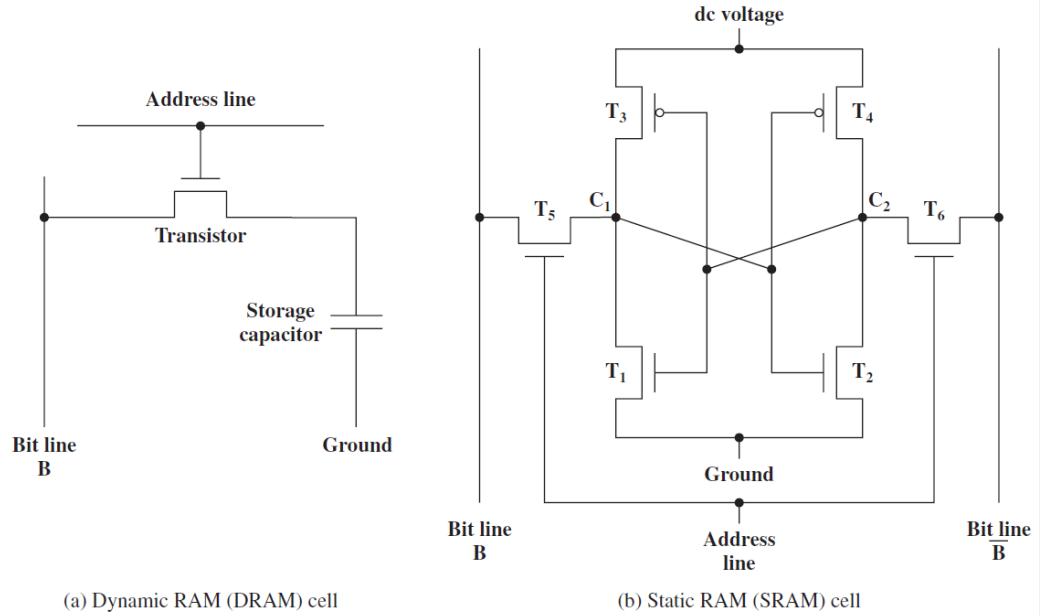


Memory Technologies

- Dynamic Random Access Memory (DRAM)
 - Less cost per bit, slow, less area per bit → larger capacity
 - Main memory made of DRAM
- Static Random Access Memory (SRAM)
 - High cost per bit, fast, more area per bit → small capacity
 - Cache memory made of SRAM

Typical access time	\$ Per GiB in 2020
0.5 – 2.5ns	\$500 – \$1000
50 – 70ns	\$3 – \$6
5μs – 50μs	\$0.06 – \$0.12
5ms – 20ms	\$0.01 – \$0.02

Memory technology	Typical access time	\$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	\$500–\$1000
DRAM semiconductor memory	50–70 ns	\$10–\$20
Flash semiconductor memory	5,000–50,000 ns	\$0.75–\$1.00
Magnetic disk	5,000,000–20,000,000 ns	\$0.05–\$0.10



Memory types

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip-level	Nonvolatile	
Electrically Erasable PROM (EEPROM)		Electrically, byte-level		
Flash memory		Electrically, block-level		

Take home assignment 2

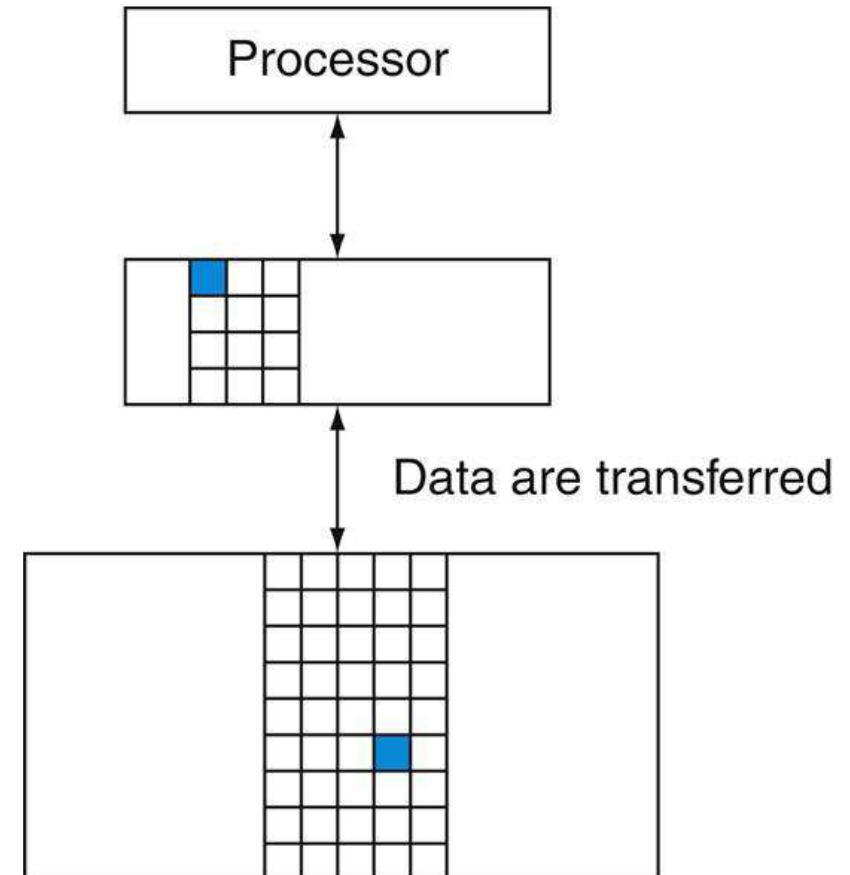
Read Section 5.2 of

Patterson, David. "Computer organization and design RISC-V edition:
the hardware." (2017) and

Answer the Quiz during my next lecture.

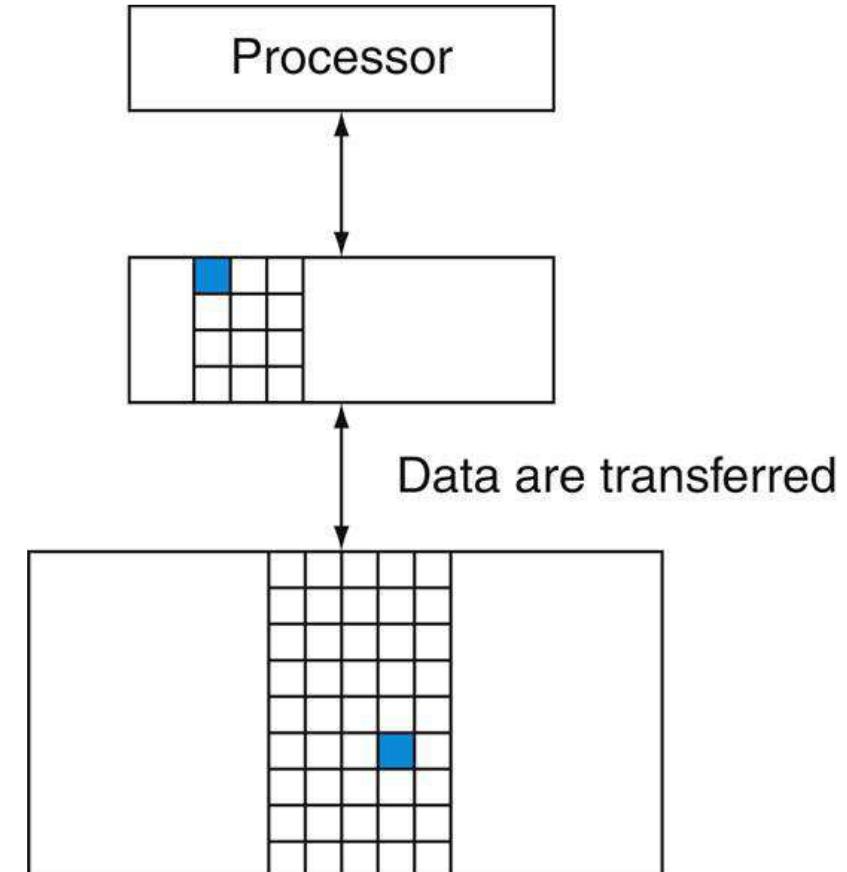
Cache

- Small amount of memory that is faster than DRAM
 - Slower than registers
 - Built using SRAM
 - Range from few KB to few MB
- Used by CPU to store frequently used instructions & data
 - Spatial & temporal locality
- Use multiple levels of cache
 - L1 Cache – Very fast, usually within CPU itself
 - L2 Cache – Slower than L1, but faster than DRAM
 - Today there's even L3 Cache

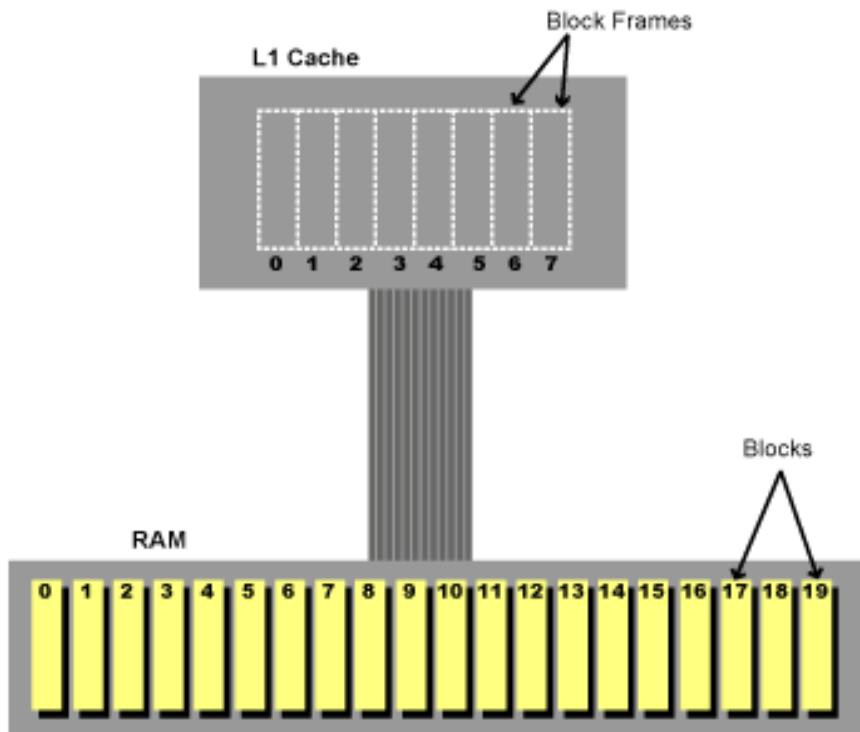


Cache - terminology

- Minimum unit of data: **block = line**
- If data requested by the processor
 - is found in Cache: **hit**
 - Is NOT found in Cache: **miss**
- **Hit ratio:** fraction of memory accesses found in cache
- **Miss ratio:** fraction of memory accesses not found in cache
- **Hit time:** time to access a block in cache
- **Miss penalty:** time to bring a block from memory plus deliver it to processor



Cache Blocks



- A collection of *words* are called a *block*
- Multiple blocks are moved between levels in cache hierarchy
- Blocks are tagged with memory address
 - Tags are searched parallel

Source:

[http://archive.arstechnica.com/paedia/c/caching
/m-caching-5.html](http://archive.arstechnica.com/paedia/c/caching/m-caching-5.html)

Cache Misses

- When required item is not found in cache
- Miss rate – fraction of cache accesses that result in a failure
- Types of misses
 - **Compulsory** – 1st access to a block
 - **Capacity** – limited cache capacity force blocks to be removed from a cache & later retrieved
 - **Conflict (collision)**– multiple blocks compete for the same set/block
- Average memory access time
= *Hit time* + *Miss rate* x *Miss penalty*

Cache Access

Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
10110_{two}	miss (5.9b)	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
11010_{two}	miss (5.9c)	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
10000_{two}	miss (5.9d)	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
00011_{two}	miss (5.9e)	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
10010_{two}	miss (5.9f)	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Cache Misses – Example

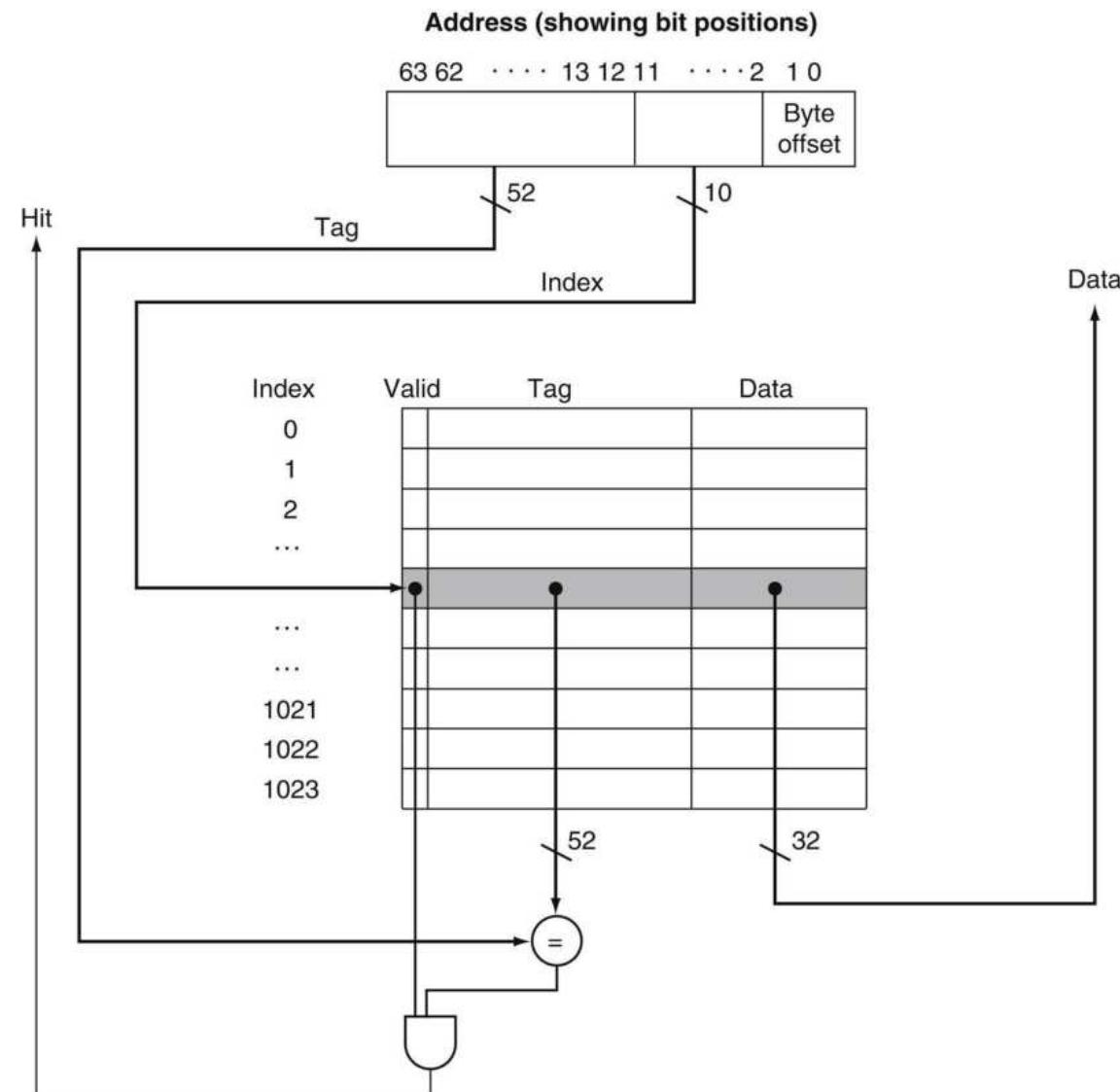
- Assume 40% of the instructions are data accessing instructions
- A hit takes 1 clock cycle & miss penalty is 100 clock cycles
- Assume instruction miss rate is 4% & data access miss rate is 12%, what is the average memory access time?

Cache Misses – Example

- Consider a cache with a block size of 4 words
- It takes 1 clock cycle to access a word in cache
- It takes 15 clock cycles to access a word from main memory
 - How much time will it take to access a word that's not in cache?
 - What is the average memory access time if miss rate is 0.4?
 - What is the average memory access time if miss rate is 0.1?

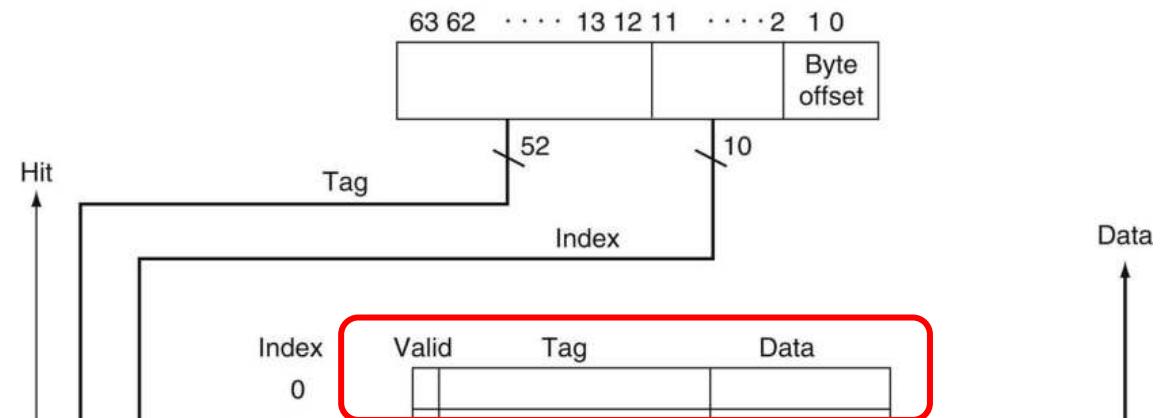
Accessing Cache

- Tag = upper portion of address
- Index = (Memory block address) modulo #Cache blocks
- Valid bit

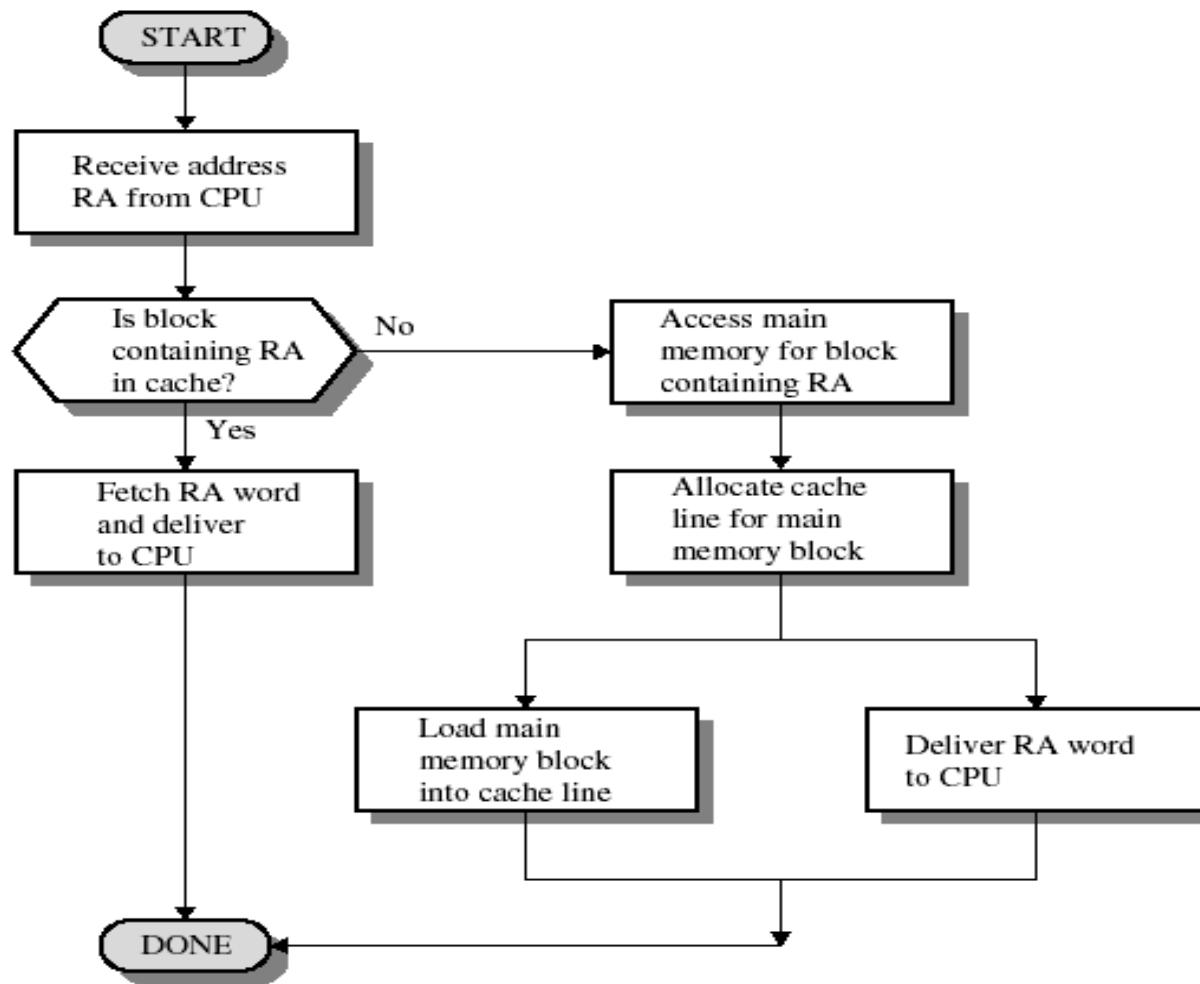


Cache organization

- In a Cache Entry
 - Valid bit
 - Tag
 - Data
- What is the size of a Cache Entry?
- Requested Address (RA) by processor is decoded as
 - Tag
 - Block index
 - Byte offset within the block

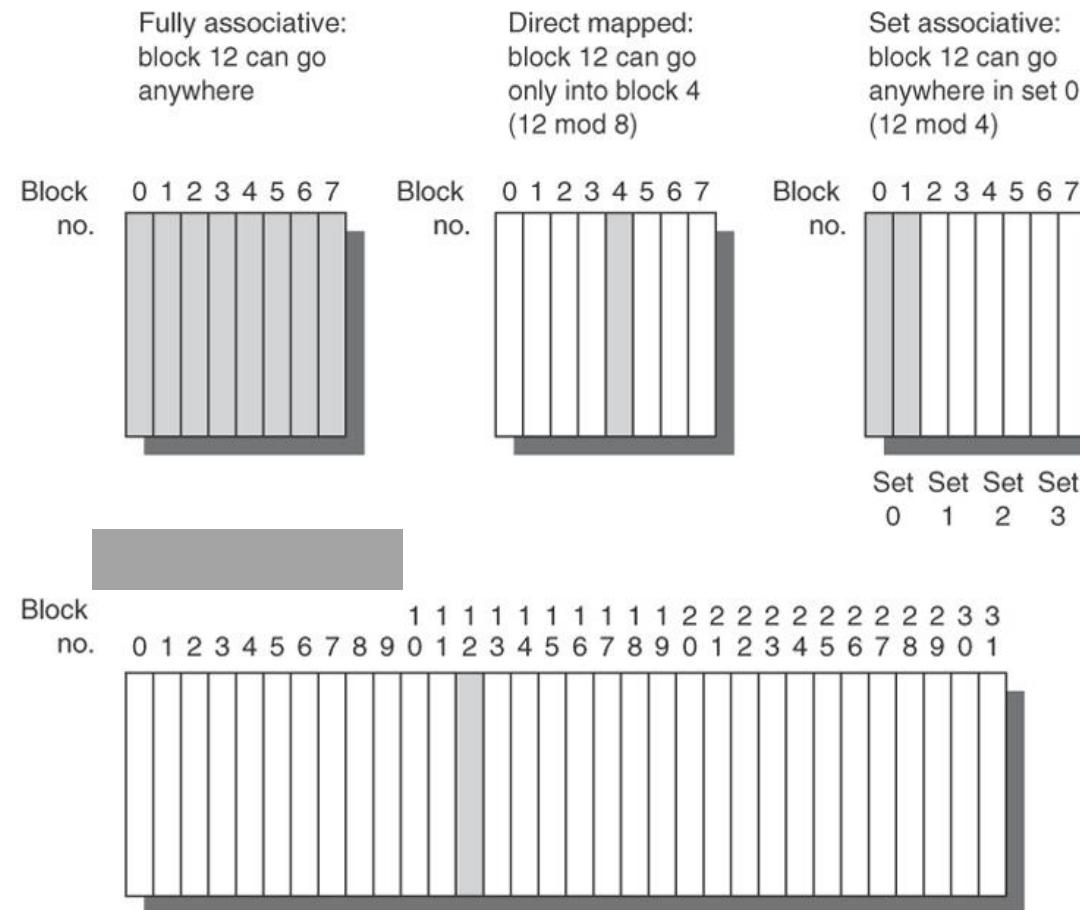


Cache Read Operations



Cache Associativity

- Defines where blocks can be placed in a cache

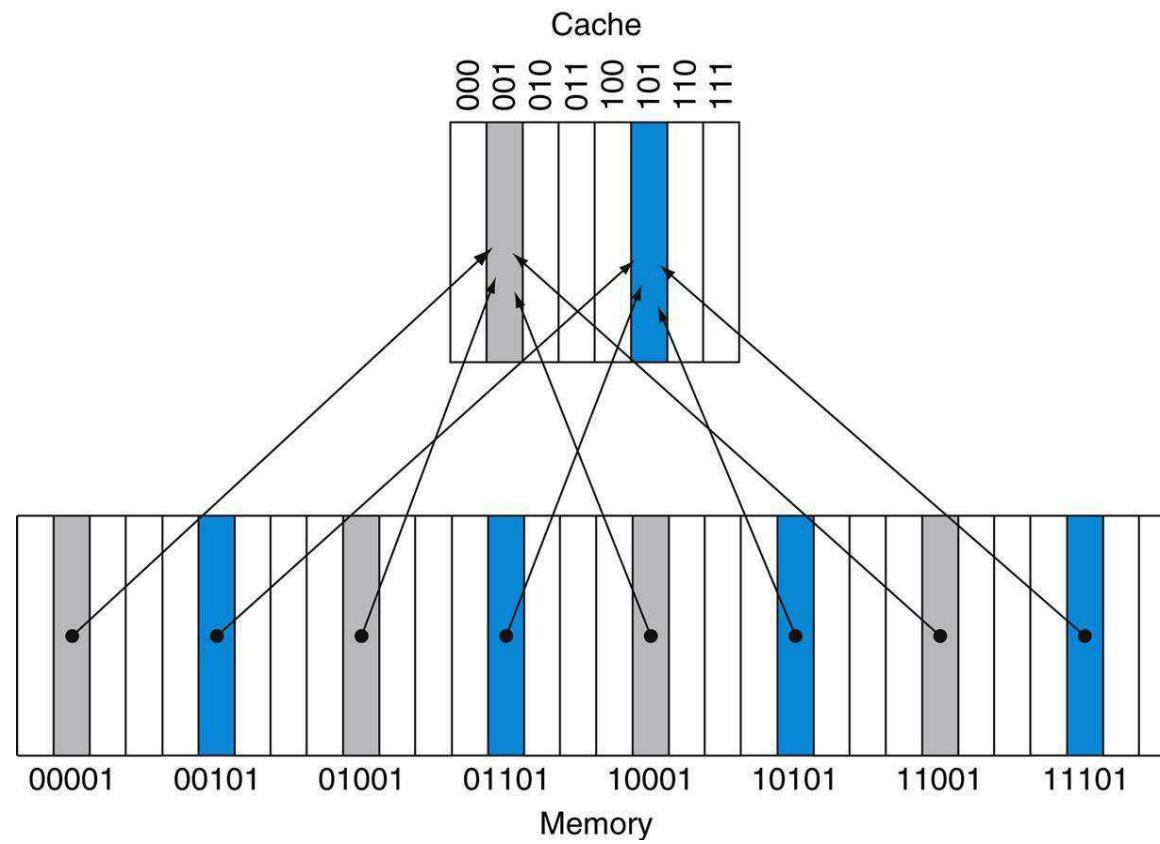


© 2007 Elsevier, Inc. All rights reserved.

Source: Computer Architecture, A Quantitative Approach by John L. Hennessy and David A. Patterson

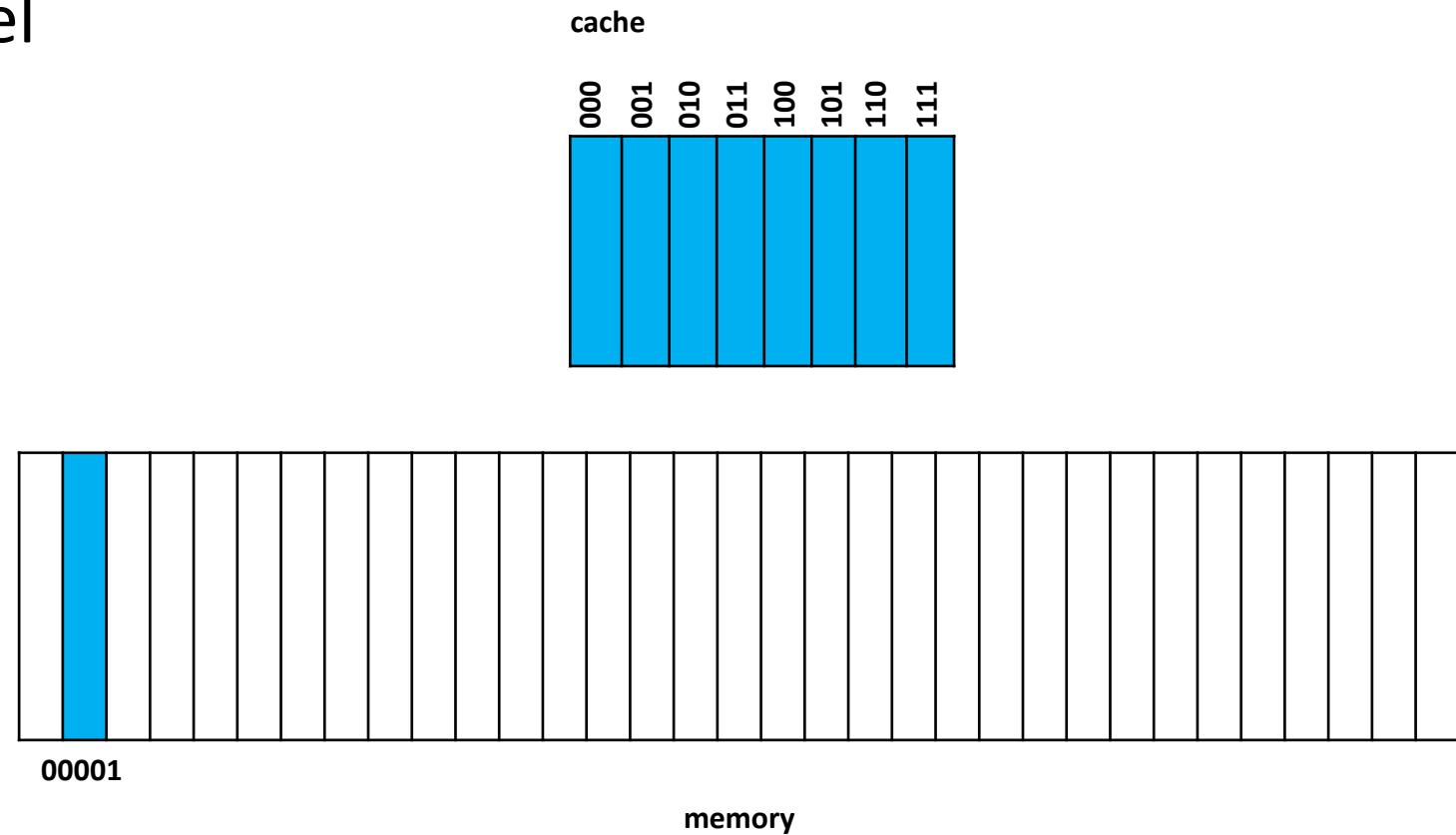
Direct mapping

- A memory block can go exactly to one place in cache



Fully associative

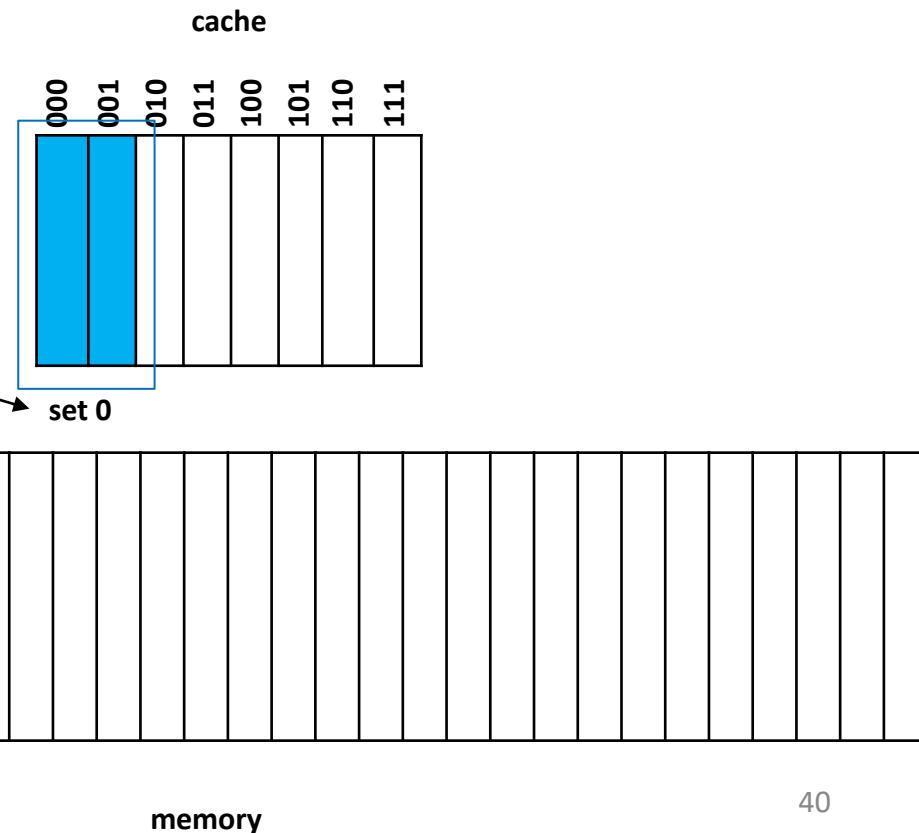
- A block in main memory can go to any block in cache
- Searched parallel



Set associative

- A block in main memory can go to any block in a set of cache blocks
- *n-way set associative*: n blocks for a set

Set index = (Memory block address) modulo #sets



Cache configuration examples

**One-way set associative
(direct mapped)**

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data														

Cache Replacement Policies

- When cache is full some of the cached blocks need to be removed before bringing new ones in
 - If cached blocks are dirty (written/updated), then they need to be written to RAM
- Cache replacement policies
 - Random
 - Least Recently Used (LRU)
 - Need to track last access time
 - Least Frequently Used (LFU)
 - Need to track no of accesses
 - First In First Out (FIFO)

Increasing Cache Performance

- Large cache capacity
- Multiple-levels of cache
- Prefetching
 - a block of data is brought into the cache before it is actually referenced
- Fully associative cache

Thank you

Memory Architecture II



CS2053 Computer Architecture
Computer Science & Engineering
University of Moratuwa

Sulochana Sooriyaarachchi
Chathuranga Hettiarachchi

Outline

- Memory types
- Memory access
- Memory hierarchy
 - Main memory
 - Cache
 - Permanent storage
- Example architecture RV32I
 - Superscalar architecture
 - Pipelining

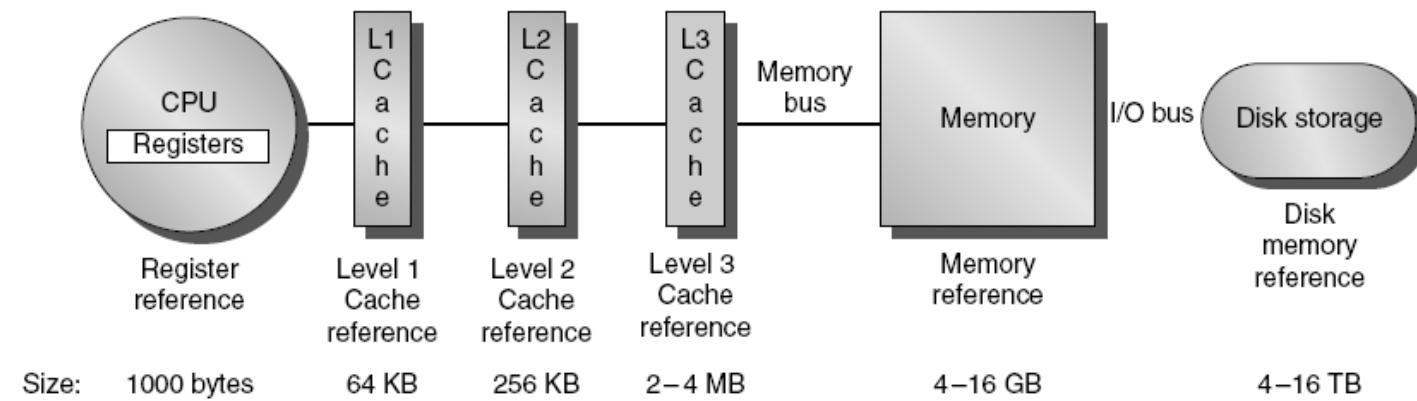
Cache



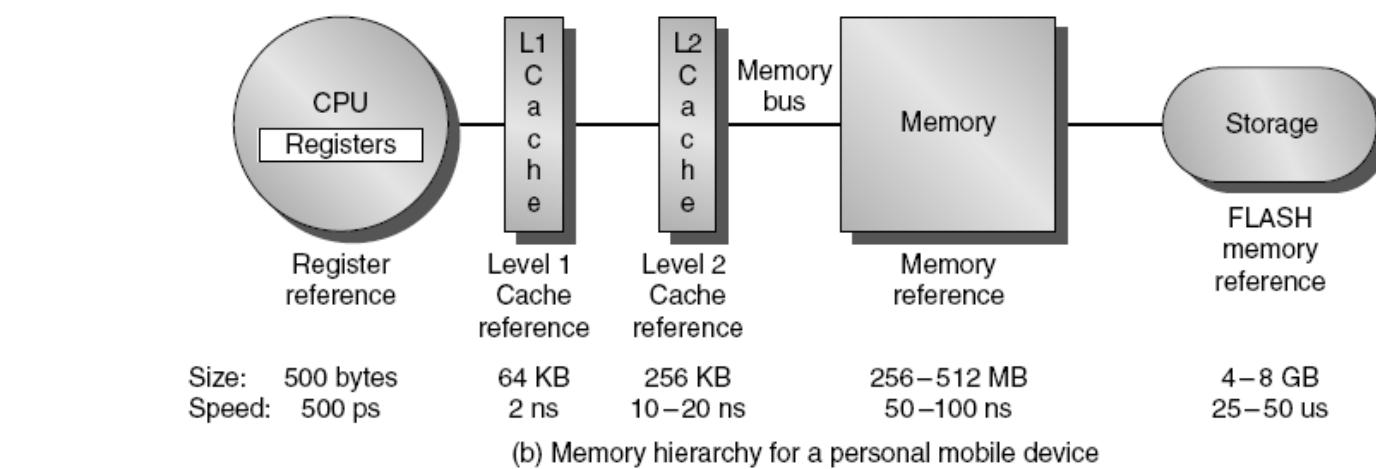
Cache Memory

- Small amount of memory that is faster than DRAM
 - Slower than registers
 - Built using SRAM
 - Range from few KB to few MB
- Used by CPU to store frequently used instructions & data
 - Spatial & temporal locality
- Use multiple levels of cache
 - L1 Cache – Very fast, usually within CPU itself
 - L2 Cache – Slower than L1, but faster than DRAM
 - Today there's even L3 Cache

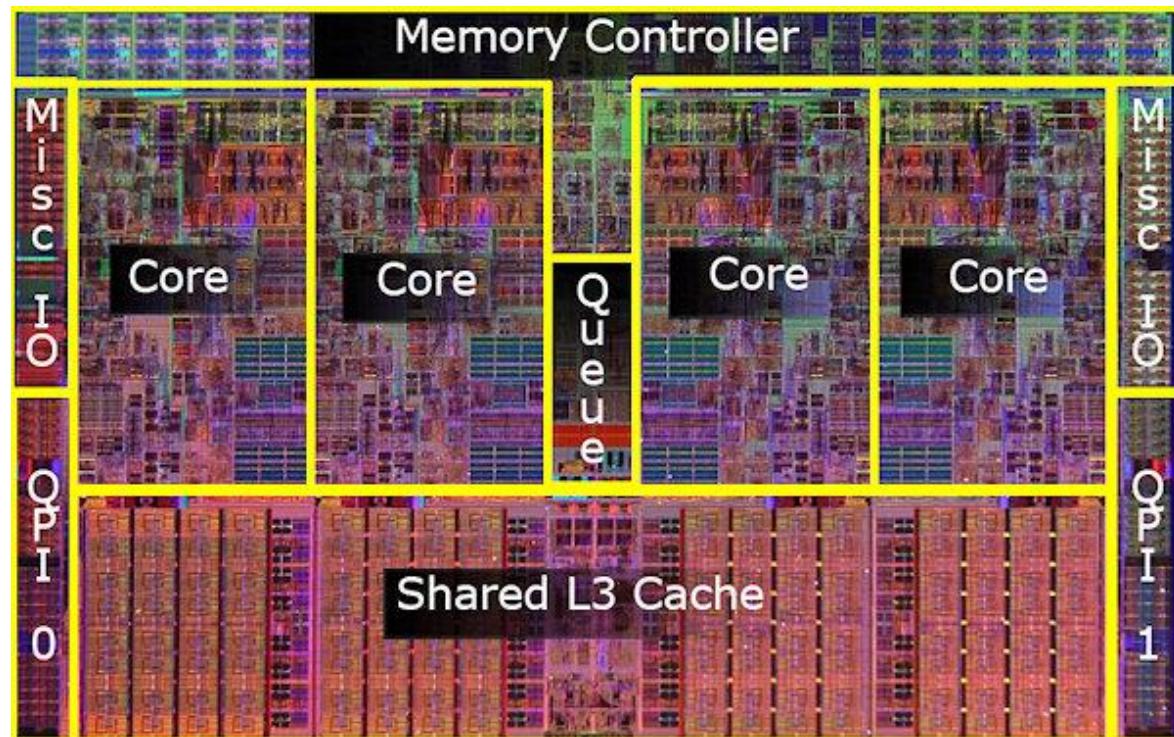
Multiple Levels of Caching



(a) Memory hierarchy for server

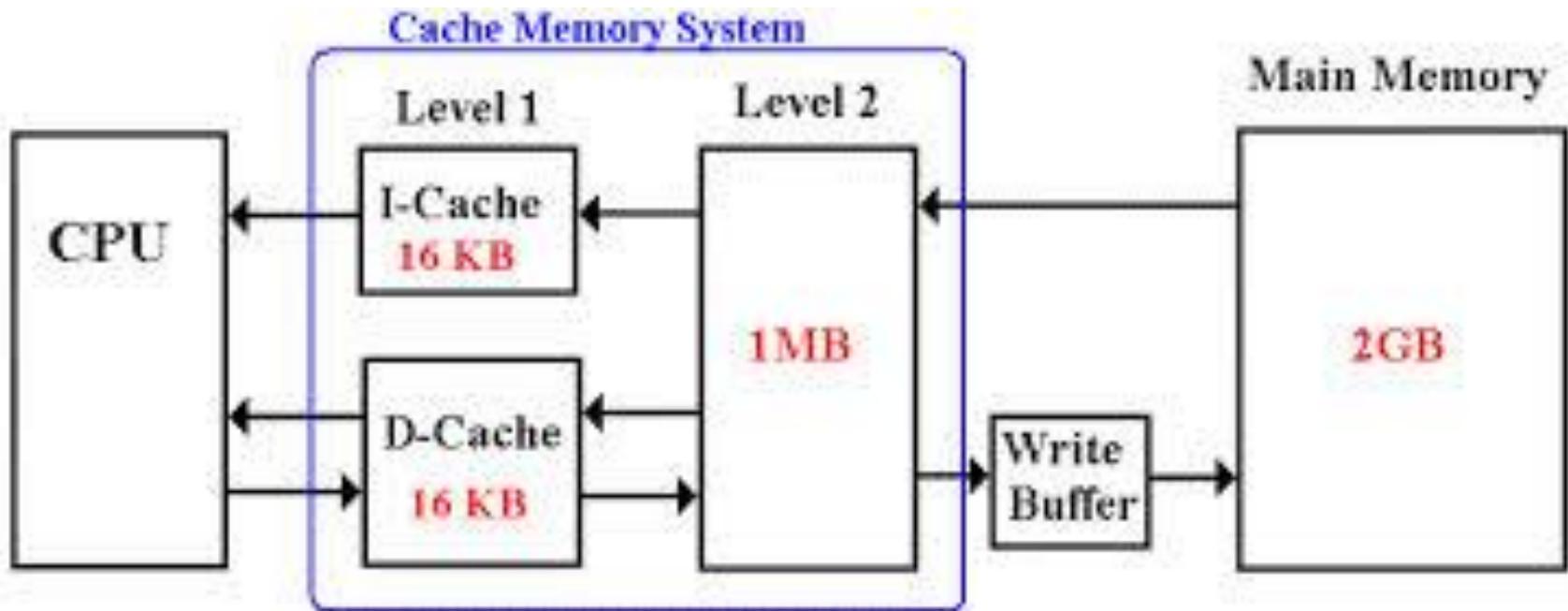


Core i7 Die & Major Components



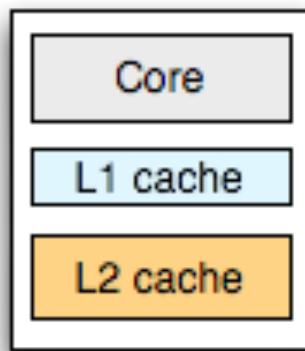
Source: Intel Inc.

