

*
$$\text{CPU Time} = \frac{\text{CPU clock Cycles} \times \text{Clock CycleTime}}{\text{Clock Period}}$$

$$= \frac{\text{CPU Clock cycles}}{\text{Clock Rate}}$$

*
$$\text{CPU clock Cycles} = \underline{IC} \times CPI$$

IC \rightarrow Instruction Count

:
$$\text{CPU Time} = IC \times CPI \times \text{Clock Period}$$

CPI \rightarrow Cycles Per Instruction
 (It's always used as average.)

$$= \frac{IC \times CPI}{\text{clock rate}}$$

* Hz \rightarrow Clock Rate

Sec \rightarrow Clock Cycle Time / Clock Period.

Sec \rightarrow CPU Time.

CPI, IC \rightarrow No unit (Just Numbers)

* $1 \text{ GHz} = 10^9 \text{ Hz}$.



Chapter 1

Computer Abstractions and Technology

The Computer Revolution

- Progress in computer technology
 - Underpinned by **Moore's Law**
 - Moore's law predicted that this trend will continue into the foreseeable future.
 - At least it has been the case for about 50 years

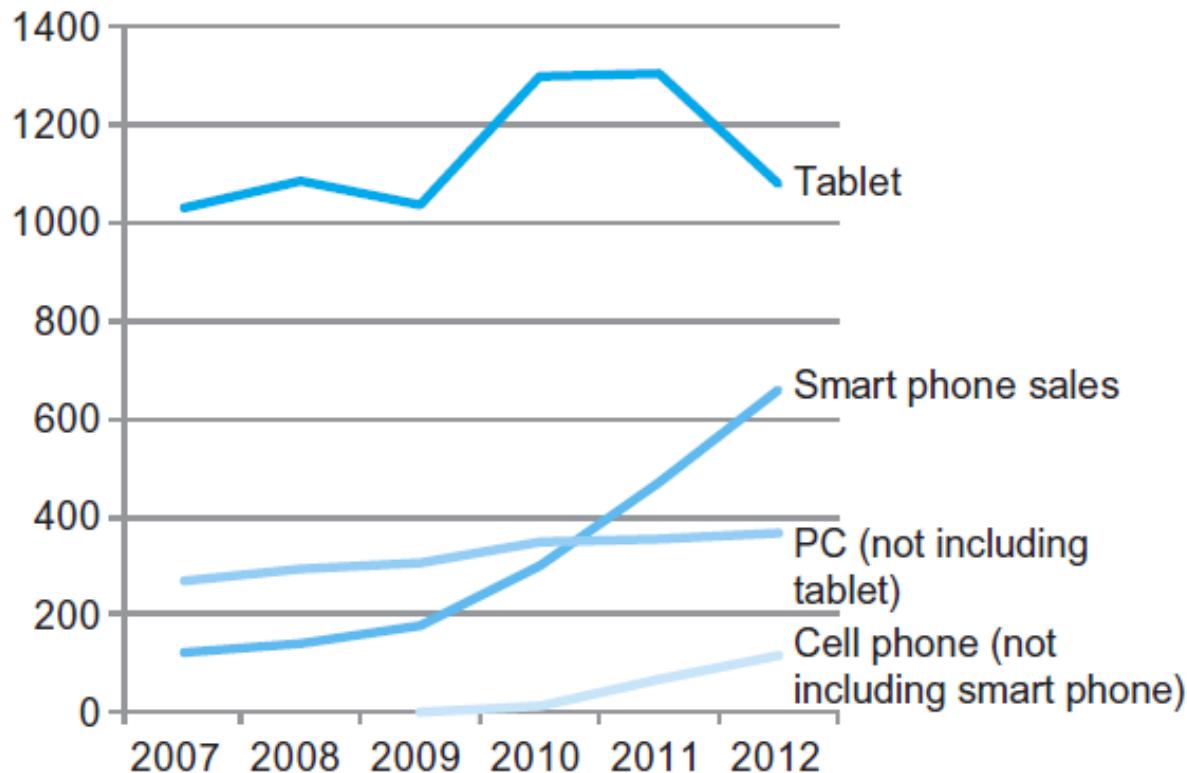
The Computer Revolution

- This progress has made novel applications feasible
 - Computers in automobiles
 - Cell phones
 - Human genome project
 - World Wide Web
 - Search Engines
- Computers are pervasive

Classes of Computers

- Personal computers
- Server computers
- Supercomputers
- Embedded computers

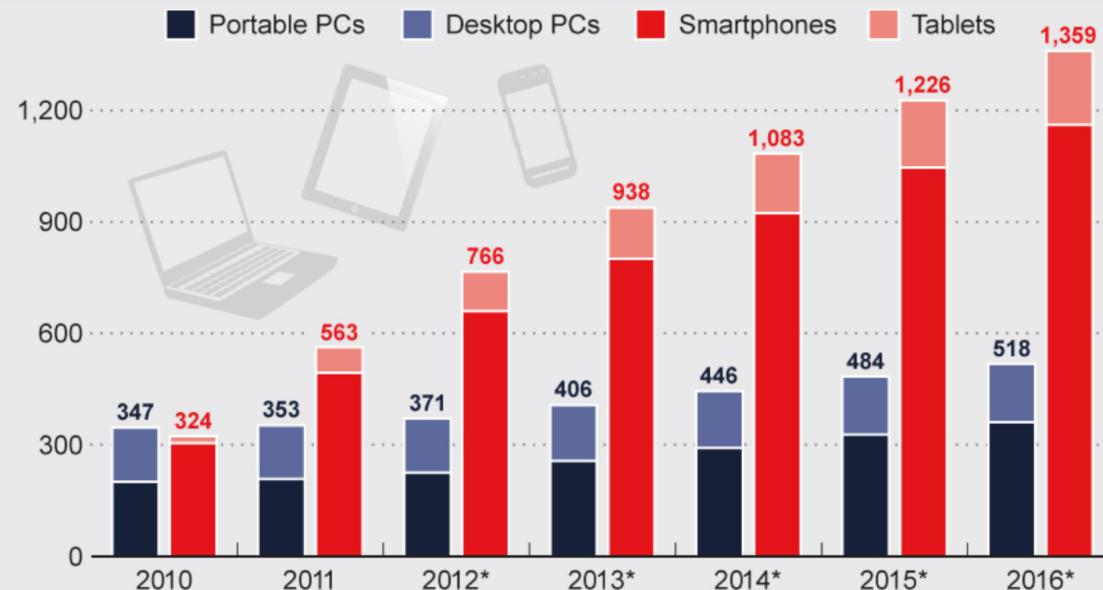
The PostPC Era



The PostPC Era

The Post-PC Era Has Arrived

Global smartphone, tablet and PC shipments (in millions)



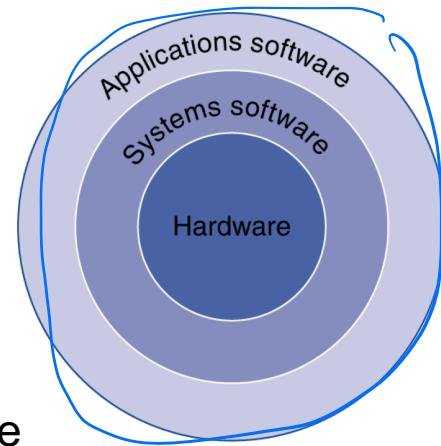
Understanding Performance

- What determines program performance?
 - And how can it be improved?
- Programs are translated into the machine language
 - And the hardware executes them
- So both HW and SW influence the performance
 - As well as the stuff interfacing them



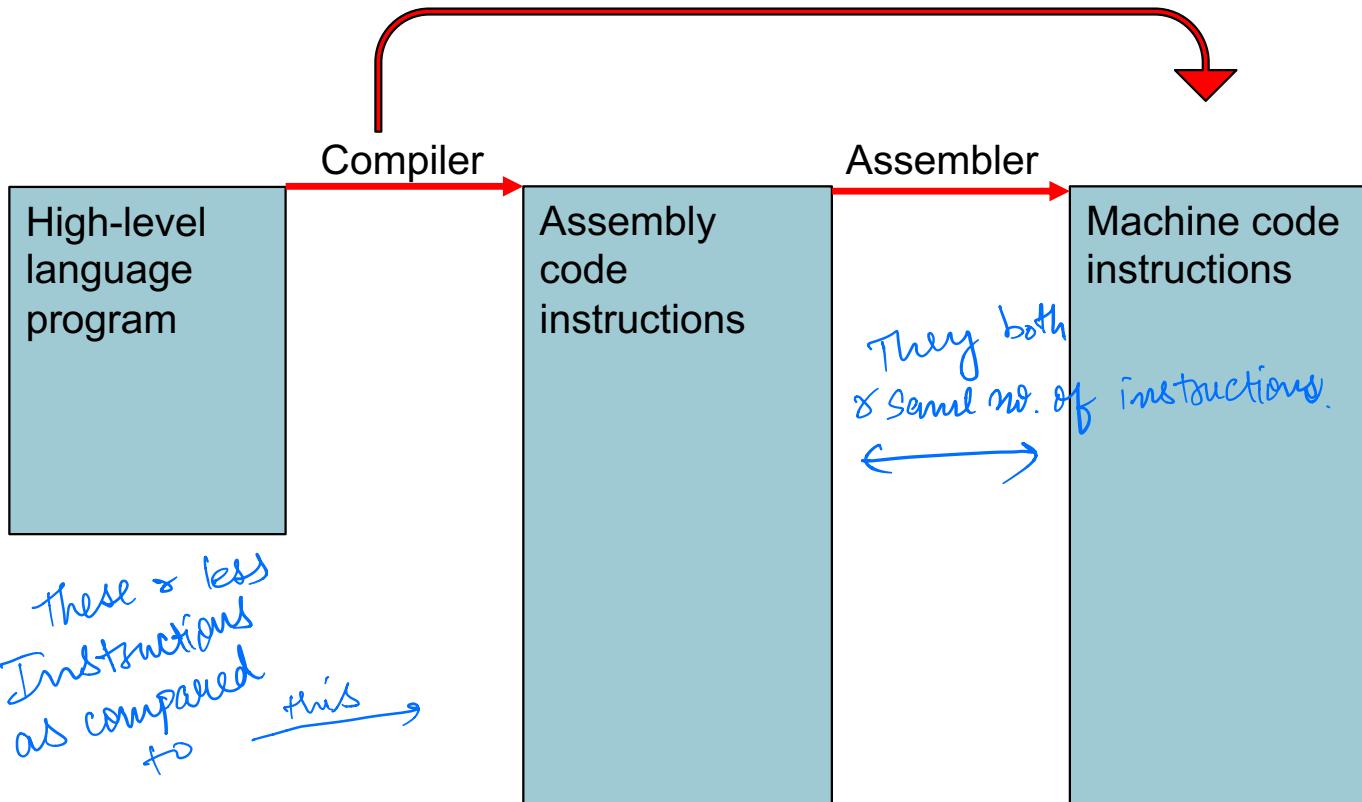
Understanding Performance

- Application software
 - Written in high-level language
- System software
 - Compiler: translates HLL code to machine code
 - Or HHL-> assembly -> machine code
 - Operating System: service code
 - Handling input/output
 - Managing memory and storage
 - Scheduling tasks & sharing resources
- Hardware
 - Processor, memory, I/O controllers

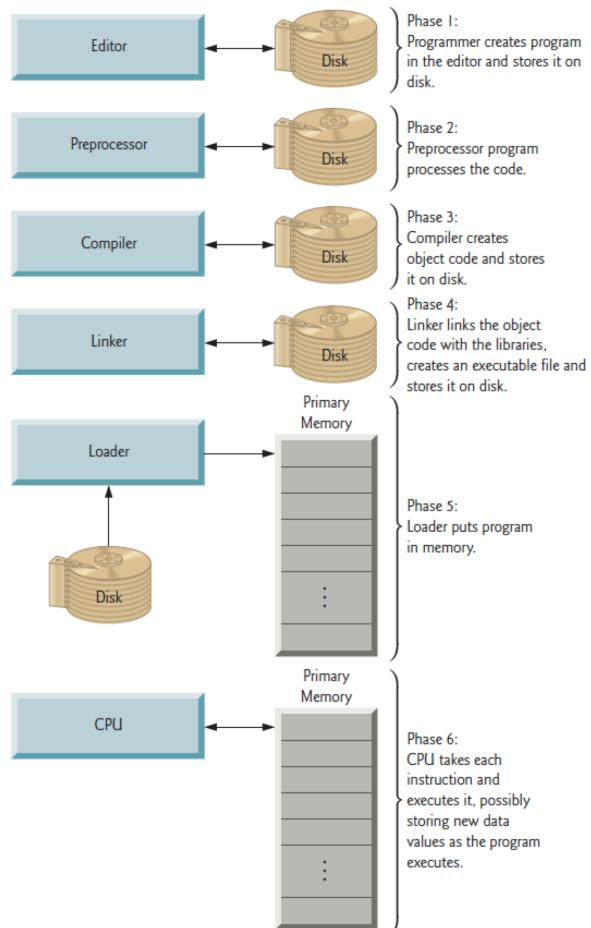


Performance
depends on these
3 things.

Understanding Performance



Understanding Performance



Understanding Performance

- Algorithm → No. of operations
- Programming language, compiler, architecture
 - ↳ ① No. of machine instruct.^u executed per oper.^u
 - ② Clock Cycle
- Processor and memory system
 - How fast the instructions are executed
 - OR
 - Clock Rate
- I/O system (including OS)
 - ↳ How fast is communication btw. CPU & I/O devices.

Understanding Performance

Software

Program Requirements



Algorithm



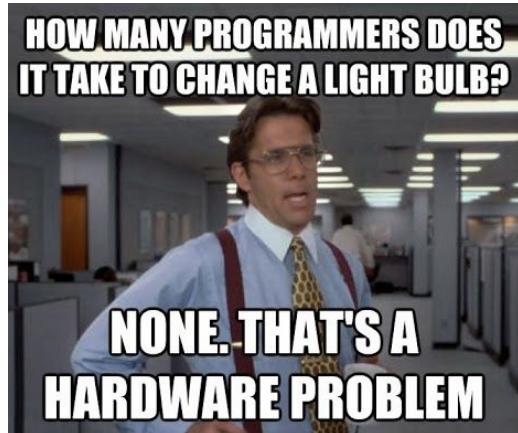
Programming language used



Operations to be executed

Compiler

→ Total of machine Instructions



Hardware

Computer Architecture
(ISA)



Instruction Set Arch!



→ Total of machine Instructions



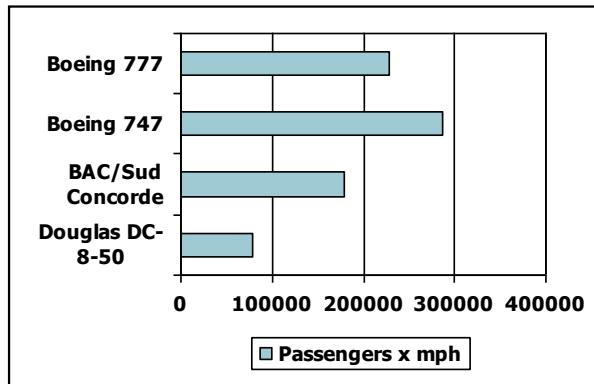
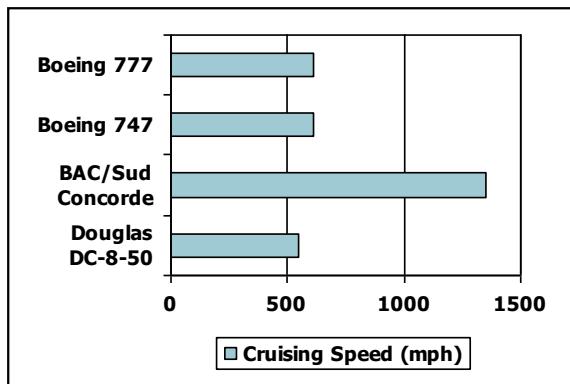
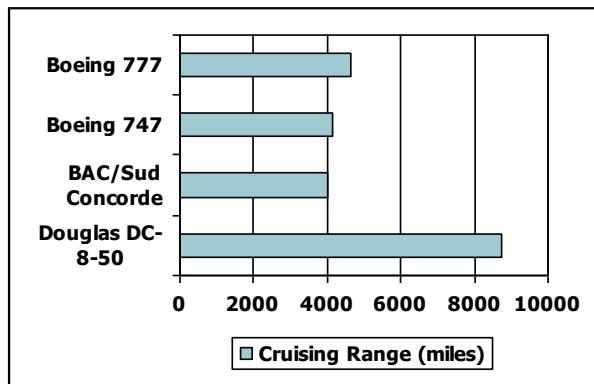
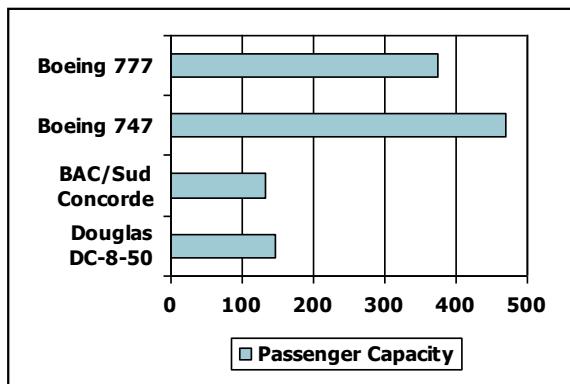
Number of clock cycles to execute each instruction



How fast and efficient are the memory and IO

Defining Performance

- Which airplane has the best performance?



Relative Performance

- Define Performance = 1/Execution Time
- Relative performance:

$$\begin{aligned} & \text{Performance}_X / \text{Performance}_Y \\ &= \text{Execution time}_Y / \text{Execution time}_X = n \end{aligned}$$

- “X is n time faster than Y”
- Example: time taken to run a program
 - 10s on computer A, 15s on computer B
 - $\text{Execution Time}_B / \text{Execution Time}_A$
 $= 15s / 10s = 1.5$
 - So computer A is 1.5 times faster than computer B



Relative Performance - SPEC

Intel It

Standard Performance Evaluation Corporation

Description	Name	Instruction Count x 10 ⁹	CPI	Clock cycle time (seconds x 10 ⁻⁹)	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	-	-	-	-	-	-	25.7

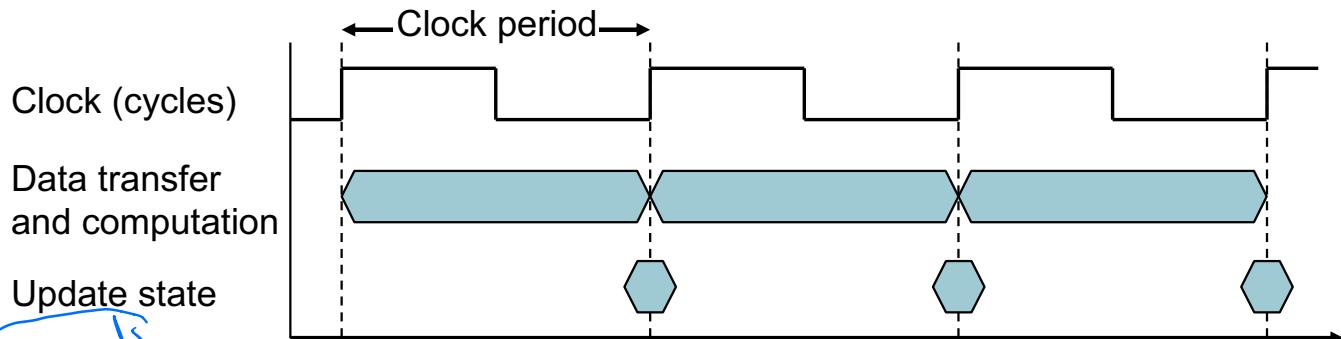
Jithna Jader Ratio
toga utna better koi.

Smaller this → Bigger this

∴ Better

CPU Clocking

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



■ Clock period: duration of a clock cycle

$$\text{e.g., } 250\text{ps} = 0.25\text{ns} = 250 \times 10^{-12}\text{s}$$

■ Clock frequency (rate): cycles per second

$$\text{e.g., } 4.0\text{GHz} = 4000\text{MHz} = 4.0 \times 10^9\text{Hz} (\#cycles/s)$$

Measuring Execution Time

- Elapsed time
 - Total response time, including all aspects
 - Processing, I/O, OS overhead, idle time
 - Determines system performance
- CPU time
 - Time spent processing a given job
 - Discounts I/O time, other jobs' shares
 - Determines CPU performance
- Different programs are affected differently by CPU and system performance

Relative Performance - SPEC

cycles per Inst.

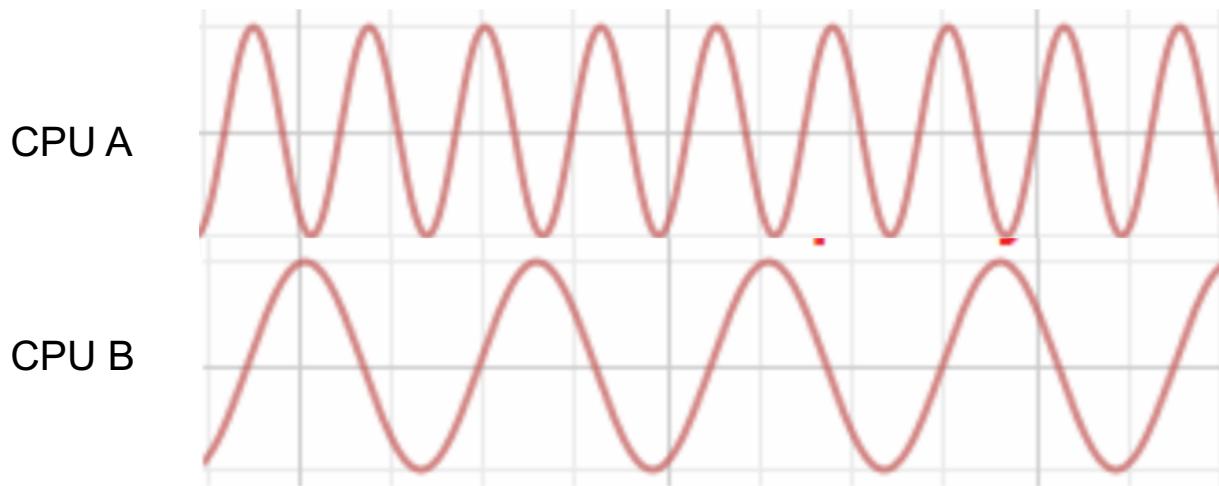
Description	Name	Instruction Count x 10 ⁹	CPI	Clock cycle time (seconds x 10 ⁻⁹)	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	-	-	-	-	-	-	25.7

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}

(Clock Period)



CPU Time

CPU Time = CPU Clock Cycles × Clock Cycle Time

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

■ How to lower CPU time to improve performance?

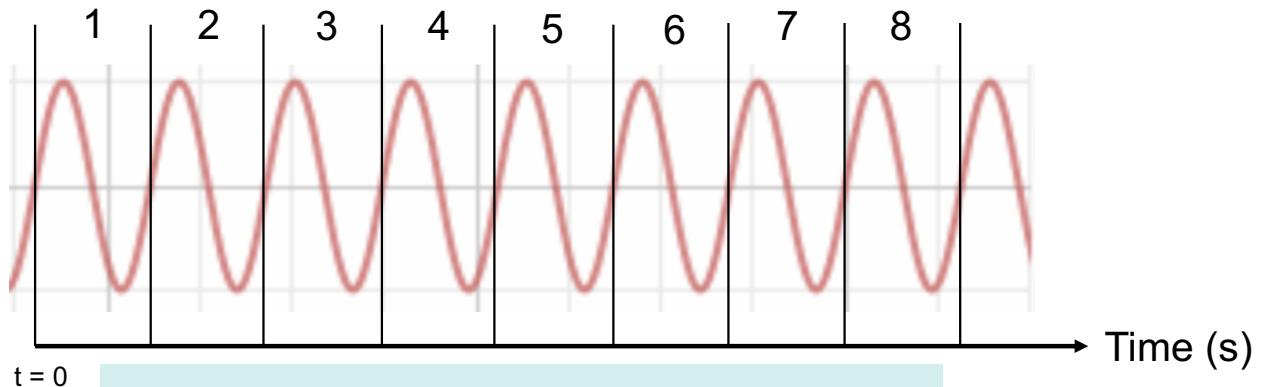
1. Reduce number of clock cycles
 - In effect by reducing clock cycles per instruction
2. Increasing clock rate (in effect, reducing clock cycle time or period)
3. Or both 1 & 2 (best case)



Instruction Count and CPI

CPU Time = **CPU Clock Cycles** × Clock Cycle Time

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$



Q2) Instruction
Count (IC) = ? 8

Q3) CPI = ? 1

```
SUB32 PROC      ; procedure begins here
        CMP AX,97    ; compare AX to 97
        JL  DONE      ; if less, jump to DONE
        CMP AX,122    ; compare AX to 122
        JG  DONE      ; if greater, jump to DONE
        SUB AX,32     ; subtract 32 from AX
DONE:   RET       ; return to main program
SUB32 ENDP      ; procedure ends here
```

Q1) Assuming my CPU can execute every instruction in 1 cycle then what is the total CPU clock cycles here to execute the entire program? 8

Instruction Count and CPI

Our original Total CPU Time definition

$$\text{CPU Time} = \boxed{\text{CPU Clock Cycles}} \times \text{Clock Cycle Time}$$
$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

Clock cycles can be replaced with

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

$$\checkmark \quad \text{CPU Time} = \text{Instruction Count} \times \boxed{\text{CPI}} \times \boxed{\text{Clock Cycle Time}}$$
$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

Our new Total CPU Time definition in terms of IC & CPI

Instruction Count and CPI

CPU Time = Instruction Count \times CPI \times Clock Cycle Time

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

```
SUB32 PROC      ; procedure begins here
        CMP AX,97    ; compare AX to 97
        JL  DONE      ; if less, jump to DONE
        CMP AX,122    ; compare AX to 122
        JG  DONE      ; if greater, jump to DONE
        SUB AX,32     ; subtract 32 from AX
DONE:   RET       ; return to main program
SUB32 ENDP      ; procedure ends here
```

- Running on a 1GHz computer (Clock rate = 1GHz) and
- Average CPI = 2

CPU Time (seconds) = (8 instructions \times 2 CPI) / 1000000000 Hz = 0.000000016 s

CPU Time (seconds) = 8 instructions \times 2 CPI \times 0.000000001 s = 0.000000016 s = 16 ns



Instruction Count and CPI

CPU Time = Instruction Count \times CPI \times Clock Cycle Time

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

IC

■ Instruction Count for a program influenced by

- Algorithm, programming language, ISA and compiler

Instruction set arch.^T

CPI

■ Cycles per instruction influenced by

- CPU hardware

■ What if different instructions have different CPI?

- We always use the average CPI of the instruction set



CPI Example

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

CPU Time = Instruction Count \times CPI \times Clock Cycle Time



CPI Example

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

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\ &= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps} \end{aligned}$$

A is faster...

$$\begin{aligned}\text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\ &= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps} \end{aligned}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

...by this much



Average CPI

- If different instruction classes take different numbers of cycles

```
SUB32 PROC      ; procedure begins here
    CMP AX,97   ; compare AX to 97
    JL  DONE    ; if less, jump to DONE
    CMP AX,122  ; compare AX to 122
    JG  DONE    ; if greater, jump to DONE
    SUB AX,32   ; subtract 32 from AX
    DONE: RET    ; return to main program
    SUB32 ENDP   ; procedure ends here
```

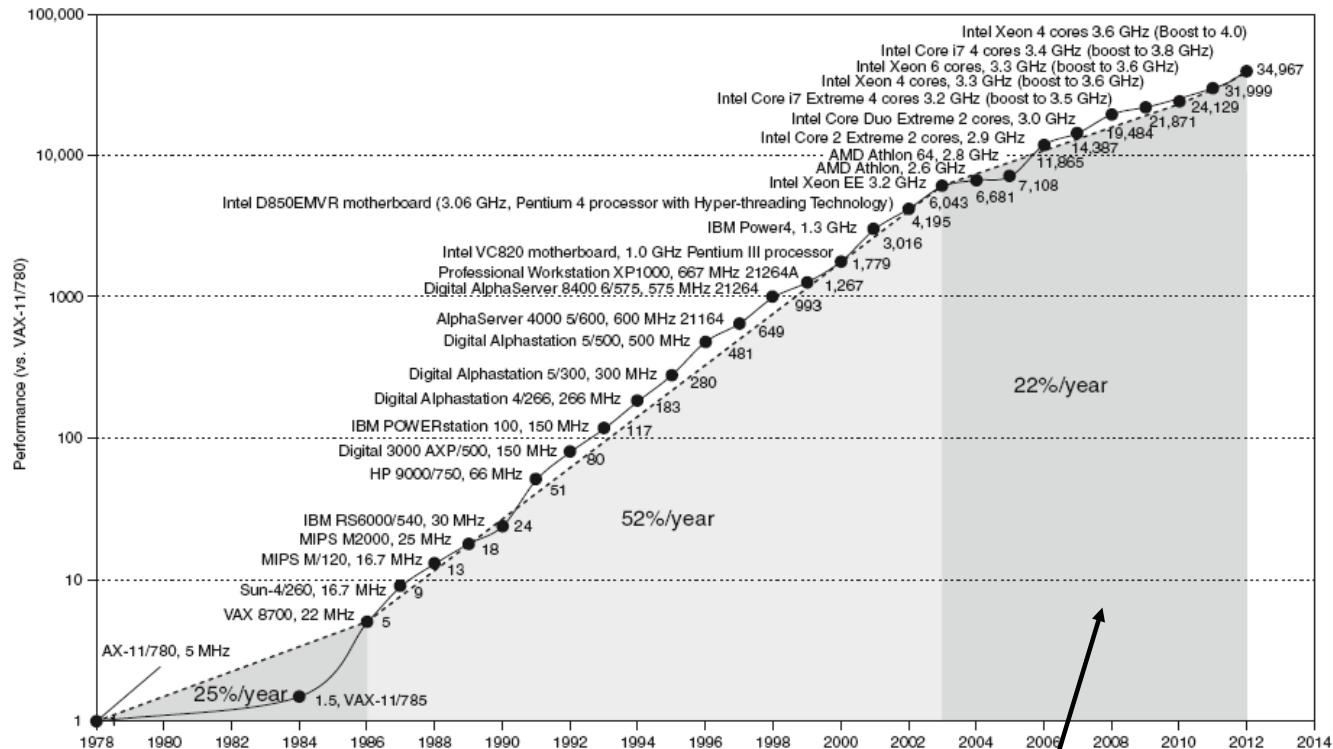
Performance Summary

The BIG Picture

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

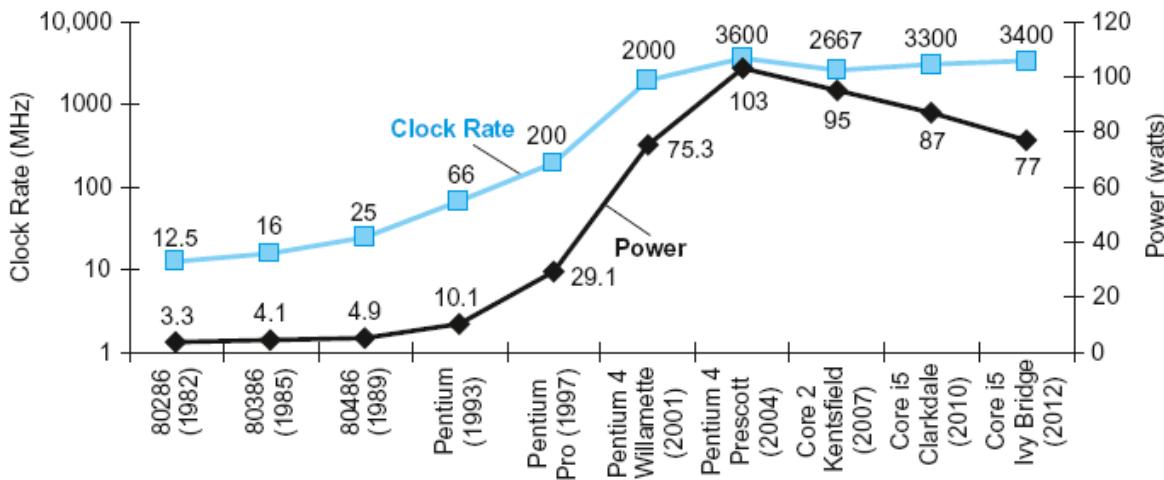
- 
- Performance depends on
 - Algorithm: affects IC, CPI
 - Programming language: affects IC, CPI
 - Compiler: affects IC, CPI
 - ISA: affects IC, CPI, T_c

Uniprocessor Performance



Constrained by power, instruction-level parallelism,
memory latency

Power Trends



- In CMOS IC technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

× 30

5V → 1V

× 1000

Reducing Power

- Suppose a new CPU has
 - 85% of capacitive load of old CPU
 - 15% voltage and 15% frequency reduction

$$\frac{P_{\text{new}}}{P_{\text{old}}} = \frac{C_{\text{old}} \times 0.85 \times (V_{\text{old}} \times 0.85)^2 \times F_{\text{old}} \times 0.85}{C_{\text{old}} \times V_{\text{old}}^2 \times F_{\text{old}}} = 0.85^4 = 0.52$$

- The power wall
 - We can't reduce voltage further
 - We can't remove more heat
- Parallel processing has come to help.

Multiprocessors

- How hardware designers improve performance
 - What is parallel processing?



Multiprocessors

- Multicore microprocessors
 - More than one processor per chip
- Requires explicitly parallel programming performed by the programmer
 - In contrast with instruction level parallelism (or pipelining performed by the CPU)
 - Hardware executes multiple instructions at once
 - Hidden from the programmer
 - Hard to do
 - Programming for performance
 - Load balancing
 - Optimizing communication and synchronization



Eight Great Ideas

1. Design for ***Moore's Law***
2. Use ***abstraction*** to simplify design
3. Make the ***common case fast***
4. Performance via ***parallelism***
5. Performance via ***pipelining***
6. Performance via ***prediction***
7. ***Hierarchy*** of memories
8. ***Dependability*** via redundancy



MOORE'S LAW



ABSTRACTION



COMMON CASE FAST



PARALLELISM



PIPELINING



PREDICTION



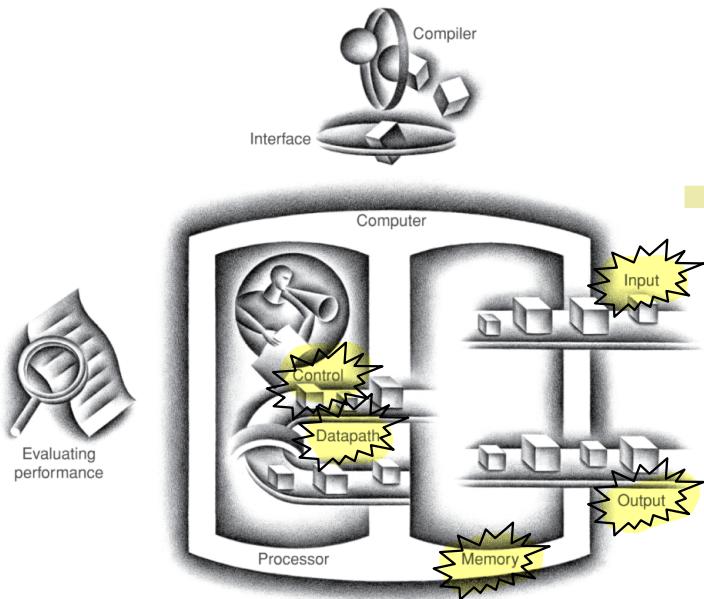
HIERARCHY



DEPENDABILITY

Components of a Computer

The BIG Picture



- Same components for all kinds of computer
 - Desktop, server, embedded
- Input/output includes
 - User-interface devices
 - Display, keyboard, mouse
 - Storage devices
 - Hard disk, CD/DVD, flash
 - Network adapters
 - For communicating with other computers

Concluding Remarks

- Cost/performance is improving
 - Due to underlying technology development
- Instruction set architecture acts as the hardware/software interface
 - A great influencer on the performance
- CPU Execution time: the best performance measure
- Power is a limiting factor
 - Use parallelism to improve performance

Logical Shift Left $\rightarrow 2^m$

g: $\rightarrow [x_1, x_2, LSL \# 3]$

$$x_1 + x_2 \ll 3$$

$\begin{array}{r} 80 \\ \times 2 \\ \hline 160 \end{array}$ $\Rightarrow 80 \times 2^3$ se multiply kar do in x_2 value.