L1 & L2 Cache

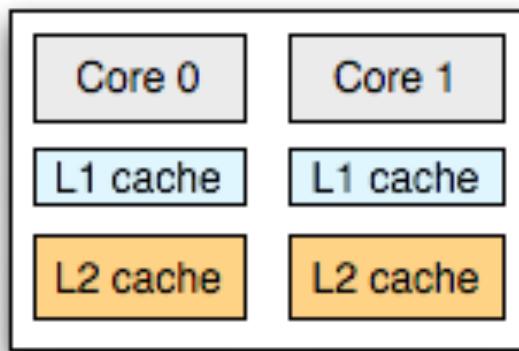


Source: www.edwardbosworth.com/CPSC2105/Lectures/Slides_06/Chapter_07/Pentium_Architecture.htm

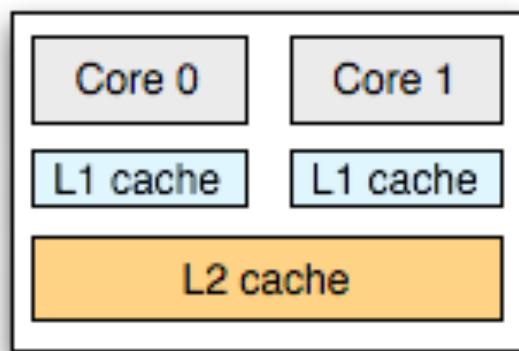
Caching in Multi-Core Systems



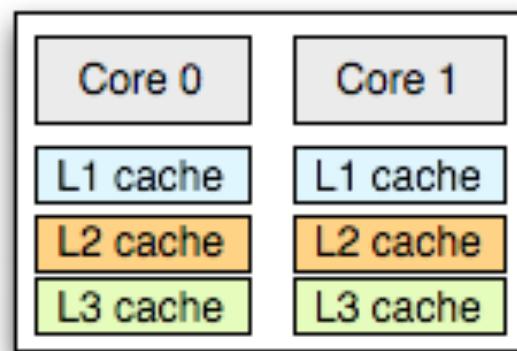
single core



AMD Opteron, Athlon

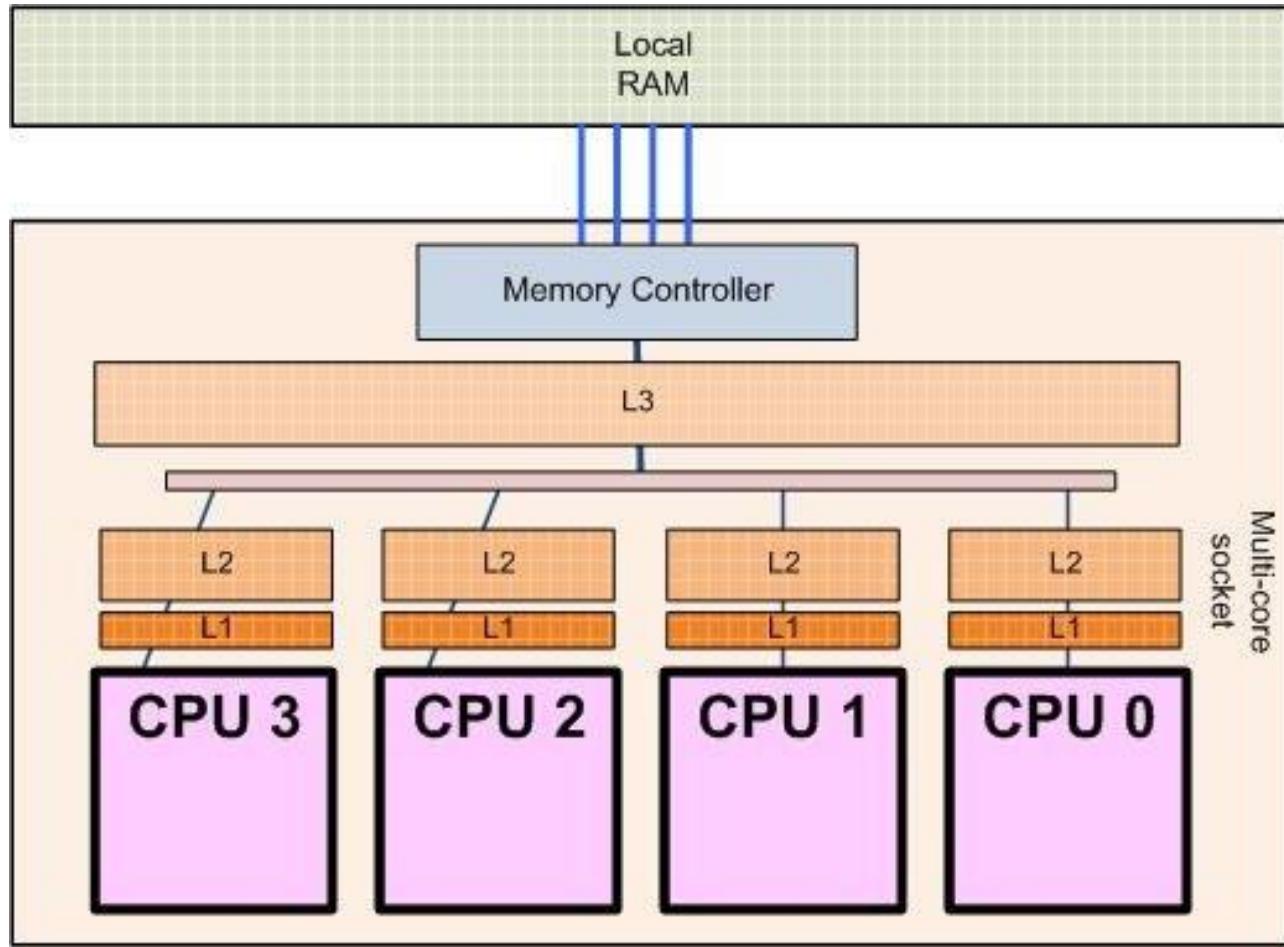


Intel Core Duo, Xeon

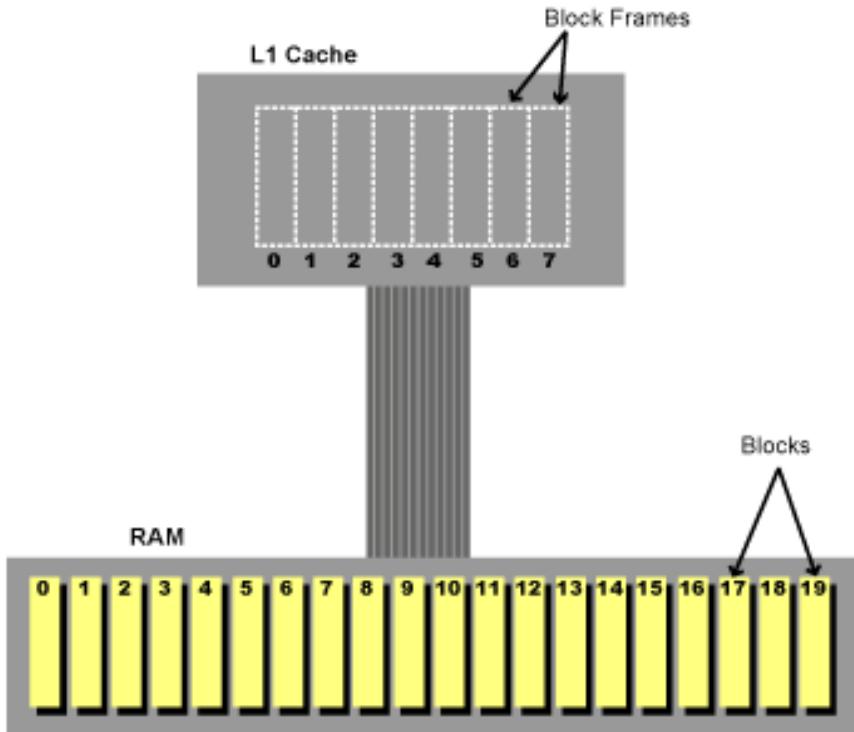


Intel Itanium 2

Caching in Multi-Core Systems (Cont.)



Cache Blocks

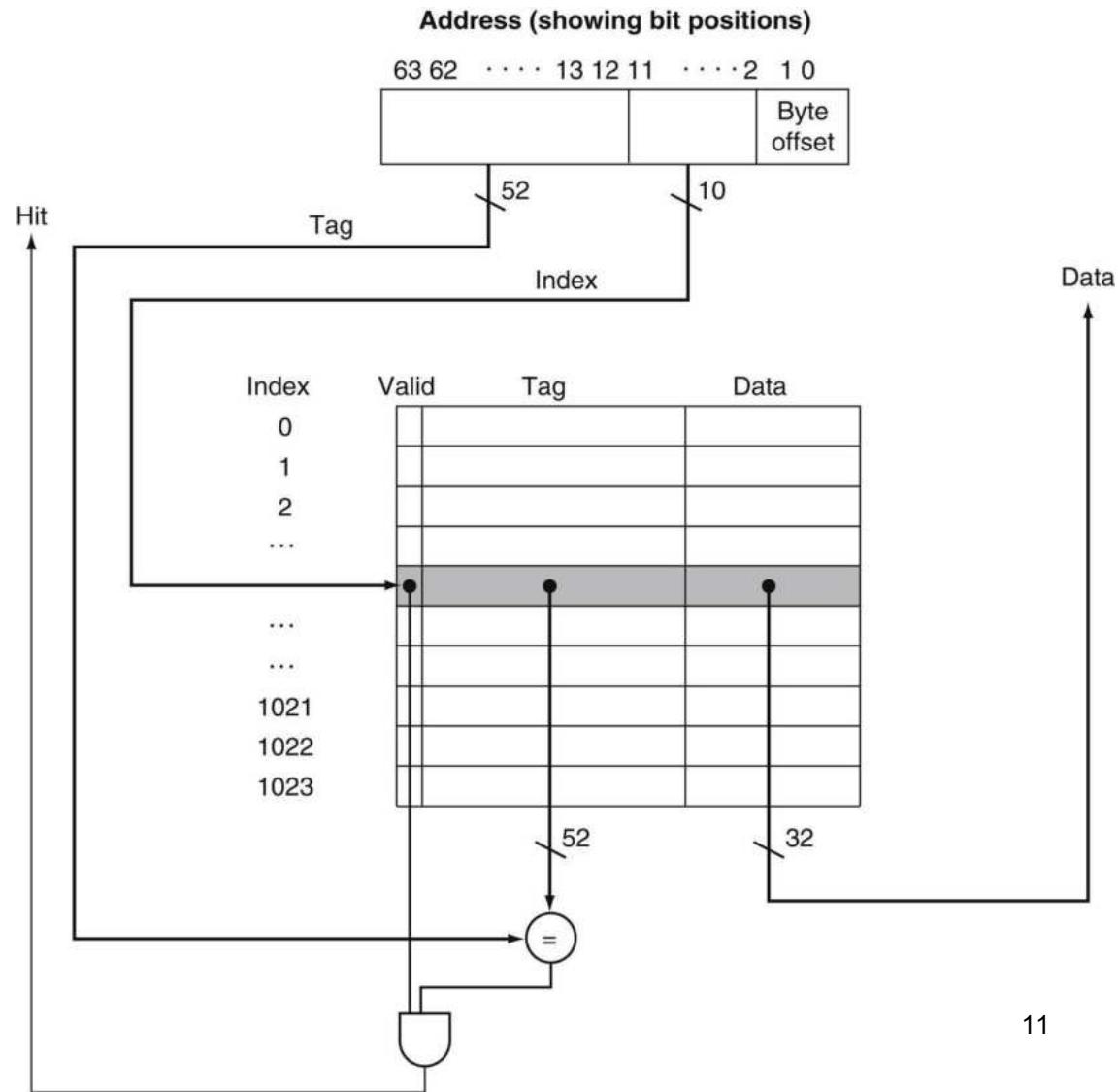


Source:
<http://archive.arstechnica.com/paedia/c/caching/m-caching-5.html>

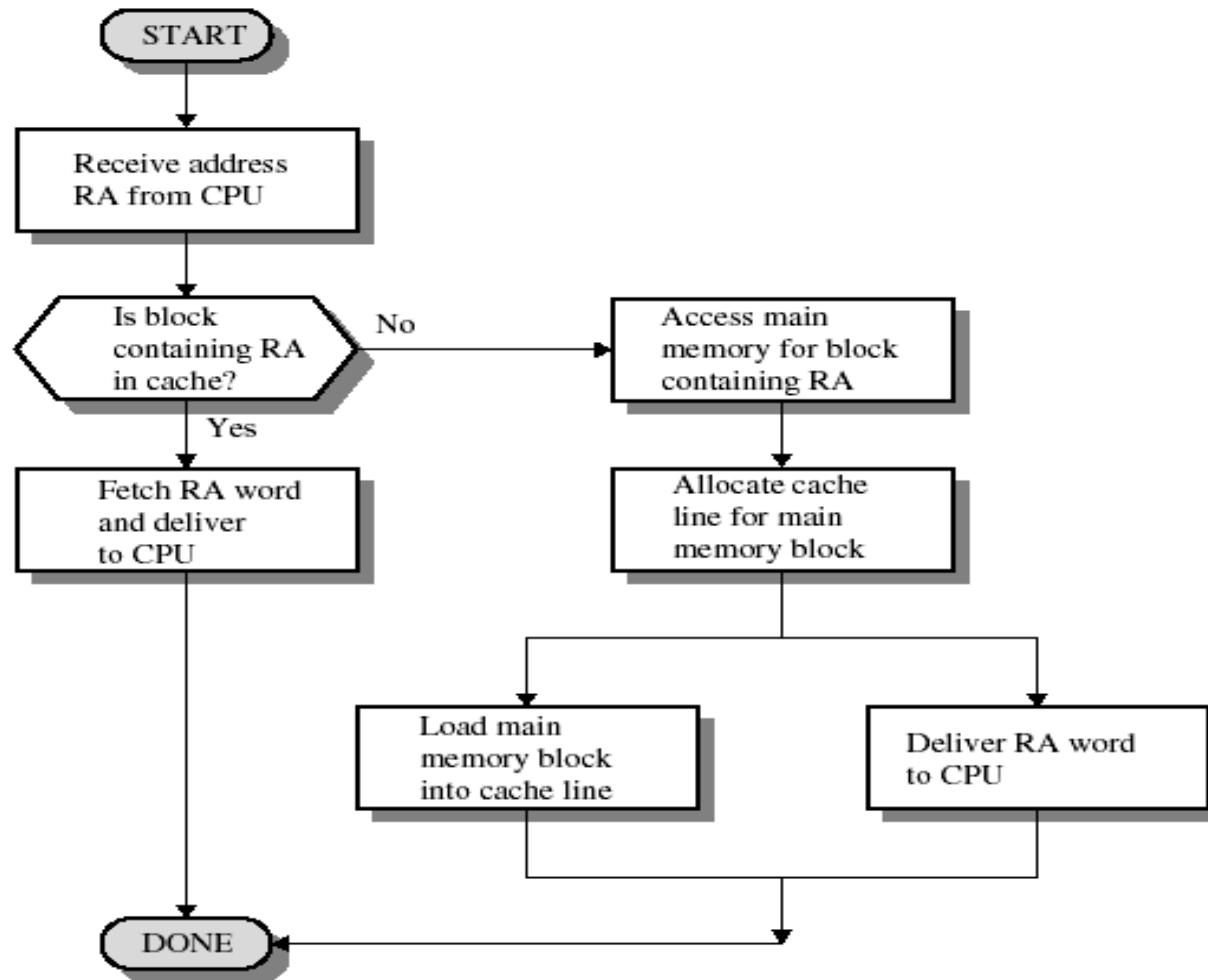
- A collection of *words* are called a *block*
- Multiple blocks are moved between levels in cache hierarchy
- Blocks are tagged with memory address
 - Tags are searched parallel

Accessing Cache

- Tag = upper portion of address
- Index = (Memory block address) modulo #Cache blocks
- Valid bit



Cache Read Operations



Cache Misses

- When required item is not found in cache
- Miss rate – fraction of cache accesses that result in a failure
- Types of misses
 - Compulsory – 1st access to a block
 - Capacity – limited cache capacity force blocks to be removed from a cache & later retrieved
 - Conflict (collision)– multiple blocks compete for the same set/block
- Average memory access time
= *Hit time* + *Miss rate* × *Miss penalty*

Cache Misses – Example

- Consider a cache with a block size of 4 words
- It takes 1 clock cycle to access a word in cache
- It takes 15 clock cycles to access a word from main memory
 - How much time will it take to access a word that's not in cache?
 - What is the average memory access time if miss rate is 0.4?
 - What is the average memory access time if miss rate is 0.1?

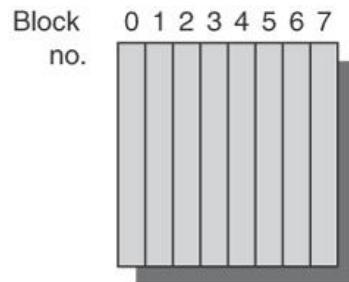
Cache Misses – Example

- Assume 40% of the instructions are data accessing instructions
- A hit takes 1 clock cycle & miss penalty is 100 clock cycles
- Assume instruction miss rate is 4% & data access miss rate is 12%, what is the average memory access time?

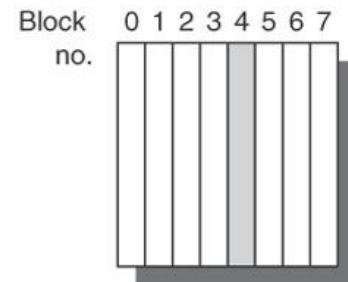
Cache Associativity

- Defines where blocks can be placed in a cache

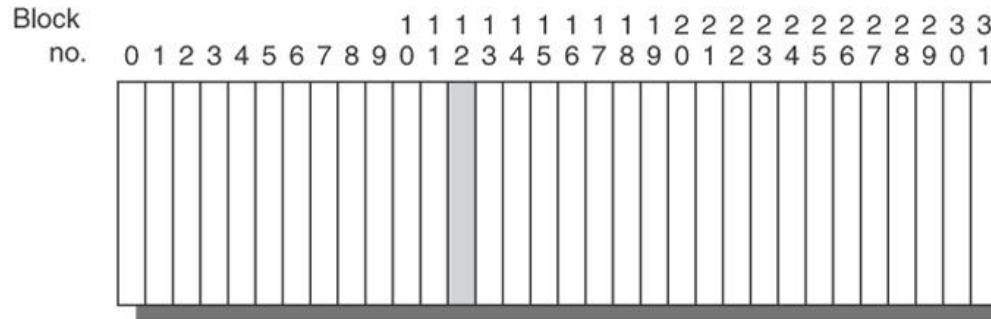
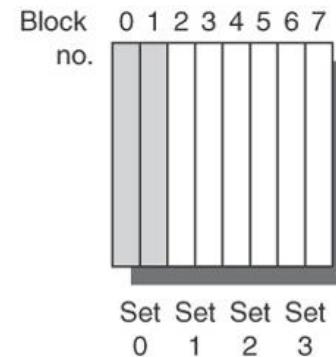
Fully associative:
block 12 can go
anywhere



Direct mapped:
block 12 can go
only into block 4
($12 \bmod 8$)



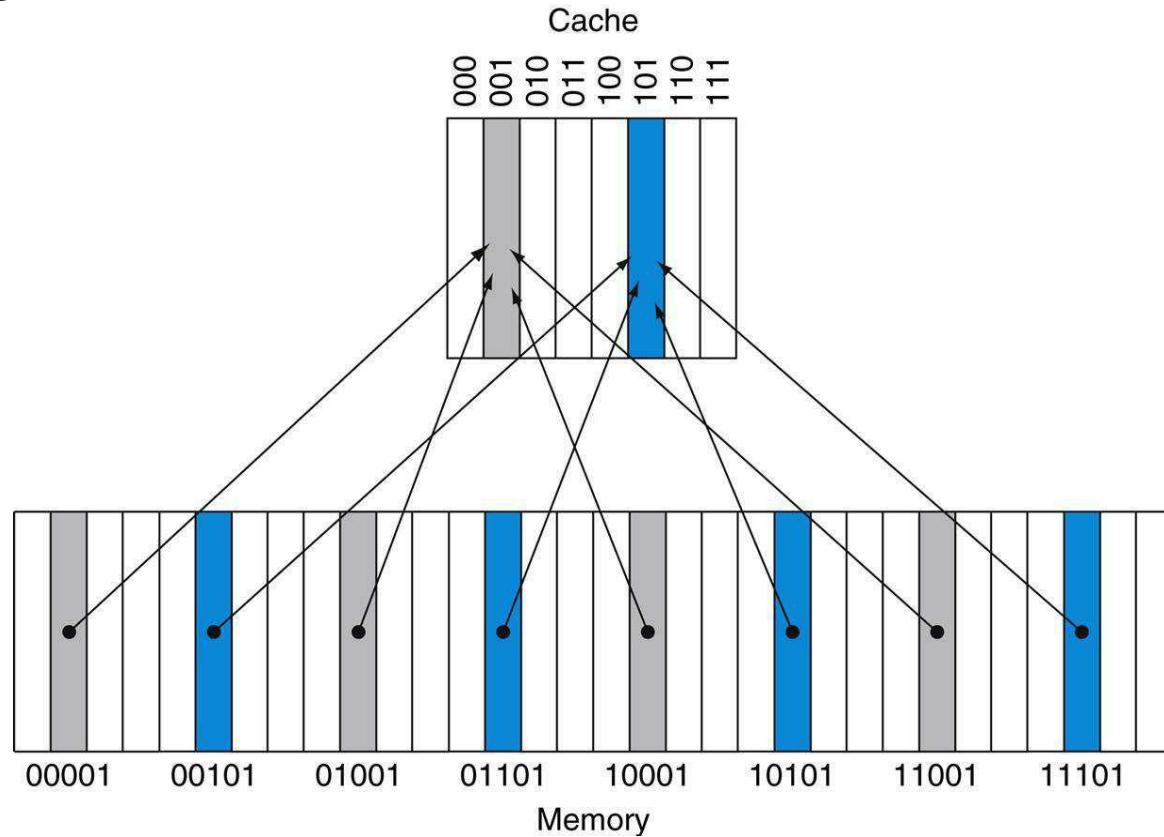
Set associative:
block 12 can go
anywhere in set 0
($12 \bmod 4$)



© 2007 Elsevier, Inc. All rights reserved.

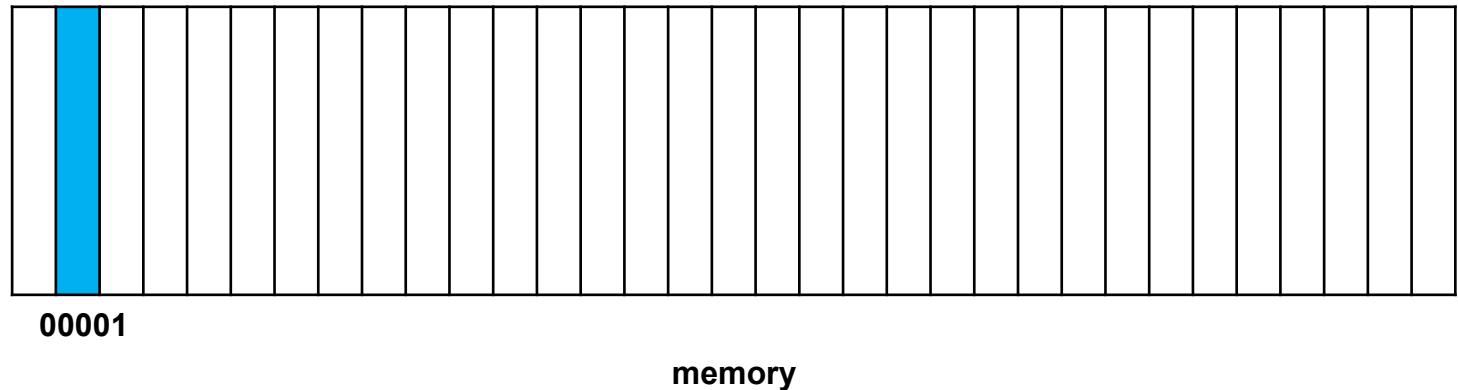
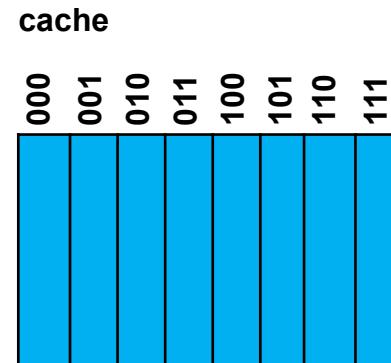
Direct mapping

- A memory block can go exactly to one place in cache



Fully associative

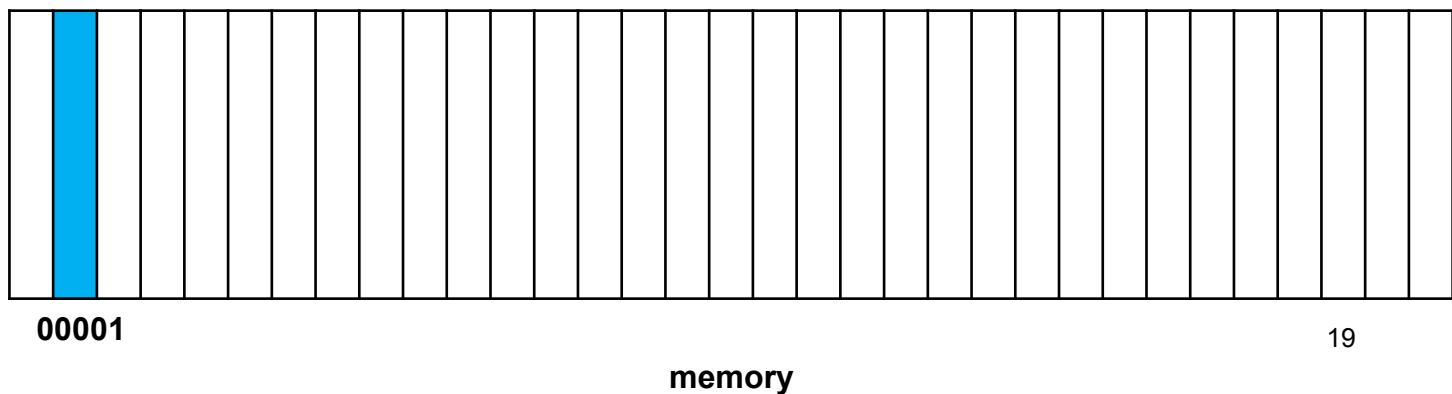
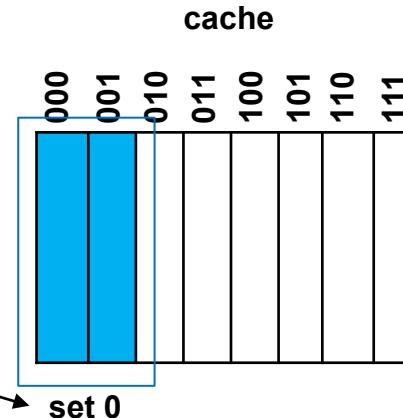
- A block in main memory can go to any block in cache
- Searched parallel



Set associative

- A block in main memory can go to any block in a set of cache blocks
- *n-way set associative*: n blocks for a set

Set index = (Memory block address) modulo #sets



Cache configuration examples

**One-way set associative
(direct mapped)**

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data														

Cache Replacement Policies

- When cache is full some of the cached blocks need to be removed before bringing new ones in
 - If cached blocks are dirty (written/updated), then they need to be written to RAM
- Cache replacement policies
 - Random
 - Least Recently Used (LRU)
 - Need to track last access time
 - Least Frequently Used (LFU)
 - Need to track no of accesses
 - First In First Out (FIFO)

Increasing Cache Performance

- Large cache capacity
- Multiple-levels of cache
- Prefetching
 - a block of data is brought into the cache before it is actually referenced
- Fully associative cache

Prefetching – Example

□ Which of the following code is faster?

```
sum = 0;  
for (i = 0; i < n; i++)  
    for (j = 0; j < m; j++)  
        sum += a[i][j];  
return sum;
```

```
sum = 0;  
for (j = 0; j < m; j++)  
    for (i = 0; i < n; i++)  
        sum += a[i][j];  
return sum;
```

		Programmer's view of matrix			
		Col 1	Col 2	Col 3	Col 4
Row 1	1, 1	1, 2	1, 3	1, 4	
	2, 1	2, 2	2, 3	2, 4	
Row 3	3, 1	3, 2	3, 3	3, 4	



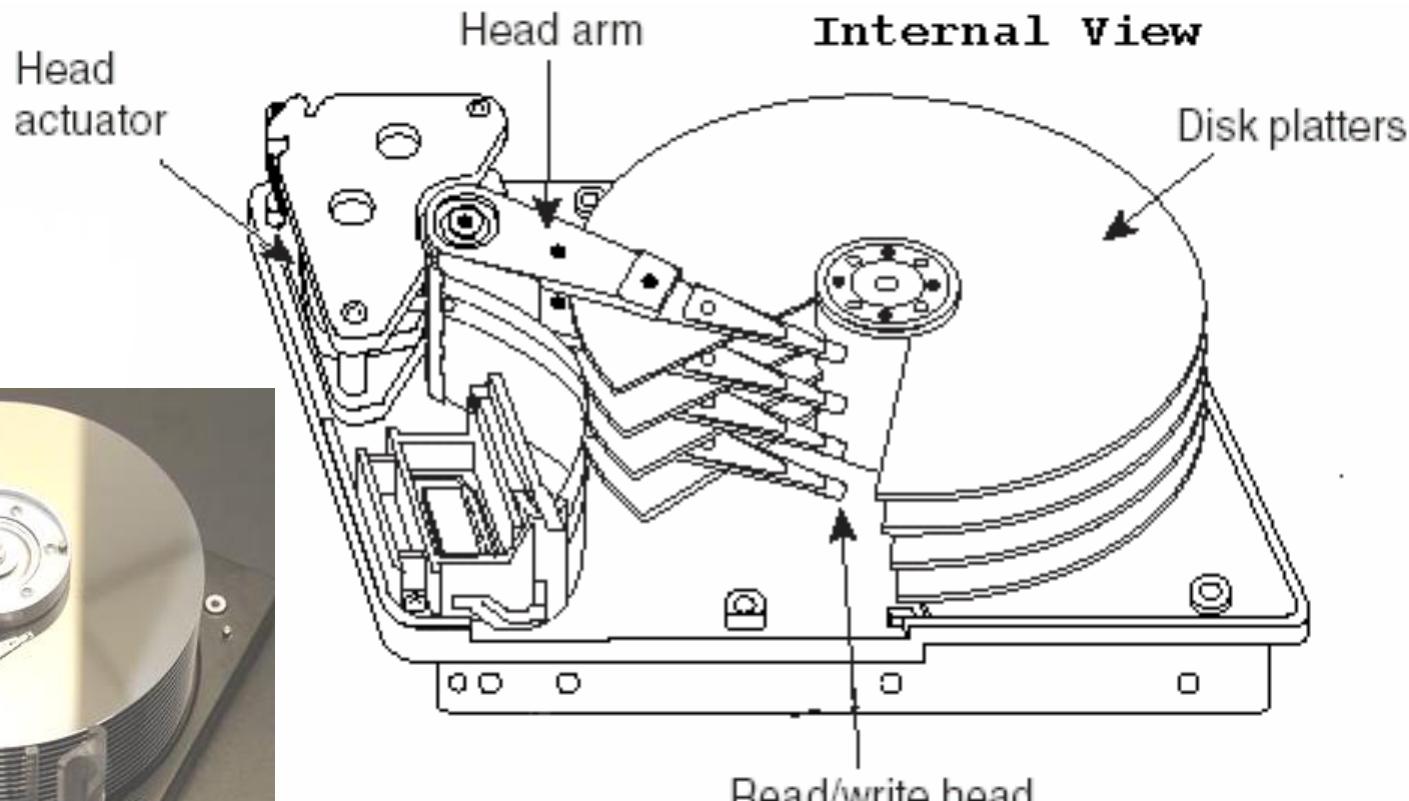
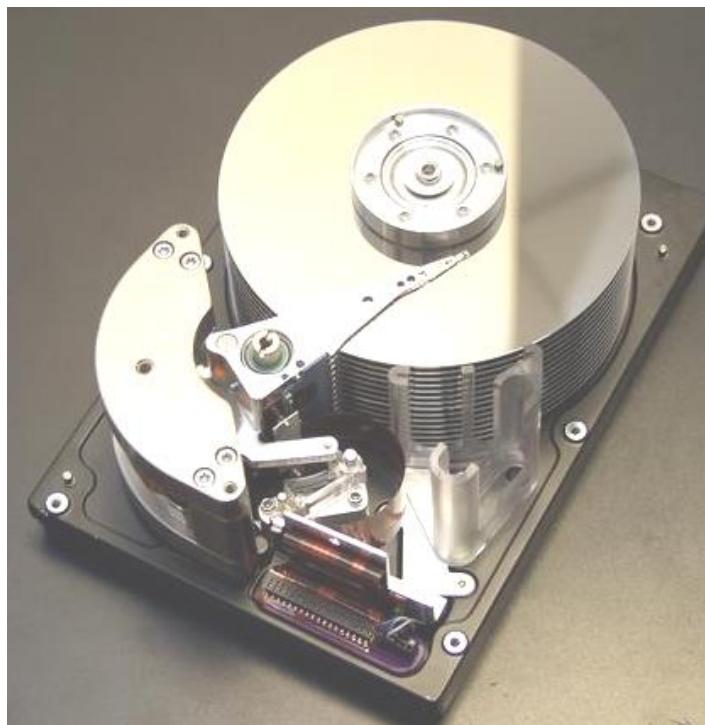
Intel Pentium 4 vs. AMD Opteron Memory Hierarchy

CPU	Pentium 4 (3.2 GHz)	Opteron (2.8 GHz)
Instruction Cache	Trace Cache 8K micro-ops	2-way associative, 64 KB, 64B block
Data Cache	8-way associative, 16 KB, 64B block, inclusive in L2	2-way associative, 64 KB, 64B block, exclusive to L2
L2 Cache	8-way associative, 2 MB, 128B block	16-way associative, 1 MB, 64B block
Prefetch	8 streams to L2	1 stream to L2
Memory	200 MHz x 64 bits	200 MHz x 128 bits

Permanent Storage

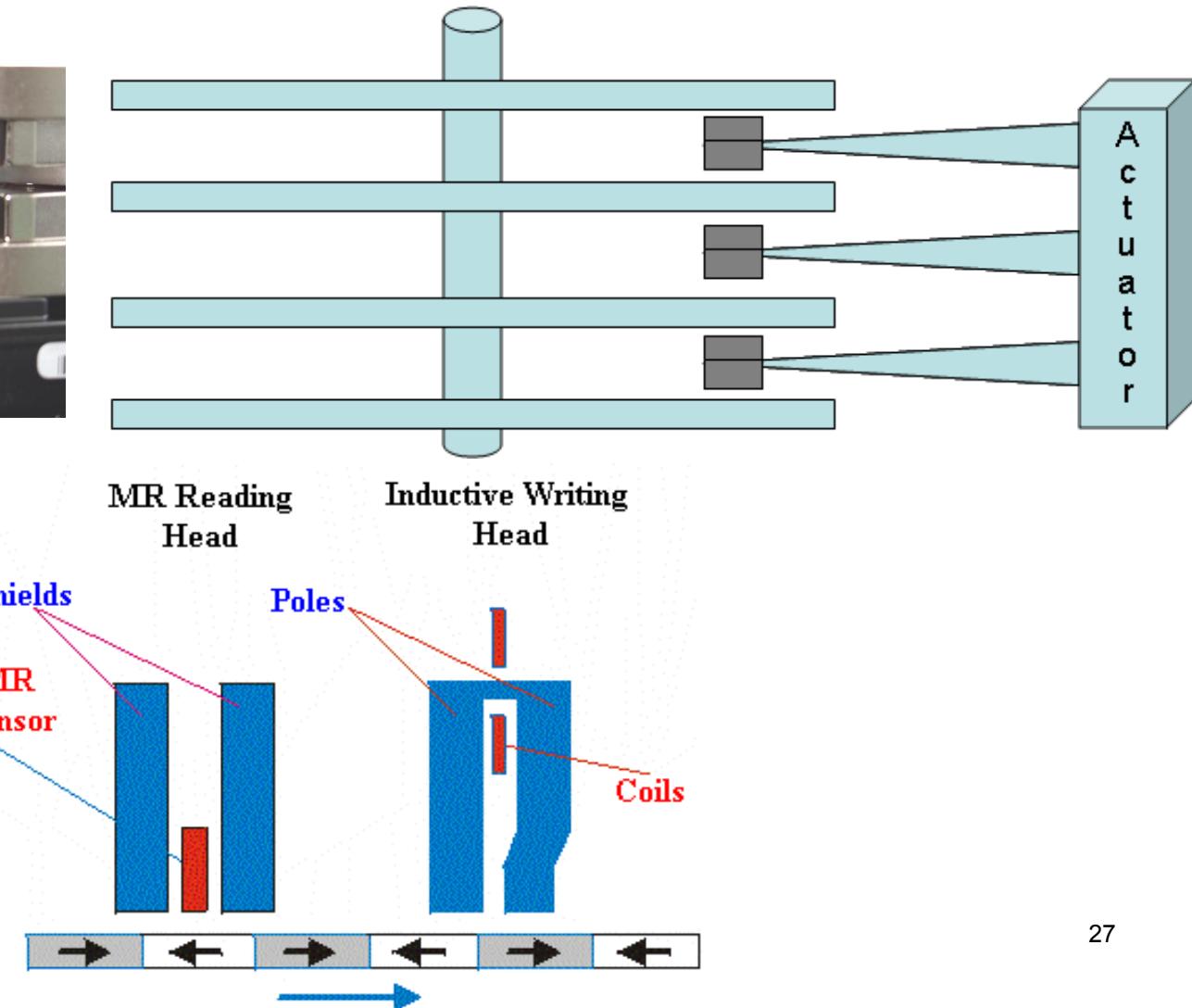


Permanent Storage – Hard Disks



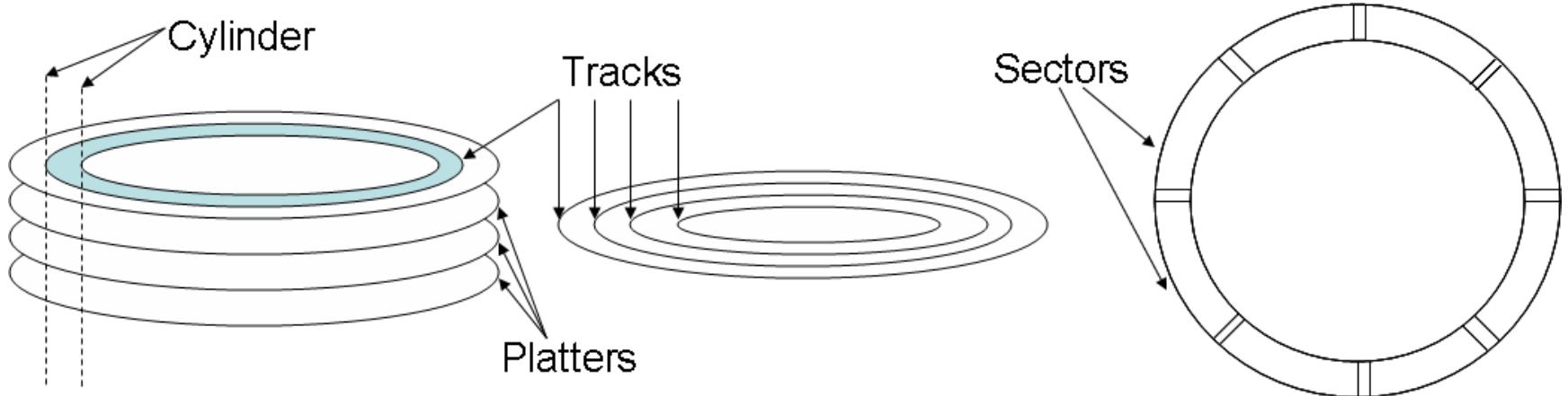
- Rigid disk (aluminum or glass) with a magnetic coating

Hard Disks (Cont.)



Source: www.dataclinic.it/data-recovery/hard-disk-functionality.htm

Cylinders, Tracks, & Sectors



- Data access time depends on
 - Seek time – time to position head over desired track
 - Rotational latency – time till desire sector is under the head
 - Read time – actual time to read data

Classification of Hard Disks

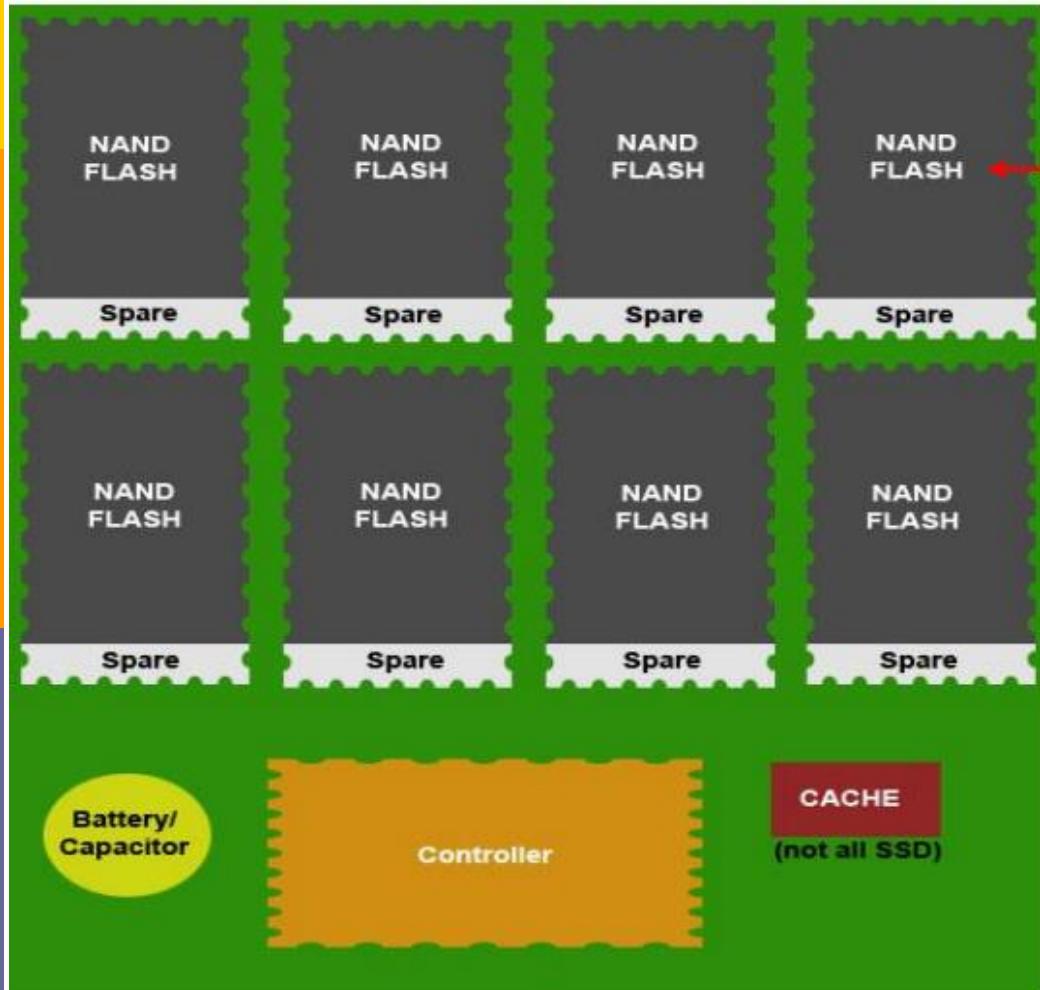
- ❑ Disk capacity
 - 250 MB, 1GB, 60GB, 500GB, 1TB, 2TB
- ❑ Type of controller
 - IDE, SCSI, SATA, eSATA, SAS
- ❑ Speed (rpm)
 - 3600, 5,400, 7,200, 10,000, 15,000

Solid State Drive (SSD)

- No moving parts
- Permanent storage
 - Based on flash memory technologies
- High speed, large capacity, & robust
- More expensive
- Used in tablets, thin laptops, laptops

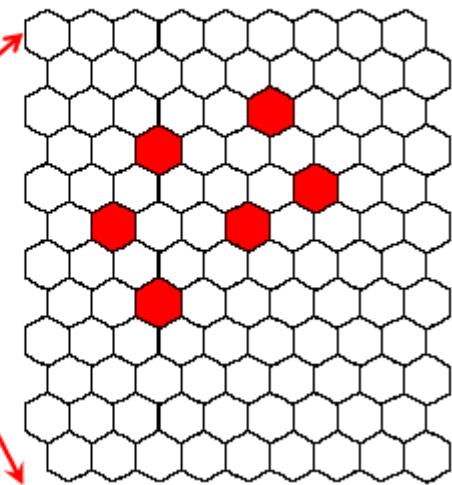
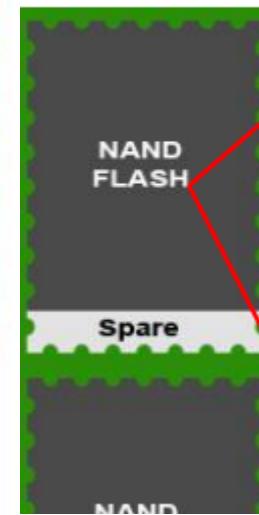


SSD Layout



Each chip
is usually
2GB to
128GB

Non volatile memory



Source:

www.blog.solidstatediskshop.com/2012/how-does-an-ssd-write-part-2/

Intel Optane (Cont.)

3D XPoint™ Technology: An Innovative, High-Density Design

Cross Point Structure

Perpendicular wires connect submicroscopic columns. An individual memory cell can be addressed by selecting its top and bottom wire.

Non-Volatile

3D XPoint™ Technology is non-volatile—which means your data doesn't go away when your power goes away—making it a great choice for storage.

High Endurance

Unlike other storage memory technologies, 3D XPoint™ Technology is not significantly impacted by the number of write cycles it can endure, making it more durable.

Stackable

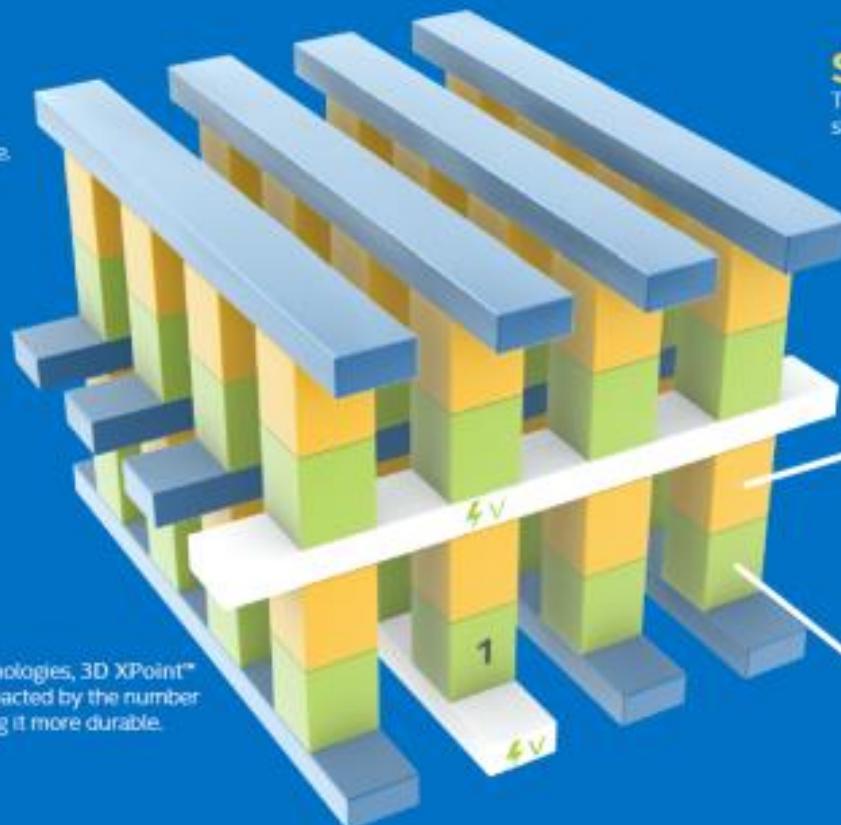
These thin layers of memory can be stacked to further boost density.

Selector

Whereas DRAM requires a transistor at each memory cell—making it big and expensive—the amount of voltage sent to each 3D XPoint™ Technology selector enables its memory cell to be written to or read without requiring a transistor.

Memory Cell

Each memory cell can store a single bit of data.





SSD VS HDD

Usually 10 000 or 15 000 rpm SAS drives

0.1 ms

Access times

SSDs exhibit virtually no access time

5.5 ~ 8.0 ms

SSDs deliver at least

6000 io/s

Random I/O Performance

SSDs are at least 15 times faster than HDDs

HDDs reach up to

400 io/s

SSDs have a failure rate of less than

0.5 %

Reliability

This makes SSDs 4 - 10 times more reliable

HDD's failure rate fluctuates between

2 ~ 5 %

SSDs consume between

2 & 5 watts

Energy savings

This means that on a large server like ours, approximately 100 watts are saved

HDDs consume between

6 & 15 watts

SSDs have an average I/O wait of

1 %

CPU Power

You will have an extra 6% of CPU power for other operations

HDDs' average I/O wait is about

7 %

the average service time for an I/O request while running a backup remains below

20 ms

Input/Output request times

SSDs allow for much faster data access

the I/O request time with HDDs during backup rises up to

400~500 ms

SSD backups take about

6 hours

Backup Rates

SSDs allows for 3 - 5 times faster backups for your data

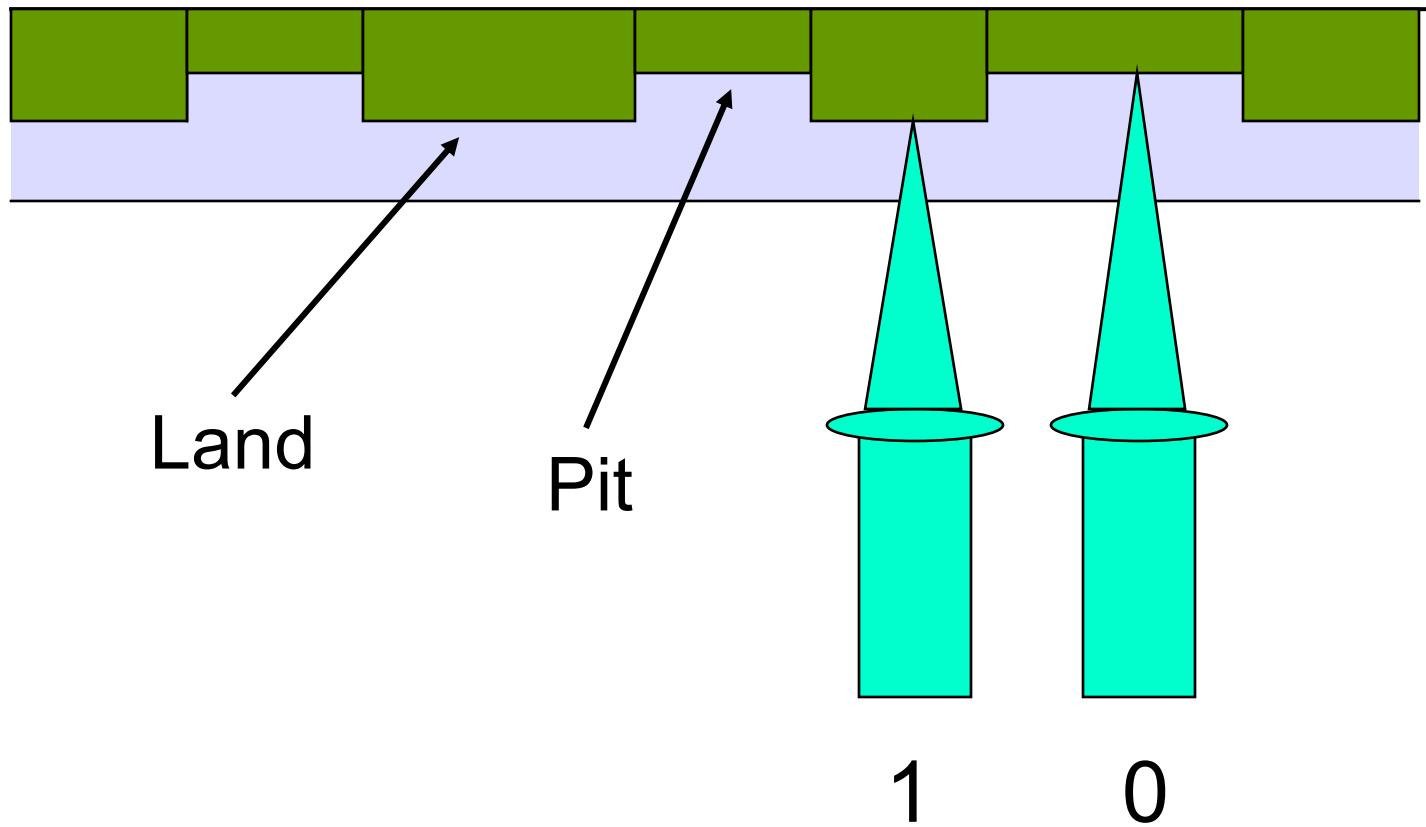
HDD backups take up to

20~24 hours

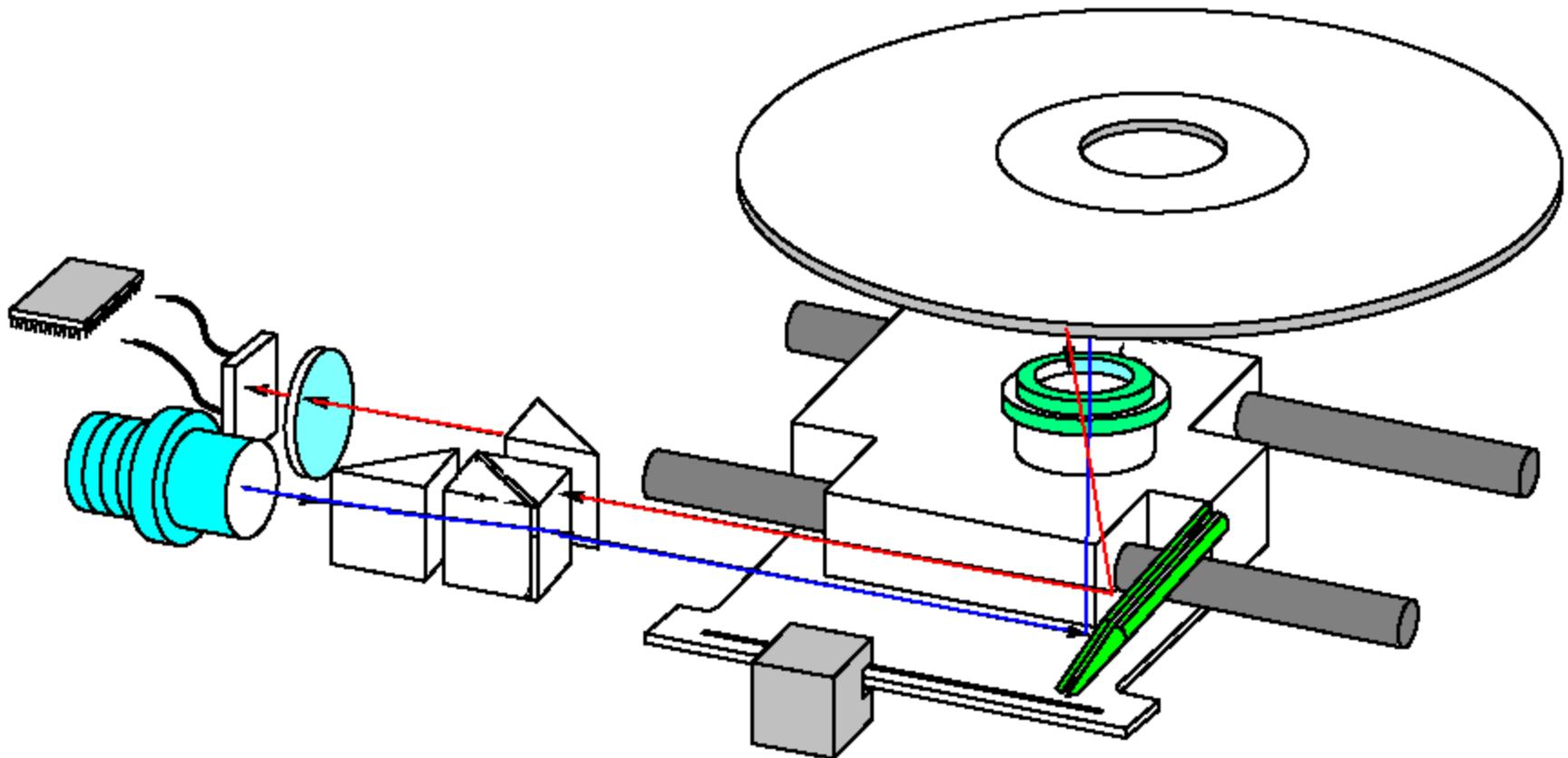
Optical Storage

- Make use of light instead of magnetism
- Different forms of optical storage
 - CD-ROM
 - CD-R – *Recordable*
 - CD-RW –*Rewritable*
 - DVD – Digital Versatile/Video Disk
 - DVD-R/DVD-RW
 - Blu-ray

Geometry of a CD



Components of a CD-ROM drive



References

- Waterman, A., Lee, Y., Patterson, D. A., & Asanovi, K. (2014). *The risc-v instruction set manual. volume 1: User-level isa, version 2.0.* California Univ Berkeley Dept of Electrical Engineering and Computer Sciences.
- Harris, S. L., Chaver, D., Piñuel, L., Gomez-Perez, J. I., Liaqat, M. H., Kakakhel, Z. L., ... & Owen, R. (2021, August). RVfpga: Using a RISC-V Core Targeted to an FPGA in Computer Architecture Education. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)* (pp. 145-150). IEEE.
- Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.
- Patterson, D. A., & Hennessy, J. L. (2016). *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann.

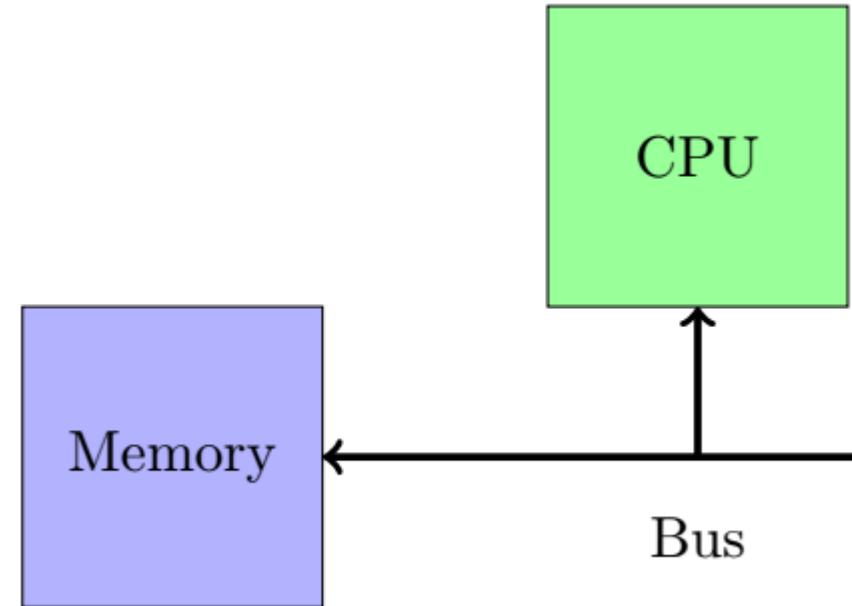
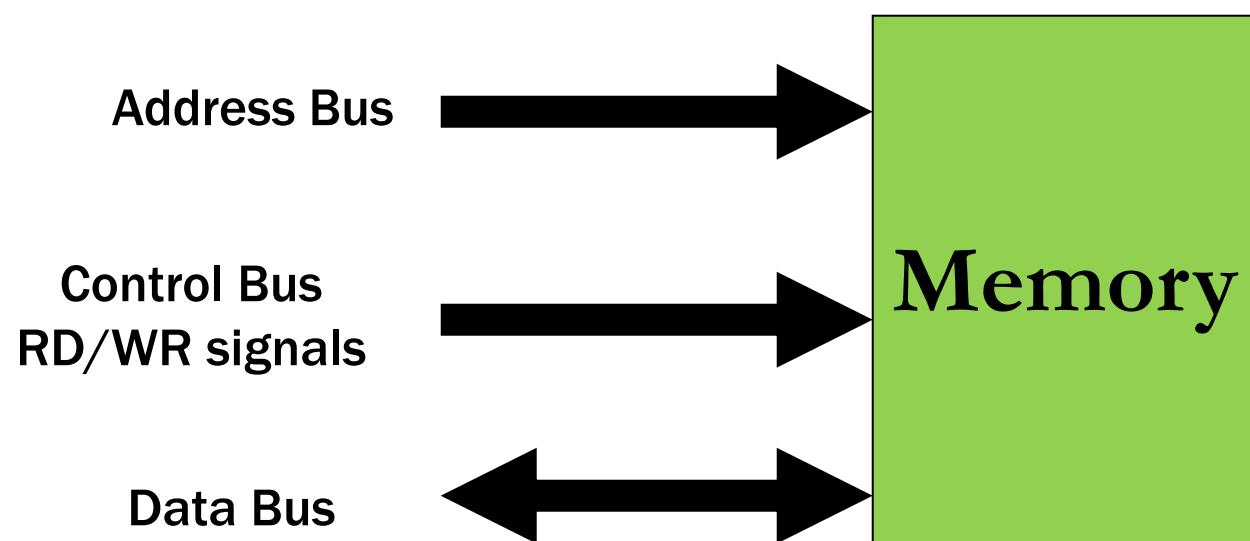
THANK YOU

CS2053 Computer Architecture

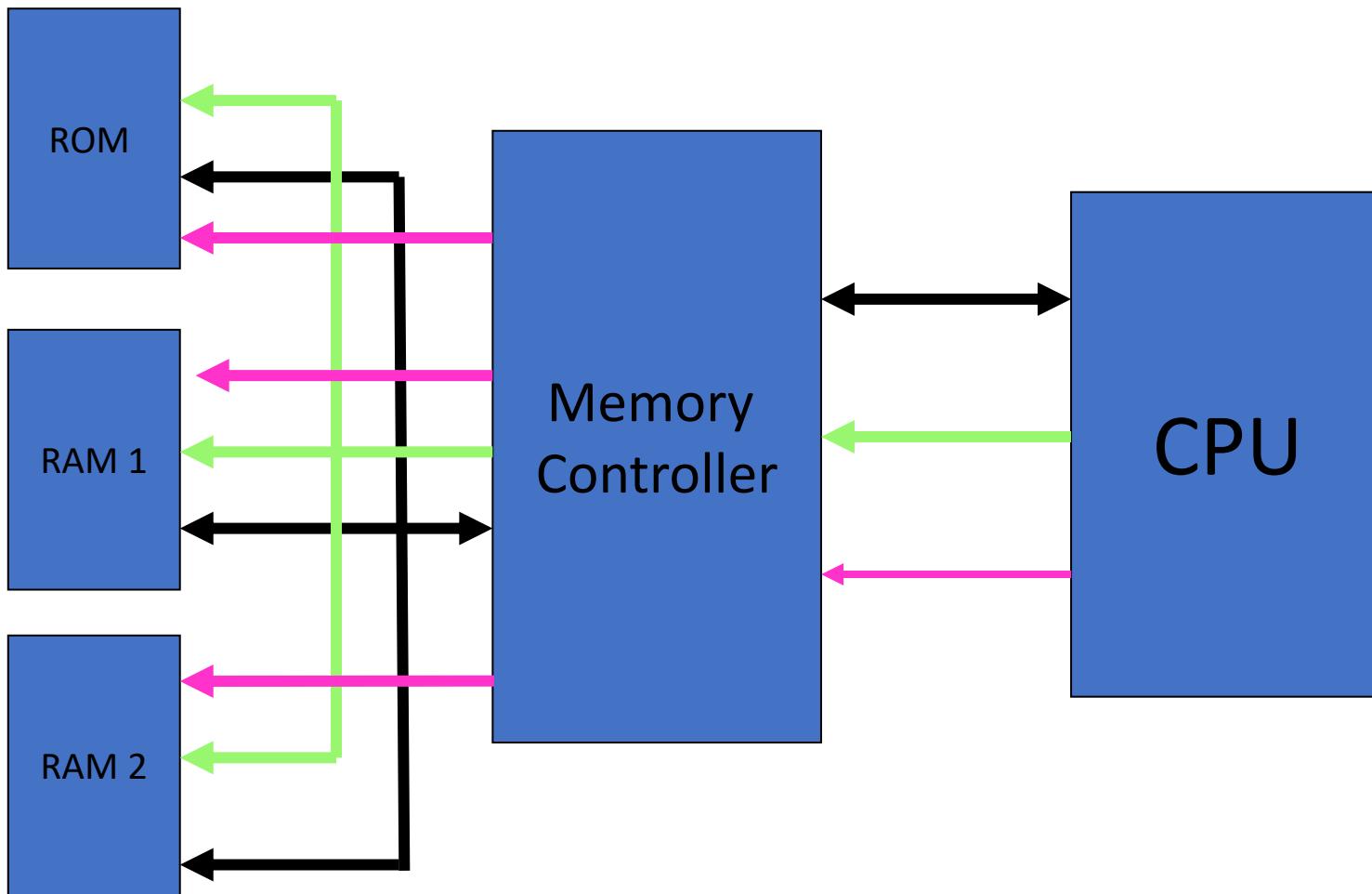
Memory Hierarchy II

Dr. Sulochana Sooriyaarachchi

Accessing Memory



Connecting Memory & CPU



Example Addressing Modes

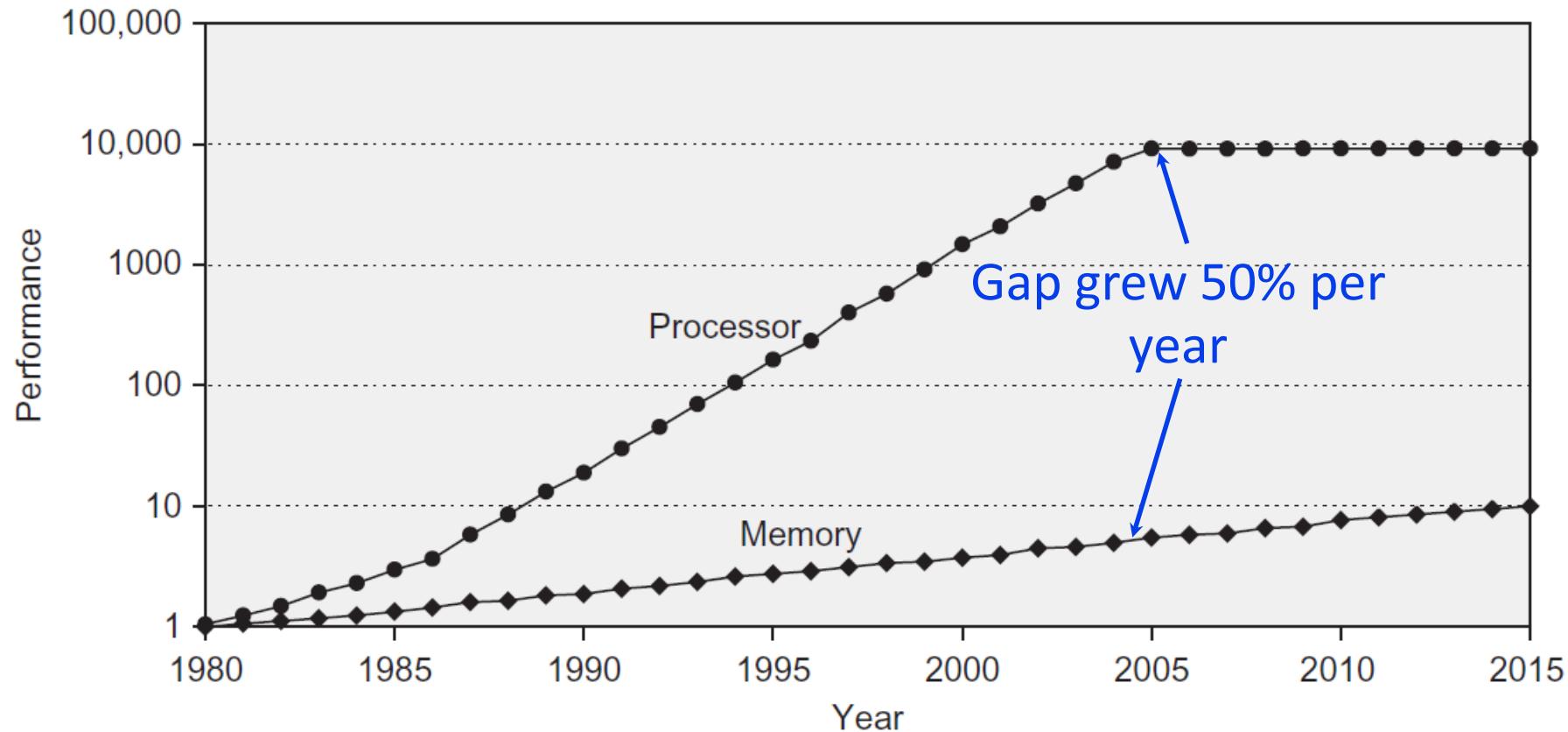
Addressing mode	Example instruction	Meaning	When used
Register	Add R4, R3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Regs}[R3]$	When a value is in a register
Immediate	Add R4, 3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + 3$	For constants
Displacement	Add R4, 100(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[100 + \text{Regs}[R1]]$	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	Add R4, (R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[\text{Regs}[R1]]$	Accessing using a pointer or a computed address
Indexed	Add R3, (R1+R2)	$\text{Regs}[R3] \leftarrow \text{Regs}[R3] + \text{Mem}[\text{Regs}[R1] + \text{Regs}[R2]]$	Sometimes useful in array addressing: R1 = base of array; R2 = index amount
Direct or absolute	Add R1, (1001)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[1001]$	Sometimes useful for accessing static data; address constant may need to be large
Memory indirect	Add R1, @(R3)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Mem}[\text{Regs}[R3]]]$	If R3 is the address of a pointer p , then mode yields $*p$
Autoincrement	Add R1, (R2)+	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$ $\text{Regs}[R2] \leftarrow \text{Regs}[R2] + d$	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d
Autodecrement	Add R1, -(R2)	$\text{Regs}[R2] \leftarrow \text{Regs}[R2] - d$ $\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	Add R1, 100(R2)[R3]	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[100 + \text{Regs}[R2] + \text{Regs}[R3] * d]$	Used to index arrays. May be applied to any indexed addressing mode in some computers

Take home assignment 1

- Study the addressing modes in the processor architecture in your Lab Series.
 - PC-relative: via *auipc*, *jal* and *br** instructions
 - Register-offset: via the *jalr*, *addi* and all memory instructions.
 - Absolute: via the *lui* instruction
- Read relevant sections in: RISCV ISA SpecificationsURL
 - <https://online.uom.lk/mod/url/view.php?id=343622>
 - Volume 1, Unprivileged Specification version 20191213 [\[PDF\]](#)

Submit a short note for each of above instructions with an illustration of addressing involved

Processor Memory Gap



Modern Memory Hierarchy



Source: <http://blog.teachbook.com.au/index.php/2012/02/memory-hierarchy/>

Principle of Locality

- Programs tend to reuse data & instructions that are close to each other or they have used recently
- Temporal locality
 - Recently referenced items are likely to be referenced in the near future
 - A block tends to be accessed again & again
- Spatial locality
 - Items with nearby addresses tend to be referenced close together in time
 - Near by blocks tend to be accessed

Summary

- **Memory hierarchy** - Illusion of a large amount of fast memory
- All data and instructions in memory are **not accessed at once with equal probability**
- **Principle of Locality** – accessing small portion of address space at any instance of time
 - **Temporal locality** – refer the same item again and again
 - **Spatial locality** – refer the items stored close to each other

Memory types

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip-level	Nonvolatile	
Electrically Erasable PROM (EEPROM)		Electrically, byte-level		
Flash memory		Electrically, block-level		

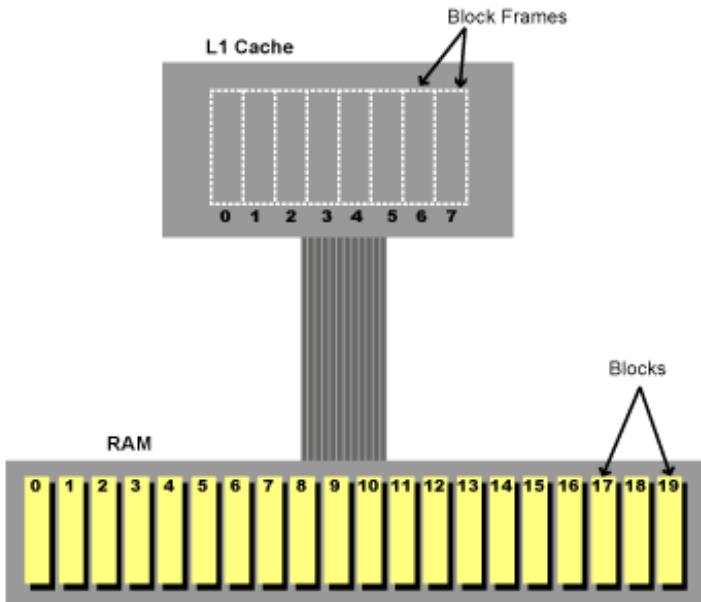
Take home assignment 2

Read Section 5.2 of

Patterson, David. "Computer organization and design RISC-V edition:
the hardware." (2017) and

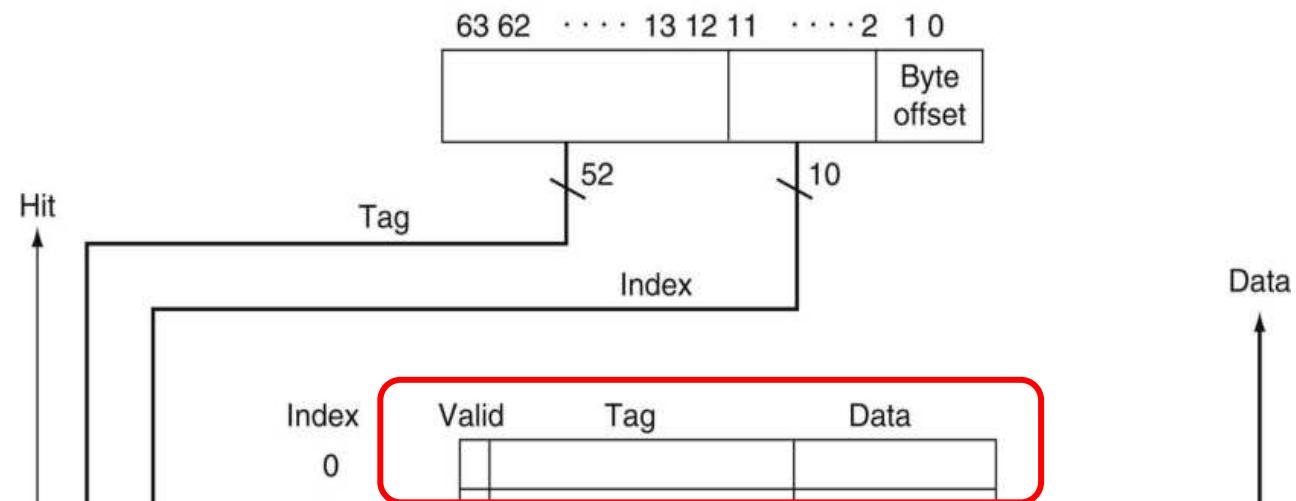
Answer the Quiz during mid semester quiz.