We use LDUR instead of LDR
(unscaled) (scaled)

In Legv8 each register is 8 bytes.



Chapter 2

Instructions: Language of the Computer

Instruction Sets

(most compiler these days can
do these in 1 step)

1. High-level language

- Level of abstraction closer to problem domain
 - Provides for productivity and portability

2. Assembly language

- Textual representation of instructions

3. Hardware representation

- Binary digits (bits)
 - Encoded instructions and data
 - What is ARM instruction size in bits?

High-level
language
program
(in C)

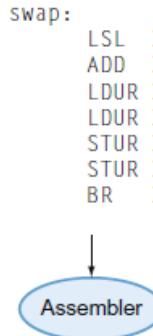
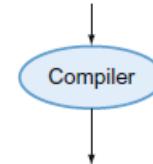
```
swap(int v[], int k)
{int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Assembly language program (for ARMv8)

```
swap:  
    LSL  X10, X1,3  
    ADD  X10, X0,X10  
    LDUR X9, [X10,0]  
    LDUR X11,[X10,8]  
    STUR X11,[X10,0]  
    STUR X9, [X10,8]  
    BR   X10
```

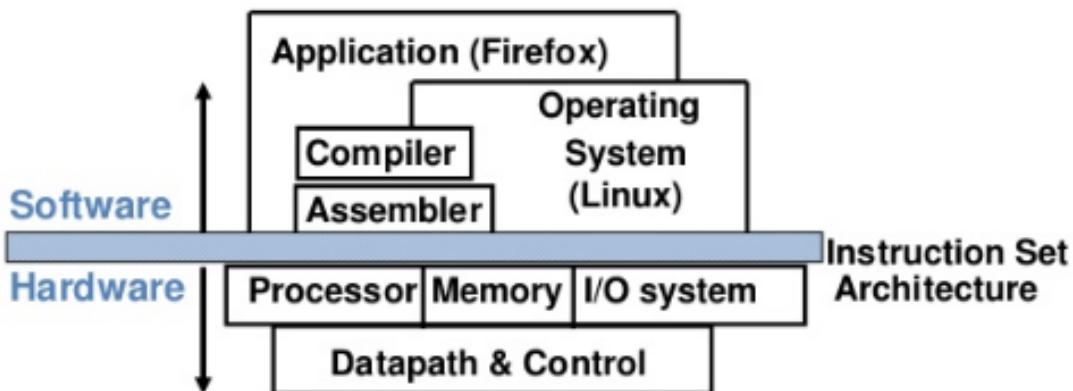
Binary machine
language
program
(for ARMv8)

```
0000000001010001000000000100011000  
000000000100000100001000000100001  
10001101111000100000000000000000000  
10001110000100100000000000000000100  
101011100001001000000000000000000000  
101011011110001000000000000000000000  
00000001111100000000000000000000000000
```



Instruction Sets

- Instruction Set is the full supply of instructions of a computer architecture
- Different computers have different instruction sets (different architectures)
 - But with many aspects in common



Instruction Sets

ISA → Instruction Set arch."

RISC

CISC

Reduced Instruction
Set Computing.

Complex Instruction Set
Computing

- The **instruction set**, also called **instruction set architecture (ISA)**, is part of a computer architecture related to programming.
- The instruction set provides commands to the processor to tell it what it needs to do.
- **CISC-based** and **RISC-based Architecture**
 1. Complex Instruction Set Computing (CISC)
 - X86 (based on the Intel 8086 CPU) → e.g. of CISC
 2. Reduced Instruction Set Computing (RISC)
 - MIPS
 - ARM→ e.g. of RISC
 3. RISC –V
 - A **free and open ISA based on RISC**
 - **Allowing anyone to design, manufacture and sell RISC-V chips or software**

Need to pay
monthly

Instruction Sets

- CISC
 - MULT mem[0] mem[1]
 - Multiply the content of memory location 0 by the content of memory location 1 and store the product in memory location 0
- RISC
 - LOAD R0, mem[0]
 - LOAD R1, mem[1]
 - PROD R0, R1
 - STORE R0, mem[0]
- RISC processors only use simple instructions that can be executed in fewer clock cycles.
 - Thus, the "MULT" command described above was divided into three separate commands: LOAD, PROD and STORE.

ARM Instruction Sets

- ARM, developed by British Co. **ARM Holdings** (originally, Acorn Risk Machine, aka Advanced Risk Machine).
 - ↗ : it uses less power, its used in phones.
- ARM is a family of RISC-based instruction set architecture (ISA) for computer processors
- Companies that make chips that implement an ARM architecture include:
 - *Analog Devices, Apple, AppliedMicro, Atmel, Broadcom, Cypress Semiconductor, Freescale Semiconductor, Nvidia, NXP, Qualcomm, Renesas, Samsung Electronics, ST Microelectronics and TI.*

ARM Instruction Sets

■ ARM Instruction Set (RISC-based)

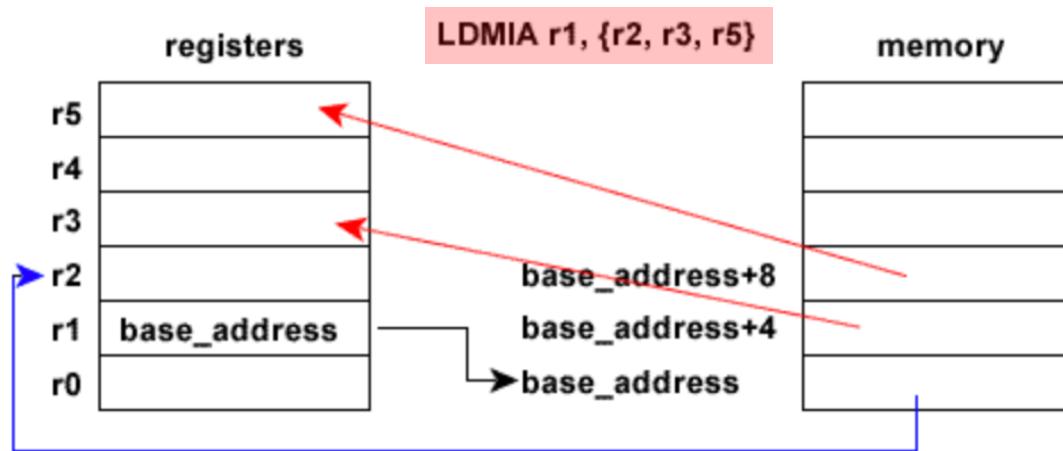
- Require significantly fewer transistors than CISC x86 processors found in most personal computers (Intel, AMD, etc.).
 - Reduces costs, heat and power use. *→ Adv.*
 - Such reductions are desirable traits for embedded systems and smaller devices, etc.
- Instructions are 32 bits wide
- Other RISC-based ISA
 - MIPS version of RISC
- “Load/Store” Architecture
 - ARM processor must first load data into one of the general-purpose registers before processing it

ARM Instruction Sets

- ARM version of RISC focuses not only on performance but also on
 - High code density (provides a 16-bit ISA as well)
 - Low power
 - Small die size
- Achieving it all through modifying RISC rules to also include
 1. Variable-cycle execution for certain instructions,
 2. An inline barrel shifter to preprocess one of the input registers,
 3. Conditional execution,
 4. A compressed 16-bit Thumb instruction set,
 5. And some enhanced DSP instructions.

ARM Instruction Sets

1. Variable-cycle execution for certain instructions



LDMIA r1, {r2, r3, r5}
Load Multiple
Increment After

=

LDR r2, [r1]
LDR r3, [r1, 4]
LDR r5, [r1, 8]

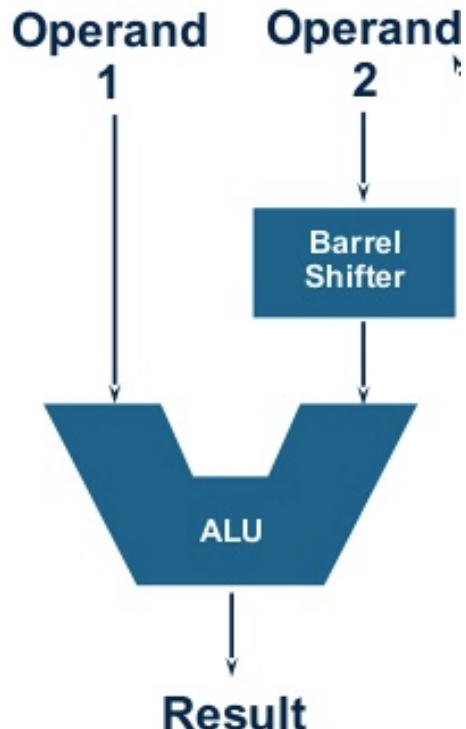
ARM Instruction Sets

2. An inline barrel shifter to preprocess one of the input registers (Operand2).

- MOV r0, r0, LSL #1 Multiply R0 by 2
- MOV r1, r1, LSR #2 Divide R1 by 4 (unsigned)
- MOV r2, r2, ASR #2 " R2 by 4 (Signed)
- MOV r3, r3, ROR #16
- ADD r4, r4, r4, LSL #4 *destination op1 op2*
- RSB r5, r5, r5, LSL #5

L SR → Logical Shift right
AS R → Arithmetic " "

Always happen on Operand 2.



ARM Instruction Sets

3. Conditional execution

- An instruction with a condition code is only executed if the condition code flags in the xPSR meet the specified condition.

Program Status Registers

Normal	Conditional
CMP r3,#0	CMP r3,#0
BEQ skip	ADDNE r0,r1,r2
ADD r0,r1,r2	
skip	

Conditional execution improves code density and performance by reducing the number of forward branch instructions.

- Each data processing instruction prefixed by one of the 16 condition codes



EQ	equal	MI	negative	HI	unsigned higher	GT	signed greater than
NE	not equal	PL	positive or zero	LS	unsigned lower or same	LE	signed less than or equal
CS	unsigned higher or same	VS	overflow	GE	signed greater than or equal	AL	always
CC	unsigned lower	VC	no overflow	LT	signed less than	NV	special purpose

ARM Instruction Sets

4. Compressed 16-bit Thumb instruction set

- All ARM instructions are 32 bits long for easy decoding and pipelining
 - at the cost of decreased code density
- Additional modes allow better code density
 - Thumb (16 bit), Thumb 2 (16/ 32 bit), Jazelle (Byte code)

ARM Instruction Sets



Highest Performance

Supreme performance at optimal power

Real-Time Processing

Reliable mission-critical performance

**Lowest Power,
Lower Cost**

Powering the most energy efficient embedded devices

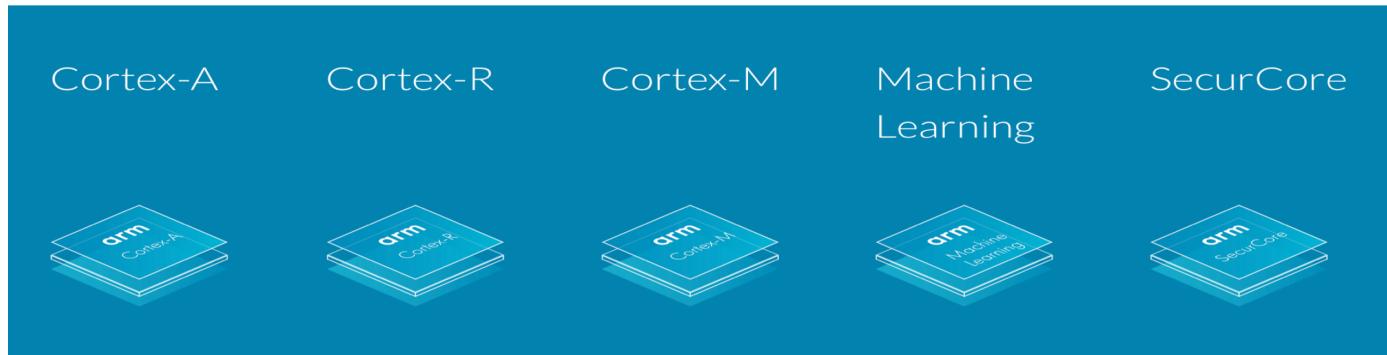
Efficiency Uplift for All Devices

Project Trillium for unmatched versatility and scalability

Tamper Resistant

Powerful solutions for security applications

ARM Instruction Sets



Example use cases:	Example use cases:	Example use cases:	Example use cases:	Example use cases:
Automotive	Automotive	Automotive Energy grid	Artificial intelligence	Advanced payment systems
Industrial	Cameras	Medical	Augmented reality	Electronic passports
Medical	Industrial	Secure embedded applications	Edge computing	SIM
Modem	Medical	Smart cards	Neural network frameworks	Smart cards
Storage		Smart devices	Object detection	
		Sensor fusion	Virtual reality	
		Wearables		

The ARMv8 Instruction Set

A short ARM processors list:

Classic: With DSP, Floating Point, TrustZone and Jazelle extensions. **(2001)**

Cortex-M: ARM Thumb®-2 technology which provides excellent code density. With Thumb-2 technology, the Cortex-M processors support a fundamental base of 16-bit Thumb instructions, extended to include more powerful 32-bit instructions. First Multi-core. **(2004)**

Cortex-R: ARMv7 Deeply pipelined micro-architecture, Flexible Multi-Processor Core (MPCore) configurations: Symmetric Multi-Processing (SMP) & Asymmetric Multi-Processing (AMP), LPAE extension. . **(2011)**

Cortex-A50: ARMv8-A 64bit with load-acquire and store-release features , which are an excellent match for the C++11, C11 and Java memory models. **(2012)**



The ARMv8 Instruction Set

ARMv8 introduced a number of profiles or flavor of the architecture that targets certain classes of workloads.

Profile	Name	Description
Application	ARMv8-A	Optimized for a large class of general applications for mobile, tablets, and servers.
Real-Time	ARMv8-R	Optimized for safety-critical environments.
Microcontroller	ARMv8-M	Optimized for embedded systems with a highly deterministic operation.

- <https://www.youtube.com/watch?v=IfHG7bj-CEI>

Intel is in serious trouble. ARM is the Future.

Dec 21, 2018



The ARMv8 Instruction Set

- A subset ARMv8 instructions, called LEGv8, used as the example throughout the book.
 - ARM
 - Advanced RISC Machine (originally, Acorn RISC Machine)
 - LEG
 - Lesser Extrinsic Garruity (less talk about non-essentials)

and was announced in 2011. For pedagogic reasons, we will use a subset of ARMv8 instructions in this book. We will use the term ARMv8 when talking about the original full instruction set, and LEGv8 when referring to the teaching subset, which is of course based on ARM's ARMv8 instruction set. (LEGv8 is intended as a pun on ARMv8, but it is also a backronym for "Lesser Extrinsic Garruity.") We'll discuss the two in the elaborations. Note that this chapter



Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination
- ADD a, b, c // *a gets b + c*
- All arithmetic operations have this form
- When it comes to ISA design simplicity favors regularity (Design Principle 1)



Arithmetic Example

- C code:

$$f = (g + h) - (i + j);$$

- Compiled LEGv8 code:

- Assuming C variables in memory g, h, i, and j are already loaded in X0, X1, X2, and X3 registers:

```
ADD X9, x0, x1    // temp x9 = g + h
```

```
ADD X10, x2, x3   // temp x10 = i + j
```

```
SUB X19, x9, x10  // x19 = x9 - x10
```

```
STR X19, f        // mem[f] <== (g + h) - (i + j)
```

Register Operands

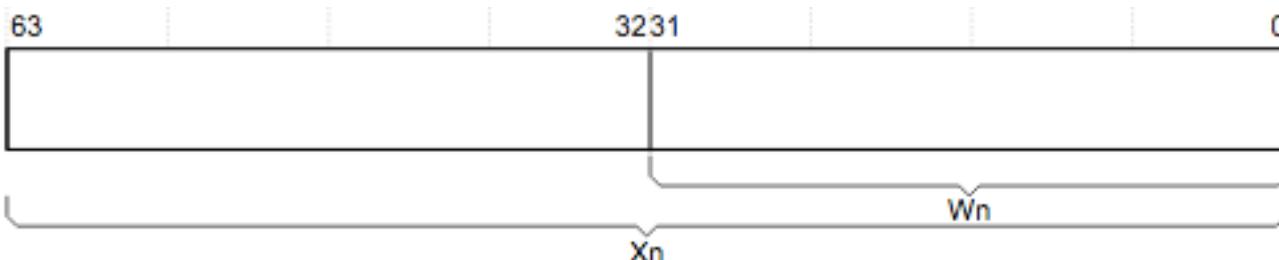
- Arithmetic instructions use register operands
- **LEGv8** has 32 registers each 64-bit long
 - A 32 x 64 bit register file
 - X registers are 64-bit long (extended/doubleword registers)
= 64 bit
- **LEGv8** has 32 registers each 32-bit long
 - W registers are 32-bit long (word registers)
- **XZR** and **WZR** (register 31)
 - Always contains the constant value 0 and can not be overwritten either

X0/W0
X1/W1
X2/W2
X3/W3
X4/W4
X5/W5
X6/W6
X7/W7
X8/W8
X9/W9
X10/W10
X11/W11
X12/W12
X13/W13
X14/W14
X15/W15
X16/W16
X17/W17
X18/W18
X19/W19
X20/W20
X21/W21
X22/W22
X23/W23
X24/W24
X25/W25
X26/W26
X27/W27
X28/W28
X29/W29
Frame pointer
Procedure link register
X30/W30

LEGv8 Registers

we use X (64 bit)
instead of W (32 bit).

- The 32-bit W register forms the lower half of the corresponding 64-bit X register.
 - That is, W_0 maps onto the lower word of X_0 , and W_1 maps onto the lower word of X_1 .
- All reads from W registers disregard the higher 32 bits of the corresponding X register and leave them unchanged.
 - 0xFFEBCD00**FFFFFABC** will be read as 0xFFFFFABC
- All writes to W registers will set the higher 32 bits of the X register to zero.
 - Writing 0xFFFFFABC into W_0 sets X_0 to 0x00000000**FFFFFABC**.



LEGv8 Registers

Temp

- **X0 – X7: procedure arguments/results**
- X8: indirect result location register
- **X9 – X15: temporaries** → anyone can overwrite them.
- X16 – X17 (IP0 – IP1): may be used by linker as a scratch register, other times as temporary register
- X18: platform register for platform independent code; otherwise a temporary register
- **X19 – X27: saved** → value is saved.
- **X28 (SP): stack pointer**
- X29 (FP): frame pointer
- **X30 (LR): link register (return address)**
- **XZR (register 31): the constant value 0**

Register Operand Example

- C code:

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

- Assumptions:

- g, h, i, and j in X0, X1, X2, X3 respectively.

- Compiled LEGv8 code:

```
ADD X9, X0, X1    // ADD result goes into temp x9
ADD X10, X2, X3   // ADD result goes into temp x10
SUB X19, X9, X10  // SUB result goes into saved x19
STR X19, f        // mem[f] <= (g + h) - (i + j)
```

Memory Operands

ARM is Load
Store Arch!

- Main memory used for composite data
 - Arrays, structures, dynamic data
- Memory is byte addressed
 - Each address identifies an 8-bit byte

Byte 3	Byte 2	Byte 1	Byte 0	
				0x00000000
				0x00000004
				0x00000008
				0x0000000C
				0x00000010
				0x00000014
				0x00000018
				0x0000001C

32 bytes

Memory Operands

- Main memory used for composite data
 - Arrays, structures, dynamic data
- Memory is byte addressed
 - Each address identifies an 8-bit byte

32 bytes

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0	
								0x00000000
								0x00000008
								0x00000010
								0x00000018

Memory Operands

- Before and after applying the arithmetic operations data transfer operations are used:
 - Load values from memory into registers
 - Store result from register to memory
 - LDR X0, [X1]
 - Unscaled no offset: *load contents of [X1] into X0*
 - LDR X0, [X1, 2] ;
 - Unscaled constant offset: *load contents of [X1 + 2] into X0*
 - LDR X0, [X1, X2] ;
 - Unscaled register offset: *load contents of [X1 + X2] into X0*
 - LDR X0, [X1, X2, LSL 3];
 - Scaled register offset: *load contents of [X1 + (X2 << 3)] into X0*
 - In scaled, you copy the content of X2 into barrel shifter to multiply it by 8 first before adding it to the address in X1 to calculate the memory address.

Memory Operands

- In LEGv8, we use **LDUR** to indicate “unscaled” operation of LDR is performed.
 - LDUR X0, [X1] ; or
 - LDUR X0, [X1, #0] ; *// Unscaled with no offset or offset 0*
 - LDUR X0, [X1, #2]: *// Unscaled with constant offset*
 - LDUR X0, [X1, X2]; *// Unscaled with register offset*
- In LEGv8, the scaled LDR instruction has been omitted and not used.
 - *Recall that LEGv8 only uses a subset of ARMv8 ISA*

Memory Operands

- In ARMv8 for scaled register offset LDR is used
 - **LDR X0, [X1, X2, LSL 3];** // load contents of $[X1 + (X2 \ll 3)]$ into X0
- *Since LDR not available in LEGv8 then the scaling operation must be done outside of the load instruction as follows:*
 - **LSL X2, X2, 3** Logical Shift Left.
 - **LDUR X0, [X1, X2]**

Memory Operand Example

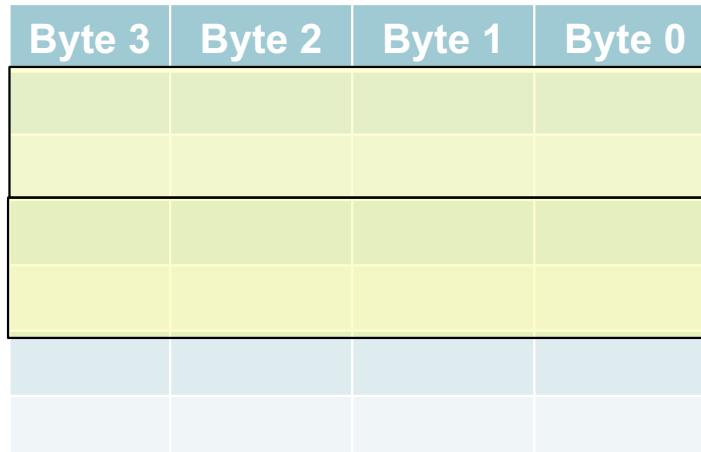
- C code:

$A[8] = h + A[4];$

- Assumptions:

1. h in $X0$, and base address of A in $X1$
2. A is an array of double words (each element in the A array is 8 bytes)

64 bits



$A[0]$

$A[1]$

Memory Operand Example

A[8] = h + A[4]; // A is an array of doubleword (8 bytes each)

Note: Variable h is in X0, and base address of array A is in X1

p. of bytes → *derefence* → *Tells value*.

LDUR x9, [x1,#32] // u for “unscaled”

ADD x9, x0, x9 // add the value found in A[4] to h

STUR x9, [x1,#64] // store the result into A[8]

Base address of A

A[0]	Mem [0] – Mem [7]
A[1]	Mem [8] – Mem [15]
A[2]	Mem [16] – Mem [23]
A[3]	Mem [24] – Mem [31]
A[4]	Mem [32] – Mem [39]
A[5]	Mem [40] – Mem [47]
A[6]	Mem [48] – Mem [55]
A[7]	Mem [56] – Mem [63]
A[8]	Mem [64] – Mem [71]

← A double word per entry

Registers vs. Memory

- Registers are faster to access than memory and require less energy
- Operating on data in memory requires loads and stores
- Compiler must use registers for variables as much as possible
 - Keep them in registers as long as possible
 - Only spill to memory for less frequently used variables or larger data sizes
- **Register optimization is important!**

Example of Register Optimization

- C code: `newAge = age + years;`

- LDUR X0, age
- LDUR X1, years
- ADD X2, X0, X1
- STUR X2, newAge

- LDUR X0, age
- LDUR X1, years
- ADD X0, X0, X1
- STUR X0, newAge

✓ Toy using
min. registers

Immediate Operands

→ Used when there is a constant numerical value involved.

- Constant data specified in an instruction

ADDI X22, X22, 4

- Immediate operand avoids a load instruction
 - LDUR X21, 4
 - ADD X22, X21, X22
- Another ISA design principle:
 - Make the **common case fast** (*Design Principle 3*)
 - Small constants are common



The ARMv8 Instruction Set

ARMv8 introduced a number of profiles or flavor of the architecture that targets certain classes of workloads.

Profile	Name	Description
Application	ARMv8-A	Optimized for a large class of general applications for mobile, tablets, and servers.
Real-Time	ARMv8-R	Optimized for safety-critical environments.
Microcontroller	ARMv8-M	Optimized for embedded systems with a highly deterministic operation.

- <https://www.youtube.com/watch?v=IfHG7bj-CEI>

Intel is in serious trouble. ARM is the Future.

Dec 21, 2018



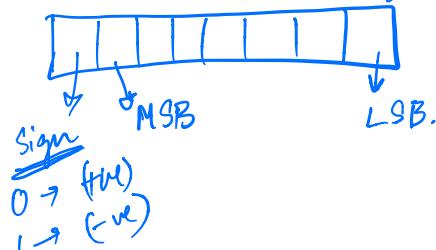
Kinds Of Data

- **Numbers**
 - Integers
 - Unsigned
 - Signed
 - Reals
 - Fixed-Point
 - Floating-Point
 - Binary-Coded Decimal
- **Text**
 - ASCII Characters
 - Strings
- **Other**
 - Graphics
 - Images
 - Video
 - Audio

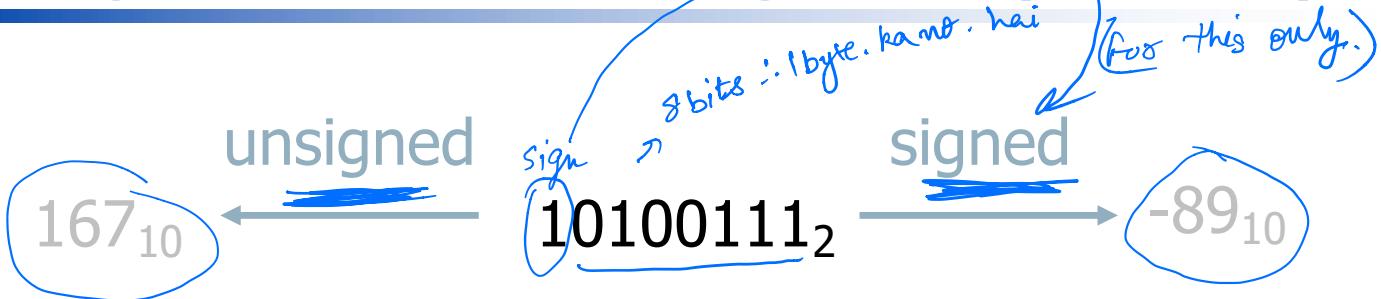
Numbers (recall)

1. Computers use binary numbers (0's and 1's).
2. Computers store and process numbers using a fixed number of digits ("fixed-precision").
1 Byte = 8 bits, Half Word = 16 bits, 1 Word = 32 bits, double word = 64 bits
3. Computers represent signed numbers using 2's complement instead of the more natural "sign-plus-magnitude" representation.

① Invert all bits of the no.
② Add 1 to LSB (Least Significant Bit)



Signed vs. Unsigned (recall)



- Signed vs. unsigned is a matter of interpretation; thus a single bit pattern can represent two different values.
- Allowing both interpretations is useful
 - Some data (e.g., count, age) can never be negative, and having a greater range is useful.

Sign Extension (recall)

- Representing a number using more bits
 - But must preserve the numeric value
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit (1111 1111) to 16-bit
 - +255: 1111 1111 => 0000 0000 1111 1111
 - -1: 1111 1111 => 1111 1111 1111 1111

1 byte → 2 byte.

MSB *Extend mein copy same.*

z y

			11111111
- In LEGv8 instruction set
 - LDURB: zero-extend loaded byte
 - LDURSB: sign-extend loaded byte

unsigned → *signed.* → *Joh MSB wko Repeat kardo.*

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111
1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111



Hexadecimal (recall)

Legv8 has 32 bit register

Base 16

- Compact representation of bit strings
- 4 bits per hex digit



0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

Example: eca8 6420

- 1110 1100 1010 1000 0110 0100 0010 0000
- 10100001010010110100111010101010

Representing Instructions

- Instructions are also encoded in binary
 - Called machine code
 - LEGv8 instructions
 - Encoded as 32-bit instruction words
 - Small number of formats, encoding operation code (opcode), register numbers, ...
 - Regularity!
- All Machine Codes
are 32 bit long.

LEGv8 Instructions Formats

~~LEGv8~~

(all are 32-bit formats)

R-format	R	Arithmetic Instruction Format
I-format	I	Immediate Format
D-format	D	Data Transfer Format
B-format	B	Unconditional Branch Format
CB-format	CB	Conditional Branch Format
IW-format	IW	Wide Immediate Format <i>→ Not used in Legv8.</i>

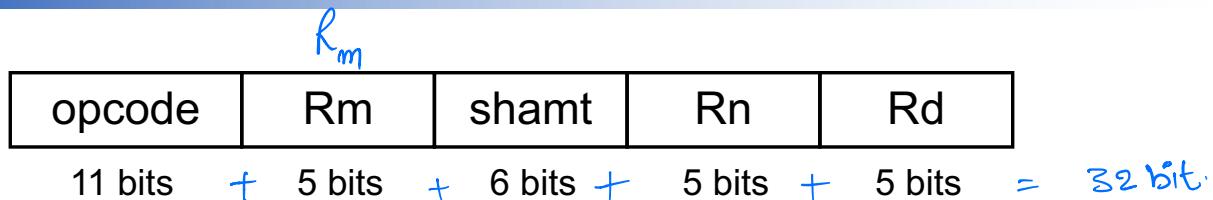
LEGv8 Instructions Formats

(3)

OPCODES IN NUMERICAL ORDER BY OPCODE

Instruction Mnemonic	Format	Width (bits)	Opcode Binary	Shamt Binary	11-bit Opcode Range (1) Start (Hex) End (Hex)
B	B	6	000101 00000 - ...1111		0A0 0BF
FMULS	R	11	00011110001	000010	0F1
FDIVS	R	11	00011110001	000110	0F1
FCMPS	R	11	00011110001	001000	0F1
FADDS	R	11	00011110001	001010	0F1
FSUBS	R	11	00011110001	001110	0F1
FMULD	R	11	00011110011	000010	0F3
FDIVD	R	11	00011110011	000110	0F3
FCMPD	R	11	00011110011	001000	0F3
FADDD	R	11	00011110011	001010	0F3
FSUBD	R	11	00011110011	001110	0F3
STURB	D	11	00111000000		1C0
LDURB	D	11	00111000010		1C2
B.cond	CB	8	01010100 000 - ...111		2A0 2A7
STURH	D	11	01111000000		3C0
LDURH	D	11	01111000010		3C2
LDURW	D	11	10001010000		4E0

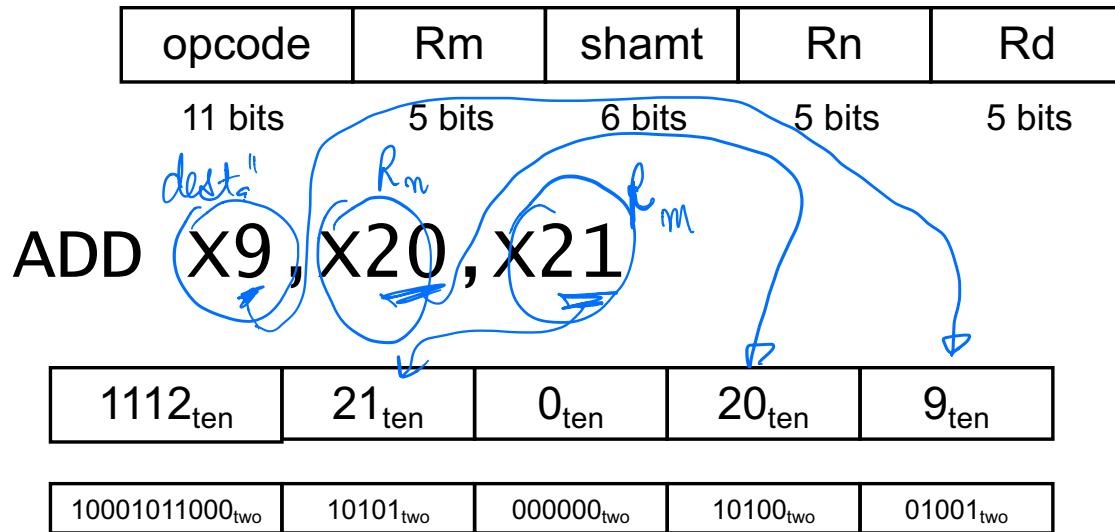
LEGv8 R-format Instructions



Instruction fields

- opcode: operation code
 - Rm: the second register source operand
 - shamt: shift amount (000000 for now)
 - Rn: the first register source operand
 - Rd: the register destination
- get from green sheet.

R-format Example



1000 1011 0001 0101 0000 0010 1000 1001₂ =

8B150289₁₆

Logical Operations (recall)

Bitwise Operators

- Bitwise operators operate on individual bit positions within the operands.
- The result in any one bit position is entirely independent of all the other bit positions.

The diagram illustrates two bitwise operations: addition and AND. On the left, a vertical column of bits is shown with the label "ADD" below it. The bits are: 1 (red), 0, 0, 1, 1. A blue bracket groups the first three bits (1, 0, 0) with the equation $1+0=0$. The next bit (1) is labeled "Carry 1". The final bit (1) is labeled $0+0=0$. Below this column is a dashed horizontal line, and at the bottom is the result 1 0 1 0 (red). On the right, another vertical column of bits is shown with the label "AND" below it. The bits are: 0, 0, 0, 1. A blue bracket groups the first three bits (0, 0, 0) with the equation $0\cdot0=0$. Below this column is a dashed horizontal line, and at the bottom is the result 0 0 0 1 (red).

Logical Operations (recall)

Instructions for bitwise manipulation

Operation	C	Java	LEGv8
Shift left	<<	<<	LSL
Shift right	>>	>>>	LSR
Bit-by-bit AND	&	&	AND, ANDI
Bit-by-bit OR			OR, ORI
Bit-by-bit NOT	~	~	EOR, EORI

Logical Operations (recall)

Q To find Least Significant 12 bits:-

❖ Example:

Jitne nibale hai woh 1 kaedo

mask:

1011 0110 1010 0100 0011	1101 1001 1010
0000 0000 0000 0000 0000	1111 1111 1111

The result of ANDing these:

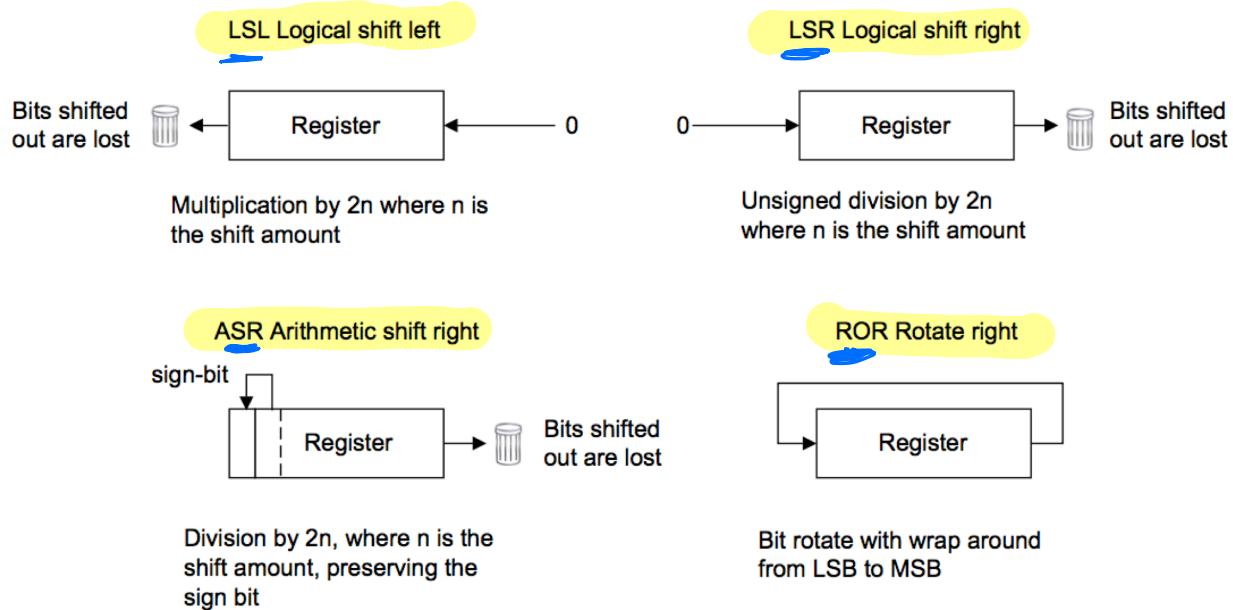
0000 0000 0000 0000 0000	1101 1001 1010
--------------------------	----------------

mask last 12 bits

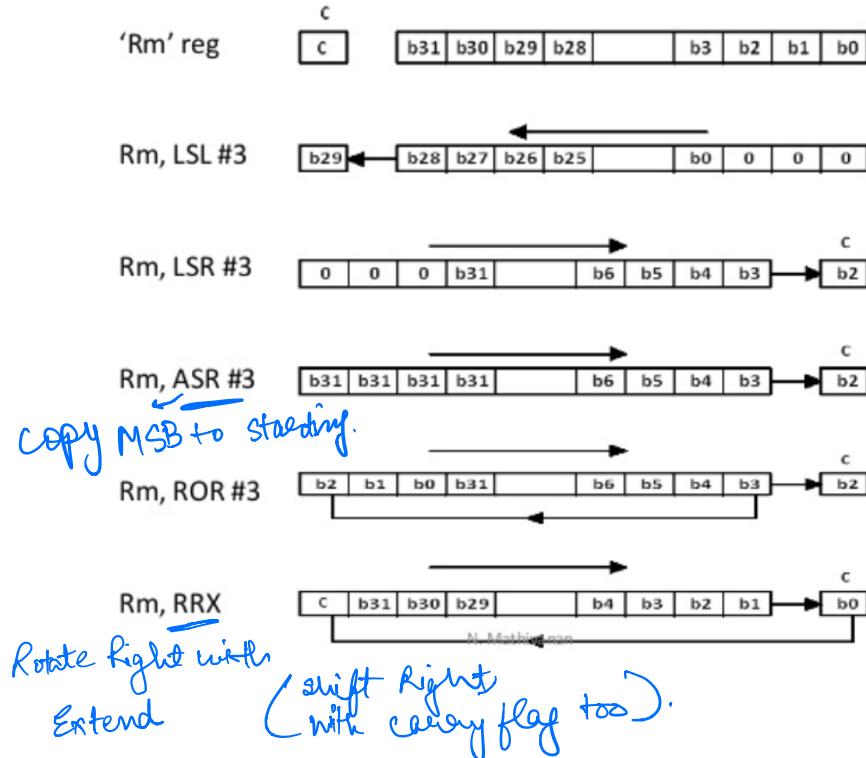
Using Bitwise OR to set the least significant half of a word:

(0x12345678 OR 0x000FFFFF) results in 0x1234FFFF

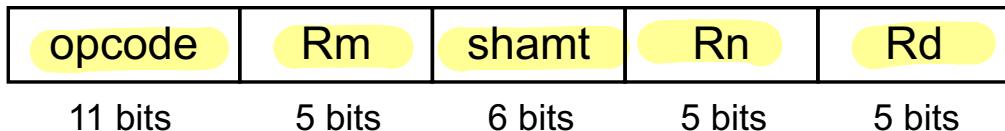
Shift Operations (recall)



Shift Operations (recall)



LEGv8 - Shift Instructions (R-Format)



- Shift machine language instructions use the R-format
- The *Rm* field is unused and is set to zero
- shamt: how many positions to shift
- The C language left-shift operator is “`<<`”
- The C language right-shift operator is “`>>`”

LEGv8 - Shift Instructions (R-Format)

Can be seen
found on shift



x_{19} does not change - A copy of x_{19} content goes into shifter & there it's shifted & then copy the result in x_{11}
[x_{19} content is not changed]

Shift left LEGv8 operator: LSL



LSL $X11, X19, \#4$ // reg $X11 = \text{reg } X19 \ll 4$ bits

- Shift left and fill with 0 bits
- LSL by i bits multiplies by 2^i



16 8 4 2 1

Seems
from green
sheet.

1691 _{ten}	0 _{ten}	4 _{ten}	19 _{ten}	11 _{ten}
---------------------	------------------	------------------	-------------------	-------------------

11010011011 _{two}	00000 _{two}	000100 _{two}	10011 _{two}	01011 _{two}
----------------------------	----------------------	-----------------------	----------------------	----------------------

LEGv8 - Shift Instructions (R-Format)

OPCODES IN NUMERICAL ORDER BY OPCODE

Instruction Mnemonic	Format	Width (bits)	Opcodes	Shamt Binary	11-bit Opcode Range (1) Start (Hex) End (Hex)
B	B	6	000101		0A0 0BF
FMULS	R	11	0001110001	000010	0F1
FDIVS	R	11	0001110001	000110	0F1
FCMPS	R	11	0001110001	001000	0F1
FADDS	R	11	0001110001	001010	0F1
FSUBS	R	11	0001110001	001110	0F1
FMULD	R	11	0001110001	000010	0F3
FDIVD	R	11	0001110001	000110	0F3
FCMPD	R	11	0001110001	001000	0F3
FADD	R	11	0001110001	001010	0F3
FSUBD	R	11	0001110001	001110	0F3
STURB	D	11	0011100000		1C0
LDURB	D	11	0011100000		1C2
B, cond	CB	8	01010100		2A0 2A7
STURH	D	11	0111100000		3C0
LDURH	D	11	0111100000		3C2
AND	R	11	1000101000		450
ADD	R	11	10001011000		458
ADDI	I	10	1001000100		488 489
ANDI	I	10	1001001000		490 491
BL	B	6	100101		4A0 4BF
SDIV	R	11	1001101010	000010	4D6
UDIV	R	11	1001101010	000001	4D6
MUL	R	11	1001101100	011111	4D8
SMULH	R	11	1001101100		4DA
UMULH	R	11	1001101110		4DE
ORR	R	11	1010101000		550
ADD\$	R	11	1010101100		558
ADDIS	I	10	101000100		588 589
ORRI	I	10	1011001000		590 591
CBZ	CB	8	10110100		5A0 5A7
CBNZ	CB	8	10110101		5A8 5AF
STURW	D	11	1011100000		5C0
LDURSW	D	11	10111000100		5C4
STURS	R	11	1011100000		5E0
LDURS	R	11	10111100010		5E2
STXR	D	11	11001000000		640
LDRX	D	11	11001000010		642
EOR	R	11	11001010000		650
SUB	R	11	11001011000		658
SUBI	I	10	1101000100		688 689
SEOR	I	10	1101001000		690 691
MOVZ	IM	9	110100101		694 697
LSR	R	11	11010011010		69A
LSL	R	11	11010011011		69B

IEEE 754 FLOATING-POINT STANDARD

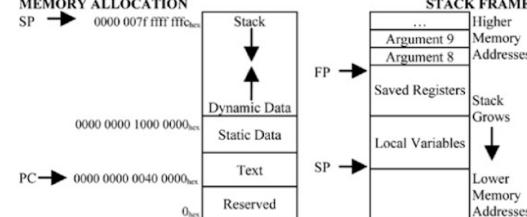
$(-1)^s \times (1 + Fraction) \times 2^{(Exponent - Bias)}$
where Single Precision Bias = 127,
Double Precision Bias = 1023

IEEE Single Precision and Double Precision Formats:

IEEE 754 Symbols		
Exponent	Fraction	Object
0	0	± 0
0	$\neq 0$	\pm Denorm
1 to MAX - 1	anything	\pm F1. Pt. Num.
MAX	0	$\pm \infty$
MAX	$\neq 0$	NaN

S.P. MAX = 255, D.P. MAX = 2047		
S	Exponent	Fraction
31 30	23 22	0
S	Exponent	Fraction
63 62	52 51	0

MEMORY ALLOCATION



DATA ALIGNMENT

Double Word							
Word				Word			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7

Value of three least significant bits of byte address (Big Endian)

EXCEPTION SYNDROME REGISTER (ESR)

Exception Class (EC)	Instruction Length (IL)	Instruction Specific Syndrome field (ISS)
31	26	25 24 0

EXCEPTION CLASS

EC	Class	Cause of Exception	Number	Name	Cause of Exception
0	Unknown	Unknown	34	PC	Misaligned PC exception
7	SIMD	SIMD/FP registers disabled	36	Data	Data Abort
14	FPE	Illegal Execution State	40	FPE	Floating-point exception
17	Sys	Supervisor Call Exception	52	WPT	Data Breakpoint exception
32	Instr	Instruction Abort	56	BKPT	SW Breakpoint Exception

LEGv8 - Shift Instructions (R-Format)

Binary to Decimal converter

Binary to decimal number conversion calculator and how to convert.

Enter binary number:

 2

Convert

Decimal number:

 10

Decimal from signed 2's complement:

 10

Hex number:

 16

LEGv8 - Shift Instructions (R-Format)

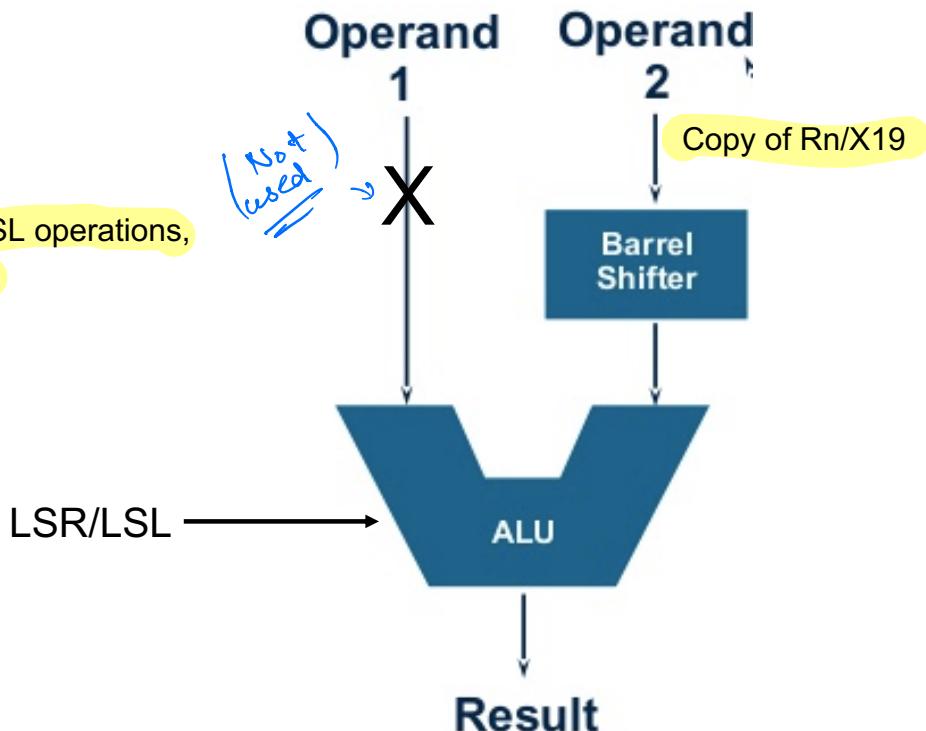
opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- Shift right LEGv8 operator: LSR
- `LSR X11,X19, 4 // reg X11 = reg X19 >> 4 bits`
- Shift right and fill with 0 bits
 - LSR by i bits divides by 2^i (unsigned only)
Arithmetic Shift Right (ASR) (if signed)
- NOTE: in both LSR and LSL operations, X19 is not altered

LEGv8 - Shift Instructions (R-Format)

LSR/LSL	0	4	19	11
11 bits	5 bits	6 bits	5 bits	5 bits

- LSR X11,X19, 4
- LSL X11,X19, 4
 - NOTE: in both LSR and LSL operations, X19 is not altered



LEGv8 D-format Instructions

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits <i>(not used in this course)</i>	5 bits <i>Base Address</i>	5 bits

Load/store instructions

- Rn: the register number that contains the base address
- address: constant offset from the base address in Rn (+/- 256 bytes)
- Rt: destination (for load) or source (for store) register number
- The 2-bit op2 is just an extension of the opcode field
 - Will be 00 in LEGv8

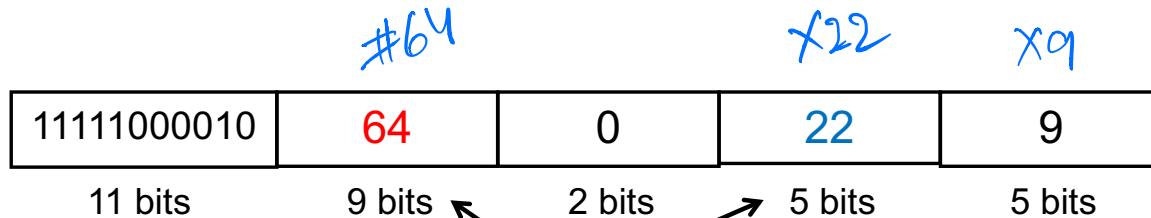
Design Principle 3: Good design demands good compromises

- Different formats complicate decoding, but allow 32-bit instructions uniformly
- Keep formats as similar as possible

LEGv8 D-format Instructions

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- LDUR X9, [X22, #64] // Temporary reg X9 gets A[8]



- Base address and offset

LEGv8 I-format Instructions

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

- Immediate instructions

- Rn: source register
- Rd: destination register

- Immediate field is zero-extended

- Example: ADDI X9, X11, #2

Add 2 to X_{11} content
→ Put it in X_9 .

1001000100	000000000010	01011	01001
------------	--------------	-------	-------

10 bits

12 bits

5 bits

5 bits

write No.in

Binary Format.

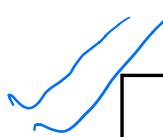
(12 bits)

∴ we have only 32 Reg. in Legv8.

LEGv8 IW-format Instructions

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

- MOVZ – Move Wide with Zero *→ overwrite the value after adding*
- MOVK – Move Wide with Keep *→ don't " " " "*



opcode	S	immediate	Rd
9 bits	2 bits	16 bits	5 bits

LEGv8 IW-format Instructions

- MOVZ – Move Wide with Zero

X9

opcode	S	immediate	Rd
9 bits	2 bits	16 bits	5 bits

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 1100 1100
Original X9 value = 460_{10}

- MOVZ X9, 255, LSL 16

Move wide with 0's

110100101	01	0000 0000 1111 1111	01001
9 bits	2 bits	16 bits	5 bits

0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 0000 0000 0000 0000
New X9 value = 255×2^{16}



LEGv8 IW-format Instructions

Original X9 value = 460_{10}

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 1100 1100

- MOVZ X9, 255, LSL 16

Move wide with 0's

110100101	01	0000 0000 1111 1111	01001
-----------	----	---------------------	-------

9 bits

2 bits

16 bits

5 bits

0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 0000 0000 0000 0000 0000

New X9 value = $255_{10} \times 2^{16}$

- MOVK X9, 255, LSL 16

Move wide with keep

111100101	01	0000 0000 1111 1111	01001
-----------	----	---------------------	-------

9 bits

2 bits

16 bits

5 bits

0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 0000 0001 1100 1100

New X9 value = $255_{10} \times 2^{16} + 460_{10}$



Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- **CBZ register, L1**
 - if (register == 0) branch to instruction labeled L1;
- **CBNZ register, L1**
 - if (register != 0) branch to instruction labeled L1;
- **B L1**
 - branch unconditionally to instruction labeled L1;

Branch Addressing

B-type

- B 10000 // go to location 10000_{ten}

5	10000 _{ten}
6 bits	26 bits

CB-type

- CBNZ x19, Exit // go to Exit if $x19 \neq 0$

181	Exit	19
8 bits	19 bits	5 bits

- Both addresses are PC-relative, and not absolute
 - Address = PC + offset (from instruction)



Branch Addressing

OPCODES IN NUMERICAL ORDER BY OPCODE

Instruction Mnemonic	Format	Width (bits)	Opcodes	Shamt Binary	11-bit Opcode Range (1) Start (Hex) End (Hex)
B	B	6	000101		0A0 - 0BF
FMULS	R	11	0001110001	000010	0F1
FDIVS	R	11	0001110001	000110	0F1
FCMPS	R	11	0001110001	001000	0F1
FADDS	R	11	0001110001	001010	0F1
FSUBS	R	11	0001110001	001110	0F1
FMULD	R	11	0001110011	000010	0F3
FDIVD	R	11	0001110011	000110	0F3
FCMPD	R	11	0001110011	001000	0F3
FADD	R	11	0001110011	001010	0F3
FSUBD	R	11	0001110011	001110	0F3
STURB	D	11	0011000000		1C0
LDURB	D	11	0011000010		1C2
B, cond	CB	8	01010100		2A0 - 2A7
STURH	D	11	0111000000		3C0
LDURH	D	11	0111000010		3C2
AND	R	11	1000101000		450
ADD	R	11	1001010100		458
ADDI	I	10	1001000100		488 - 489
ANDI	I	10	1001001000		490 - 491
BL	B	6	100101		4A0 - 4BF
SDIV	R	11	1001101010	000010	4D6
UDIV	R	11	1001101010	000011	4D6
MUL	R	11	1001101000	011111	4D8
SMULH	R	11	1001101010		4DA
UMULH	R	11	1001101110		4DE
ORR	R	11	1010101000		550
ADD\$	R	11	1010101100		558
ADDIS	I	10	1010001000		588 - 589
ORRI	I	10	1011001000		590 - 591
CBZ	CB	8	10110100		5A0 - 5A7
CBNZ	CB	8	10110101		5A8 - 5AF
STURW	D	11	1011100000		5C0
LDURSW	D	11	1011100010		5C4
STURS	R	11	1011100000		5E0
LDURS	R	11	10111100010		5E2
STXR	D	11	1100100000		640
LDRX	D	11	11001000010		642
EOR	R	11	11001010000		650
SUB	R	11	1101011000		658
SUBI	I	10	1101000100		688 - 689
SEOR	I	10	1101001000		690 - 691
MOVZ	IM	9	110100101		694 - 697
LSR	R	11	11010011010		69A
LSL	R	11	11010011011		69B

IEEE 754 FLOATING-POINT STANDARD

$(-1)^s \times (1 + Fraction) \times 2^{(Exponent - Bias)}$
where Single Precision Bias = 127,
Double Precision Bias = 1023

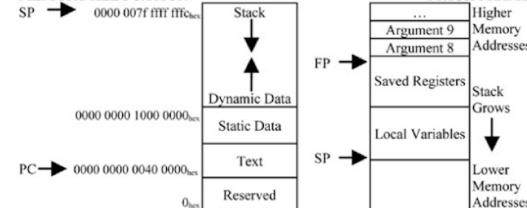
IEEE Single Precision and Double Precision Formats:

S		Exponent		Fraction	
31	30	23	22		0
S		Exponent		Fraction	
63	62	52	51		0

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	$\neq 0$	\pm Denorm
1 to MAX - 1	anything	\pm F1. Pt. Num.
MAX	0	$\pm \infty$
MAX	$\neq 0$	NaN

MEMORY ALLOCATION



DATA ALIGNMENT

Double Word							
Word				Word			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7

Value of three least significant bits of byte address (Big Endian)

EXCEPTION SYNDROME REGISTER (ESR)

Exception Class (EC)	Instruction Length (IL)	Instruction Specific Syndrome field (ISS)
31	26	25 - 24

EXCEPTION CLASS

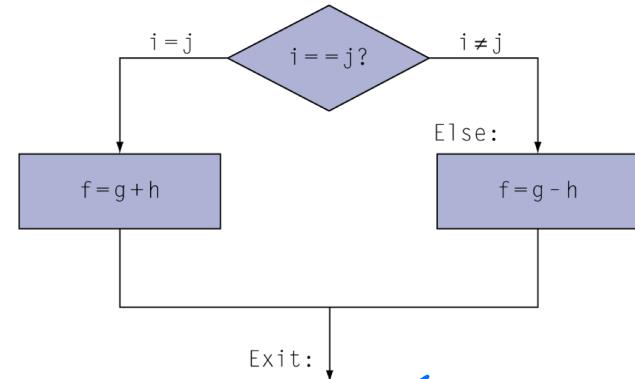
EC	Class	Cause of Exception	Number	Name	Cause of Exception
0	Unknown	Unknown	34	PC	Misaligned PC exception
7	SIMD	SIMD/FP registers disabled	36	Data	Data Abort
14	FPE	Illegal Execution State	40	FPE	Floating-point exception
17	Sys	Supervisor Call Exception	52	WPT	Data Breakpoint exception
32	Instr	Instruction Abort	56	BKPT	SW Breakpoint Exception

Compiling If Statements

C code:

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

- i in X20, j = X21, g in X22, h in X23



Compiled LEGv8 code:

```
SUB X9, X20, X21 // i-j, set Z Flag  
CBNZ X9, Else  
ADD X19, X22, X23 // g+h  
B Exit Save Reg.
```

Annotations in blue ink:

- "Temp Reg." is written above the first SUB instruction.
- "set Z Flag" is crossed out with a large red X.
- "Put value in X9." is written below the CBNZ instruction.
- "Save Reg." is written below the B Exit instruction.

Else: SUB X19, X22, X23 // g-h

Exit: ...

STUR X19, f

Assembler calculates addresses

Compiling Loop Statements

- C code: *(Search anomaly) → Take diff. array k se waha stop.*
while (save[i] == k) i += 1; *64 bit size of each element.*
 - The save array, an array of 8 bytes long elements
 - k** = 11100000 11111111 00000000 11111111 11111111 11111111 11110010 00111111

save[i] 11100000 11111111 00000000 11111111 11111111 11111111 11110010 00111111

.

save[2] 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111

save[1] 11100000 11111111 00000000 11111111 11111111 11111111 11110010 00111111

save[0] 11100000 11111111 00000000 11111111 11111111 11111111 11110010 00111111



Compiling Loop Statements

- C code:

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

- i in x22, 64-bit k in x24, address of save in x25

- Compiled LEGv8 code:

// get to starting address of the ith element in save keeping in mind that each element of the save array is 8 bytes long or 64 bits.



```
Loop: LSL    x10,x22,3 //calculate the offset for ith element
      ADD    x10,x10,x25 //x10 gets (starting address of save +
                           //offset) = address of ith element
      LDUR   x9,[x10,#0] //x9 gets the content of ith element
      SUB    x11,x9,x24 //is save[i] == k?
      CBNZ   x11,Exit   //if not, we are done (anomaly found)
      ADDI   x22,x22,1 //if equal increment i(check next item)
      B      Loop       //go back to check the (i+1)th element
```

Exit: ...



Compiling Loop Statements

- C code:

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

- i in x22, 64-bit k in x24, address of save in x25

- Compiled ARMv8 code:

Loop: // LSL X10,X22,3
// ADD X10,X10,X25
LDR x9,[X25,X22,LSL 3]
SUB x11,x9,x24 // is save[i] == k?
CBNZ x11,Exit // if not, we are done, was found
ADDI x22,x22,1 // if equal increment i, continue
B Loop

Exit: ...



Conditional Codes and Flags

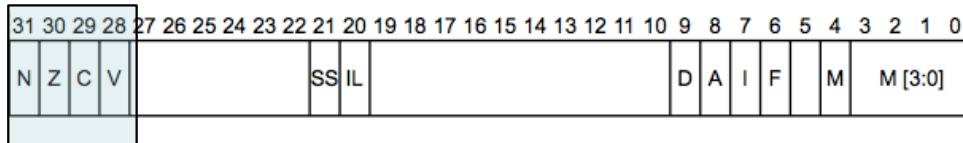


Figure 4-4 SPSR

The individual bits represent the following values for AArch64:

N	Negative result (N flag).
Z	Zero result (Z) flag.
C	Carry out (C flag).
V	Overflow (V flag).

Safe Program Status Register.

- SS** Software Step. Indicates whether software step was enabled when an exception was taken.
- IL** Illegal Execution State bit. Shows the value of PSTATE.IL immediately before the exception was taken.
- D** Process state Debug mask. Indicates whether debug exceptions from watchpoint, breakpoint, and software step debug events that are targeted at the Exception level the exception occurred in were masked or not.
- A**SError (System Error) mask bit.
- I** IRQ mask bit.
- F**FIQ mask bit.
- M[4]** Execution state that the exception was taken from. A value of 0 indicates

Conditional Codes and Flags

- Flags are set after the execution of the following LEGv8 instructions:
 - ADDS, ADDIS, ANDS, ANDIS, SUBS, SUBIS
 - negative (N): result had 1 in MSB
 - zero (Z): result was 0
 - overflow (V): result overflowed
 - carry (C): result had carryout from MSB, or borrow into MSB
- This is for
Set flag.

Signed vs. Unsigned

- Example
 - $X22 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$
 - $X23 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$
 - Is $X22 > X23$?
 - $x22 < x23 \# \text{signed}$
 - $-1 < +1$
 - $x22 > x23 \# \text{unsigned}$
 - $+4,294,967,295 > +1$



Conditional Codes and Flags

SUB X11,X9,X24

CBNZ X11, Exit or CBZ X11, Done

- Can equally use the S-suffix to set the flags used by the condition codes such as **SUBS X11, X9, X24**
 - Negative (N=1): result had 1 in MSB
 - Zero (Z=1): result was 0
 - Both C and V flags will also be updated
- Use **subtract** to set flags, then conditionally branch:
 - **B.EQ** Done (will branch to Done label if Z = 1)
 - **B.NE.** Exit. (will branch to Exit label if Z = 0)
 - **B.LT** (less than, signed), **B.LO** (less than, unsigned)
 - **B.LE** (less than or equal, signed), **B.LS** (less than or equal, unsigned)
 - **B.GT** (greater than, signed), **B.HI** (greater than, unsigned)
 - **B.GE** (greater than or equal, signed),
 - **B.HS** (greater than or equal, unsigned)

Conditional Codes and Flags

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	~(Z=0 & N=V)	B.LS	~(Z=0 & C=1)
>	B.GT	(Z=0 & N=V)	B.HI	(Z=0 & C=1)
≥	B.GE	N=V	B.HS	C=1

GT (s greater than), GE (s greater than or equal), LT (s less than), LE (s less than or =), HI (u higher), HS (u higher or same), LO (u lower), LS (u lower or same).

Conditional Codes Using Flags

Condition Code		Opposite	
Code	Description	Code	Description
eq	Equal.	ne	Not equal.
hs (or cs)	Unsigned higher or same (or carry set).	lo (or cc)	Unsigned lower (or carry clear).
mi	Negative.	pl	Positive or zero.
vs	Signed overflow.	vc	No signed overflow.
hi	Unsigned higher.	ls	Unsigned lower or same.
ge	Signed greater than or equal.	lt	Signed less than.
gt	Signed greater than.	le	Signed less than or equal.
a1 (or omitted)	Always executed.	<i>There is no opposite to a1.</i>	

Conditional and Unconditional Summary

B.cond label

Branch: conditionally jumps to program-relative label if cond is true.

CBNZ Xn, label

Compare and Branch Not Zero (extended): conditionally jumps to label if X_n is not equal to zero.

CBZ Xn, label

Compare and Branch Zero (extended): conditionally jumps to label if X_n is equal to zero.

B label

Program Counter

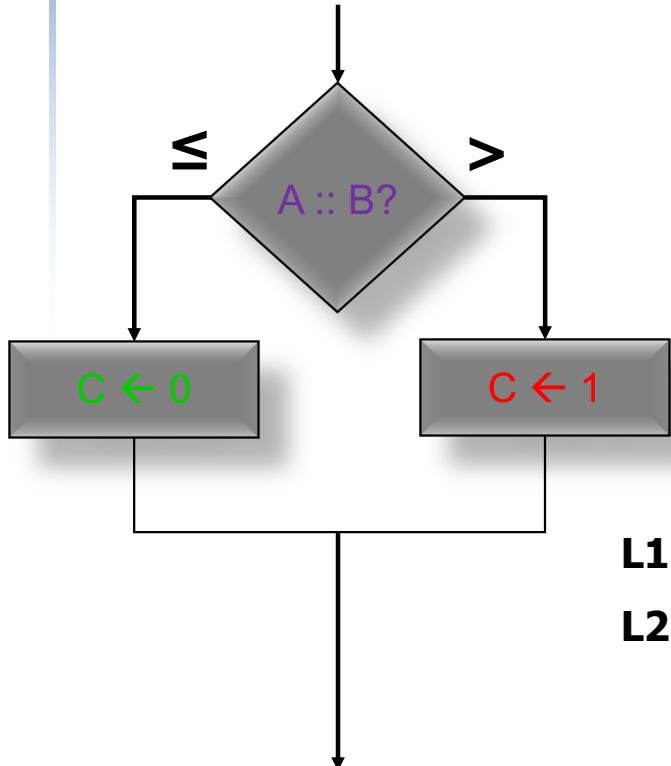
Branch: unconditionally jumps to pc-relative label.

BL label

Branch and Link: unconditionally jumps to pc-relative label, writing the address of the next sequential instruction to register X_{30} .

Conditional Code Example

if-then-else statement



LDUR	X0,A
LDUR	X1,B
SUBS	X2,X0,X1 // x0-x1 & set flags
//CMP	X0, X1 // (A-B) is A LE B
B.LE	L1 //if Z=1 or N !=V
LDUR	X0,=1 (MOV X0, 1)
B	L2
LDUR	X0,=0 (MOV X0, 0)
STUR	X0,C
...	

Opposite Conditional Code Example

- if ($a > b$) $a += 1$;
 - Assuming a is already loaded in X22, and b in X23

LDUR X22, a

LDUR X23, b

SUBS X9,X22,X23 // *(a-b) use subtract to make comparison*

B.LE Exit *// if $a \leq b$ (if opposite condition true) then
exit (conditional branch)*

ADDI X22, X22,1

Exit:



Function Call and Return

Function Call: “BL function”

- Loads program counter (pc) with entry point address of function.
- Saves return address in the link register (LR aka X30).

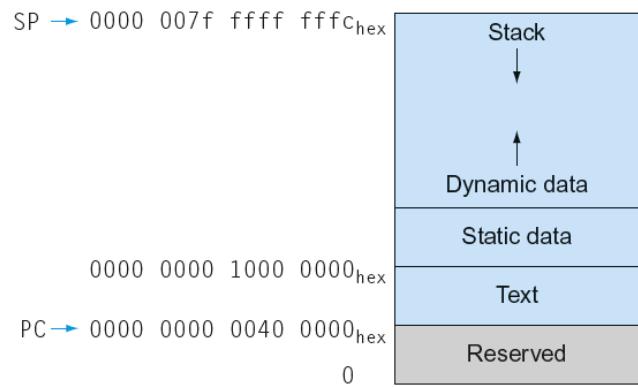
Branch with Link

Function Return: “BR LR”

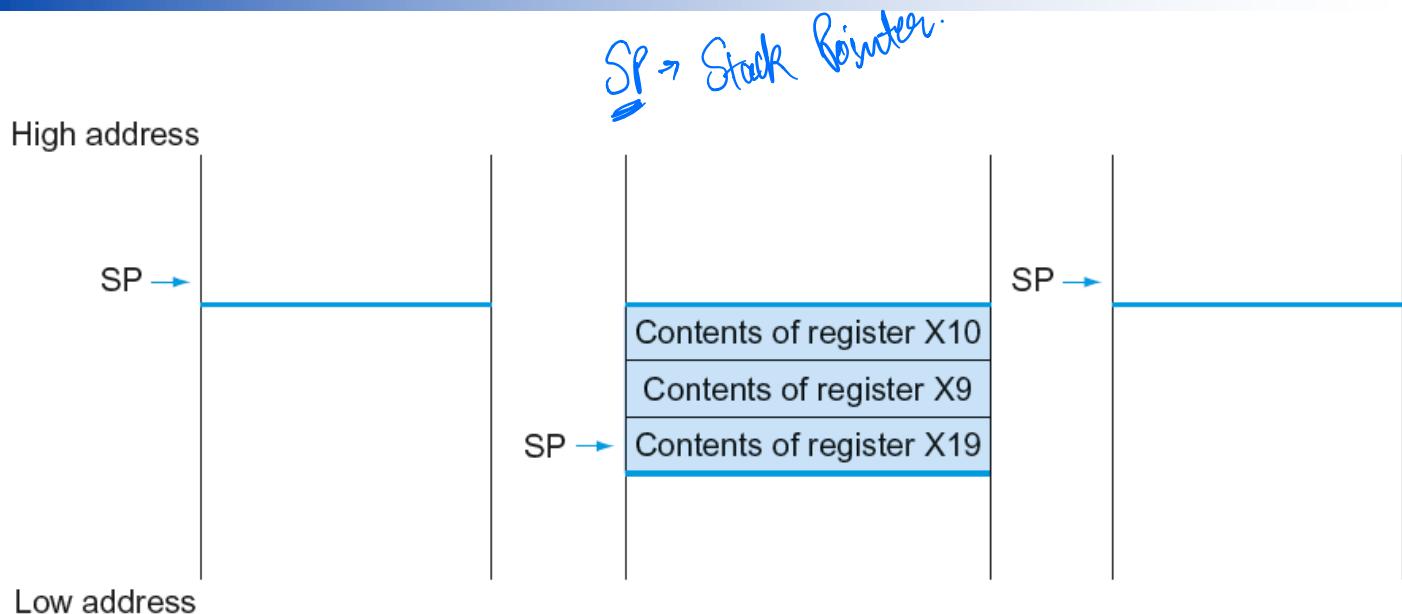
- Copies link register (X30 aka LR) back into program counter.

Memory Layout (recall)

- Text: program code
- Static data: global variables
 - e.g., static variables in C, constant arrays and strings
- Dynamic data: heap
 - E.g., malloc in C, new in Java
- Stack: automatic storage



Storage on the Stack (recall)



Push {X10, X9, X19}

Pop {X19, X9, X10}

LIFO

X19, X9, X10
and SP are
restored with
their original
values

Register Usage (Recall)

- X0 – X7: procedure arguments/results
- X9 to X15: temporary registers
 - Not preserved by the called function
- X19 to X27: saved registers
 - If used, the called function saves and restores them
- X30 aka LR contains the return address in the calling function.



Function Call and Return

- Steps required
 1. Place parameters in registers X0 to X7
 2. Transfer control to function (BL func)
 3. Acquire storage for function (on stack)
 4. Perform function's operations
 5. Place result in register for caller (X0...X7)
 6. Return to place of call (address in X30 aka LR)

Function Call and Return