Cache



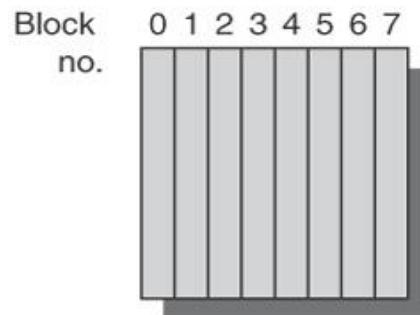
Source:
<http://archive.arsTechnica.com/paedia/c/caching/m-caching-5.html>

- Cache hits/misses
- Average memory access time
- Accessing cache
- Cache organization

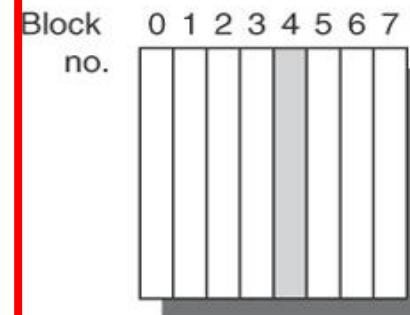


Cache Associativity

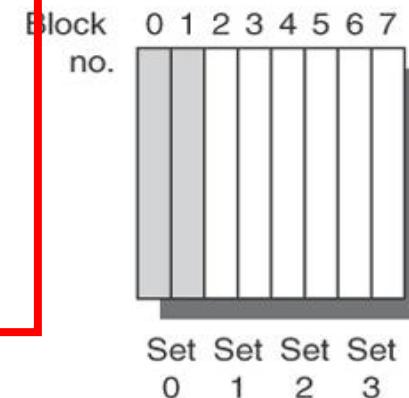
Fully associative:
block 12 can go
anywhere



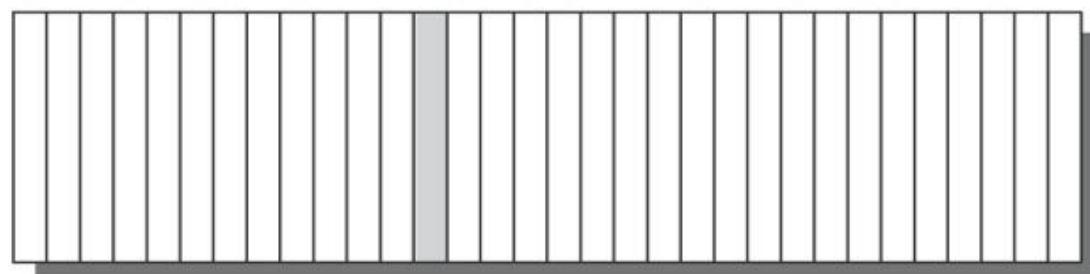
Direct mapped:
block 12 can go
only into block 4
($12 \bmod 8$)



Set associative:
block 12 can go
anywhere in set 0
($12 \bmod 4$)



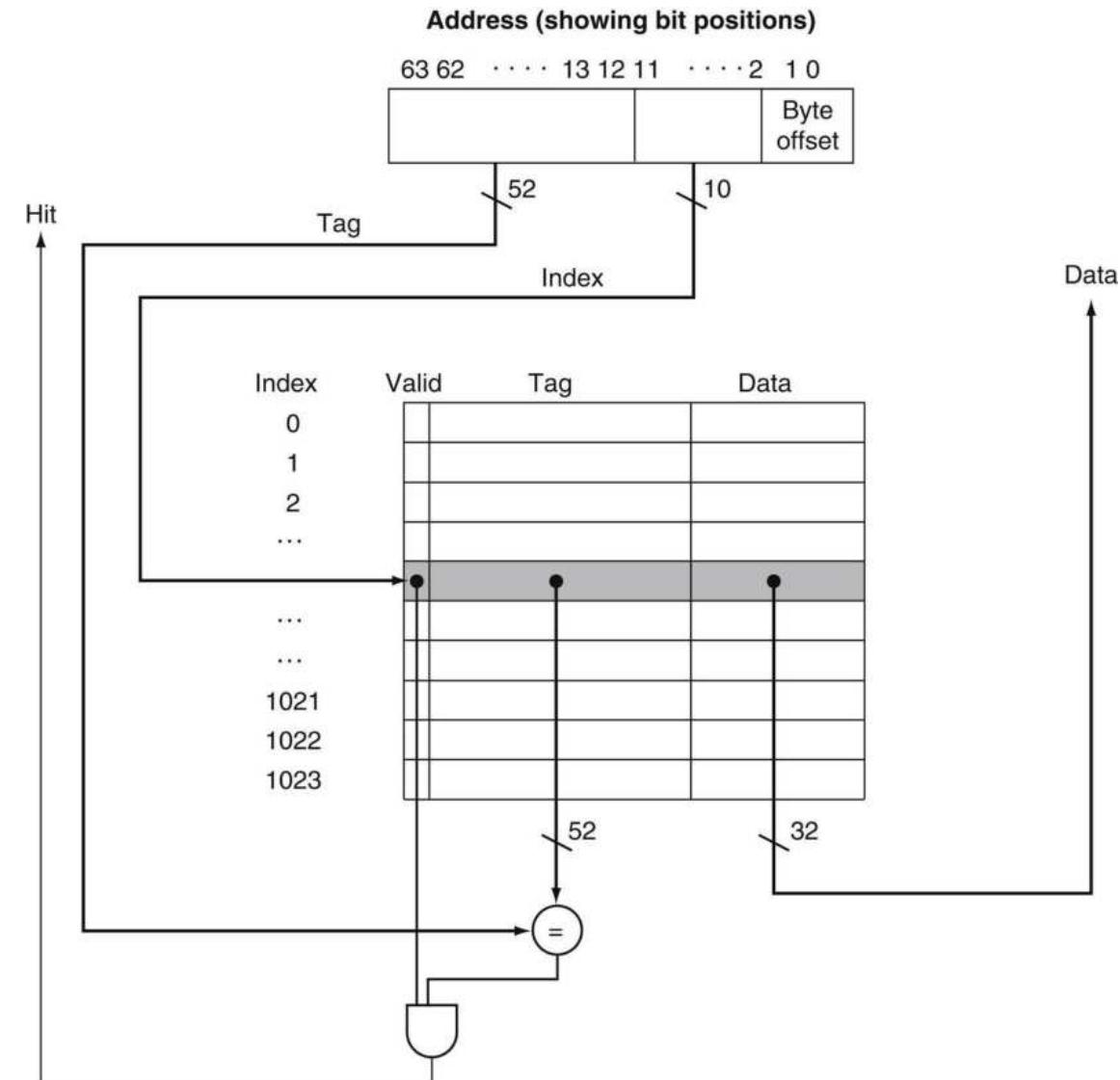
Block no. 0 1 2 3 4 5 6 7 8 9 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3



© 2007 Elsevier, Inc. All rights reserved.

Accessing Cache

- Tag = upper portion of address
- Index = (Memory block address) modulo #Cache blocks
- Valid bit
- Example:
 1. How many bits are required for a direct-mapped cache with 16KiB of data and four-word blocks (addresses are 64bit)



Handling Cache Misses - Read

- Cache miss → processor stall
 - Freeze the register content & wait for memory in **in-order processors** (**opposite: out-of-order processors**)
- Steps:
 1. Processor generates the address to access the memory
 2. Cache miss occurs (=tag not matched)
 3. Instruct memory to do a read operation
 4. Wait for memory action to complete
 5. Write cache entry: data → data portion, upper bits of address → tag, valid bit → 1
 6. Processor resume from where stalled

Handling Cache Misses - Write

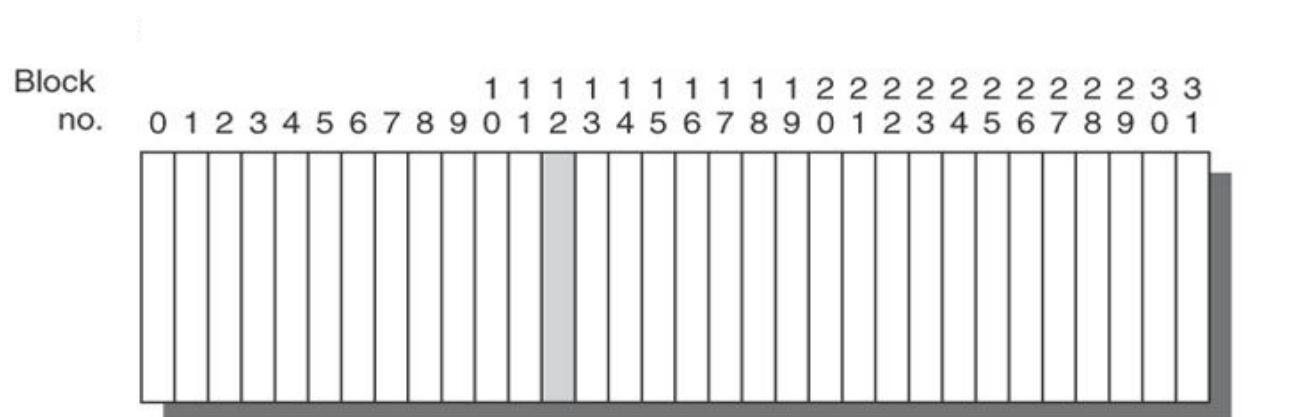
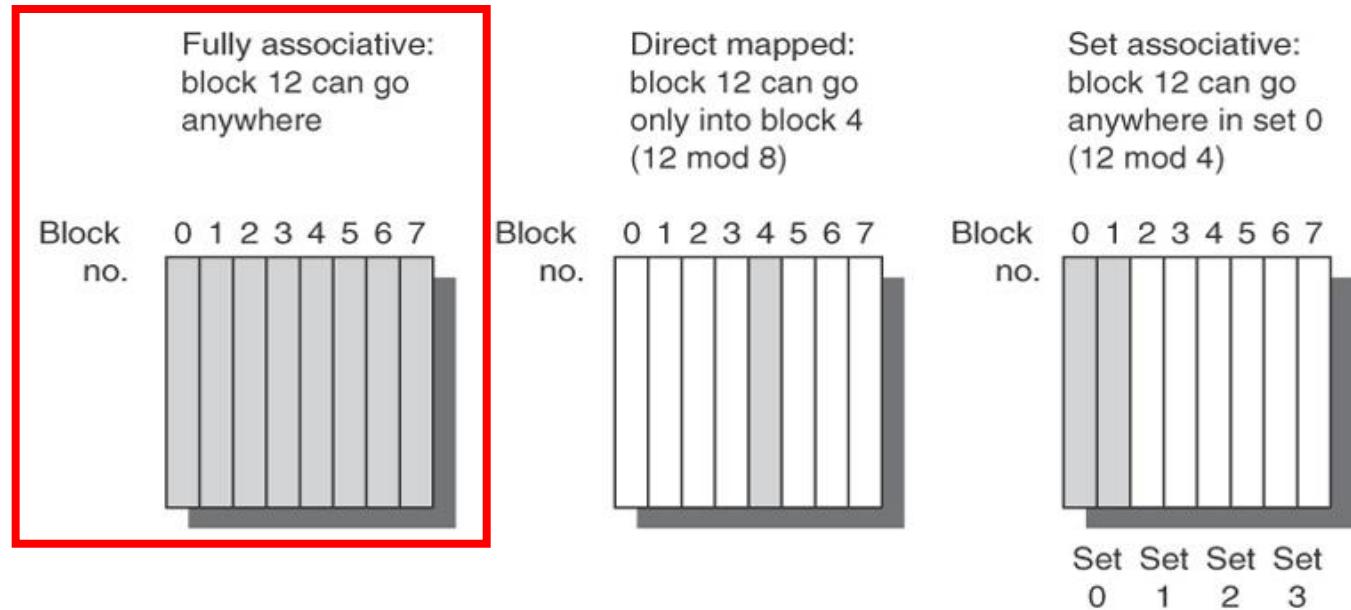
- Processor writes ONLY to the cache (no change in main memory)
→ Cache Inconsistency
- Write-through: write both to Cache and Memory at the same time
→ poor performance
 - Write-buffer – temporarily store data while waiting to write to memory
 - Frees the write-buffer only after memory write successfully completes
 - Otherwise, write-buffer is full → next write-through operation causes processor stall
- Write-back: new value written only to cache, memory is written only when the block is replaced
 - Performs better but complex to implement

Cache Replacement Policies

- When cache is full some of the cached blocks need to be removed before bringing new ones in
 - If cached blocks are dirty (written/updated), then they need to be written to RAM
- Cache replacement policies
 - Random: simple to build in hardware
 - Least Recently Used (LRU)
 - Need to track last access time
 - Least Frequently Used (LFU)
 - Need to track no of accesses
 - First In First Out (FIFO)

Fully Associative Cache

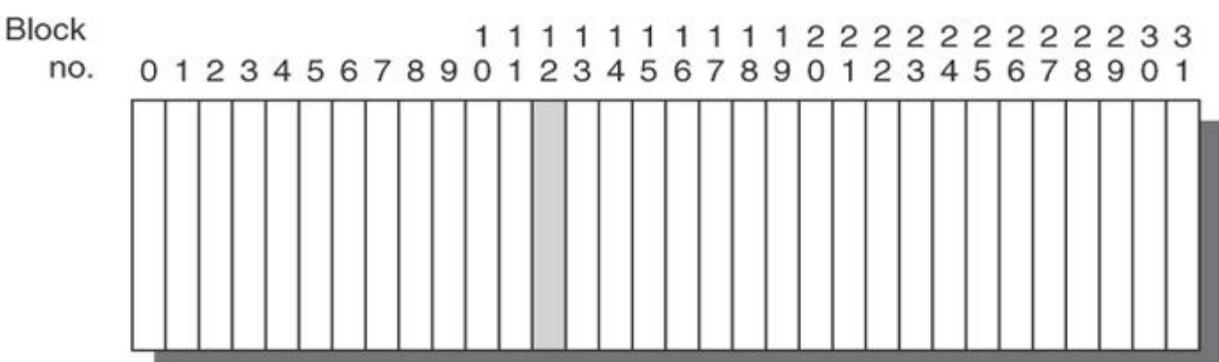
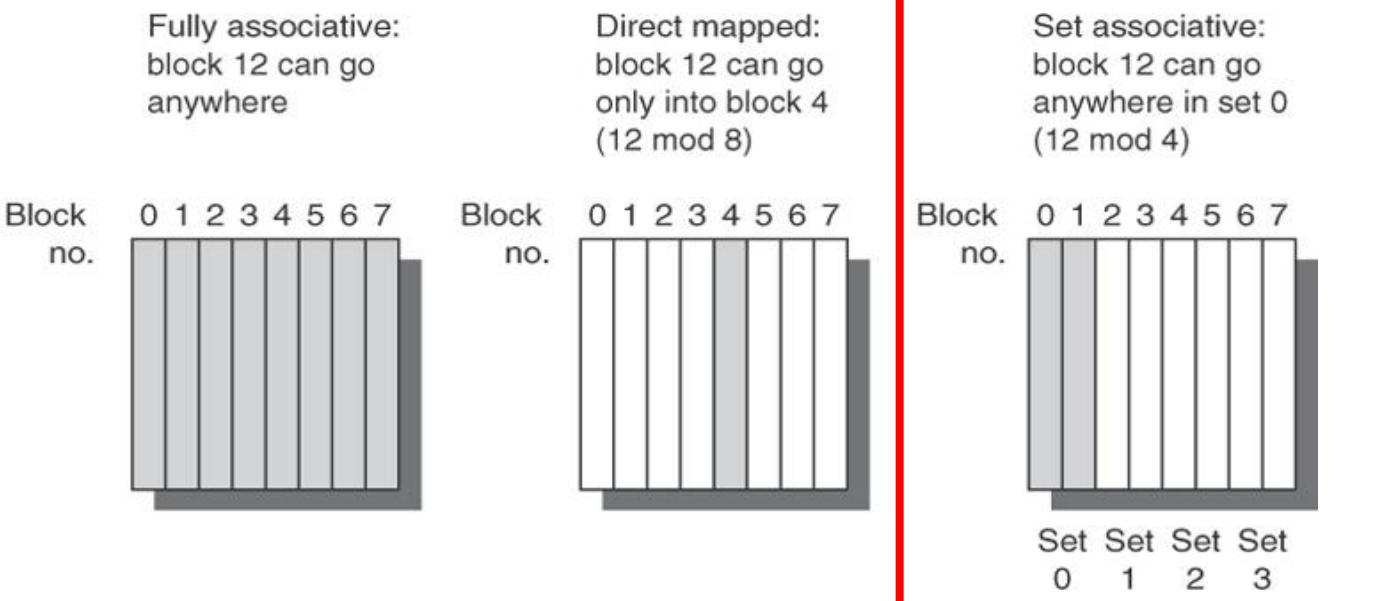
- Parallel tag search using hardware comparators
→ Hardware costs high
- Practical when #blocks in cache is less



© 2007 Elsevier, Inc. All rights reserved.

Set Associative Cache

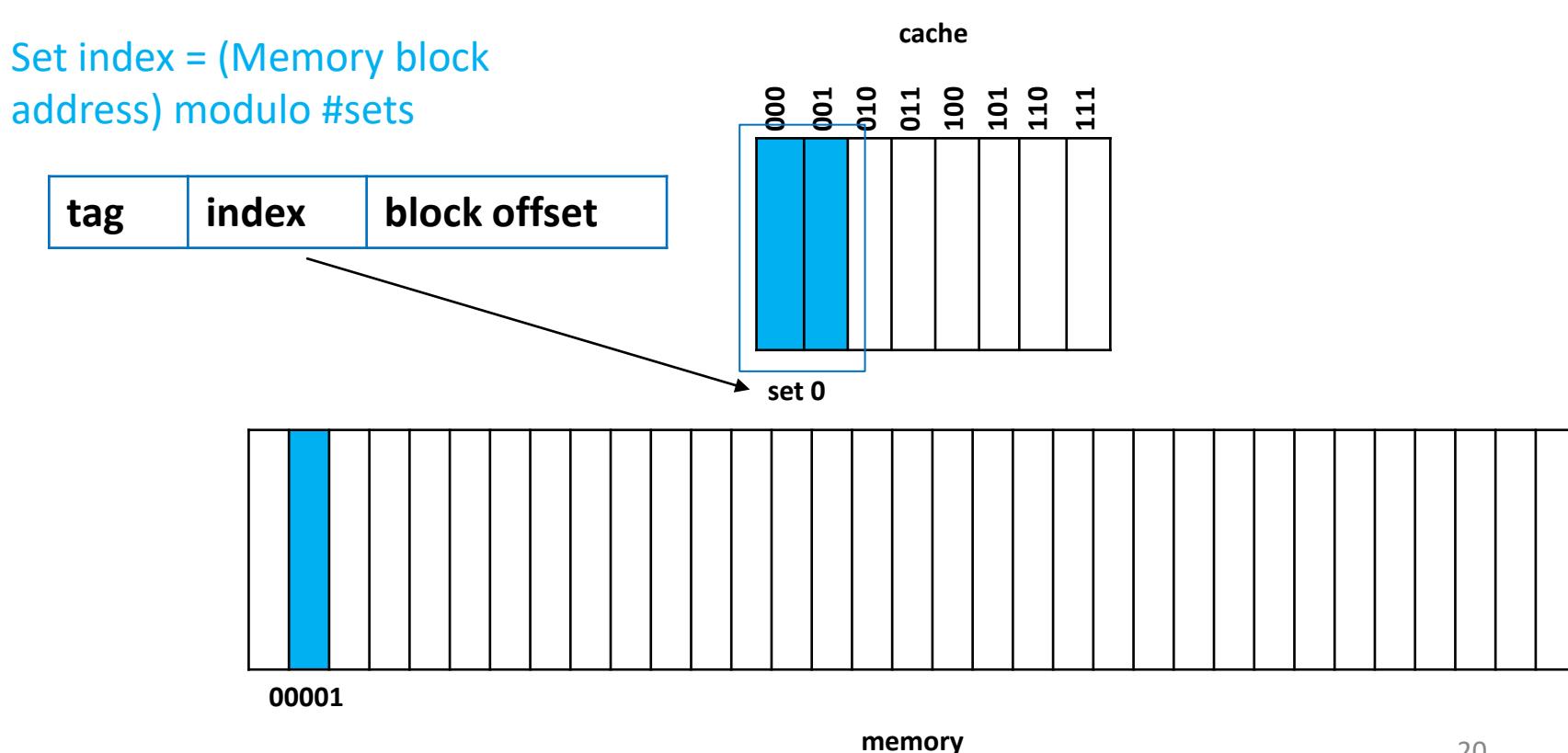
- A block in main memory can go to any block in a set of cache blocks
- Combines **Direct-mapping** (Sets) and **Fully associativity** (Blocks)



© 2007 Elsevier, Inc. All rights reserved.

Set associative

- *n-way set associative*: n blocks for a set
- All tags of all elements in a set must be searched



Cache configuration examples

**One-way set associative
(direct mapped)**

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

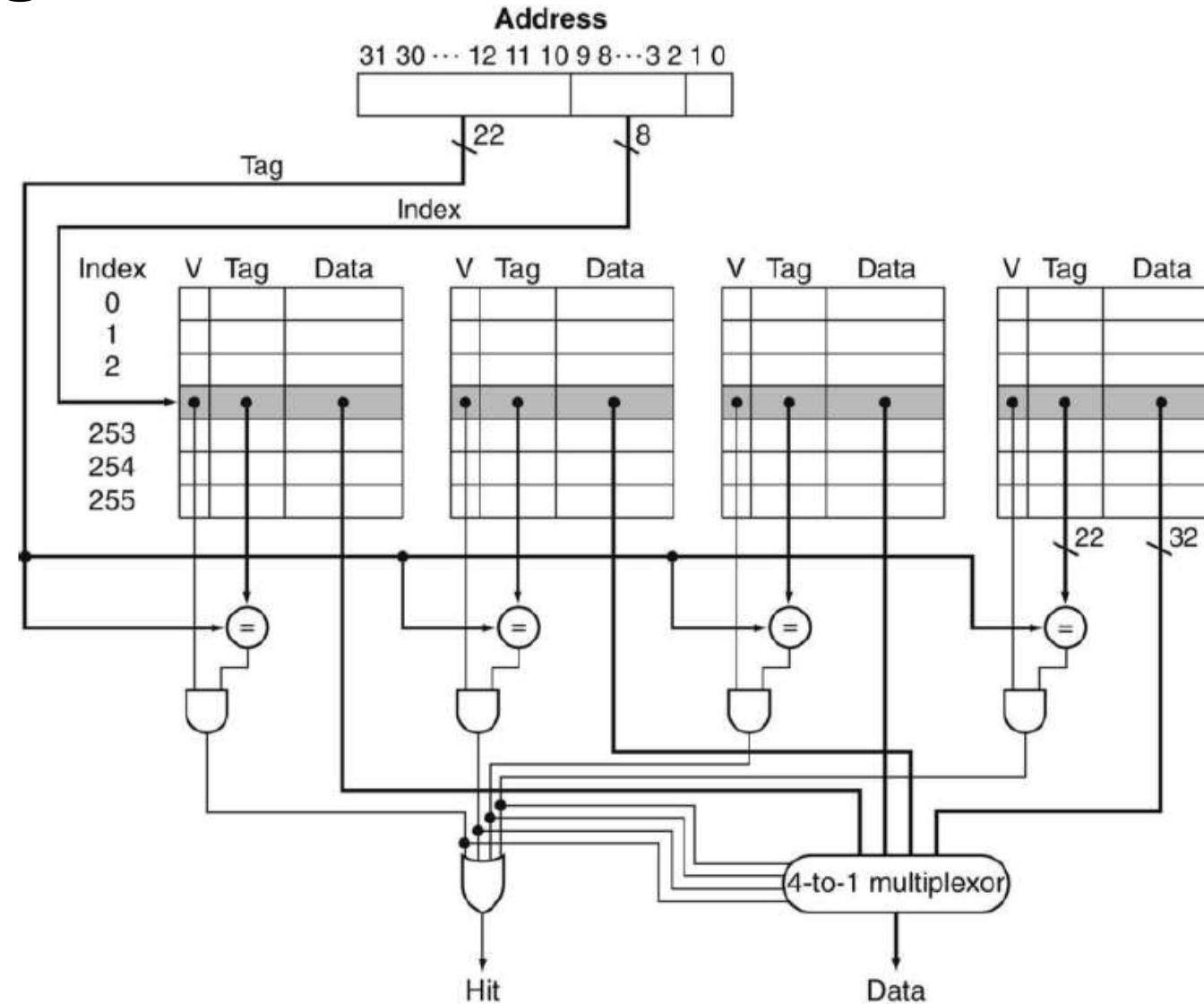
Eight-way set associative (fully associative)

Tag	Data														

Associativity Vs Performance

- Refer the example in pages 785-787 and upload a similar analysis where block address sequence is A,B,C,X,Y,Z (ABCXYZ is the numerical part of your index number) – replace digit 9 with 8
- Take home assignment 3!

Accessing Set-Associative Cache



Size of Tags Vs Set Associativity - Exercise

- Cache size = 4096 Blocks, Block size = 4 words, Address size = 64 bits
- Calculate
 1. Total number of sets
 2. Total number of tag bits

Where the cache is

- (a) Direct mapped
- (b) Two-way set associative
- (c) Fully associative

Least Recently Used (LRU)

- Replaces the block **unused for the longest time**
- Keeps track of usage of blocks
 - Expensive when #blocks increases
- Relies on temporal locality

Increasing Cache Performance

- Large cache capacity
- Multiple-levels of cache
- Prefetching
 - a block of data is brought into the cache before it is actually referenced
- Fully associative cache

Thank you

Memory Architecture

IV



CS2053 Computer Architecture
Computer Science & Engineering
University of Moratuwa

Sulochana Sooriyaarachchi
Chathuranga Hettiarachchi

Outline

- Memory types
- Memory access
- Memory hierarchy
 - Main memory
 - Cache
 - Permanent storage
- Example architecture RV32I
 - Superscalar architecture
 - Pipelining
- Virtual memory

Virtual Memory



Virtual Memory

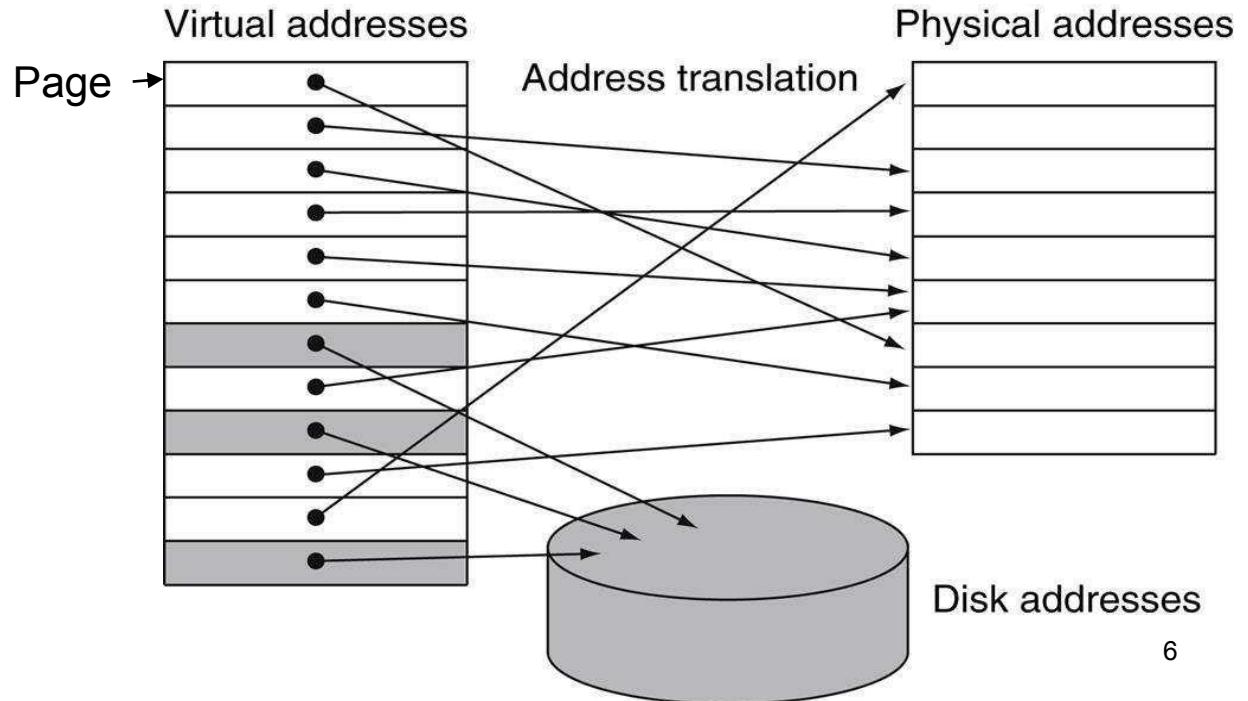
- Main memory act as “cache” for secondary storage
- Motivation
 - To share memory among several programs
 - To relieve the programmer from limited memory constraint
- Protection needed to avoid read/ write to memory portions of other programs/ virtual machines
 - Programs have own **address space**

Program's address space to physical address mapping

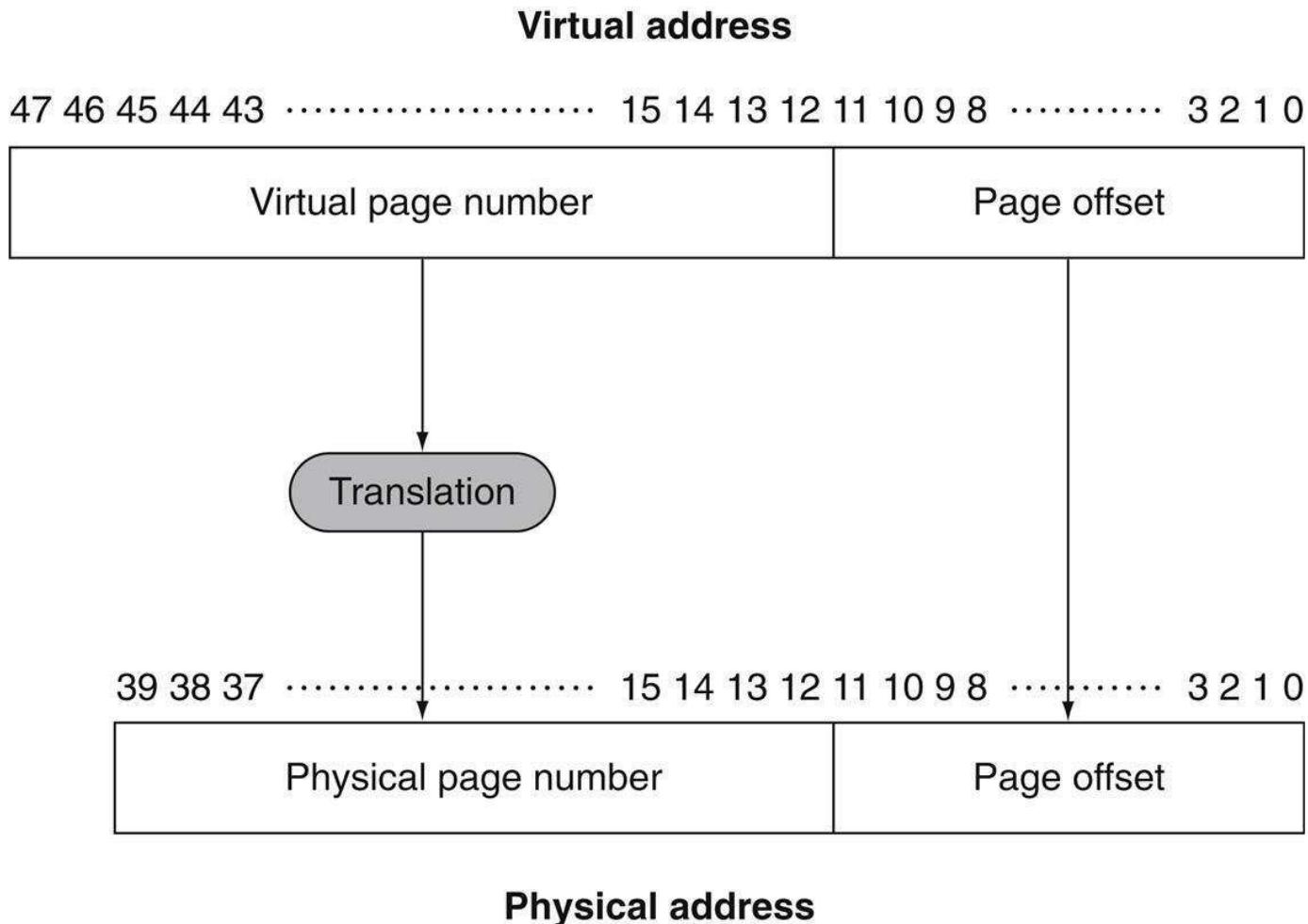
- **Physical address**: an address is main memory
- **Protection**: a set of *mechanisms* to ensure that
 - multiple processes *sharing*
 - the processor, memory, or I/O devices
 - **cannot interfere** intentionally/ unintentionally with one another
 - by reading/ writing to each other's data
 - Protection also isolates OS from user processes
- Operation of cache and virtual memory have similarities, but terminology is different

Virtual memory terminology

- Main memory = physical memory
- Secondary storage: e.g. magnetic disks
- Virtual memory blocks = *pages*
- A page not in main memory = *page fault*

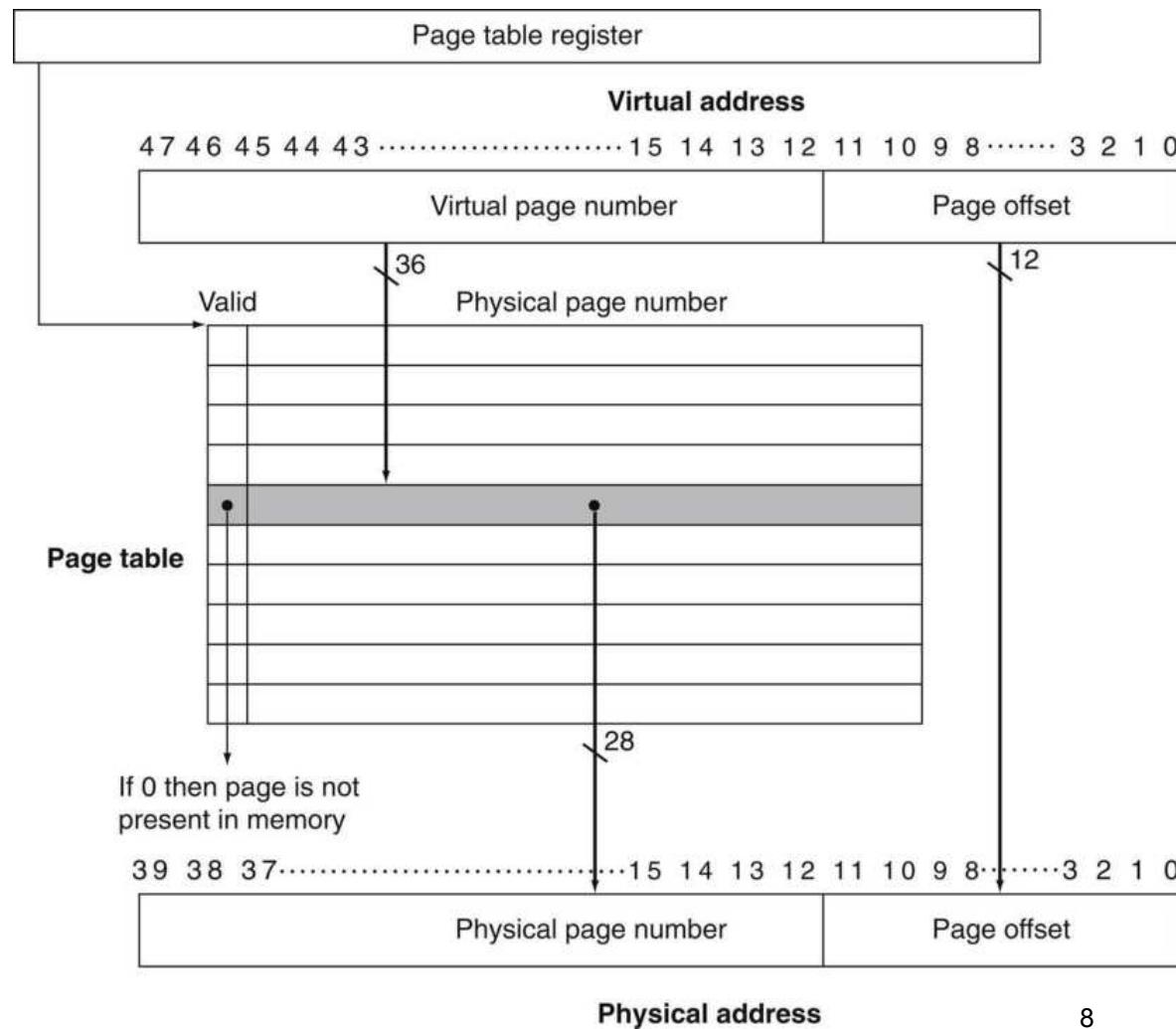


Address translation



Virtual memory configuration

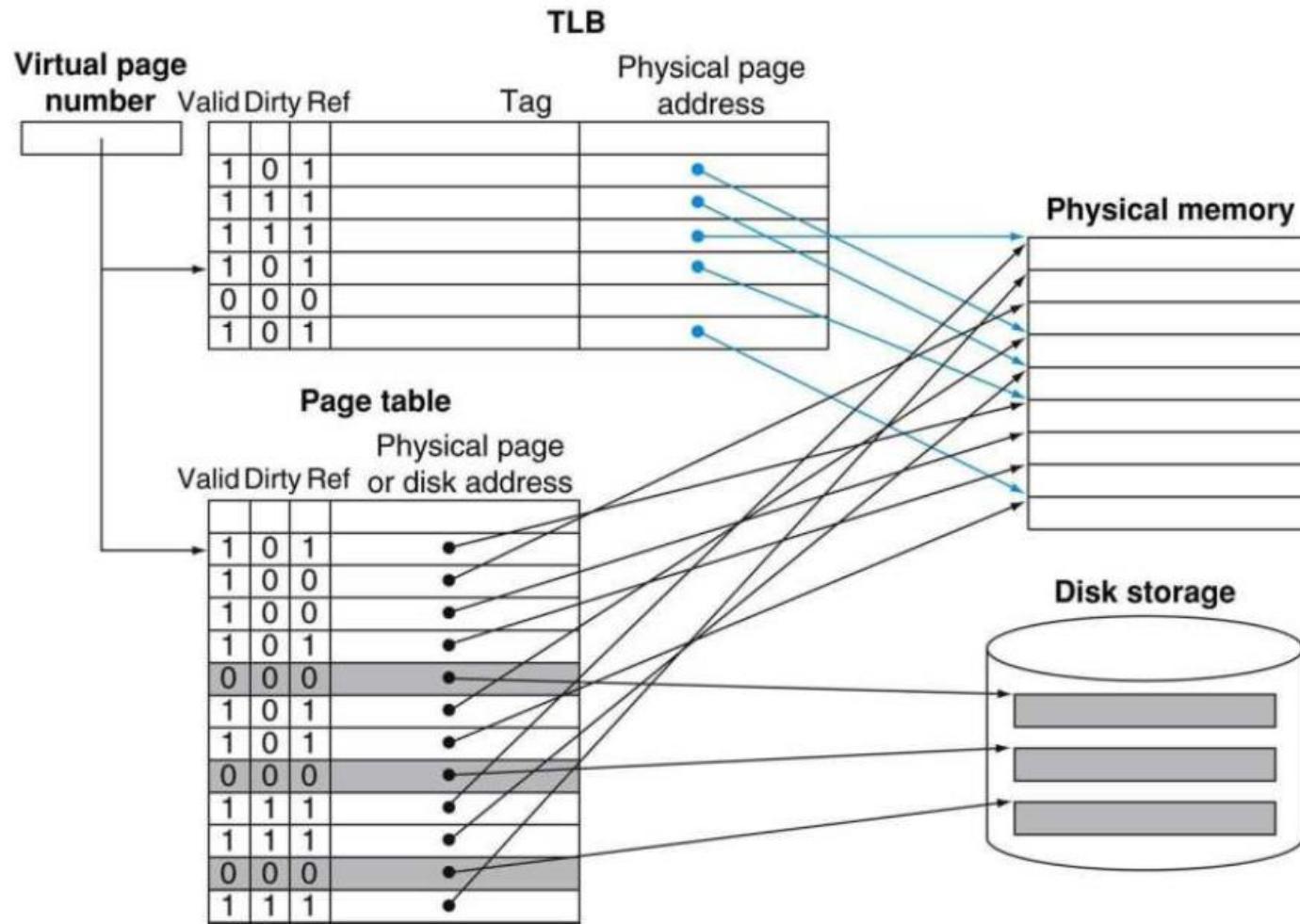
- Locate pages using a table that indexes the main memory = **page table**
- Page table resides in main memory
- Valid bit = 0 → page fault
 - OS should manage
 - Swap space



Page Table

- Maps virtual address to physical address
- Stored in main memory
- Virtual page number to index page table entry
 - Assume Byte-addressing
 - 4KiB pages → 2^{12} bytes in a page
 - 48-bit Virtual address → 12 bit page offset
 - 36 bits for indexing page table entries
 - 2^{36} page table entries ~ 64 billion entries!
- Techniques to reduce page table size
 - E.g. page the page table, multi-level table

Translation Look-aside Buffer



TLB

- A special cache to keep address translations
- Accesses within a page may have
 - Temporal locality
 - Spatial locality
- Reduce memory accesses needed to lookup the page table
- TLB includes additional status bits
 - dirty bit, reference bit

References

- Waterman, A., Lee, Y., Patterson, D. A., & Asanovi, K. (2014). *The risc-v instruction set manual. volume 1: User-level isa, version 2.0.* California Univ Berkeley Dept of Electrical Engineering and Computer Sciences.
- Harris, S. L., Chaver, D., Piñuel, L., Gomez-Perez, J. I., Liaqat, M. H., Kakakhel, Z. L., ... & Owen, R. (2021, August). RVfpga: Using a RISC-V Core Targeted to an FPGA in Computer Architecture Education. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)* (pp. 145-150). IEEE.
- Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.
- Patterson, D. A., & Hennessy, J. L. (2016). *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann.

THANK YOU

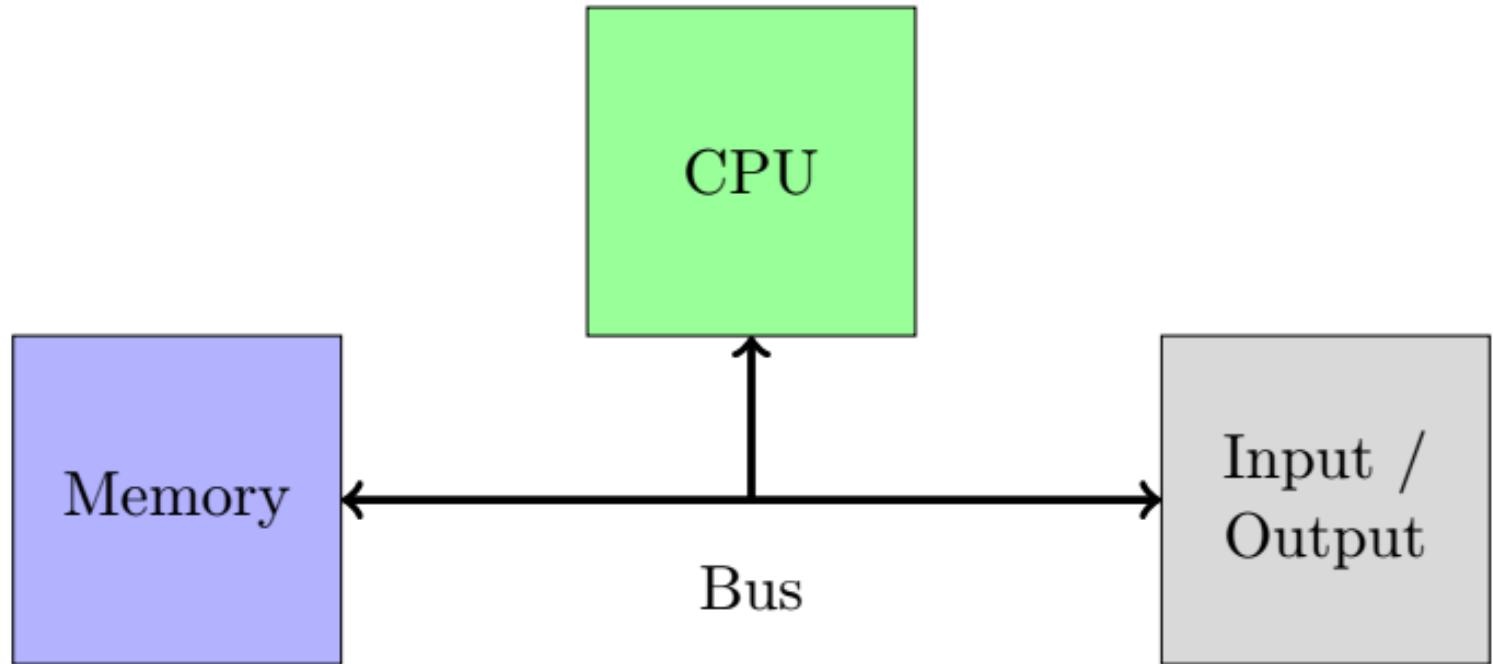
CPU Performance Enhancements



CS2052 Computer Architecture
Computer Science & Engineering
University of Moratuwa

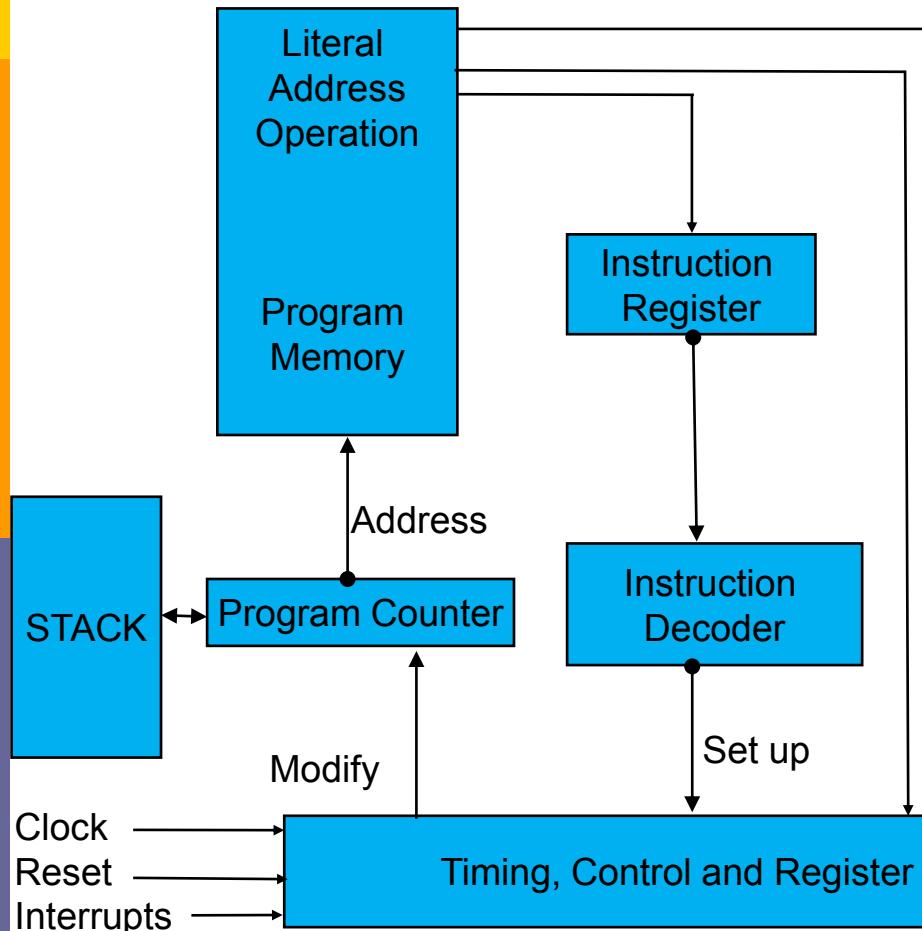
Dilum Bandara
Dilum.Bandara@uom.lk

Introduction

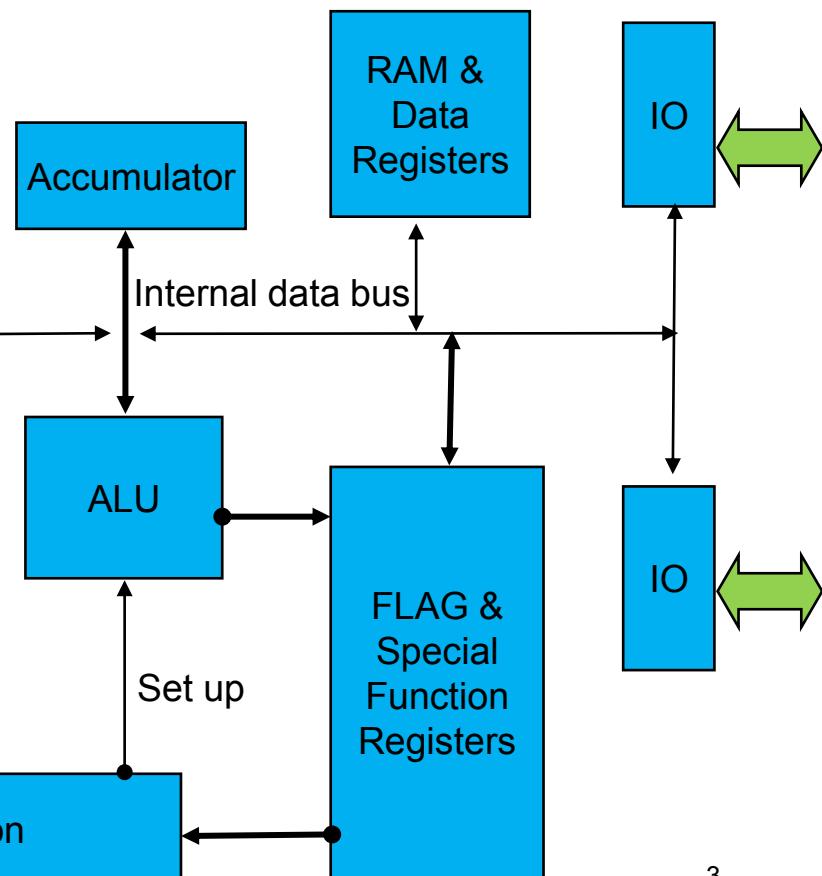


Introduction (Cont.)

Program Execution Section



Register Processing Section

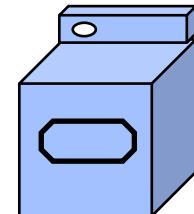


Introduction (cont.)

- ❑ Pipelining
- ❑ Parallelism
- ❑ Advanced processor architectures

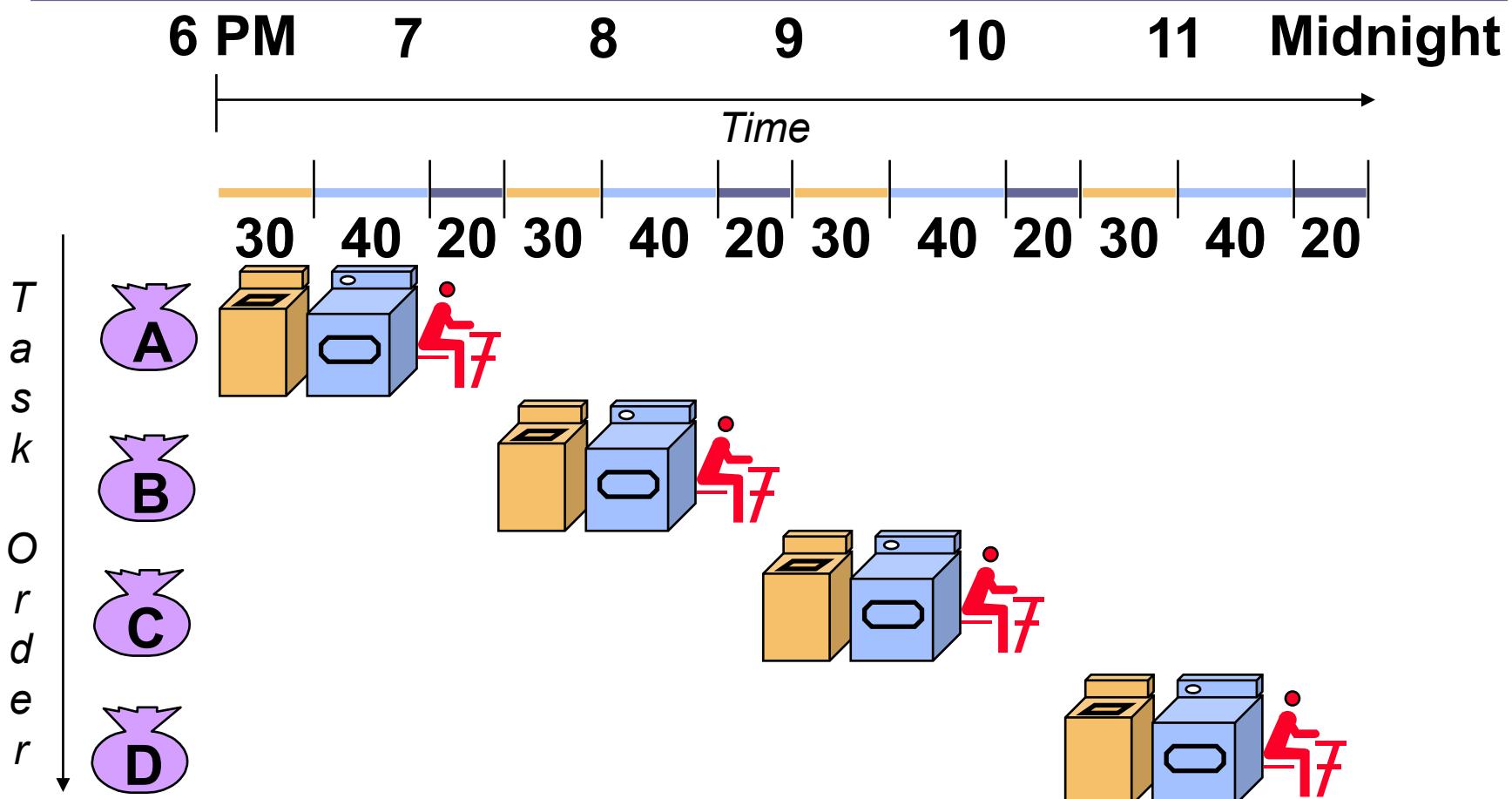
Pipelining – It's Natural!

- Laundry example
- Amal, Bimal, Chamal, & Dinal each have one load of clothes to wash, dry, & fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- Folder takes 20 minutes



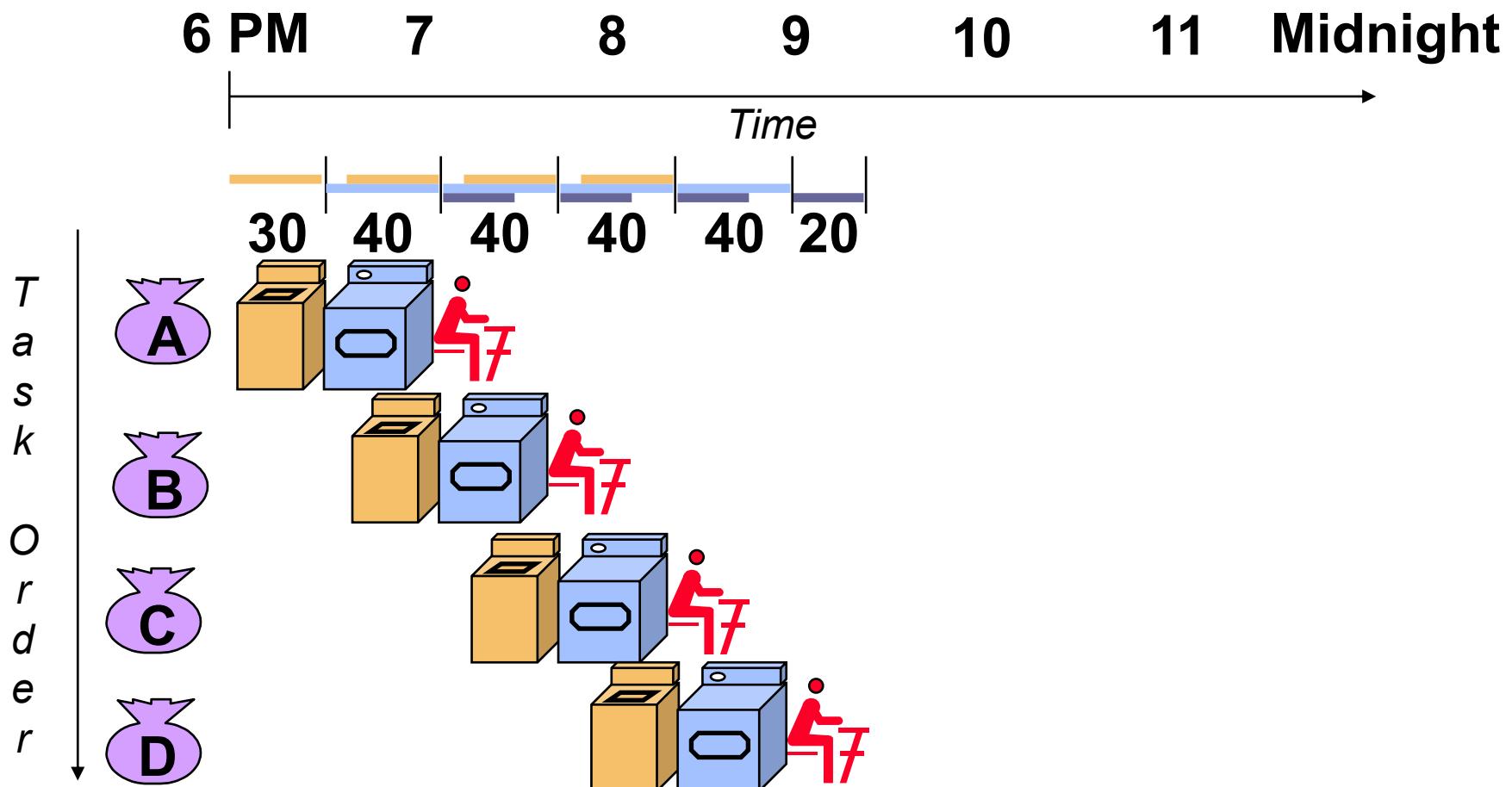
<https://www.youtube.com/watch?v=ztw0QG0E23o>

Sequential Laundry



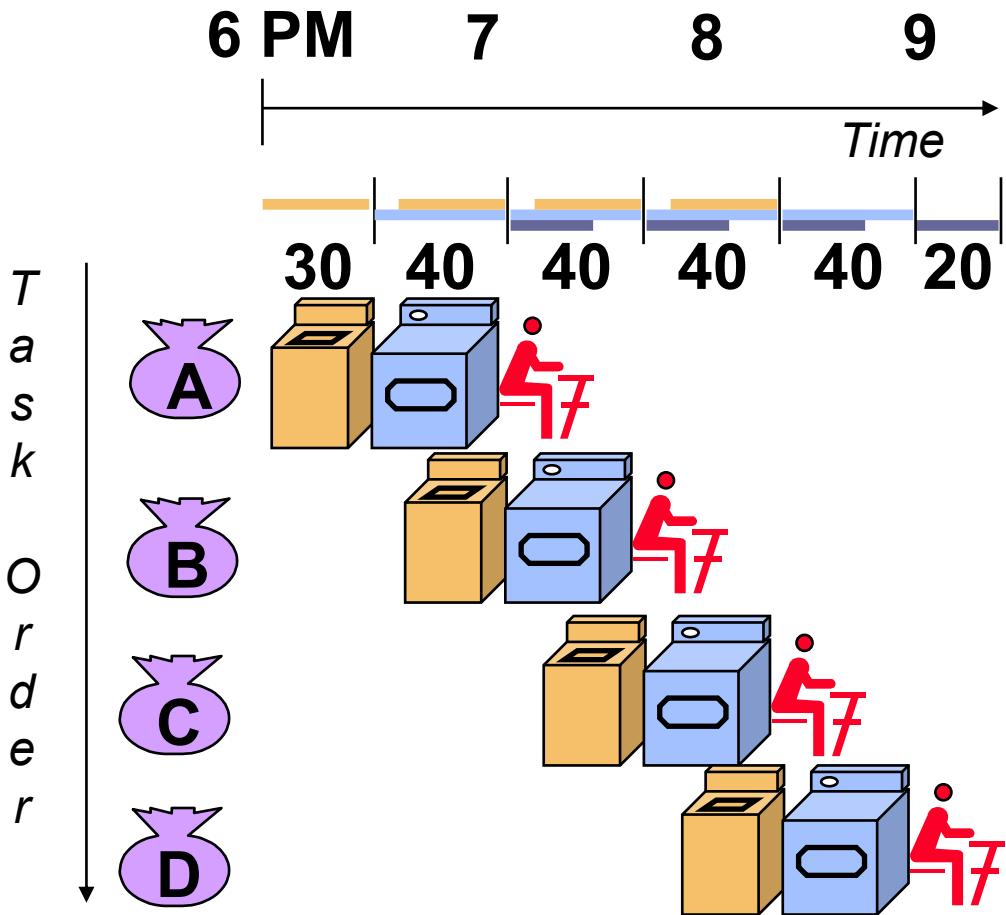
- ❑ Sequential laundry takes 6 hours for 4 loads
- ❑ If they learned pipelining, how long would laundry take?

Pipelined Laundry – Start Work ASAP



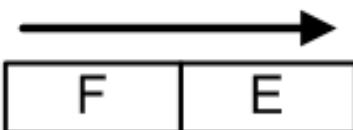
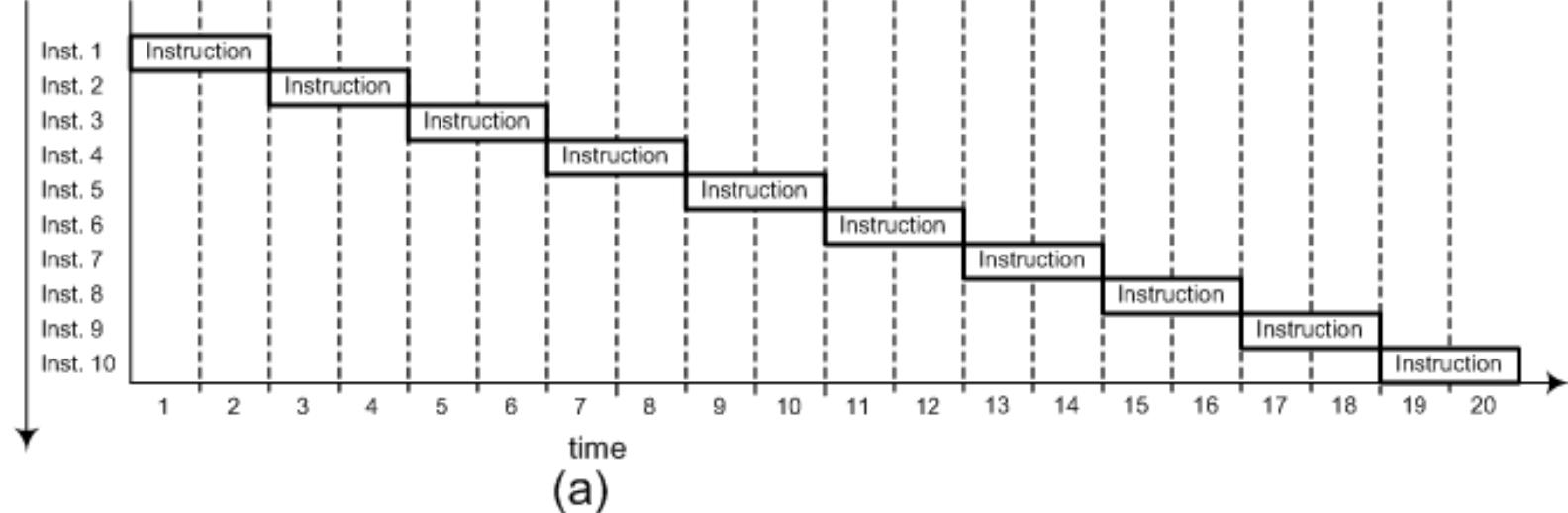
- ❑ Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



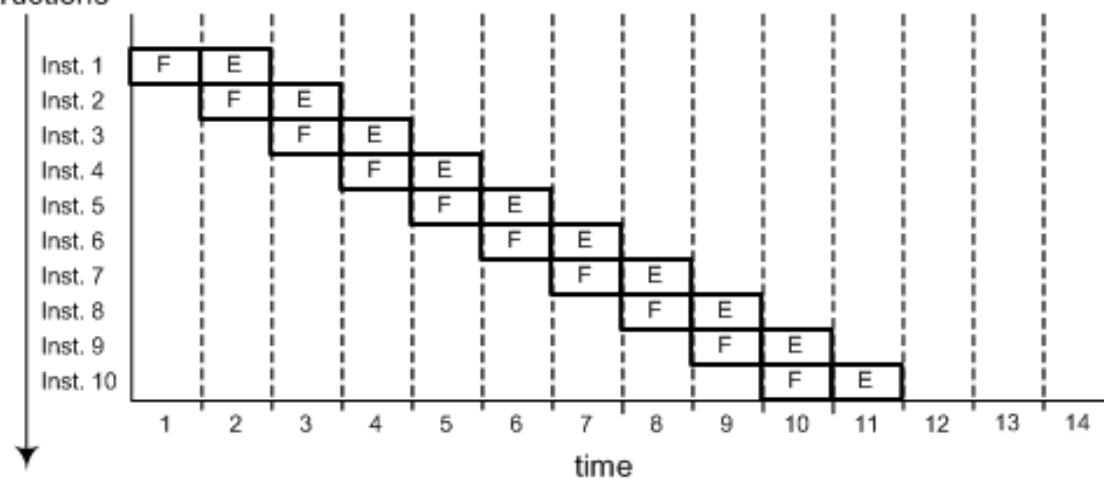
- ❑ Pipelining doesn't reduce **latency** of a single task
- ❑ Improve **throughput** of entire workload
- ❑ Pipeline rate limited by **slowest** pipeline stage
- ❑ Multiple tasks operating simultaneously
- ❑ Potential speedup = No pipe stages
- ❑ Unbalanced lengths of pipe stages reduces speedup
- ❑ Time to **fill** pipeline & time to drain/flush it reduces speedup

Program Instructions



Program Instructions

Instruction Fetch Instruction Execute



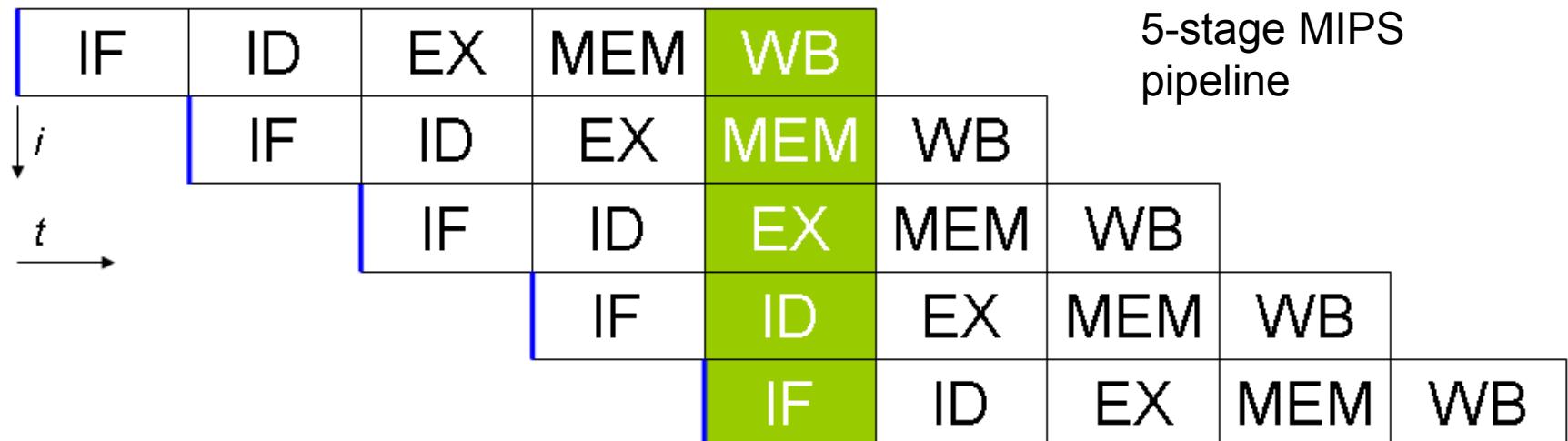
Instruction Level Parallelism (ILP)

Source:

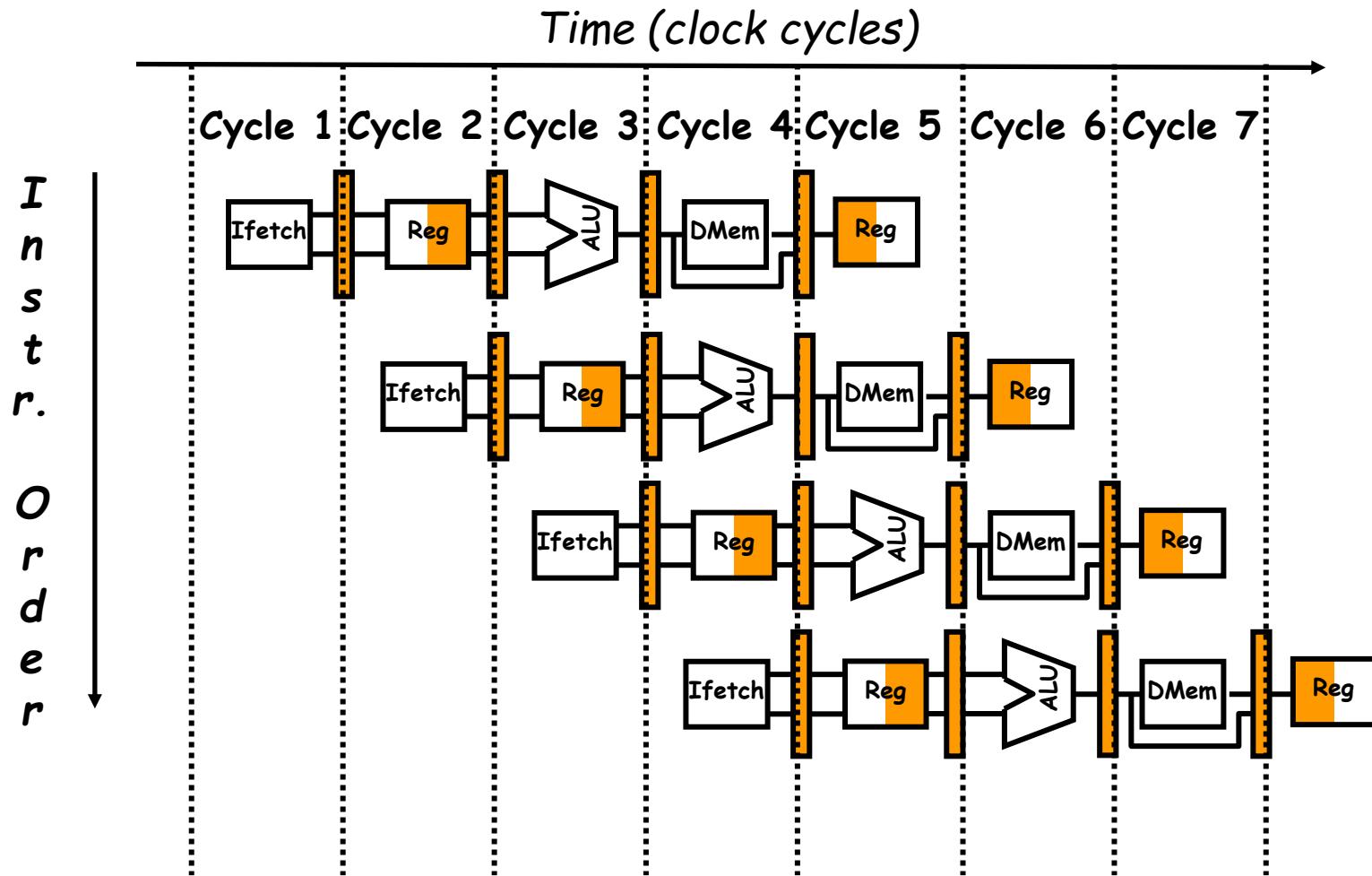
<http://mail.humber.ca/~paul.michaude/Pipeline.htm>

CPU Pipelines

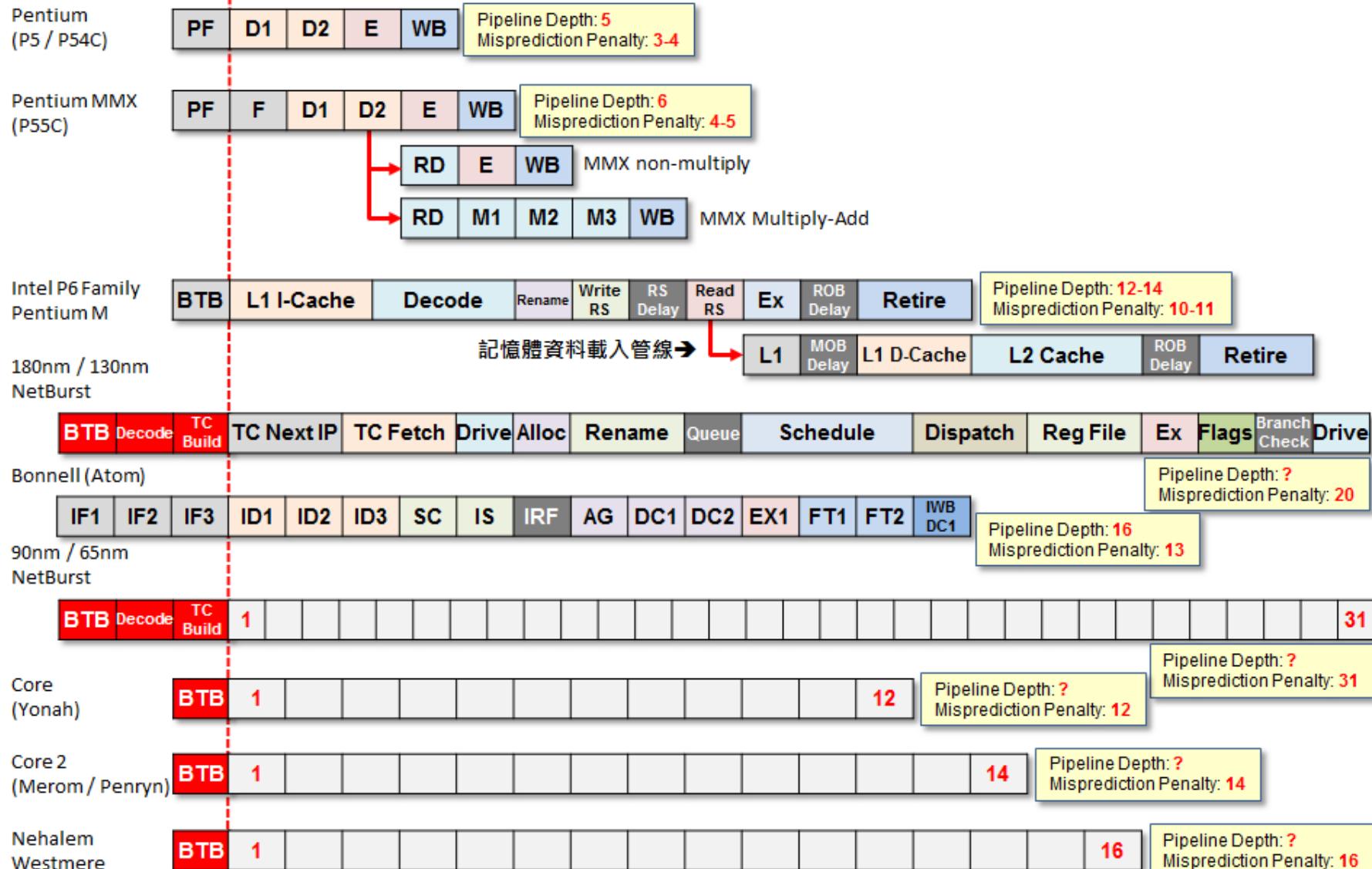
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
Instruction 1	Fetch	Decode	Execute		
Instruction 2		Fetch	Decode	Execute	
Instruction 3			Fetch	Decode	Execute



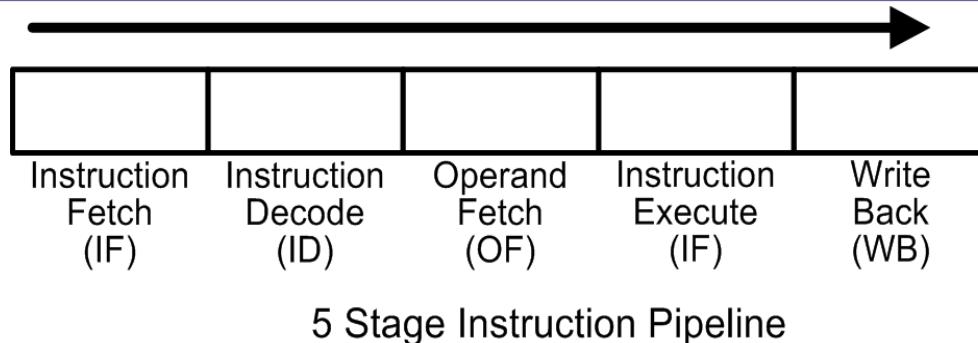
Pipelined Instruction Execution



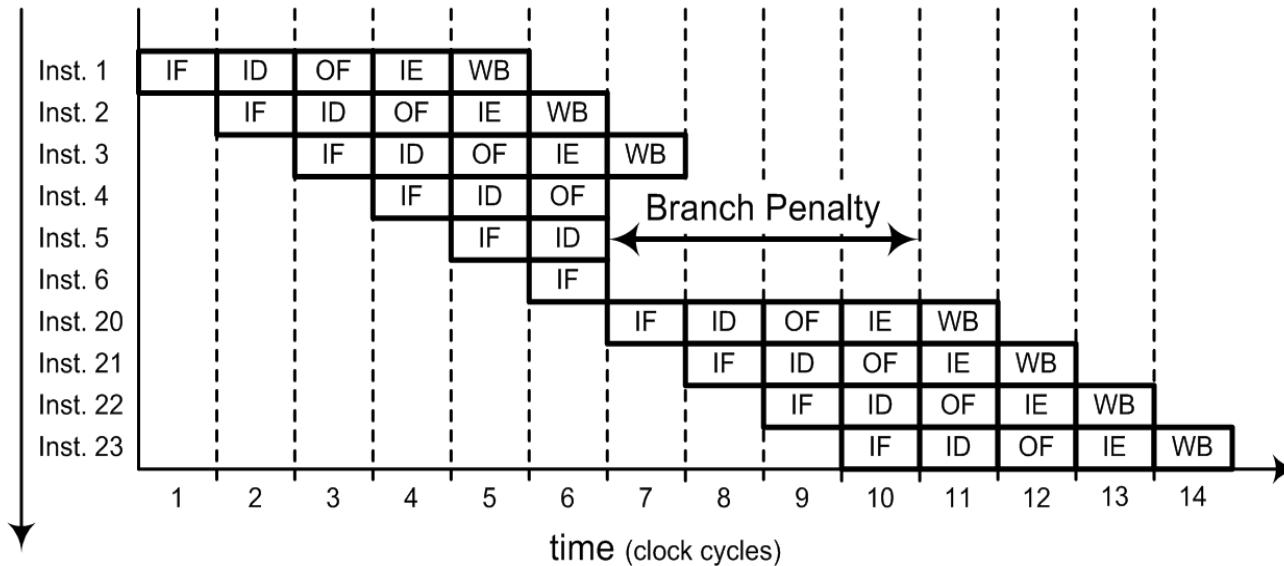
Intel x86 Pipeline History



Pipeline With a Branch Penalty Due to a Taken Branch



Program Instructions



PIC instructions
have a branch
penalty of 1 CC

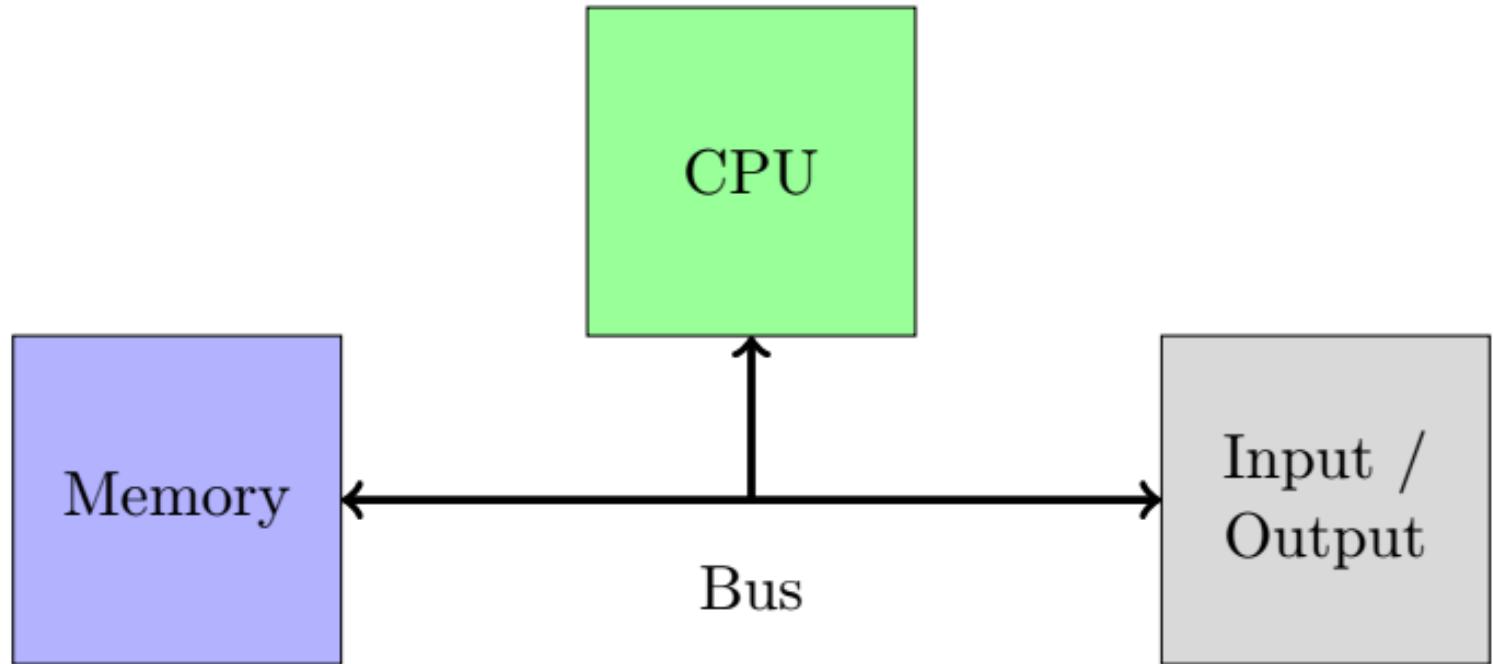
Pipelining



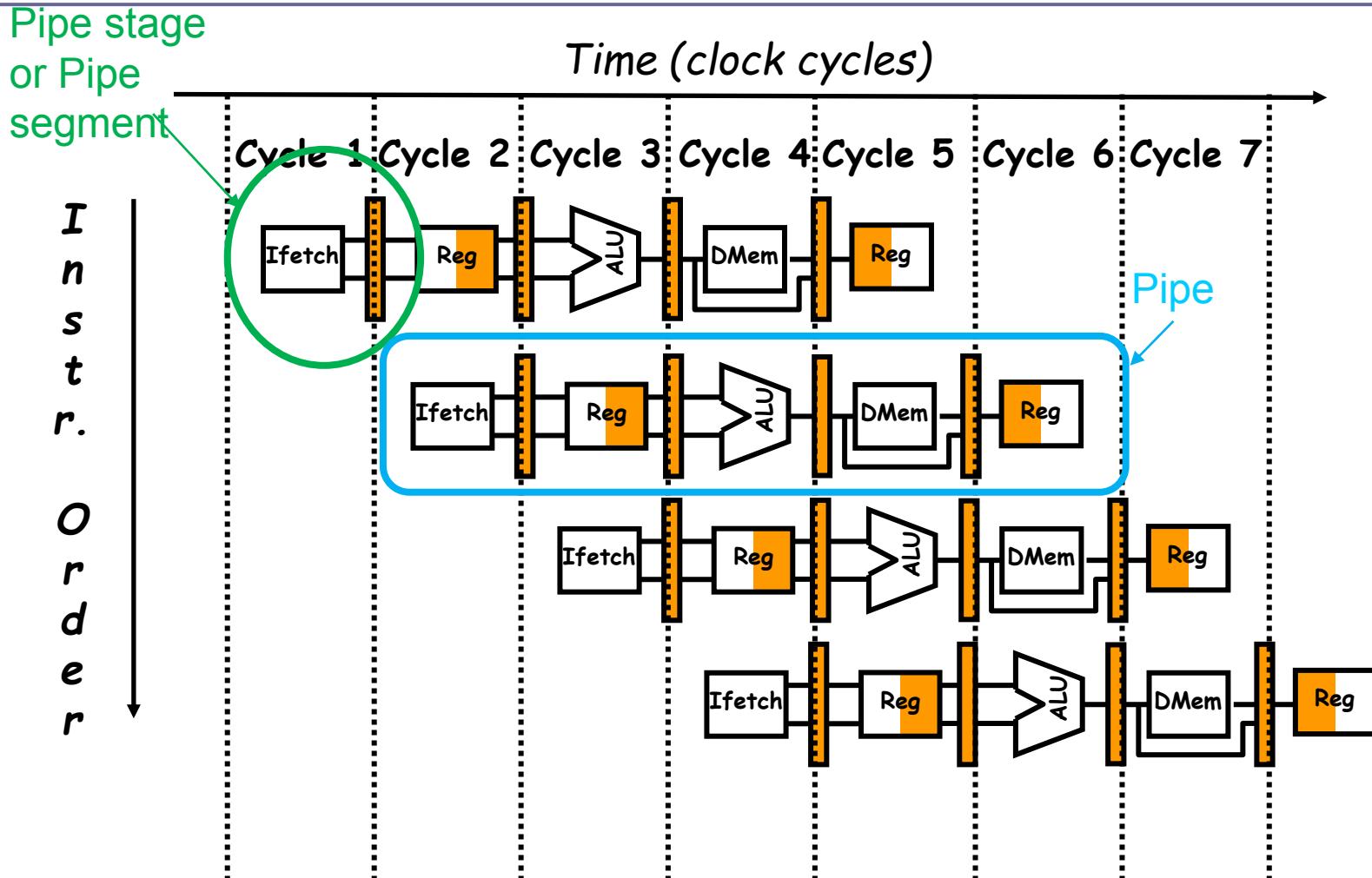
CS2052 Computer Architecture
Computer Science & Engineering
University of Moratuwa

Sulochana Sooriyaarachchi
sulochanas@cse.mrt.ac.lk

Introduction



Pipelining concepts and terms



Terminology

- Throughput = number of instructions coming out of the pipe per unit time
- Processor cycle = time for moving an instruction one step down the pipeline
 - Depends on slowest stage
- Balanced pipeline = all the pipeline stages have the same duration
 - Time per instruction is given by;
$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

Terminology cntd.