```
void enable(void) ;           //defining enable()
```

• • •

```
enable() ;
```

```
//calling enable()
```

• • •

Function Call and Return

```
void enable(void) ;           //defining enable()
```

• • •

```
enable() ;                  //calling enable()
```

• • •

 *Compiler*

• • •

BL enable

• • •

Function Call and Return

```
void enable(void) ;
```

//defining enable()

• • •

enable() ;

• • •



Compiler

//calling enable()

• • •

BL

enable

PC →

Address

saved in LR

enable

export enable

• • •

• • •

BR

LR

Leaf Procedure Example

- C code:

```
long long int leaf_example (long long int  
x0 g, long long int h, long long int i, long  
long int j)           X1                      X2  
{                     X3  
    long long int f;  
    f = (g + h) - (i + j);  
    return f;  
}
```

- Arguments g, ..., j in X0, ..., X3



Leaf Procedure Example

■ LEGv8 code:

leaf_example:

```
ADD X9,X0,X1      // X9 = g + h
ADD X10,X2,X3     // X10 = i + j
SUB X11,X9,X10    // f = X9 - X10
ADD X0,X11,XZR    // copy f to X0 to return it
BR LR              // Return to caller
```

```
{long long int f;
f = (g + h) - (i + j);
return f;}
```

*(Using scratch
Reg.)*

Leaf Procedure Example

LEGv8 code:

leaf_example:

```
Saving in stack
few stack slots
the value
gets
LDUR X20, [SP, #16]
LDUR X19, [SP, #8]
LDUR X21, [SP, #0]
SUBI SP, SP, #24
STUR X20, [SP, #16]
STUR X19, [SP, #8]
STUR X21, [SP, #0]
ADD X19, X0, X1
ADD X20, X2, X3
SUB X21, X19, X20
ADD X0, X21, XZR
LDUR X20, [SP, #16]
LDUR X19, [SP, #8]
LDUR X21, [SP, #0]
ADDI SP, SP, #24
BR LR
```

{long long int f;
f = (g + h) - (i + j);
return f;} *Each 8bytes.*

(Using
Stack)

// Make room to save X20, X19, X21 on stack
// Save X20 on the stack, SP [#16-#23]
// Save X19 on the stack, SP [#8-#15]
// Save X21 on the stack, SP [#0-#7]
// X19 = g + h
// X20 = i + j
// f = X19 - X20
// copy f to X0 to return it
// Restore X20 from stack
// Restore X19 from stack
// Restore X21 from stack
// Update the stack pointer
// Return to caller

Leaf Procedure Example

- In earlier versions of ARM:

leaf_example:

```
PUSH {x19, x20, x21} // Save X19, X20, and X21 on the stack
ADD X19, x0, x1          // X19 = g + h
ADD X20, x2, x3          // X20 = i + j
SUB X21, x19, x20         // f = X19 - X20
ADD X0, x21, XZR          // copy f to X0 to return it
POP {X19, x20, x21} // Restore X10, X9, X19 from stack
BR LR                  // Return to caller
```



C Sort Example

- Illustrates use of assembly instructions for a C sort function
- Swap procedure (leaf)

```
void swap(long long int v[], long long
int k)
{
    long long int temp;
    temp = v[k];//save the kth element
    v[k] = v[k+1];//copy (k+1)th to kth
    v[k+1] = temp;//copy temp to (k+1)th
}
```

- Starting address of v in X0, and k in X1

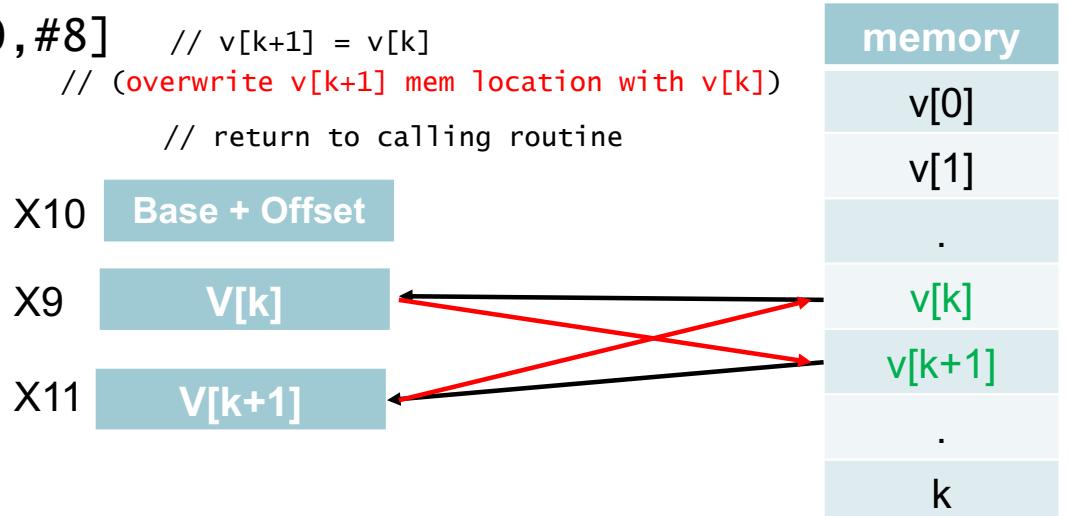


The Procedure Swap

***Assumption: Starting address of the array of double words v in X0, and k in X1

swap:

LSL X10, X1, #3 // $x_{10} = k * 8$ (find byte offset of the $v[k]$)
ADD X10, X0, X10 // $x_{10} = \text{address of } v[k] = \text{address of } v[0] + \text{offset of } v[k]$
LDUR X9, [X10, #0] // $x_9 = v[k]$ (save content of $v[k]$ into x_9 temp register)
LDUR X11, [X10, #8] // $x_{11} = v[k+1]$ (load content of $v[k+1]$ into x_{11})
STUR X11, [X10, #0] // $v[k] = v[k+1]$ (overwrite $v[k]$ mem location with $v[k+1]$)
STUR X9, [X10, #8] // $v[k+1] = v[k]$
// (overwrite $v[k+1]$ mem location with $v[k]$)
BR LR // return to calling routine



Passing Parameters to a Function

```
void display(uint8_t *p, int32_t);
```

• • •

```
display(ptrToBuffer, 5);
```

• • •

↓ Compiler

• • •

```
LDR X0, =ptrToBuffer  
LDVR X1, =5.  
BL display
```

Registers X0-X7
used for first 8
parameters; any
more have to be
pushed on stack

export display
display ...
...
BR LR

• • •

Returning a Value from Functions

```
int32_t random(void) ;
```

• • •

```
numb = random() ;
```

• • •

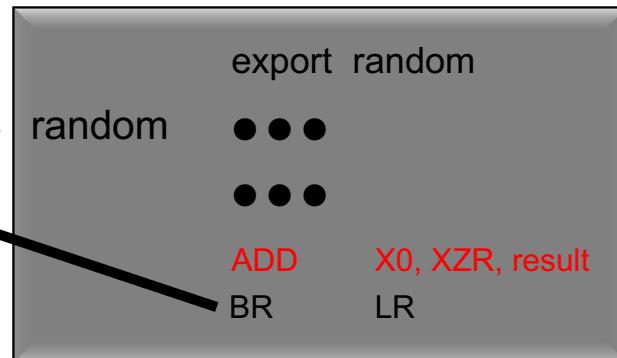
↓ *Compiler*

• • •

```
BL random
```

```
STUR X0, numb
```

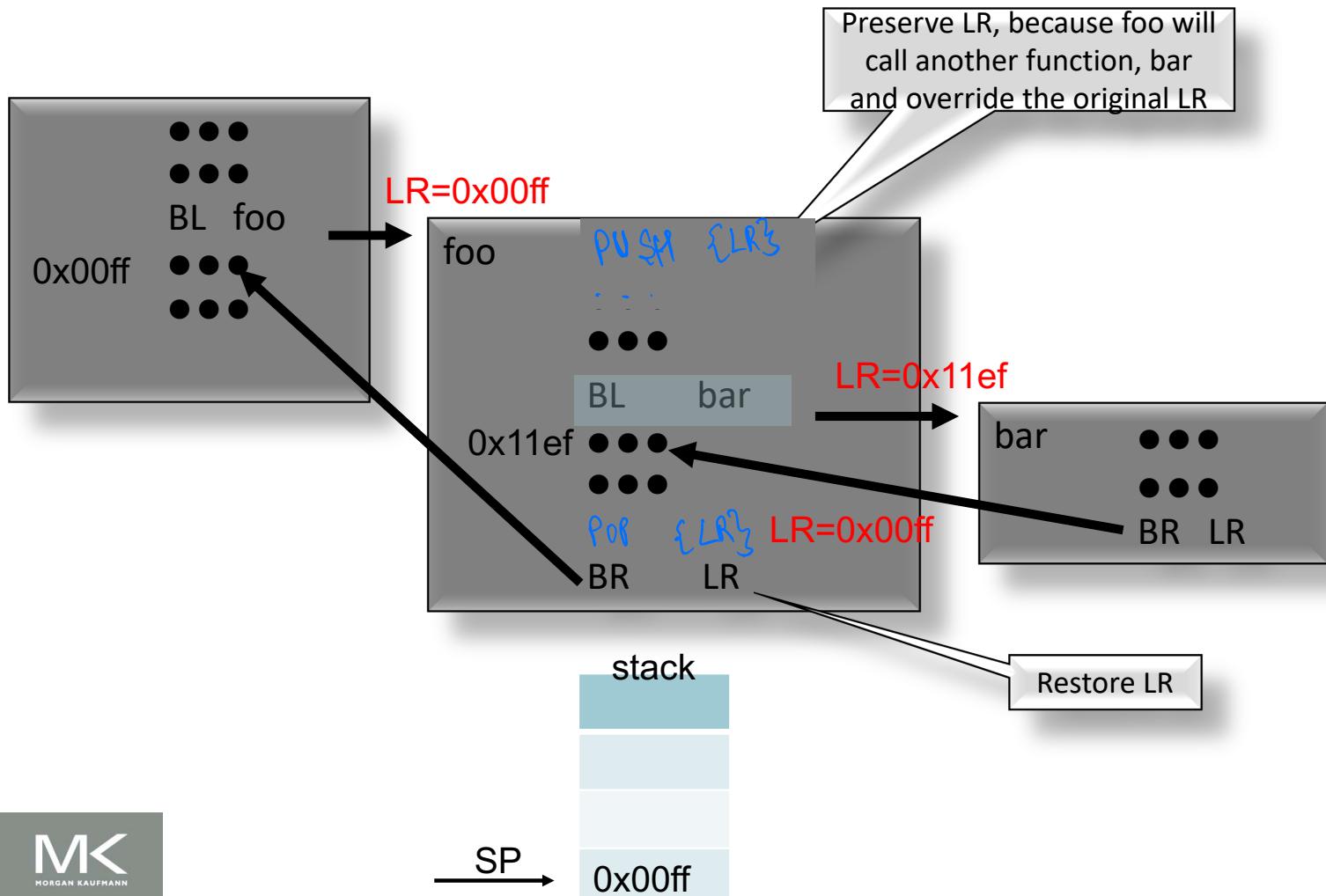
• • •



Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
 - Its return address
 - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example



Non-Leaf Procedure Example

```
Main()
{
    .
    y = f1(2);
    .
}
```

Two functions with parameters and return values, one calling the other.

C Version