- Speedup
 - Time for an instruction in non pipelined method / time for an instruction in pipelined method
 - For a balanced pipeline: speedup = #number of stages

Pipelining concept

- Overlap instruction processing
- Instruction level parallelism (ILP)
- Datapath implications
- Hazards
- Performance

Pipelining with RISC architecture

□ RISC characteristics

- All operations on data apply to entire data register
- Only operations on memory are *store* and *load*
 - memory  register
 - Can transfer data < full register
- All instructions are typically one size and register specifiers are always in the same place

□ Above leads to simplified pipeline implementation

□ Example: 5-stage pipeline

- IF → ID → EX → MEM → WB

Example Pipeline Implementation

□ Stages

- Instruction Fetch (IF)
 - Refer PC in memory, fetch the instruction from memory, update the PC to next instruction address
- Instruction Decode/ Register Fetch (ID)
 - Fixed field decoding
 - Parallel decoding of opcode and registers
- Execution/ Effective Address (EX)
 - ALU operates on operands (mem ref, reg-reg, ref-imm, cond branch)
- Memory Access (MEM) – use the effective addr
- Write Back (WB) – write results to register file

Pipeline issues

□ What can go wrong?

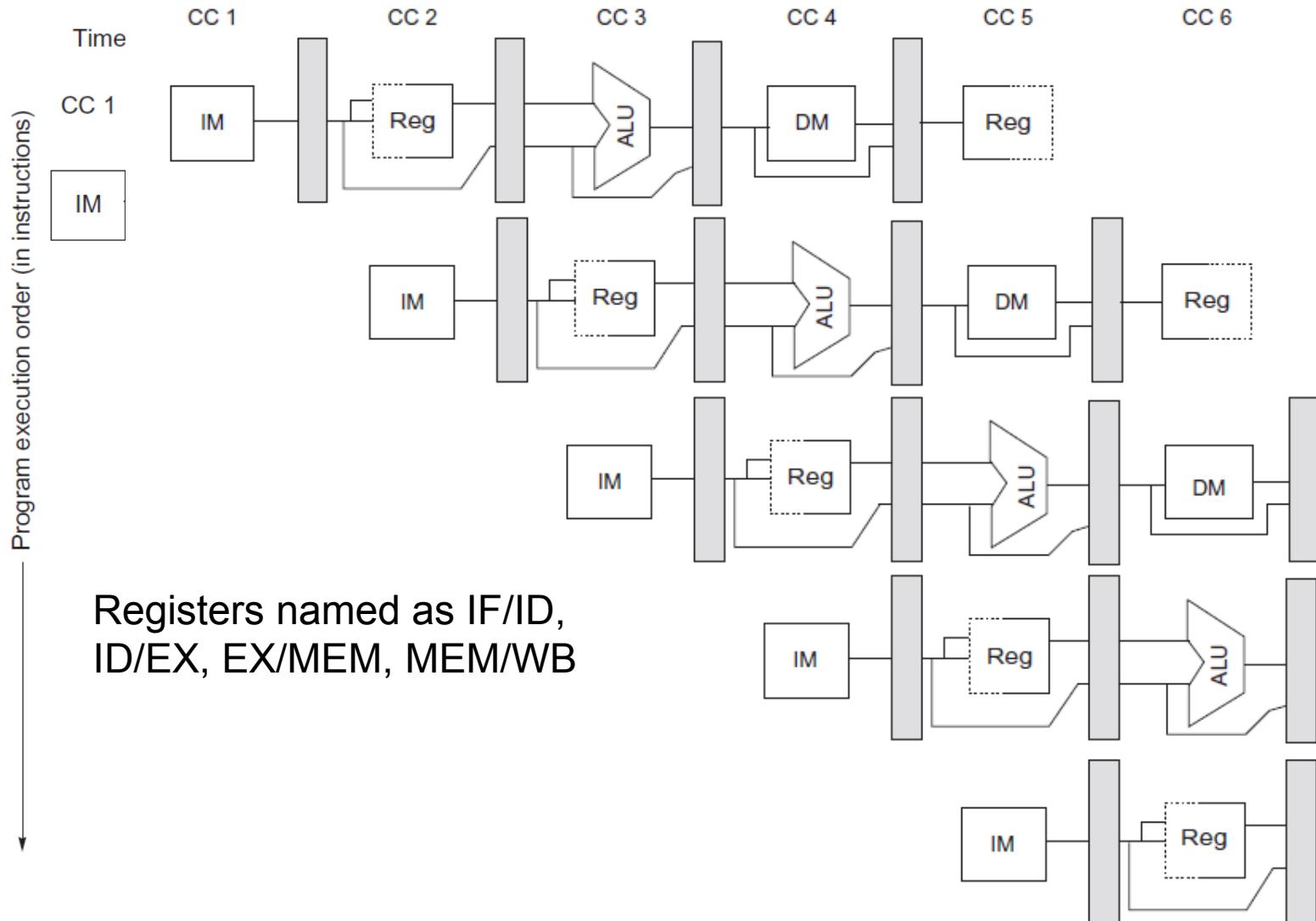
- Two different operations with same data path resources on the same clock cycle
- Overlapping operations have conflicts?

Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

How to omit Pipeline Issues

- Use separate instruction and data memories with separate caches
- Register file usage:
 - Reading from and writing to the same register – half of the clock cycle
- Instructions in different stages interfering with each other
 - → use pipeline registers between successive stages

Pipeline registers



Performance Issues

- Pipelining increases throughput but introduces overhead to control the pipeline
- Imbalance of pipeline stages
 - Speedup depends on slowest pipeline stage
- Pipeline register delay and clock skew
 - Registers need setup time and propagation time of clock cycle
 - Clock skew – time between arrival of clock edge at two registers

In class quiz

Example Consider the unpipelined processor in the previous section. Assume that it has a 2GHz clock (or a 0.5 ns clock cycle) and that it uses four cycles for ALU operations and branches and five cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.1 ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Answer The average instruction execution time on the unpipelined processor is

$$\begin{aligned}\text{Average instruction execution time} &= \text{Clock cycle} \times \text{Average CPI} \\ &= 0.5 \text{ ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5] \\ &= 0.5 \text{ ns} \times 4.4 \\ &= 2.2 \text{ ns}\end{aligned}$$

Speed up

Average instruction time unpipelined

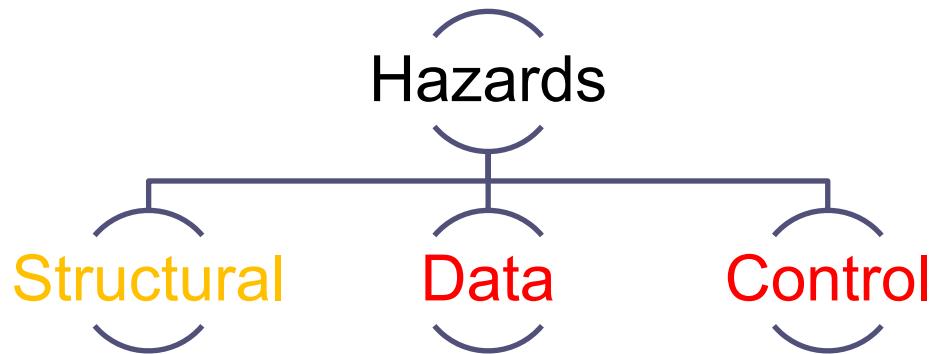
Average instruction time pipelined

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be $0.5 + 0.1$ or 0.6 ns; this is the average instruction execution time. Thus, the speedup from pipelining is

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \\ &= \frac{2.2 \text{ ns}}{0.6 \text{ ns}} = 3.7 \text{ times}\end{aligned}$$

Pipeline Hazards

- **Hazards** – situations preventing next instruction execution in designated clock cycle



- Structural hazard – resource conflict
- Data hazard – next instruction depending on result of current instruction during overlap
- Control hazards – branch and PC modifying instructions

Pipeline Stall

- During a hazard some instructions need delaying – stall
- All instructions after a stalled instruction also stop
- Instructions issued earlier than the stalled one must continue → clear the stall
- No new instructions fetched during stall

Data Hazards

Instruction i on register x

Instruction j on register x

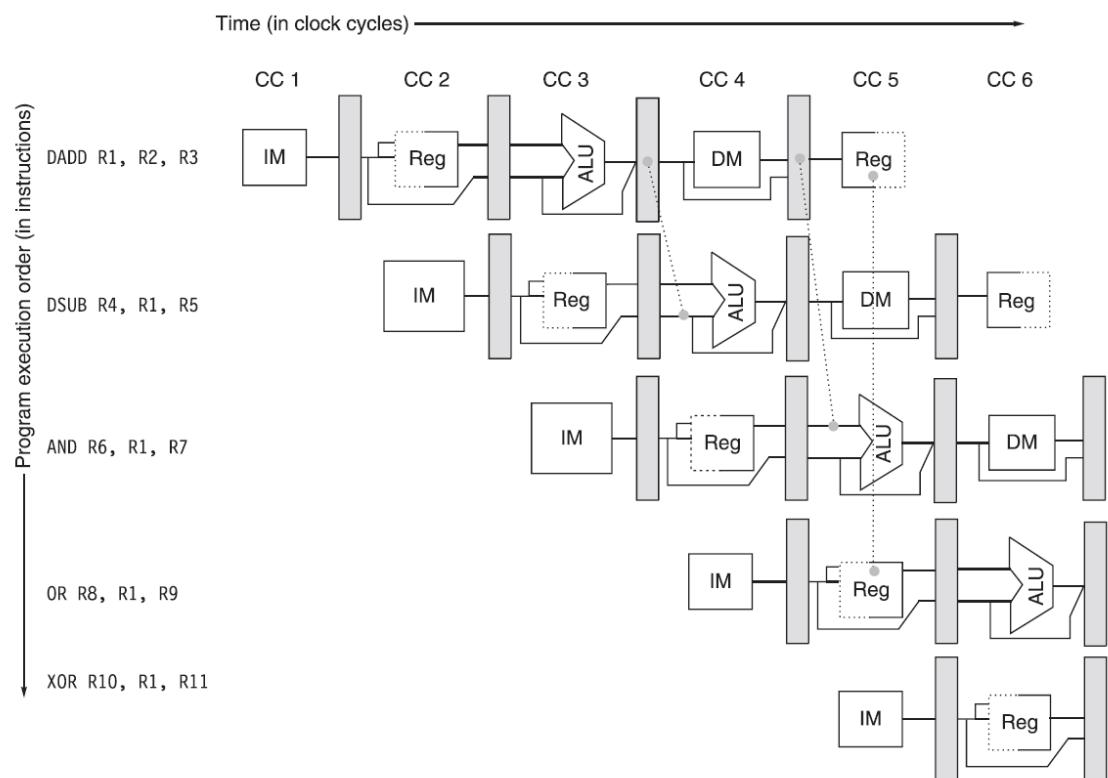
- Read after write (RAW) – j reads before i writes
- Write after read (WAR) – i reads after j writes
- Write after write (WAW) – i writes after j writes

Minimizing Data Hazard Stalls

□ Forwarding (=bypassing = short-circuiting)

- Copying the pipeline register where a result is generated to where the result is used

add	x1,x2,x3
sub	x4,x1,x5
and	x6,x1,x7
or	x8,x1,x9
xor	x10,x1,x11



Branch Hazard

- Greatest performance loss
- If a branch changes PC → *taken* branch
- Otherwise *untaken*
- PC changes only after ID

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Minimizing Control Hazard Stalls

- Freeze or flush the pipeline
- Predicted-untaken
- Predicted-taken
- Delayed-branch

Read appendix C in recommended text
Take home assignment – short note!

THANK YOU

Performance of Modern Computer Systems



CS2053 Computer Architecture

Computer Science & Engineering
University of Moratuwa

Sulochana Sooriyaarachchi
Chathuranga Hettiarachchi

Outline

- Instruction Level Parallelism : Beyond Pipelining
- Other Performance enhancements

Beyond pipelining

- What are the limitations of performance so far?
 - Pipelining
 - Clocks per Instruction (CPI)? Instructions per clock (IPC)?

Beyond pipelining

- What are the limitations of performance so far?

- Pipelining

- Clocks per Instruction (CPI)? Instructions per clock (IPC)?

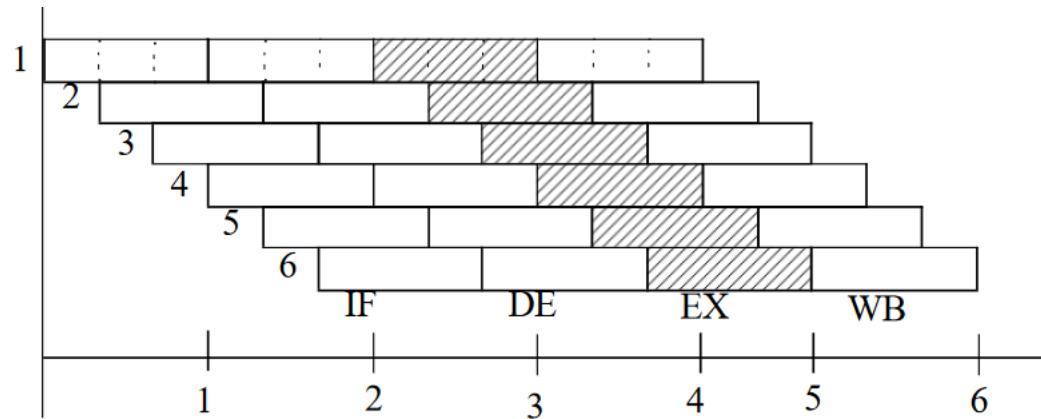
- Upper-bound on throughput

- $IPC \leq 1$ or $CPI \geq 1$

- A.k.a. “Flynn’s Bottleneck”

- Super-pipelining?

- Minor cycles



Beyond pipelining

- What are the limitations of performance so far?
 - Pipelining
 - Clocks per Instruction (CPI)? Instructions per clock (IPC)?

- Superscalar
 - Can we execute multiple instructions parallelly?
 - Two (or more) parallel pipelines?



Superscalar design

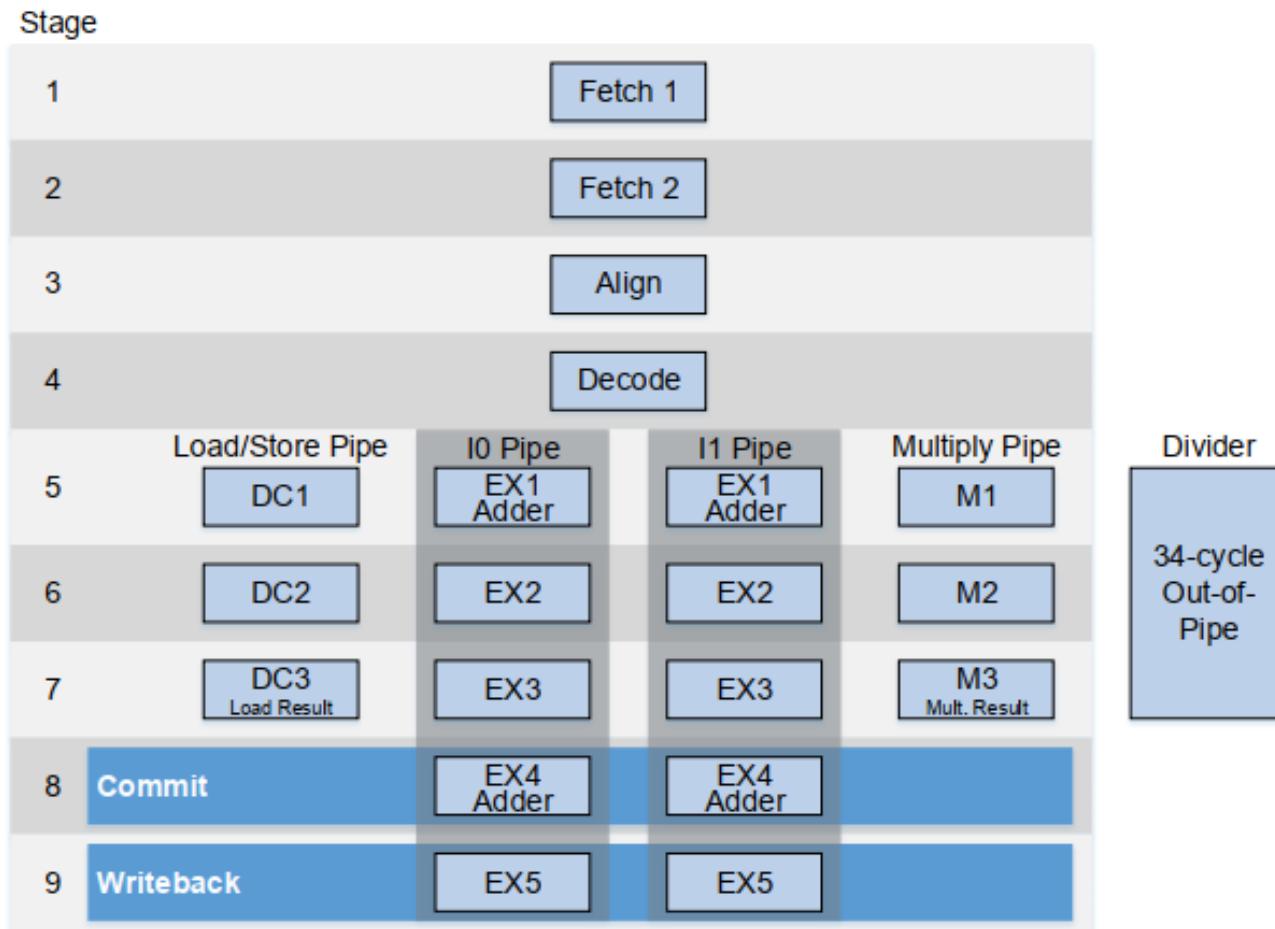


Figure 1-2 SweRV EH1 Core Pipeline

Superscalar design

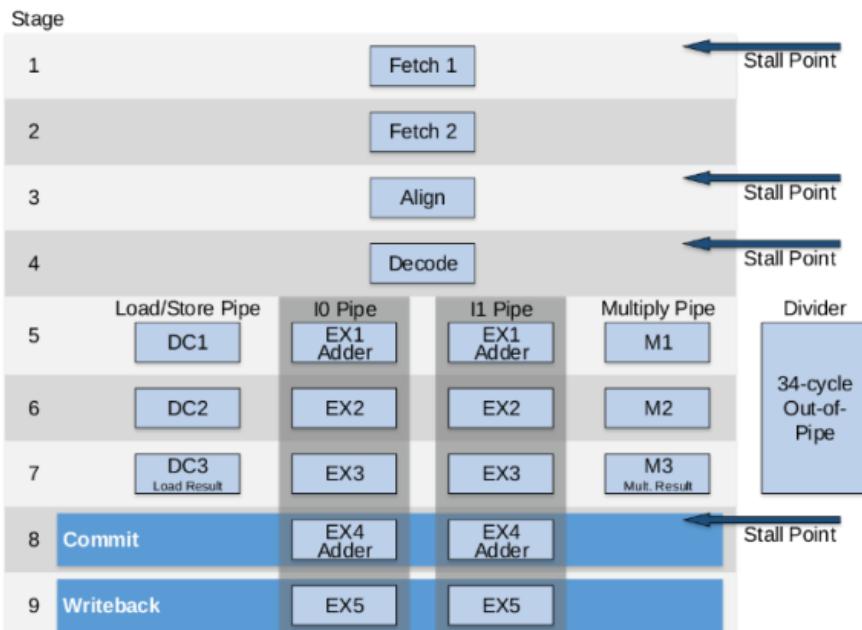


Figure 18. SweRV EH1 core microarchitecture

- ❑ Pipeline: Single issue; in-order
- ❑ Superscalar: Multiple issue; in-order
 - Multiple pipelines
- ❑ Stall Points?
- ❑ Out of order execution?

Single-core systems

- Pipelining
- Superscalar execution units
 - Dispatch multiple instructions during a single clock cycle
- Out of order execution
 - Execute instructions as soon as their operands are ready
- Large cache

Von Neumann bottleneck

- Data accessing is slow
- Overcoming the bottleneck
 - Caching
 - Prefetching
 - Speculative execution (Branch prediction)
 - New types of RAM
 - Multithreading
 - Near data processing
 - Hardware acceleration
 - System on a chip

Parallelism

- Classes of parallelism in applications
 - Data-Level Parallelism (DLP)
 - Task-Level Parallelism (TLP)
- Classes of architectural parallelism
 - Instruction-Level Parallelism (ILP)
 - Exploits DLP in pipelining & speculative execution
 - Vector architectures/Graphic Processor Units (GPUs)
 - Exploit DLP by applying same instruction on many data items
 - Thread-Level Parallelism
 - Exploit DLP & TLP in cooperative processing by threads
 - Request-Level Parallelism
 - Parallel execution of tasks that are independent

Flynn's Taxonomy

		Instruction Streams	
		one	many
Data Streams	one	SISD traditional von Neumann single CPU computer	MISD May be pipelined Computers
	many	SIMD Vector processors fine grained data Parallel computers	MIMD Multi computers Multiprocessors

Multithreading



Multithreading

- So far, in the processor we run a single program sequence
 - Single thread
- Can we run two programs at once?
 - Ex:
 - Operating system / GUI / User applications
 - Listen to music while editing a document
 - Download files via network while browsing another web page

Multi-threading

How to implement multithreading?

- Time sharing schemes in a single processor/core/ heart
 - Usually, processor core can run much faster than the human response times
 - OS supported context switching

- Run two threads in two different processors at once
 - Two computers?

Multithreading

- Simultaneous multi-threading/ Hyperthreading
 - CPU divides up its physical cores into virtual cores that are treated as if they are actually physical cores by the operating system
 - Intel proprietary technology

Complications in Multi-threading

- Communicate between different applications
 - Communicate between different threads

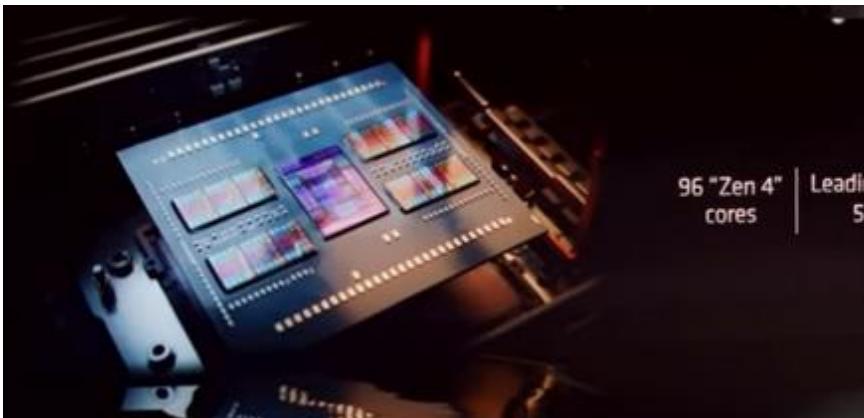
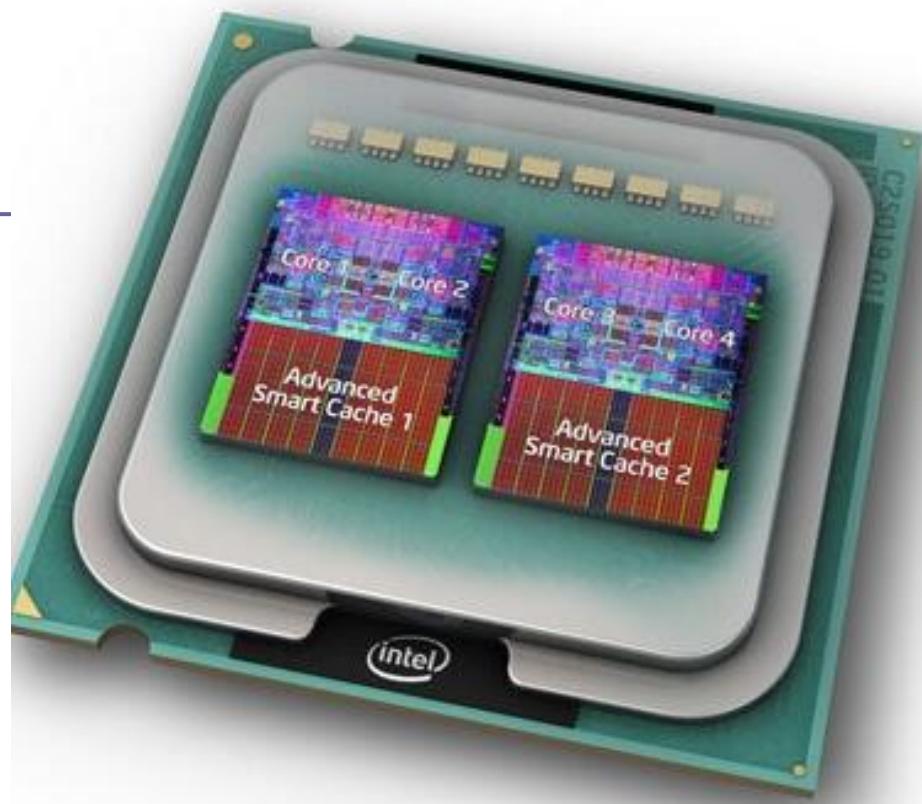
- Shared memory
 - Each program should have its own data and instruction memory
 - Different programs need to communicate using a shared memory
 - Need to ensure programs do not corrupt other programs (Should maintain access rights)

Multi-core processors



Multi-core processors

- More than one processor chip (Dual CPU)
 - Motherboard can mount two different processors
- Multiple cores / hearts in a single chip
- If there is more than one core, each thread can be assigned to run in a different core.
 - Pros:
 - Alternative to time sharing scheme
 - Own register banks. So, no need to switch contexts
 - Cons:
 - Cache coherence?



4TH GEN AMD EPYC™

96 "Zen 4"
cores

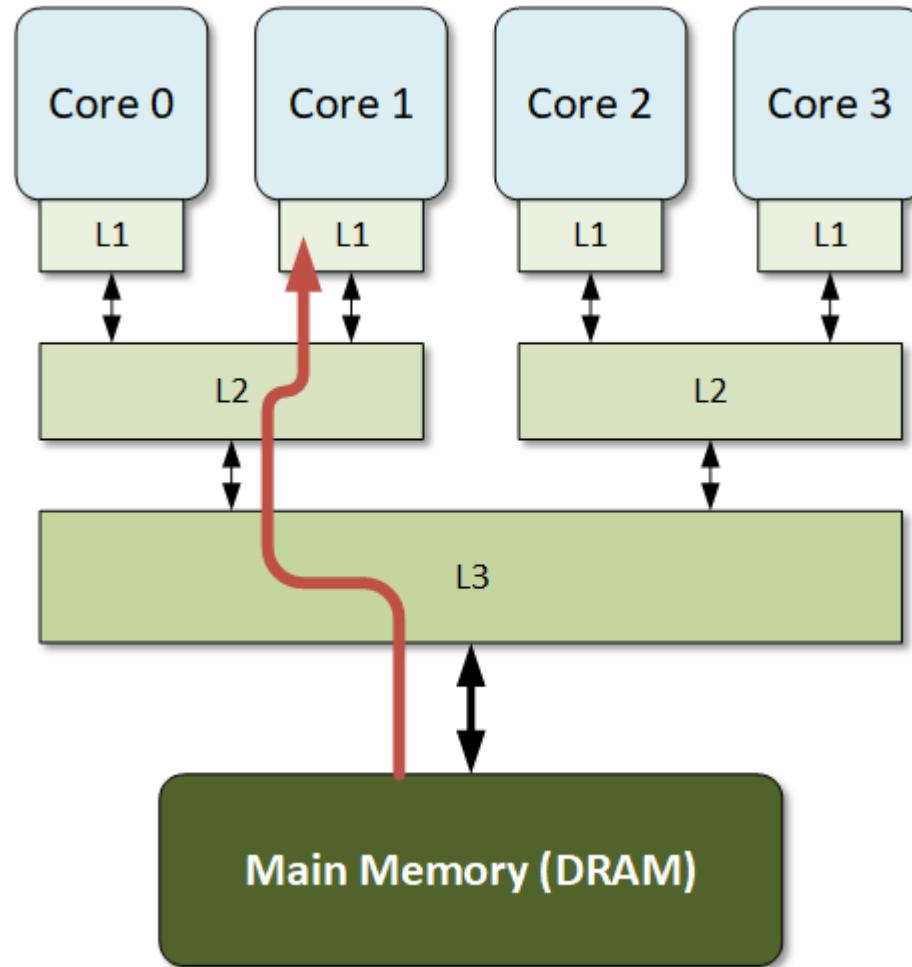
Leading edge
5nm

PCIe®5.0 | CXL™ memory
expansion

12 channels DDR5

~2x confidential
VMs

Cache placement options



Programming Model

- Programs which is written for serial processing can run without a problem.
 - OS can assist the management of the thread
- Programs which are parallelly executable can be written using multi-thread support
 - “Concurrent programming”
 - Shared memory concerns must be handled
 - Atomicity
 - Mutual exclusion
 - Synchronization

Instructions for multithreading

RV32A Atomic Extension

31		27 26		25 24		20 19		15 14		12 11		7 6		0		
funct5		aq	rl	rs2		rs1		funct3		rd		opcode				
5		1	1	5		5		3		5		7				
Inst	Name			FMT	Opcode	funct3	funct5	Description (C)								
lwr.w	Load Reserved			R	0101111	0x2	0x02	rd = M[rs1], reserve M[rs1]								
sc.w	Store Conditional			R	0101111	0x2	0x03	if (reserved) { M[rs1] = rs2; rd = 0 }			else { rd = 1 }					
amoswap.w	Atomic Swap			R	0101111	0x2	0x01	rd = M[rs1]; swap(rd, rs2); M[rs1] = rd								
amoadd.w	Atomic ADD			R	0101111	0x2	0x00	rd = M[rs1] + rs2; M[rs1] = rd								
amoand.w	Atomic AND			R	0101111	0x2	0x0C	rd = M[rs1] & rs2; M[rs1] = rd								
amoor.w	Atomic OR			R	0101111	0x2	0x0A	rd = M[rs1] rs2; M[rs1] = rd								
amoxor.w	Atomix XOR			R	0101111	0x2	0x04	rd = M[rs1] ^ rs2; M[rs1] = rd								
amomax.w	Atomic MAX			R	0101111	0x2	0x14	rd = max(M[rs1], rs2); M[rs1] = rd								
amomin.w	Atomic MIN			R	0101111	0x2	0x10	rd = min(M[rs1], rs2); M[rs1] = rd								

ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

Other instructions like csrrs
Privileged instructions

Many-core Architectures

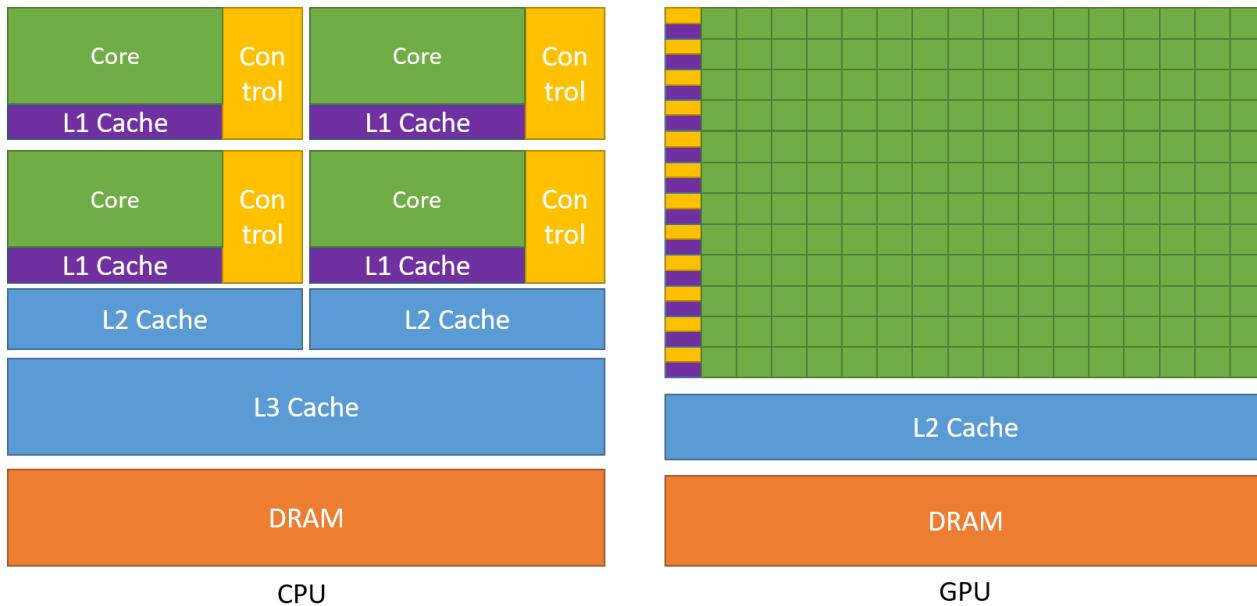


Multi-core vs Many-core machines

Multi-Core Processors	Many-Core Processors
<ul style="list-style-type: none">• As an extension of single core processors• Designed to run both parallel and serial code• Performance emphasis on single thread performance (out-of-order execution, pipelining, superscalar execution units, larger cache, shared memory.)• Few cores• Complemented by manycore accelerators (ex. GPU)	<ul style="list-style-type: none">• Intended for higher degree of explicit parallelism• Higher throughput• Low power consumption• High latency and lower single thread performance

Graphical Processing Units (GPU)

- Graphics are inherently parallel
 - Drawing each pixel can be done independently.
 - Scanning row by row is not efficient
 - Divide the image to number of sections and assign to number of threads/processors
 - If we have
 - Two cores: Twice the performance as single core
 - Four cores: Four times the single core
 - N cores: N times the single core
- Offload the graphics processing entirely to a specialized co-processor(s)



- For applications with high degree of parallelism
- Large number of cores allow massively parallel programs to execute in parallel

Programming for GPUs

- For applications with high degree of parallelism
- Large number of cores allow massively parallel programs to execute in parallel

Traditional loops

```
void
.mul(const int n,
      const float *a,
      const float *b,
      float *c)
{
    int i;
    for (i = 0; i < n; i++)
        c[i] = a[i] * b[i];
}
```

OpenCL Kernel

```
__kernel void
mul(__global const float *a,
     __global const float *b,
     __global      float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
}
```

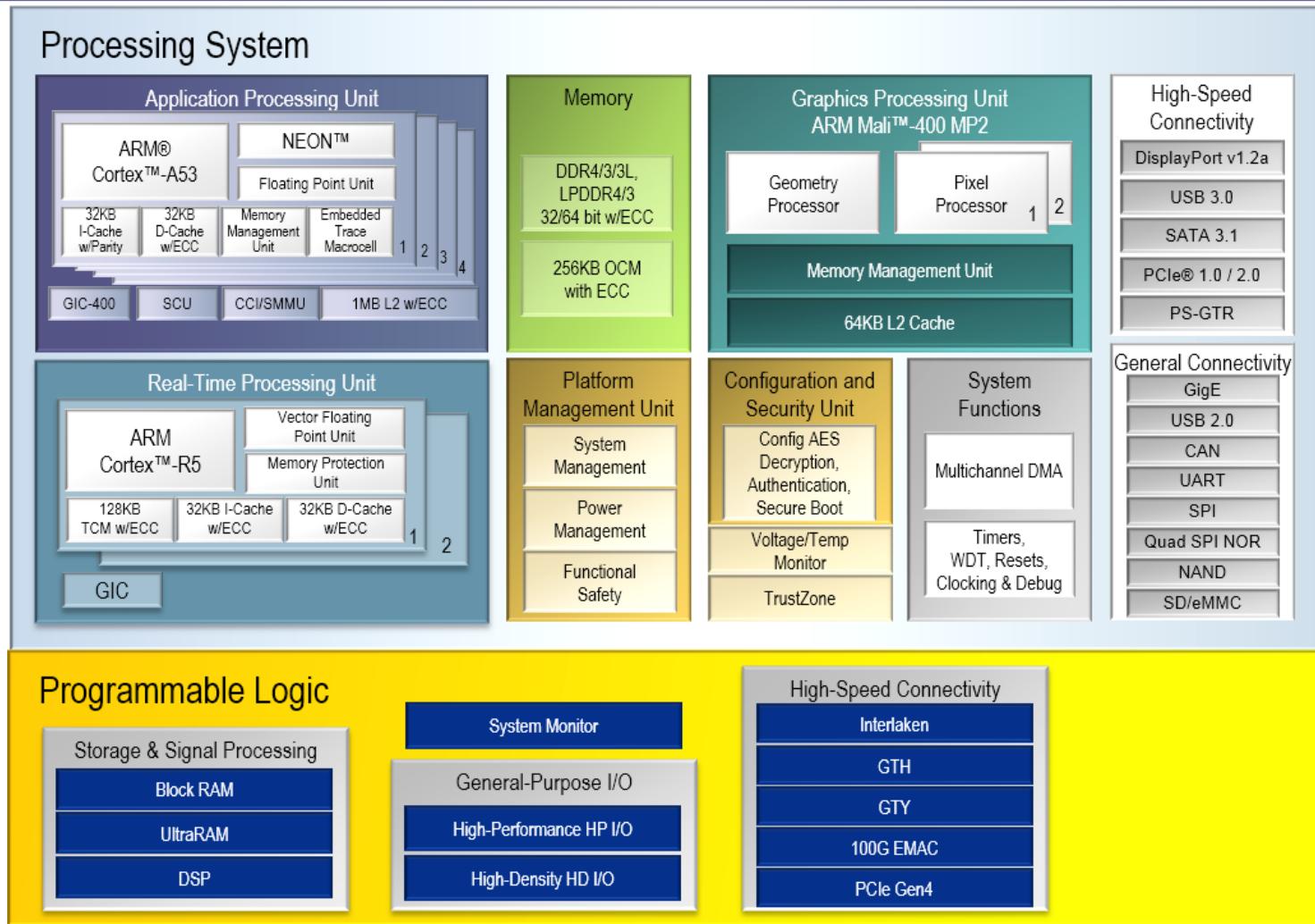
Heterogeneity



Heterogeneity

- Symmetric multiprocessing (SMP)
 - Two (or more) copies of similar processors
 - Typical in Many-core/multi-core systems
- Asymmetric multiprocessing(AMP)
 - Processors for specific tasks
 - Audio processor
 - Graphic co-processor
 - Direct Memory Access (DMA)
 - Cryptographic co-processors
- MPSoCs
 - Heterogenous Multi Processor Systems on Chip.

System-on-Chip (SoC)s



Other Co-Processing Units

- Neural Processing Units / Tensor Processing Units / Systolic arrays / Data Processing Units
 - Neural Network on hardware
 - Network on a chip
 - Multiply and accumulate applications
 - Massively parallel integration
 - Convolution, correlation, matrix multiplication or data sorting tasks.
 - Dynamic programming algorithms, used in DNA and protein sequence analysis

Warehouse scale parallelism

- Cloud computing/Distributed computing approaches
- Different Programming models
 - Example:
 - Map - Reduce
 - Lambda architecture

New Paradigms

❑ Processing In Memory

- Can we bring processing and memory together.
 - ❑ Brain does not have separate memory and processing units

Thank you!

Input & Output



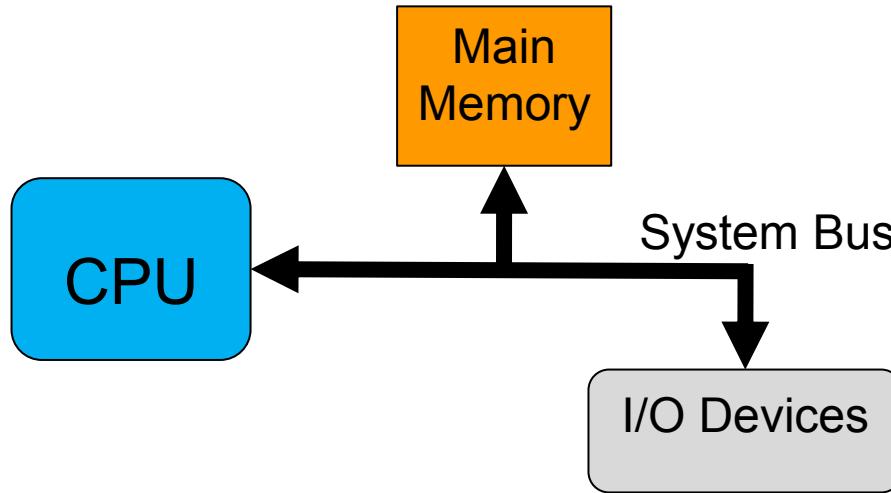
CS2053 Computer Architecture
Computer Science & Engineering
University of Moratuwa

Dr. Sulochana Sooriyaarachchi

Dr. Chathuranga Hettiarachchi

Acknowledgement: Dr. Dilum Bandara

Input & Output

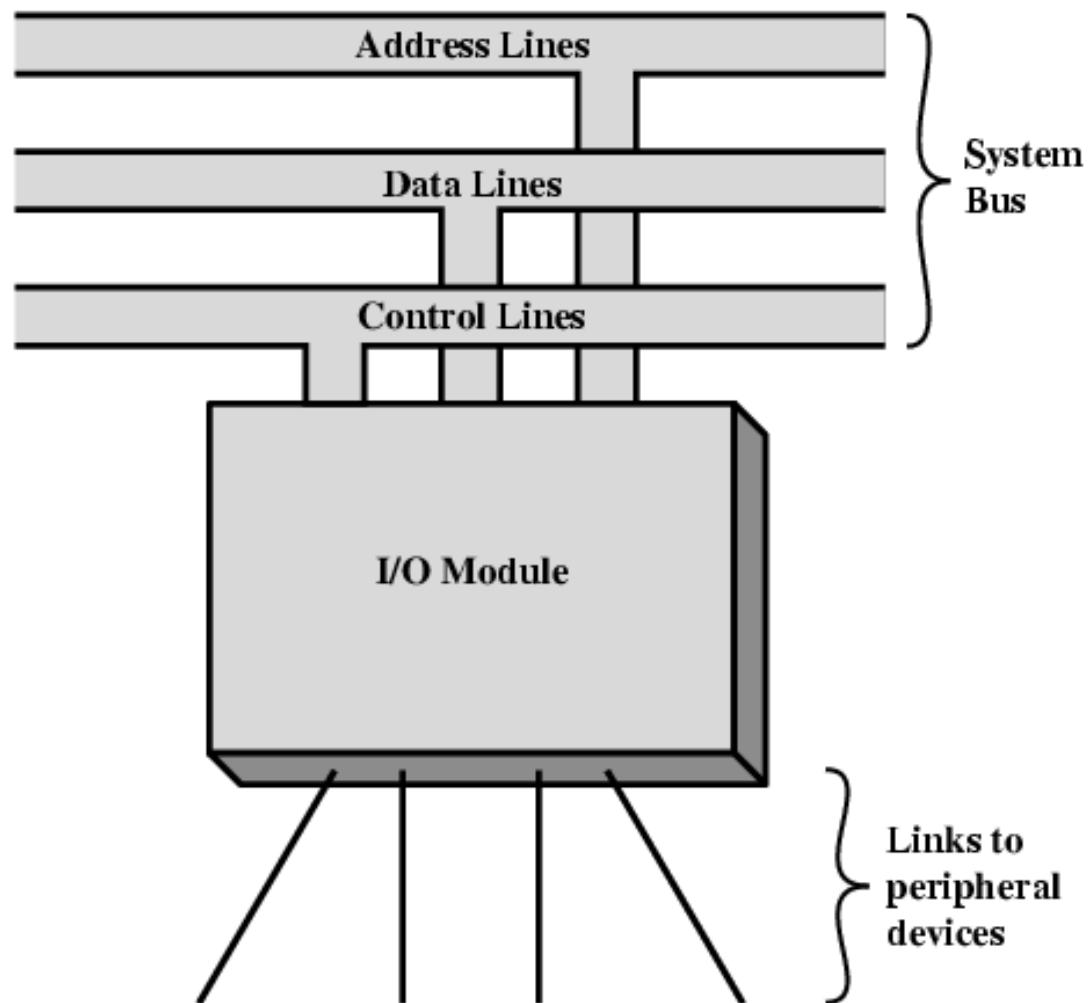


- ❑ Wide variety of peripherals
 - Different volumes of data, in different formats, & different speeds
- ❑ All slower than CPU & RAM
- ❑ Controlled via I/O modules/controllers
 - Interface to CPU & Memory
 - Interface to 1 or more peripherals

External Devices

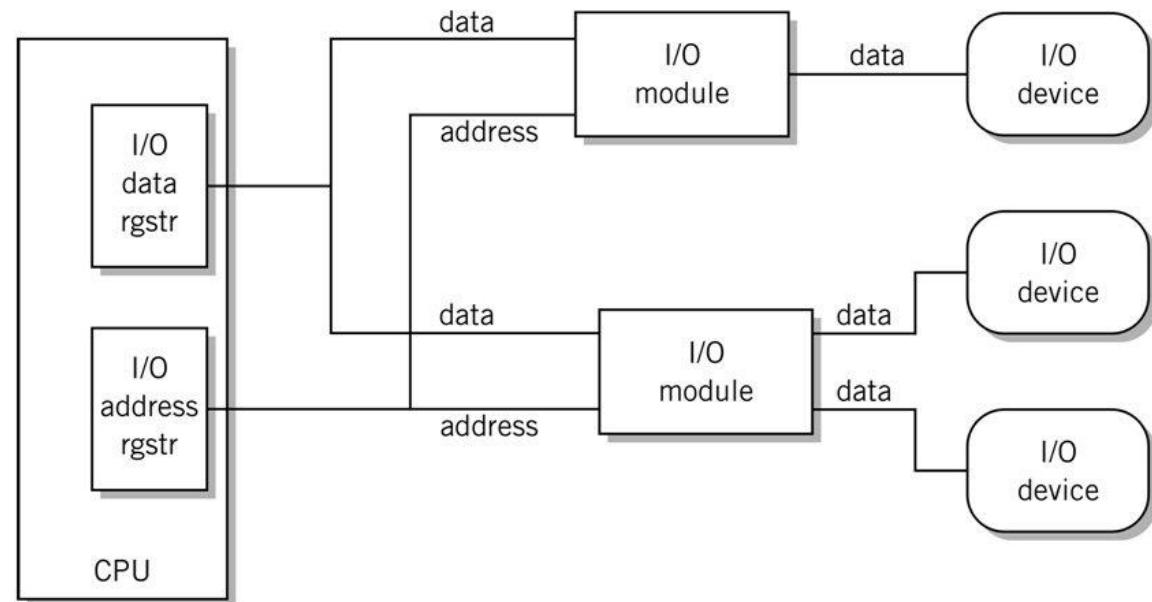
- Interact with humans
 - Monitor, printer, keyboard, mouse
- Machine readable
 - Monitoring & control
 - e.g., process scheduling, CPU/casing temperature monitoring, fan speed control
- Communication
 - Network Interface Card (NIC)
 - Modems
 - Dongles – Bluetooth, Wi-Fi, 3G, 4G

Generic Model of I/O Module

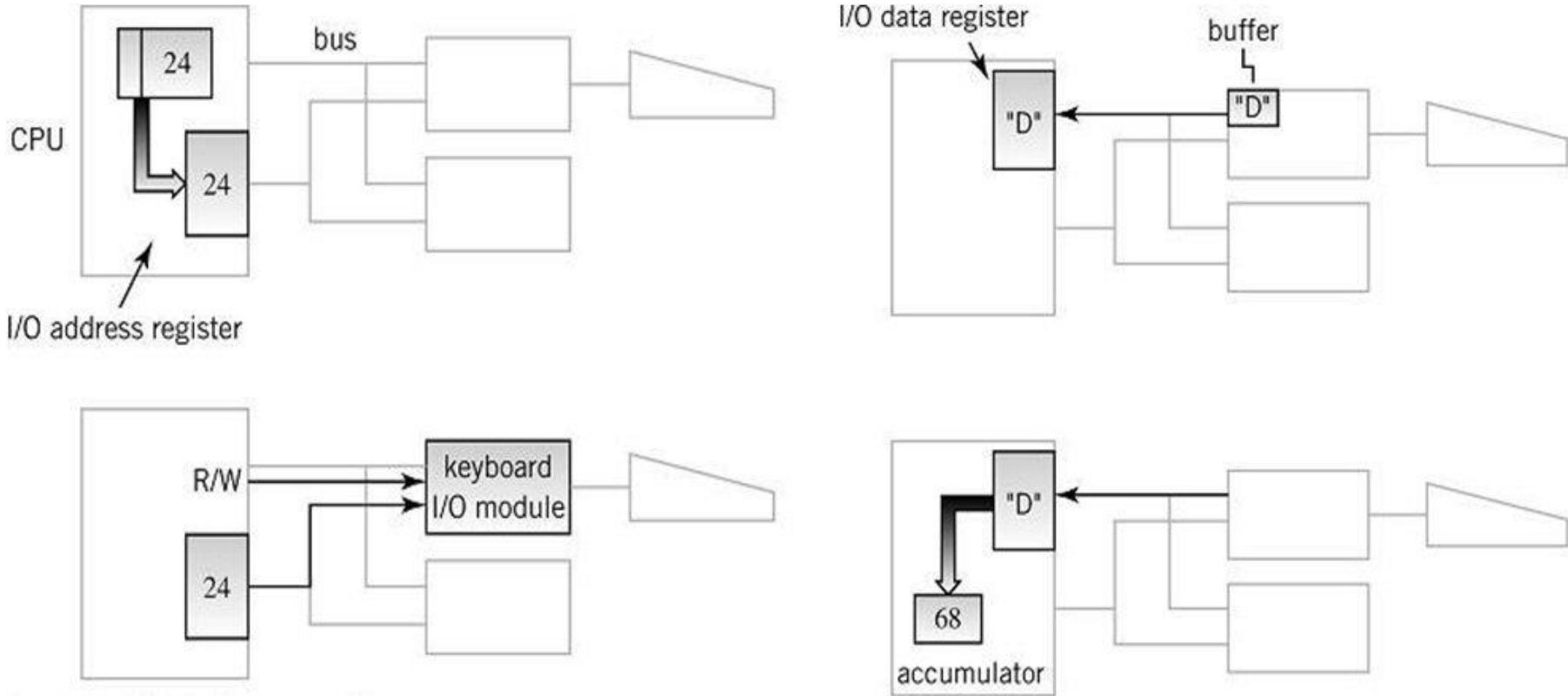


I/O Module Functions

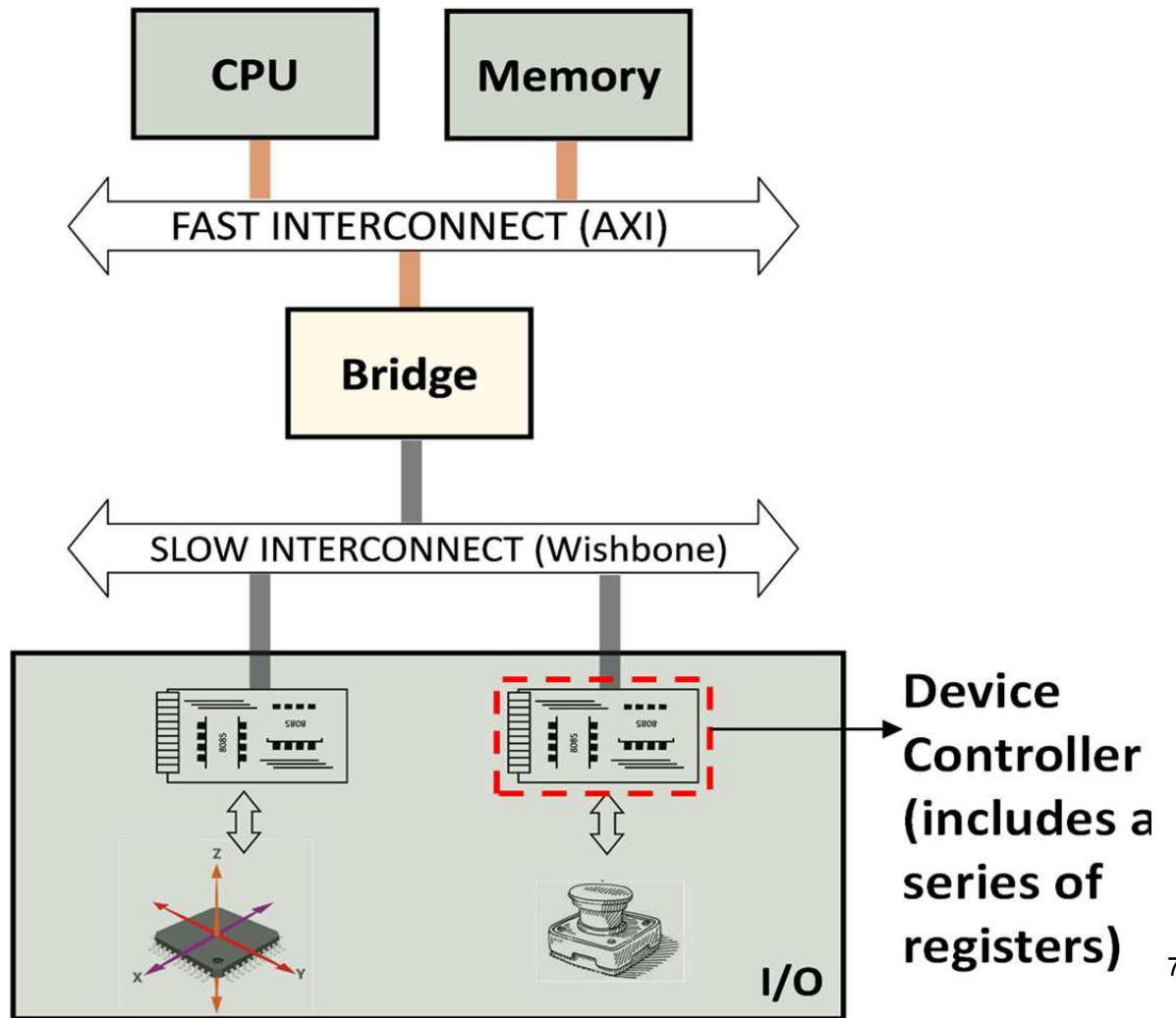
- Control & Timing
- CPU communication
- Device communication
- Data buffering
- Error detection



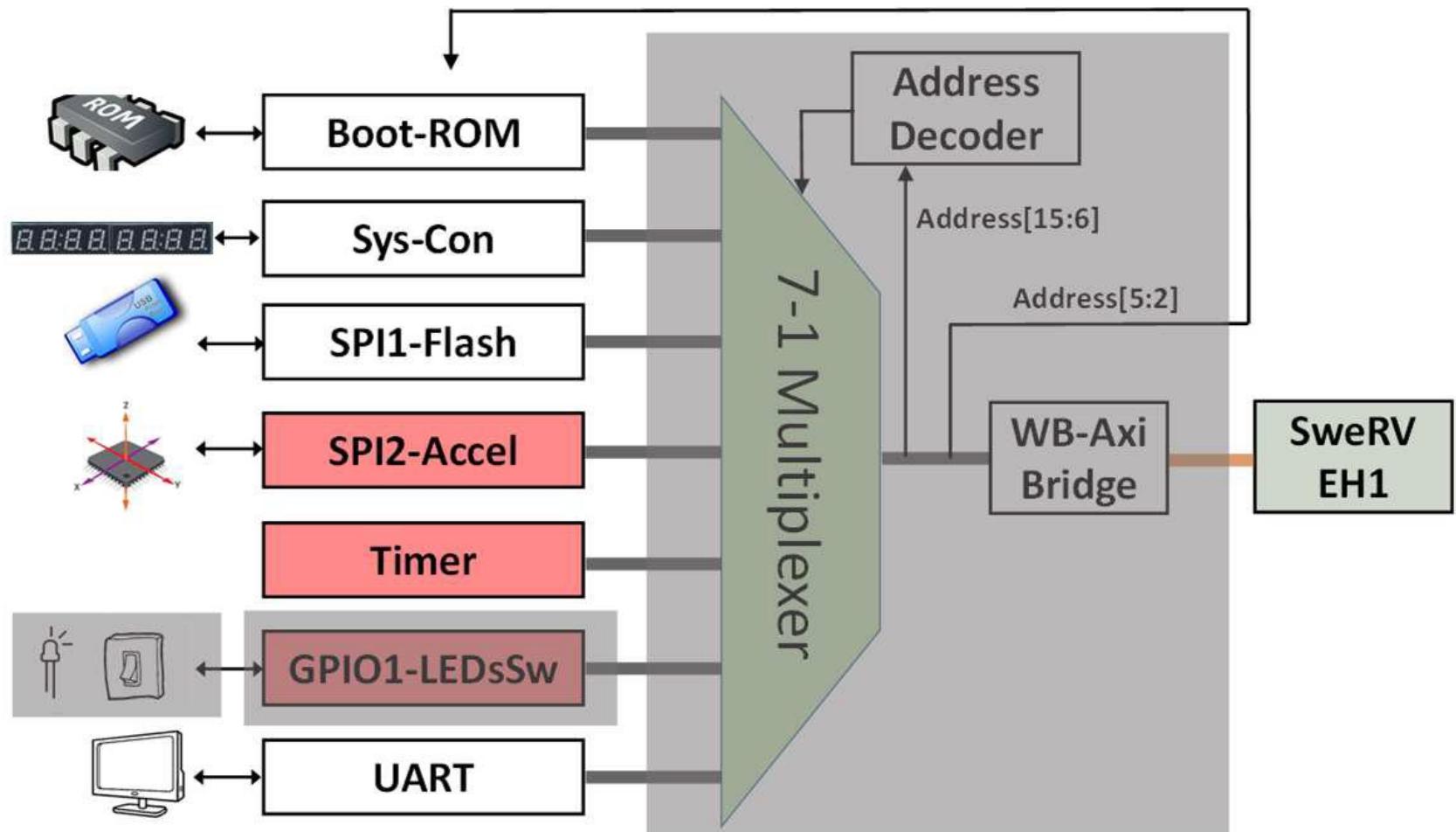
I/O Module Functions – Example



SweRVolfX



SweRVolfX IO at low level



Input Output Techniques

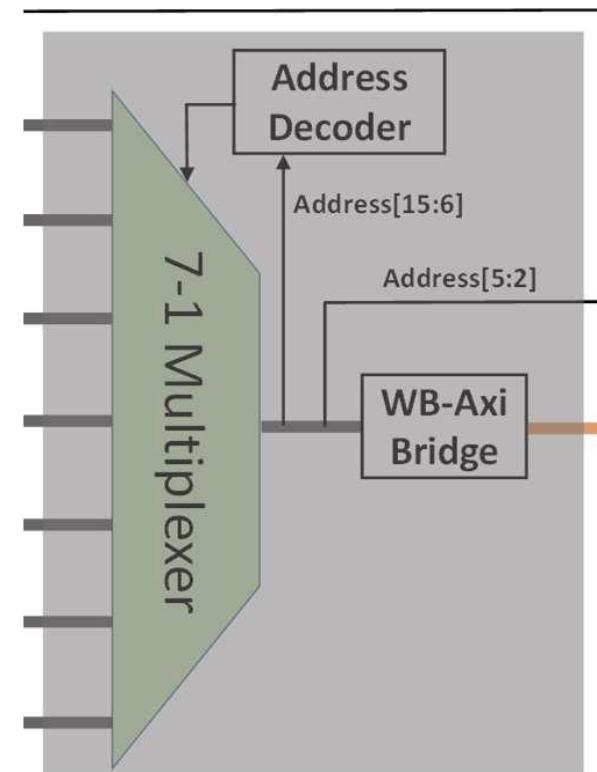
- Programmed I/O
 - Polling
- Interrupt driven I/O
- Direct Memory Access (DMA)

Programmed I/O

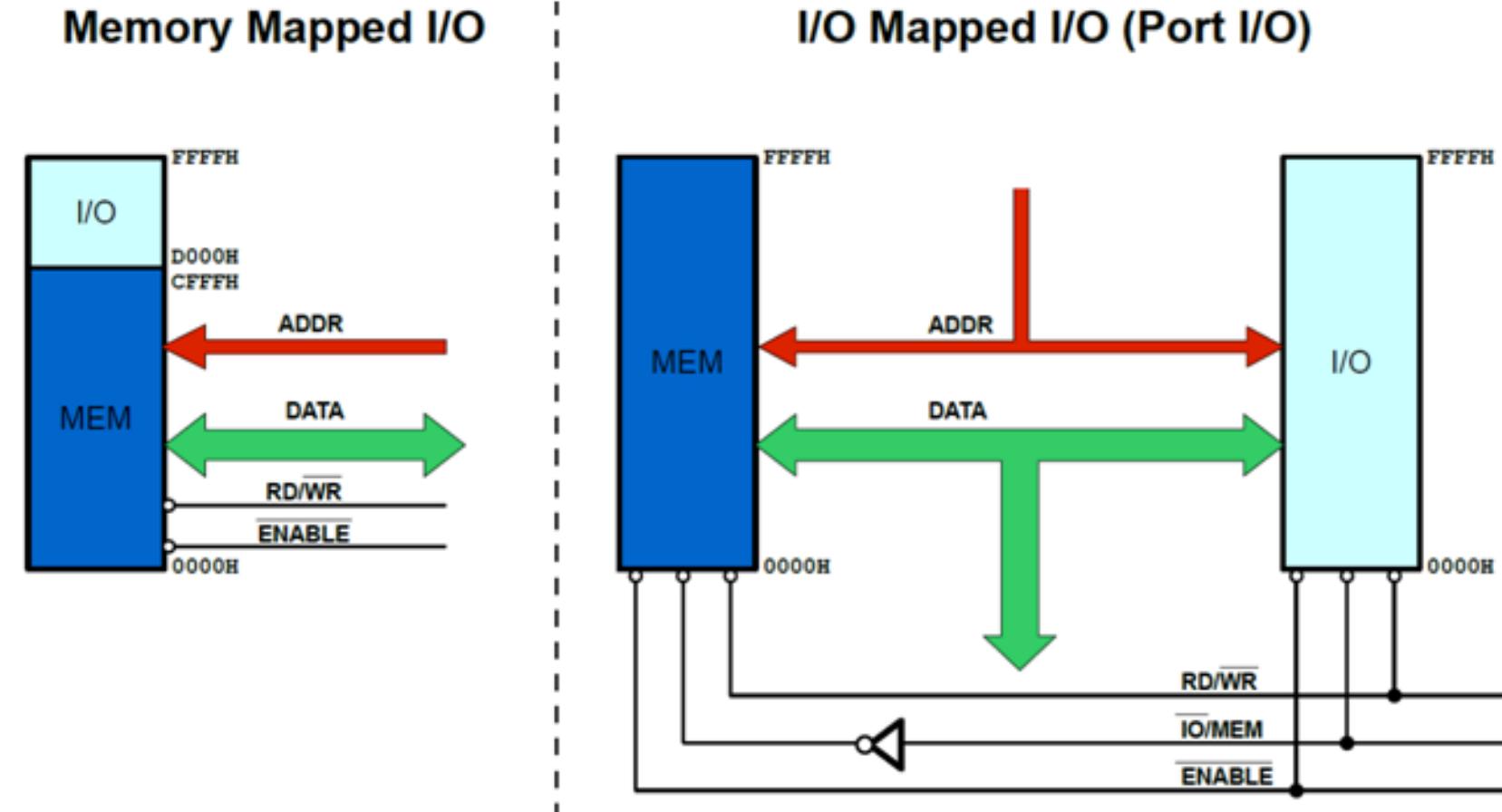
- ❑ CPU has direct control over I/O
 - Continuously sense status – Poll
 - Read/write commands
 - Transferring data
- ❑ CPU waits for I/O module to complete operation
- ❑ Wastes CPU time

Addressing I/O Devices

- Under programmed I/O data transfer is like memory access
 - Use LOAD/STORE instructions
- Each device is given a unique identifier
 - Address[15:6]
- CPU commands contain identifier (address)
- Registers within I/O controller are identified by
 - Address[5:2]



I/O Mapping



Source: <http://me-lrt.de/memory-map-port-isolated-input>

I/O Mapping (Cont.)

Memory Mapped I/O

- Devices & memory share same address space
- I/O looks just like memory read/write
- No special commands for I/O
 - Large selection of memory access commands

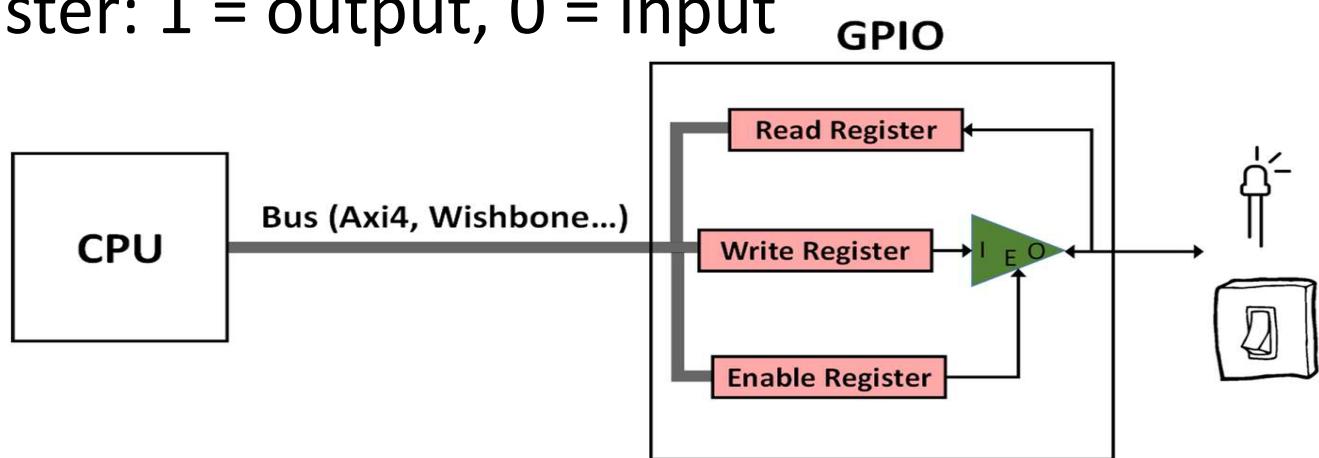
Isolated I/O

- Separate address spaces
- Need I/O or memory select lines
- Special commands for I/O
 - Limited set

E.g. Memory mapped - GPIO

Three memory-mapped registers:

- Read Register: value read from pin
- Write Register: value to write to pin
- Enable Register: 1 = output, 0 = input

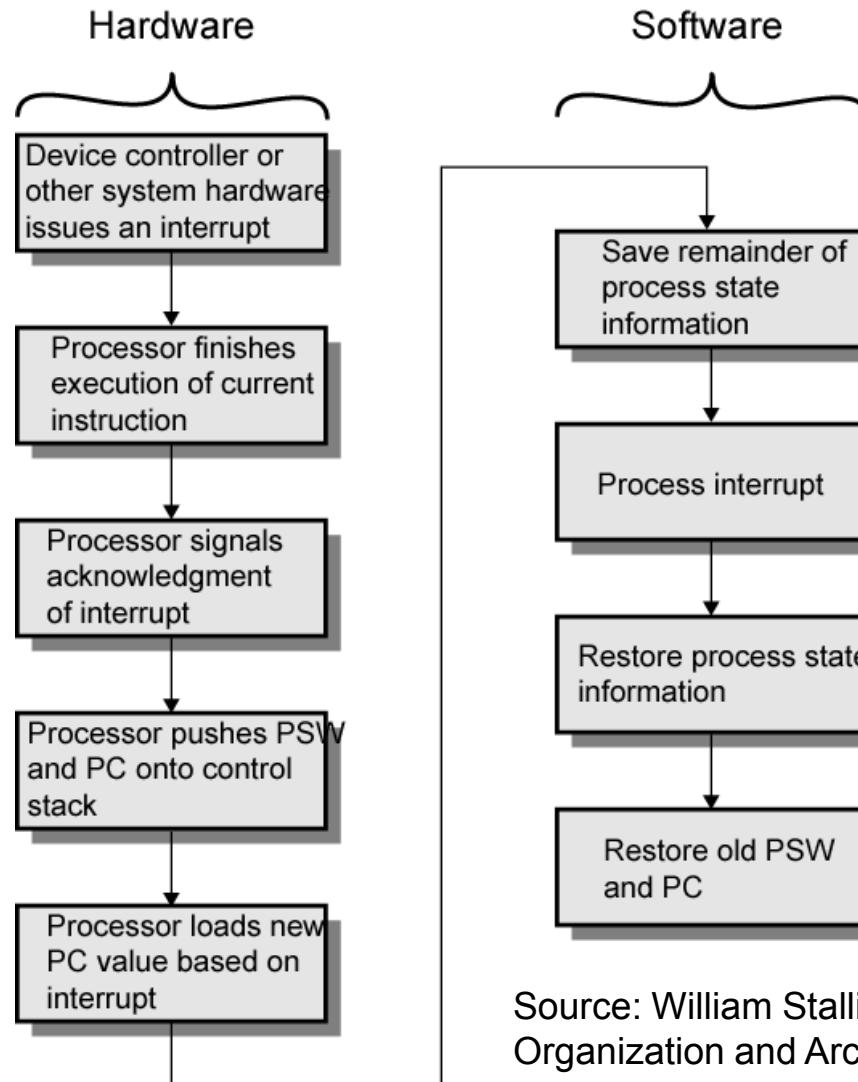


Register	Memory-Mapped Address
Read Register	0x80001400
Write Register	0x80001404
Enable Register	0x80001408

Interrupt Driven I/O

- No repeated CPU checking of device
 - No waiting
 - CPU does its own work
- I/O module interrupts CPU when ready
- Steps
 - CPU issues read command
 - I/O module gets data from peripheral whilst CPU does other work
 - I/O module interrupts CPU
 - CPU requests data
 - I/O module transfers data

Interrupt Processing (Cont.)



Source: William Stallings, Computer Organization and Architecture, 8th Edition

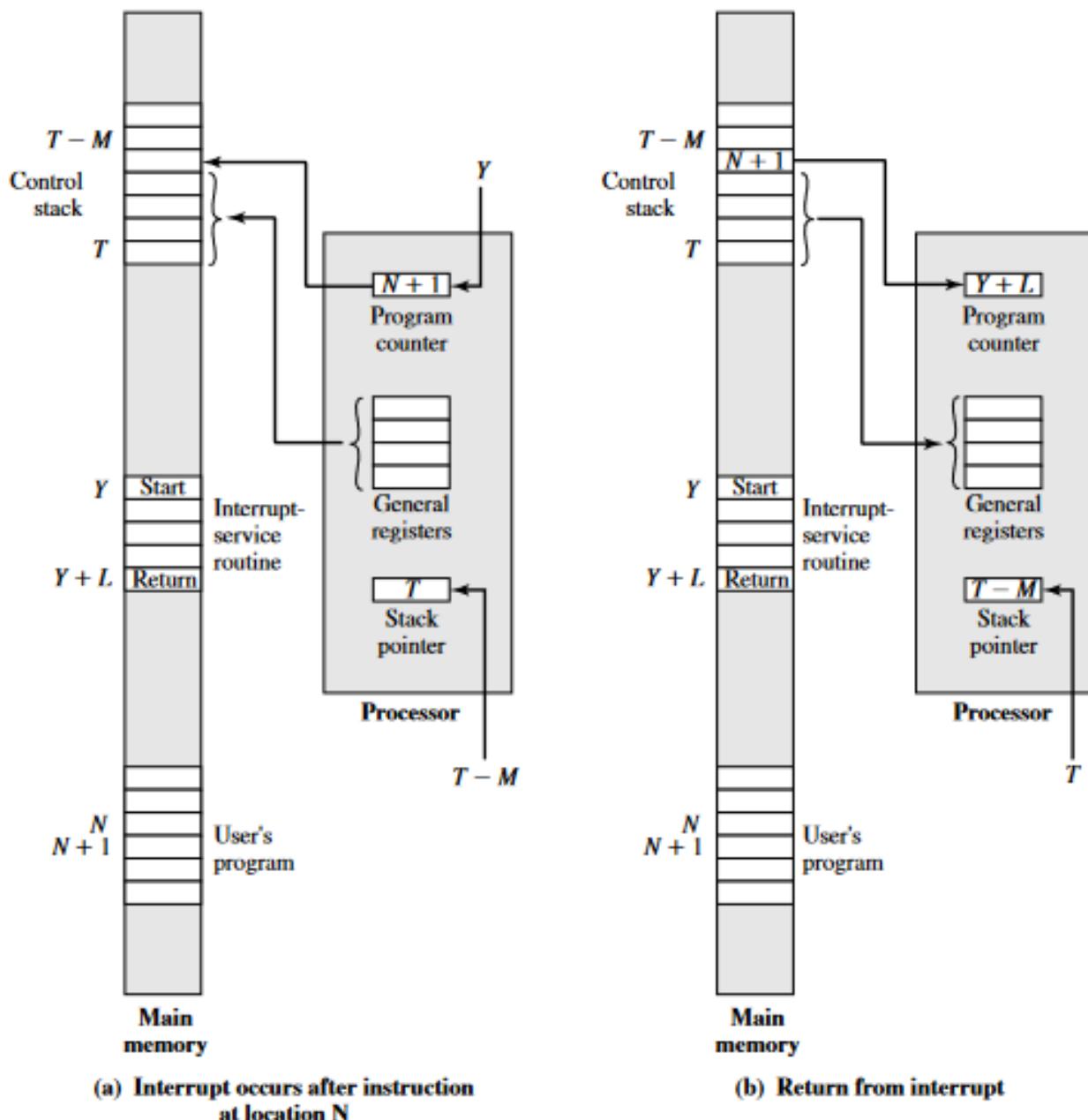
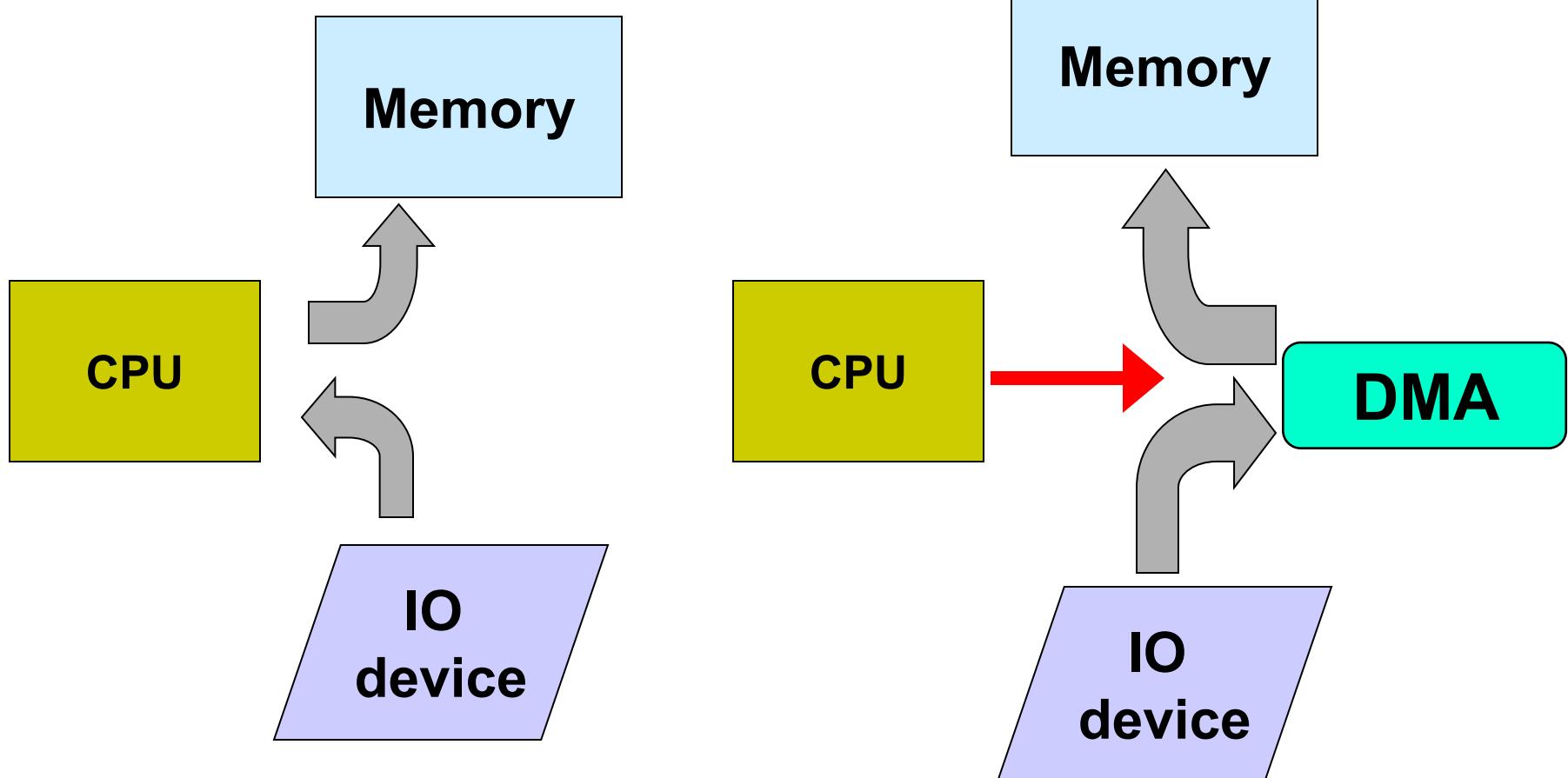


Figure 7.7 Changes in Memory and Registers for an Interrupt

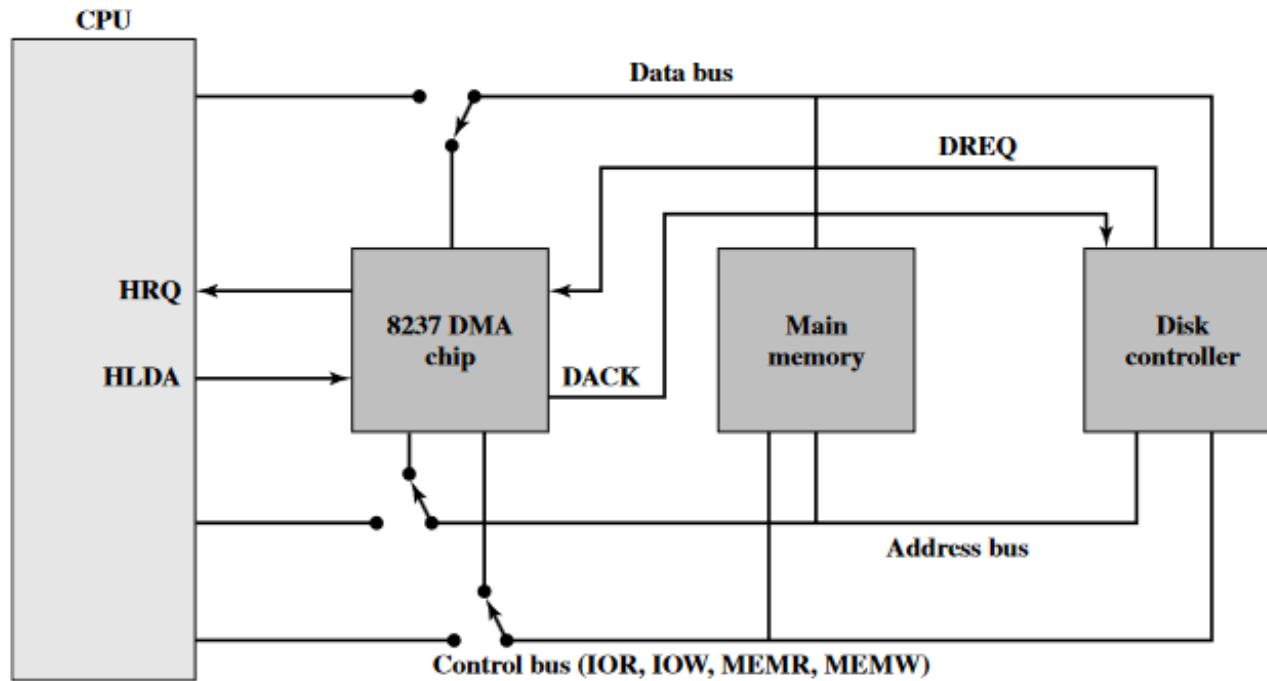
Direct Memory Access (DMA)

- Programmed & interrupt driven I/O require active CPU intervention
 - Transfer rate is limited
 - CPU is tied up in data transfer
- DMA is the answer
 - Additional module (hardware) on bus
 - DMA controller takes over from CPU for I/O
 - Provide a way of bypassing CPU when transferring data between memory & IO

DMA (Cont.)



Example bus configuration



DACK = DMA acknowledge

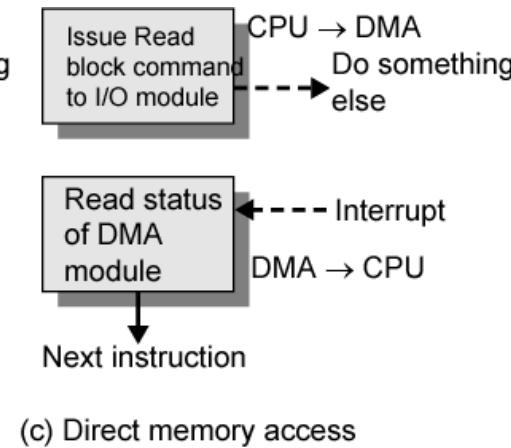
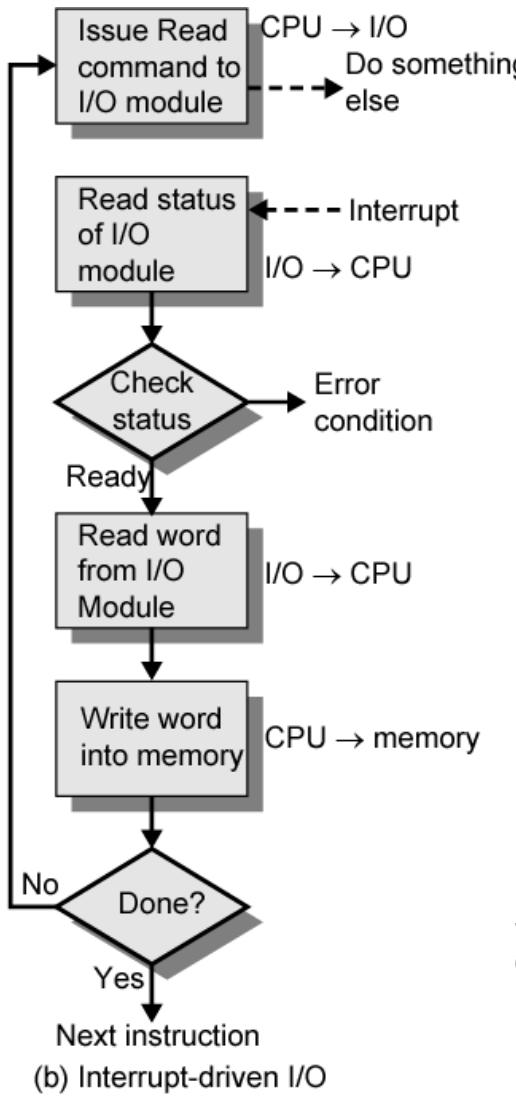
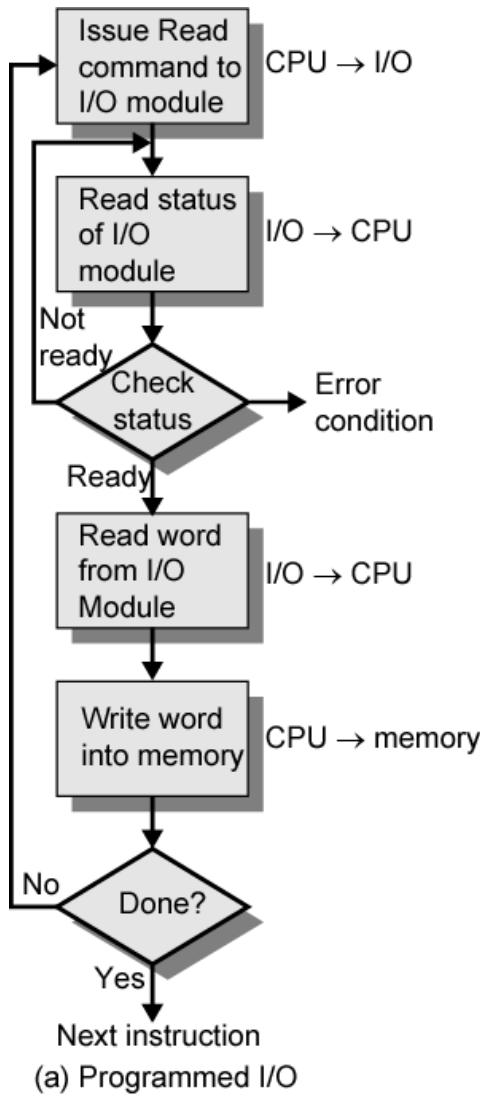
DREQ = DMA request

HLDA = HOLD acknowledge

HRQ = HOLD request

Figure 7.14 8237 DMA Usage of System Bus

3 Techniques for Input of a Block of Data



Source: William Stallings, Computer Organization and Architecture, 8th Edition

Blocks of a Computer

Further Reading

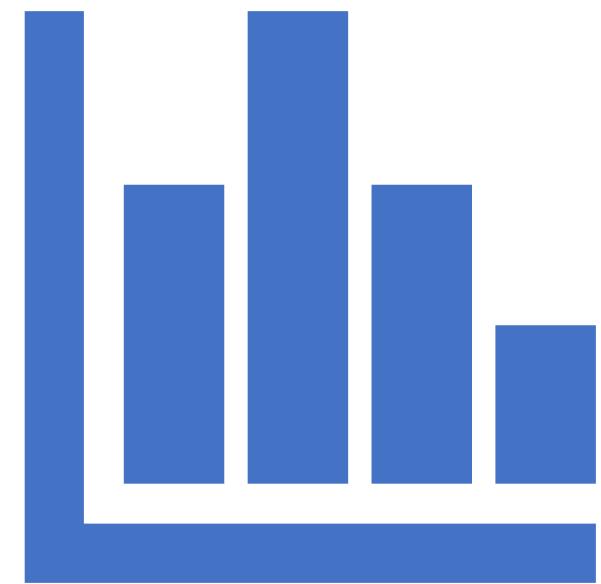
Chapter 7 : Computer Organization and Architecture : Designing for Performance (8th Edition), by William Stallings

THANK YOU

Computer Architecture

Performance Assessment : Quantitative Approach

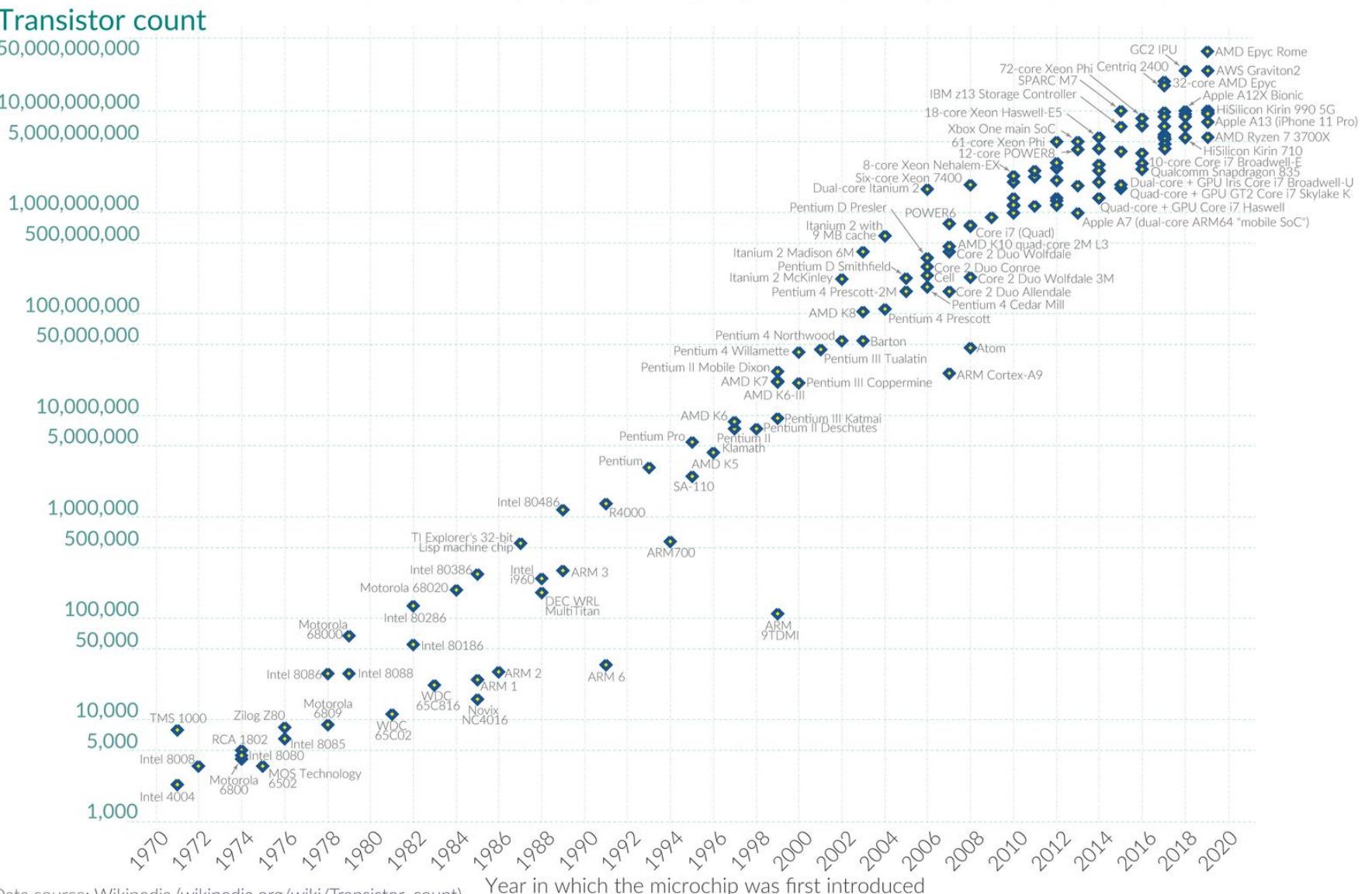
1. Limits of performance
2. Micro-architecture Design
3. Domain Specific Architectures



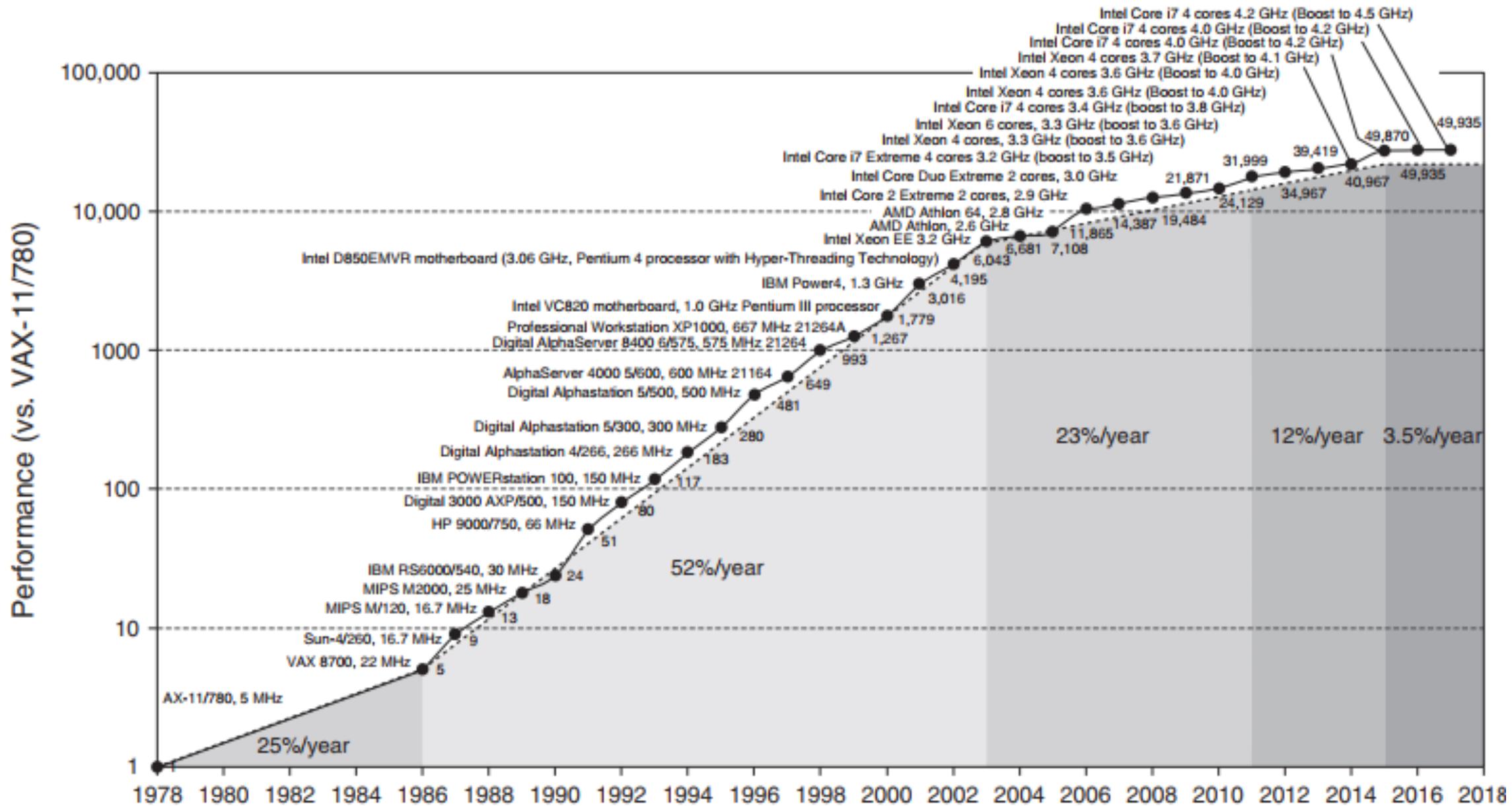
Moore's Law

Moore's Law: The number of transistors on microchips doubles every two years
 Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
 This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World
in Data



Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.



Limits of performance?

- 1st-level, 2nd-level, 3rd-level, and even 4th-level caches
- 512-bit SIMD floating-point units
- 15+ stage pipelines
- Branch prediction
- Out-of-order execution
- Speculative prefetching
- Multithreading
- Multiprocessing

Moore's Law can't continue forever ... We have another 10 to 20 years before we reach a fundamental limit

Gordon Moore,
Intel Co-Founder (2005)

Feature	Personal mobile device (PMD)	Desktop	Server	Clusters/warehouse-scale computer	Internet of things/embedded
Price of system	\$100–\$1000	\$300–\$2500	\$5000–\$10,000,000	\$100,000–\$200,000,000	\$10–\$100,000
Price of microprocessor	\$10–\$100	\$50–\$500	\$200–\$2000	\$50–\$250	\$0.01–\$100
Critical system design issues	Cost, energy, media performance, responsiveness	Price-performance, energy, graphics performance	Throughput, availability, scalability, energy	Price-performance, throughput, energy proportionality	Price, energy, application-specific performance

Figure 1.2 A summary of the five mainstream computing classes and their system characteristics. Sales in 2015 included about 1.6 billion PMDs (90% cell phones), 275 million desktop PCs, and 15 million servers. The total number of embedded processors sold was nearly 19 billion. In total, 14.8 billion ARM-technology-based chips were shipped in 2015. Note the wide range in system price for servers and embedded systems, which go from USB keys to network routers. For servers, this range arises from the need for very large-scale multiprocessor systems for high-end transaction processing.

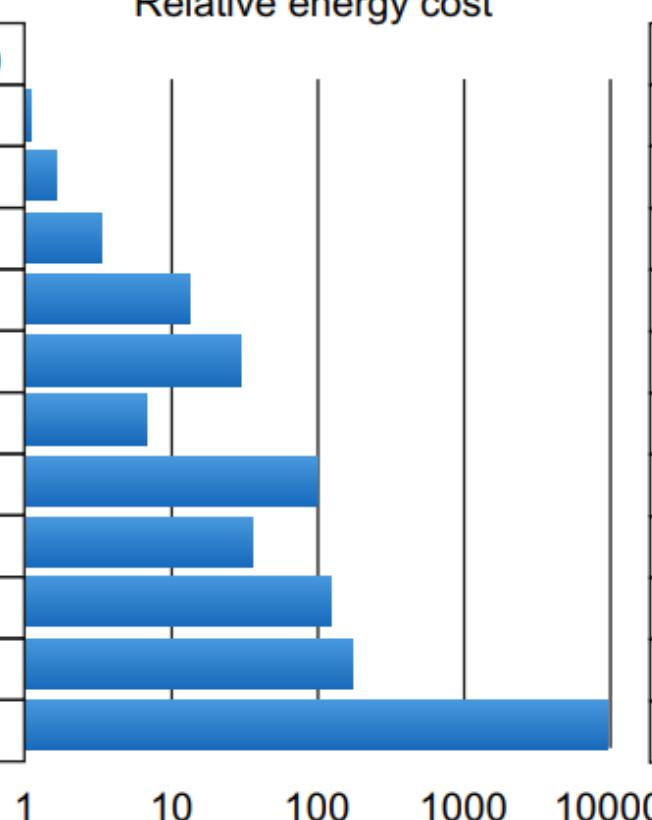
Microprocessor	16-Bit address/ bus, microcoded	32-Bit address/ bus, microcoded	5-Stage pipeline, on-chip I & D caches, FPU	2-Way superscalar, 64-bit bus	Out-of-order 3-way superscalar	Out-of-order superpipelined, on-chip L2 cache	Multicore OOO 4-way on chip L3 cache, Turbo
Product	Intel 80286	Intel 80386	Intel 80486	Intel Pentium	Intel Pentium Pro	Intel Pentium 4	Intel Core i7
Year	1982	1985	1989	1993	1997	2001	2015
Die size (mm ²)	47	43	81	90	308	217	122
Transistors	134,000	275,000	1,200,000	3,100,000	5,500,000	42,000,000	1,750,000,000
Processors/chip	1	1	1	1	1	1	4
Pins	68	132	168	273	387	423	1400
Latency (clocks)	6	5	5	5	10	22	14
Bus width (bits)	16	32	32	64	64	64	196
Clock rate (MHz)	12.5	16	25	66	200	1500	4000
Bandwidth (MIPS)	2	6	25	132	600	4500	64,000
Latency (ns)	320	313	200	76	50	15	4

Memory module	DRAM	Page mode DRAM	Fast page mode DRAM	Fast page mode DRAM	Synchronous DRAM	Double data rate SDRAM	DDR4 SDRAM
Module width (bits)	16	16	32	64	64	64	64
Year	1980	1983	1986	1993	1997	2000	2016
Mbits/DRAM chip	0.06	0.25	1	16	64	256	4096
Die size (mm ²)	35	45	70	130	170	204	50
Pins/DRAM chip	16	16	18	20	54	66	134
Bandwidth (MBytes/s)	13	40	160	267	640	1600	27,000
Latency (ns)	225	170	125	75	62	52	30

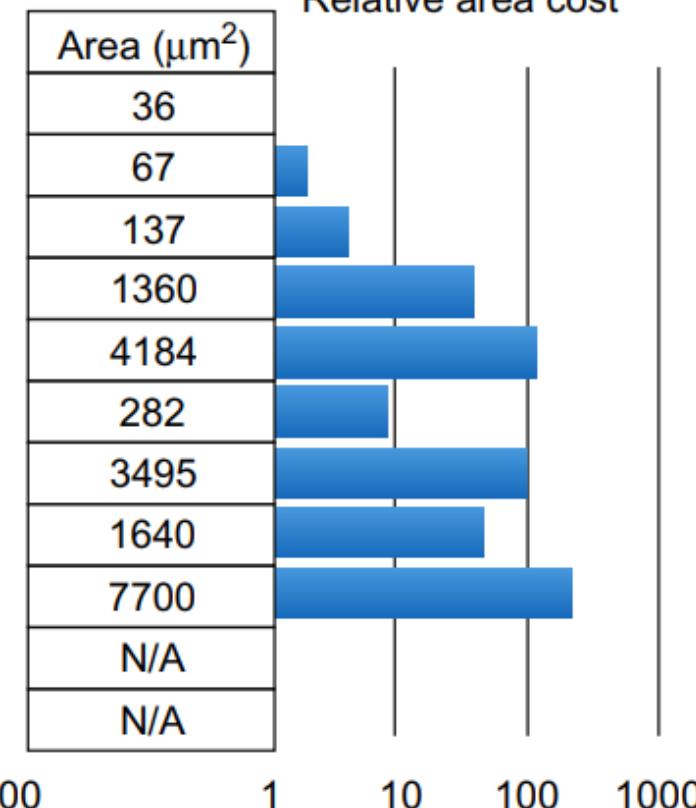
Local area network	Ethernet	Fast Ethernet	Gigabit Ethernet	10 Gigabit Ethernet	100 Gigabit Ethernet	400 Gigabit Ethernet
IEEE standard	802.3	803.3u	802.3ab	802.3ac	802.3ba	802.3bs
Year	1978	1995	1999	2003	2010	2017
Bandwidth (Mbits/seconds)	10	100	1000	10,000	100,000	400,000
Latency (μs)	3000	500	340	190	100	60
Hard disk	3600 RPM	5400 RPM	7200 RPM	10,000 RPM	15,000 RPM	15,000 RPM
Product	CDC WrenI 94145-36	Seagate ST41600	Seagate ST15150	Seagate ST39102	Seagate ST373453	Seagate ST600MX0062
Year	1983	1990	1994	1998	2003	2016
Capacity (GB)	0.03	1.4	4.3	9.1	73.4	600
Disk form factor	5.25 in.	5.25 in.	3.5 in.	3.5 in.	3.5 in.	3.5 in.
Media diameter	5.25 in.	5.25 in.	3.5 in.	3.0 in.	2.5 in.	2.5 in.
Interface	ST-412	SCSI	SCSI	SCSI	SCSI	SAS
Bandwidth (MBytes/s)	0.6	4	9	24	86	250
Latency (ms)	48.3	17.1	12.7	8.8	5.7	3.6

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FB Add	0.4
32b FB Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FB Mult	1.1
32b FB Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640

Relative energy cost



Relative area cost



Energy numbers are from Mark Horowitz *Computing's Energy problem (and what we can do about it)*. ISSCC 2014

Area numbers are from synthesized result using Design compiler under TSMC 45nm tech node. FP units used DesignWare Library.

Figure 1.13 Comparison of the energy and die area of arithmetic operations and energy cost of accesses to SRAM and DRAM. [Azizi][Dally]. Area is for TSMC 45 nm technology node.

CISC ISA

When there is a large number of dedicated hardware co-processors/modules, it needs to use more instructions .

- CISC : Complex Instruction Set Computer
- Consumes more power, because large number of modules are active
- Large number of instructions for dedicated hardware operations
- Difficult to program. Too many ways to choose from!

RISC ISA

When we have a minimal set of hardware we can use a simple set of instructions.

- RISC : Reduced Instruction Set Computer
- Consumes less power, hardware is simple and minimal
- Only a handful of instructions
- Easy to program

MicroArchitecture Design

Some questions

- How many cycles does the fastest instruction take?
- How many cycles does the slowest instruction take?
- Why does a specific instruction take as long as it takes?
- What determines the clock cycle time?

Micro Architecture Design Principles

- Critical Path
 - Find and decrease the maximum combinational logic delay
 - Break a path into multiple cycles if it takes too long
- Common Case vs Uncommon Case
 - Spend time and resources on where it matters most
 - No point in putting much effort to improve rarely used instructions
- Eliminate the bottlenecks

Single Cycle vs Multi-Cycle microarchitectures

- Some instructions are fast, some are slow.
 - Example: ADDI instruction is fast. LOAD/ STORE instructions are slow
- Single cycle processors need to adjust to the slowest stage.
- Idea
 - Determine clock cycle time independently of instruction processing time
 - Each instruction takes as many clock cycles as it needs to take
 - Multiple state transitions per instruction
 - The states followed by each instruction is different
- Example:
 - Load Instruction may take 8 clock cycles (Or even an undetermined time period)
 - ADDI instruction takes only 1 clock cycle

Programmable Control Unit

- Three components:
 - Microinstruction, control store, micro-sequencer
 - **Microinstruction**: control signals that control the datapath and help determine the next state
 - Each microinstruction is stored in a unique location in the **control store** (a special memory structure)
 - **Micro-sequencer** determines the address of the next microinstruction
 - Who is programming the microinstruction sequences?
 - Processor manufacturers themselves usually do the micro-coding
 - Processor “firmware”

microprogramming

- The concept of a control store of microinstructions enables the hardware designer with a new abstraction:
microprogramming
- The designer can translate any desired operation to a sequence of microinstructions
- All the designer needs to provide is
 - The sequence of microinstructions needed to implement the desired operation
 - The ability for the control logic to correctly sequence through the microinstructions
 - “Soft” and dynamic datapath control signals (no need of additional hardware control signals if the operation can be “translated” into existing control signals)

- Drawbacks of micro-coded architectures
 - Complex Instruction Set (Leading to CISC)
 - High Power Consumption
 - General Purpose optimizations, only the hardware vendor can re-purpose the instructions
 - High-level language compilers might not use the optimizations available in the ISA
- The other end?

Domain specific Architectures

- Use dedicated memories to minimize the distance over which data is moved
- Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories
- Use the easiest form of parallelism that matches the domain
- Reduce data size and type to the simplest needed for the domain
- Use a domain-specific programming language to port code to the DSA

- Shared Memory and Cache Hierarchy
- Performance measurements
 - Benchmarking How to compare two processors/systems
 - Read

Performance Equation

- CPU time = CPU clock cycles for a program x Clock cycle time
- Clock cycles per instruction
 - CPI = CPU clock cycles for a program / Instruction count
 - Instructions per clock (IPC) = 1/ CPI
- CPU time = Instruction count x Cycles per instruction x Clock cycle time

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

- *Clock cycle time*—Hardware technology and organization
- *CPI*—Organization and instruction set architecture
- *Instruction count*—Instruction set architecture and compiler technology

- Benchmarks

- Requirements change!
 - with time
 - type of device

	SPEC2017	SPEC2006	Benchmark name by SPEC generation			
			SPEC2000	SPEC95	SPEC92	SPEC89
GNU C compiler						gcc
Perl interpreter				perl		espresso
Route planning		mcf				li
General data compression	XZ		bzip2			eqntott
Discrete Event simulation - computer network		omnetpp	vortex	go		
XML to HTML conversion via XSLT		xalancbmk		jpeg	compress	
Video compression	X264	h264ref	gzip	sc		
Artificial Intelligence: alpha-beta tree search (Chess)	deepsjeng	sjeng	eon			
Artificial Intelligence: Monte Carlo tree search (Go)	leela	gobmk	twolf			
Artificial Intelligence: recursive solution generator (Sudoku)	exchange2	astar	vortex			
		hmmer	vpr			
		libquantum	crafty			
			parser			
Explosion modeling		bwaves				fppp
Physics: relativity		cactuBSSN				tomcatv
Molecular dynamics		namd				doduc
Ray tracing		povray				nasa7
Fluid dynamics		lbm				spice
Weather forecasting		wrf				matrix300
Biomedical imaging: optical tomography with finite elements	parest	gamess				
3D rendering and animation	blender					
Atmosphere modeling	cam4	milc	wupwise	apsi		
Image manipulation	imagick	zeusmp	apply	mgrid		
Molecular dynamics	nab	gromacs	galgel	applu		
Computational Electromagnetics	fotonik3d	leslie3d	mesa	turb3d		
Regional ocean modeling	roms	dealII	art			
		soplex	equake			
		calculix	facerec			
		GemsFDTD	ammp			
		tonto	lucas			
		sphinx3	fma3d			
			sixtrack			

Benchmarks	Sun Ultra Enterprise 2 time (seconds)	AMD A10- 6800K time (seconds)	SPEC 2006Cint ratio	Intel Xeon E5-2690 time (seconds)	SPEC 2006Cint ratio	AMD/Intel times (seconds)	Intel/AMD SPEC ratios
perlbench	9770	401	24.36	261	37.43	1.54	1.54
bzip2	9650	505	19.11	422	22.87	1.20	1.20
gcc	8050	490	16.43	227	35.46	2.16	2.16
mcf	9120	249	36.63	153	59.61	1.63	1.63
gobmk	10,490	418	25.10	382	27.46	1.09	1.09
hmmer	9330	182	51.26	120	77.75	1.52	1.52
sjeng	12,100	517	23.40	383	31.59	1.35	1.35
libquantum	20,720	84	246.08	3	7295.77	29.65	29.65
h264ref	22,130	611	36.22	425	52.07	1.44	1.44
omnetpp	6250	313	19.97	153	40.85	2.05	2.05
astar	7020	303	23.17	209	33.59	1.45	1.45
xalancbmk	6900	215	32.09	98	70.41	2.19	2.19
Geometric mean			31.91		63.72	2.00	2.00

Figure 1.19 SPEC2006Cint execution times (in seconds) for the Sun Ultra 5—the reference computer of SPEC2006—and execution times and SPECRatios for the AMD A10 and Intel Xeon E5-2690. The final two columns show the ratios of execution times and SPEC ratios. This figure demonstrates the irrelevance of the reference computer in relative performance. The ratio of the execution times is identical to the ratio of the SPEC ratios, and the ratio of the geometric means ($63.72/31.91 = 2.00$) is identical to the geometric mean of the ratios (2.00). Section 1.11 discusses libquantum, whose performance is orders of magnitude higher than the other SPEC benchmarks.

	System 1	System 2		System 3	
Component	Cost (% Cost)	Cost (% Cost)		Cost (% Cost)	
Base server	PowerEdge R710	\$653 (7%)	PowerEdge R815	\$1437 (15%)	PowerEdge R815
Power supply	570 W		1100 W		1100 W
Processor	Xeon X5670	\$3738 (40%)	Opteron 6174	\$2679 (29%)	Opteron 6174
Clock rate	2.93 GHz		2.20 GHz		2.20 GHz
Total cores	12		24		48
Sockets	2		2		4
Cores/socket	6		12		12
DRAM	12 GB	\$484 (5%)	16 GB	\$693 (7%)	32 GB
Ethernet Inter.	Dual 1-Gbit	\$199 (2%)	Dual 1-Gbit	\$199 (2%)	Dual 1-Gbit
Disk	50 GB SSD	\$1279 (14%)	50 GB SSD	\$1279 (14%)	50 GB SSD
Windows OS		\$2999 (32%)		\$2999 (33%)	
Total		\$9352 (100%)		\$9286 (100%)	\$12,658 (100%)
Max ssj_ops	910,978		926,676		1,840,450
Max ssj_ops/\$	97		100		145

Figure 1.20 Three Dell PowerEdge servers being measured and their prices as of July 2016. We calculated the cost

Performance is Domain Specific

- Examples:
 - Neural Networks
 - Security /Networking
 - Video Encoding. Decoding Etc.

Programming Models for Various Computer Systems

Computer Architecture

Department of Computer Science and Engineering

Programming Models

- A sequence of instructions
 - Branching?
- Single program
 - Branching
 - Sub-routines / Function calls ?
- Interruptable programs?
 - Separate Interrupt service routine(s).
 - Processor's ability to perform a context switch

Programming Models...

- Multi-threaded programs
 - **Memory Safety?**
 - Multiple programs originated from different sources
- Multi-core programs
 - Homogenous multi-core processors
 - How to write programs for multi core processors?
- Heterogenous systems
 - GPU-CPU co-processing
- System On Chip (SoC) architectures
- Virtualization architectures
 - **Memory Safety?**
 - Multiple virtual machines running on same shared hardware

Memory Safety

- Memory safety is based on Virtual Memory Page System
 - What happens if a program tries to access a memory location of another program?
 - If its for communication purpose, and coordinated well, it could be useful
 - However, it could be intentionally or unintentionally harmful as well
 - Corrupt the programs
 - Tamper with data
 - Who will manage the memory safety? And How?
 - Program is a collection of instructions
 - We have Load Store instructions
 - Somebody should avoid using unauthorized memory regions

Memory segmentation and paging (Virtual Memory)

- Caching mechanism between Main memory and hard disk
- **Doubles as a security mechanism**

Memory segmentation and paging (Virtual Memory)

- Caching mechanism between Main memory and hard disk
- **Doubles as a security mechanism**

Figure 8.20 Virtual and physical pages

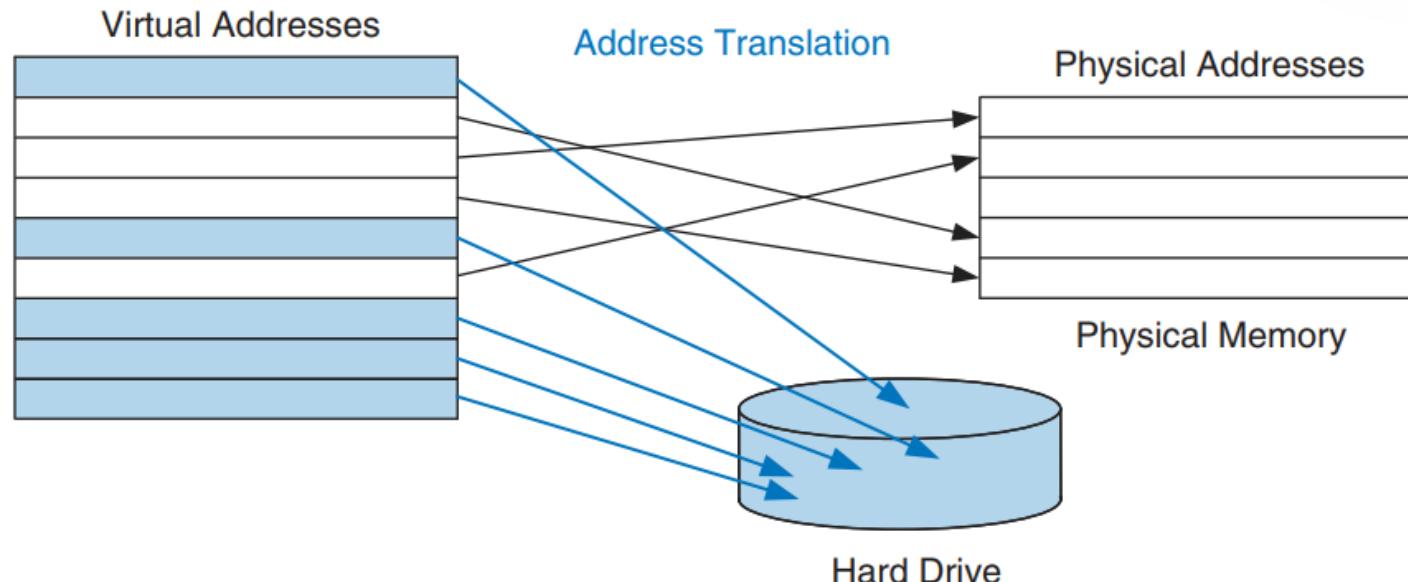


Table 8.4 Analogous cache and virtual memory terms

Cache	Virtual Memory
Block	Page
Block size	Page size
Block offset	Page offset
Miss	Page fault
Tag	Virtual page number

As you probably know, modern computers typically run several programs or *processes* at the same time. All of the programs are simultaneously present in physical memory. In a well-designed computer system, the programs should be protected from each other so that no program can crash or hijack another program. Specifically, no program should be able to access another program's memory without permission. This is called *memory protection*.

Virtual memory systems provide memory protection by giving each program its own *virtual address space*. Each program can use as much memory as it wants in that virtual address space, but only a portion of the virtual address space is in physical memory at any given time. Each program can use its entire virtual address space without having to worry about where other programs are physically located. However, a program can access only those physical pages that are mapped in its page table. In this way, a program cannot accidentally or maliciously access another program's physical pages, because they are not mapped in its page table. In some cases, multiple programs access common instructions or data. The operating system adds control bits to each page table entry to determine which programs, if any, can write to the shared physical pages.

- <https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions>
- <https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf>

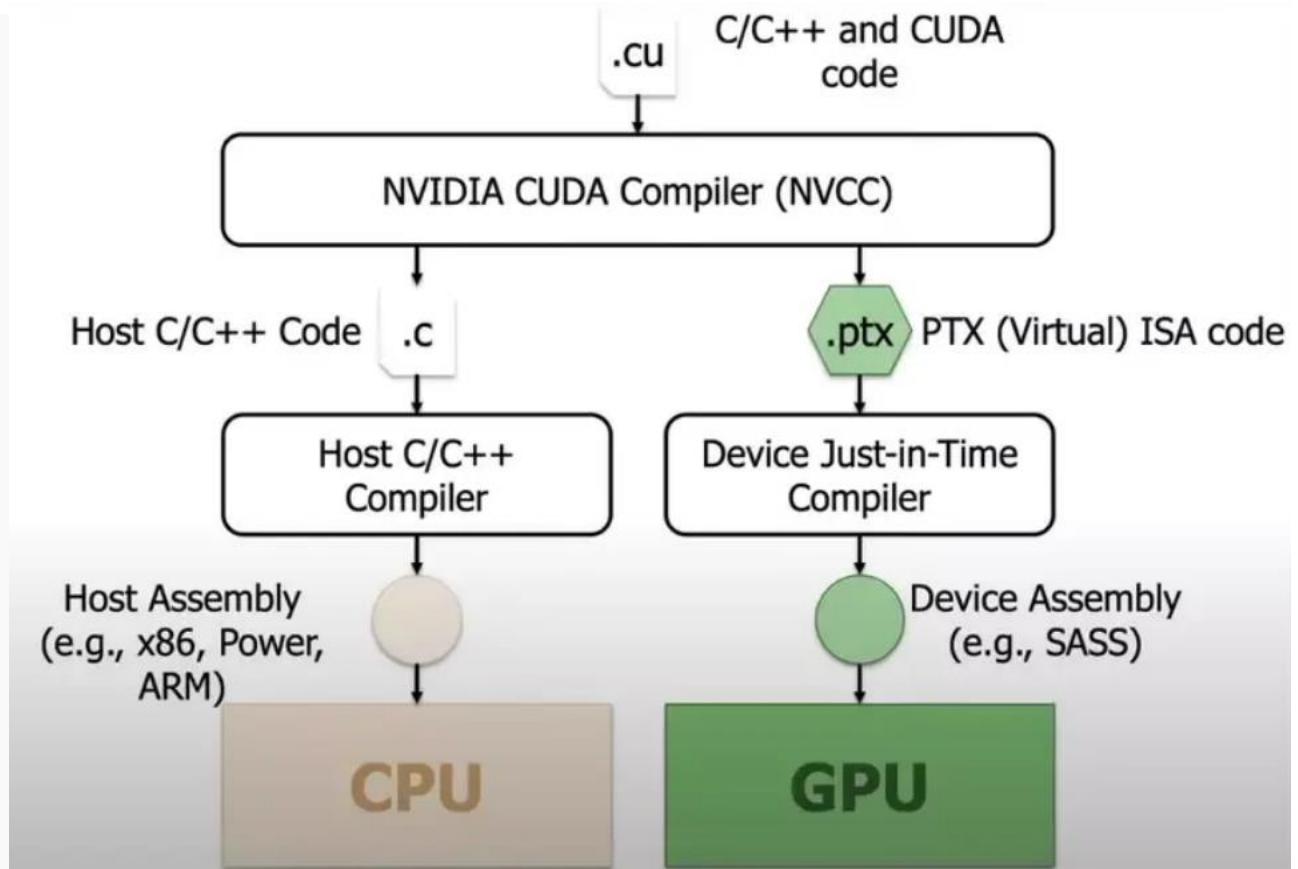
Kernels

```
#include <stdio.h>

__global__ void kernel()
{
    printf("hello world");
}

int main()
{
    kernel<<<1,1>>>();
    cudaDeviceSynchronize();

    return 0;
}
```



- What are the programming models for System On Chip Designs?