```
int32_t f1(int32_t i)
{
    return f2(4) + i ;
}

***assume i in X0
```

ARMv8 Assembly Version

	f1:
	PUSH {X4,LR} // Preserve X4 & LR
	ADD X4, XZR, X0 //Keep i safe in X4
	ADDI X0,XZR,#4 //X0 <- f2's arg
	BL f2 // X0 <- f2(4)
	ADD X0,X0,X4 // X0 <- f2(4) + i
	POP {X4,LR} // Restore X4 & LR
	BR LR

1999 ISO (C99) New Data Types

#include <stdint.h>

1 byte
8-bits
1/2 word
16-bits
1 word
32-bits
double word
64-bits

Signed
int8_t
int16_t
int32_t
int64_t

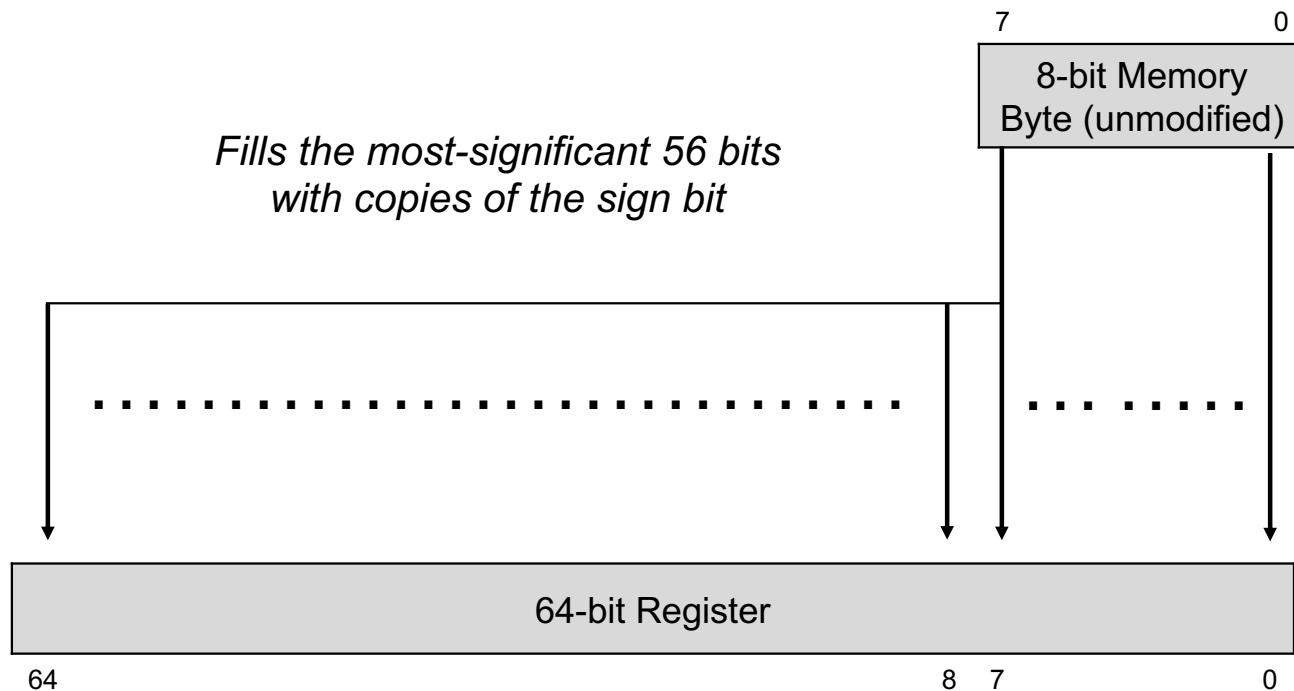
Unsigned
uint8_t
uint16_t
uint32_t
uint64_t

DATA TYPE	SIZE (IN BYTES)	RANGE
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	

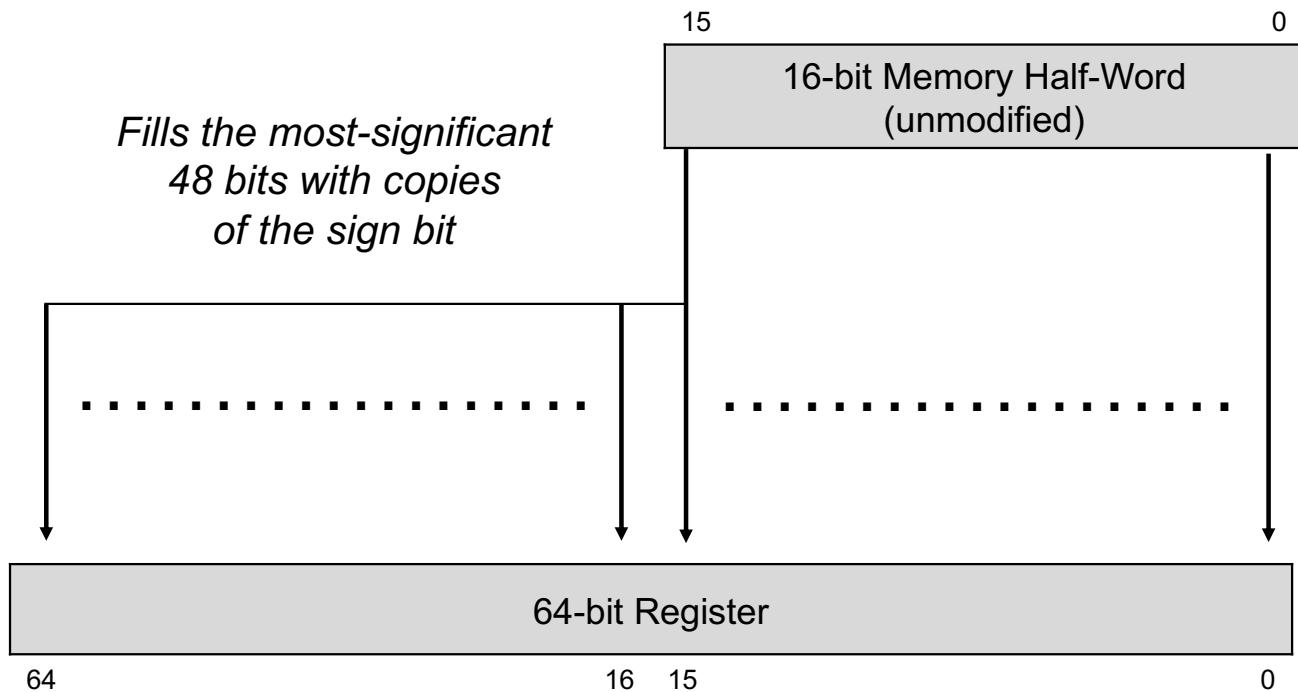
Byte/Halfword/Word Operations

- LEGv8 byte/halfword/word load/store
 - Load/Store **byte**:
 - LDURB Rt, [Rn, offset] //Zero extend to 64 bits in rt
 - LDURSB Rt, [Rn, offset] //Sign extend to 64 bits in rt
 - STURB Rt, [Rn, offset] //Store just rightmost byte, same size as dest. *in mem.*
 - Load/Store **halfword**:
 - LDURH Rt, [Rn, offset] //Zero extend to 64 bits in rt
 - LDURSH Rt, [Rn, offset] //Sign extend to 64 bits in rt
 - STURH Rt, [Rn, offset] //Store just rightmost h-word, same size as dest. *in mem.*
 - Load/Store signed **word**: *LDUR w0, a // unsigned.*
 - LDURSW Rt, [Rn, offset] //Sign extend to 64 bits in rt
 - STURW Rt, [Rn, offset] //Store just rightmost word, same size as dest.
 - Where both Rt and Rn are 64-bit X registers

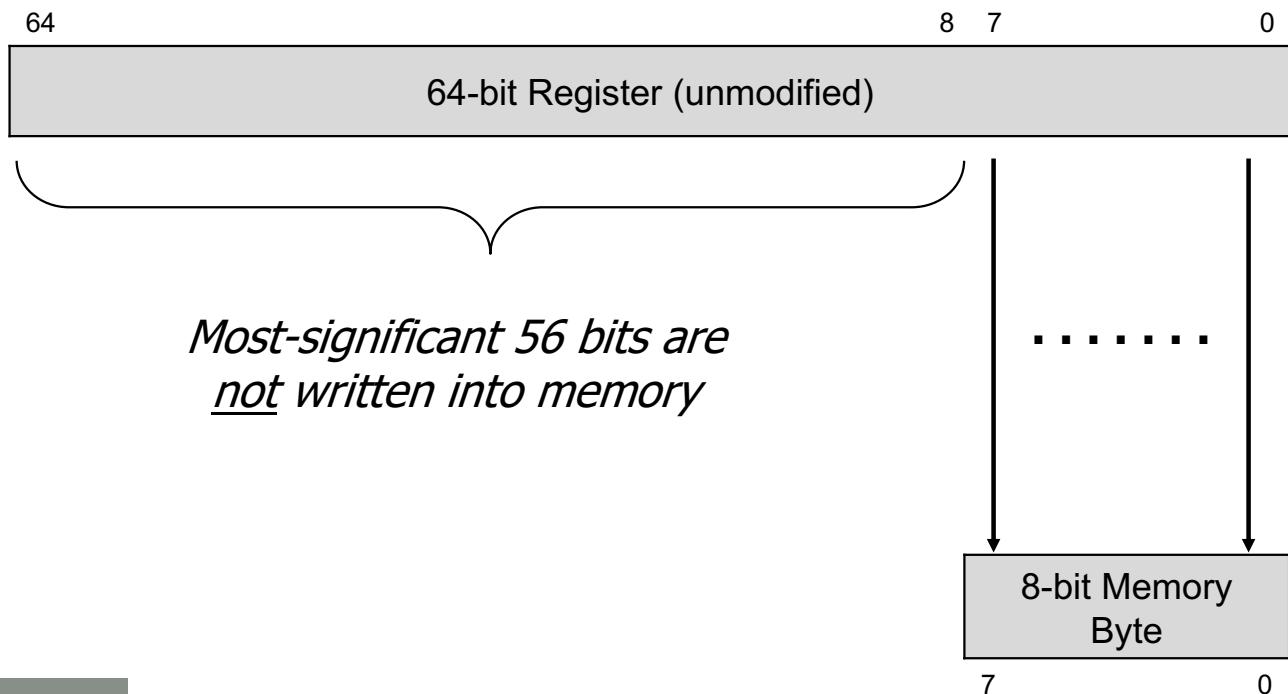
LDURSB: Load Register with Signed Byte



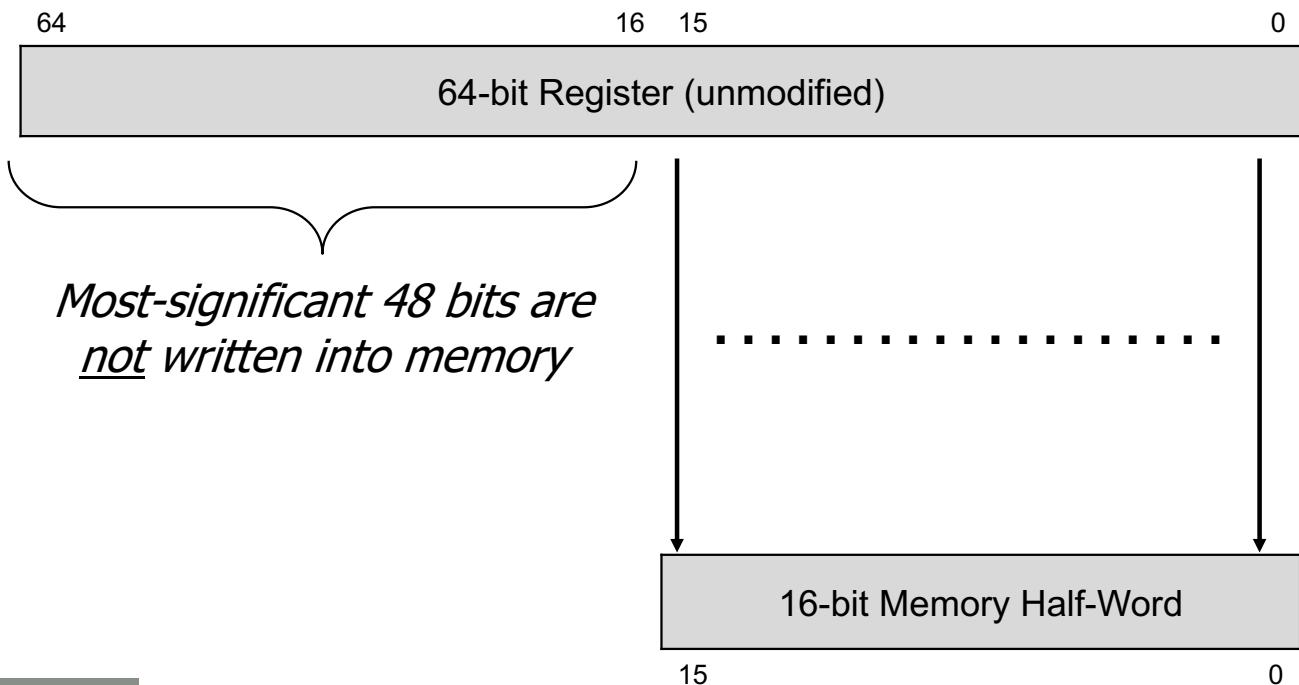
LDURSH: Load Register with Signed Half-Word



STURB: Store Register to Byte



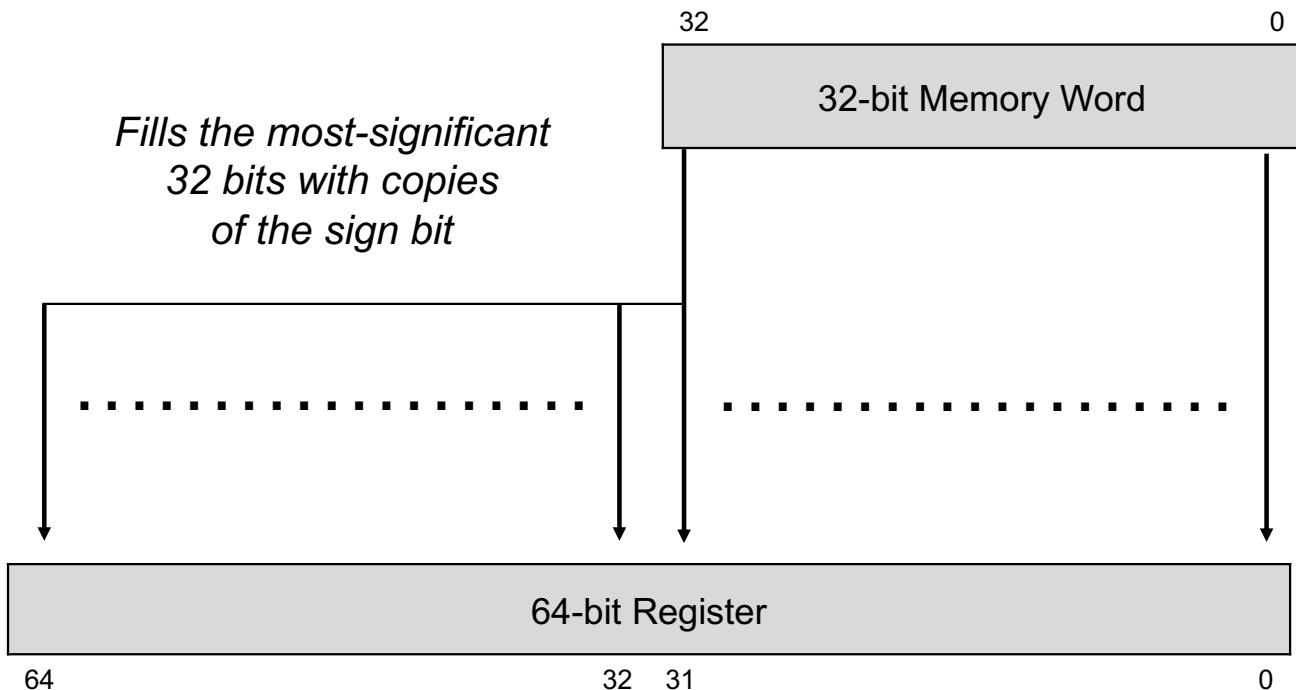
STURH: Store Register to Half-Word



LDURSW: Load Register with Signed Word

LDURW W0, a

*Fills the most-significant
32 bits with copies
of the sign bit*



LEGv8 Addressing Summary

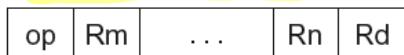
1. Immediate addressing



ADDI X0, X1, #5

No need to load operand first & then use it.

2. Register addressing



LDUR X4, [X0, X1]

Registers

Register

3. Base addressing



LDUR X4, [X0, #24]

Memory

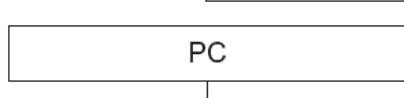


4. PC-relative addressing



CBNZ X0, 1000 (if X0 content NZ goto PC+1000)

Memory



Inter-Process Communication (IPC)

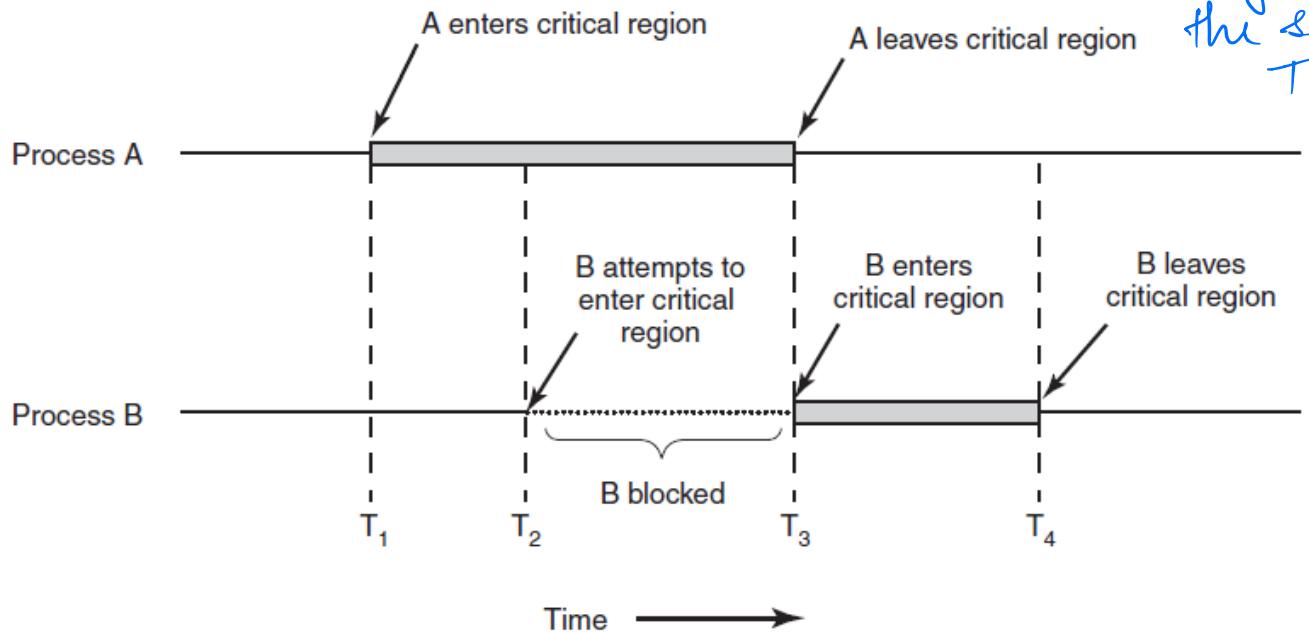
- Race Conditions
- Critical Regions
- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- The Producer-Consumer Problem
- Semaphores
- Mutexes
- Monitors

Critical Regions (recall)

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

Critical Regions (recall)

Mutual exclusion \Rightarrow No 2 processes in a critical region at the same time.



Synchronization (recall)

- Two processors (or processes, threads, etc. for that matter) sharing an area of memory
 - P1 writes, then P2 reads
 - Data race if P1 and P2 don't synchronize
 - Result depends of order of accesses
- Hardware support required
 - Atomic read/write memory operation
 - No other access (processor, process, or thread) to the location allowed between the read and write
- An atomic pair of instructions
 - E.g., atomic swap of register ↔ memory



Synchronization in LEGv8

- Load exclusive register: LDXR
- Store exclusive register: STXR
- To use:
 - Execute LDXR then STXR with same address
 - If there is an intervening change to the content of address, store fails
 - STXR fails with a return value of 1 stored in a register specified by the STXR instruction
 - Only register instructions allowed (not memory/data transfer instructions) on the loaded register in between LDXR and STXR.
 - ADD, SUB, LSL, etc.



Synchronization in LEGv8

- Example 1: atomic swap

X20 0x0000...008

X23 21 ←

y
x

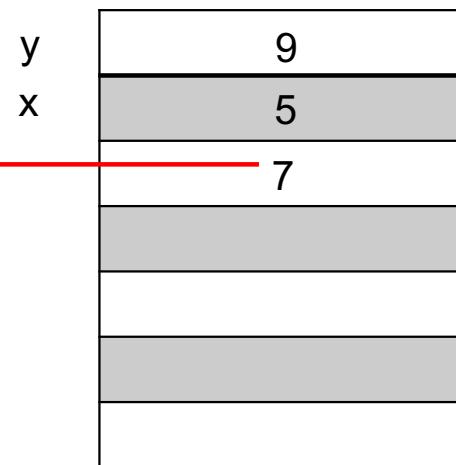
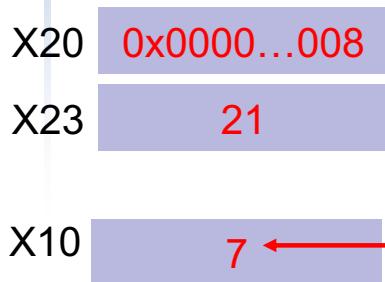
9
5
7

0x0000000000000000
0x0000000000000004
0x0000000000000008
0x000000000000000C
0x0000000000000010
0x0000000000000014

Synchronization in LEGv8

- Example 1: atomic swap

again: LDXR X10, [X20,#0] //load mem content pointed
to by [X20, #0] into X10



0x0000000000000000
0x0000000000000004
0x0000000000000008
0x000000000000000C
0x0000000000000010
0x0000000000000014

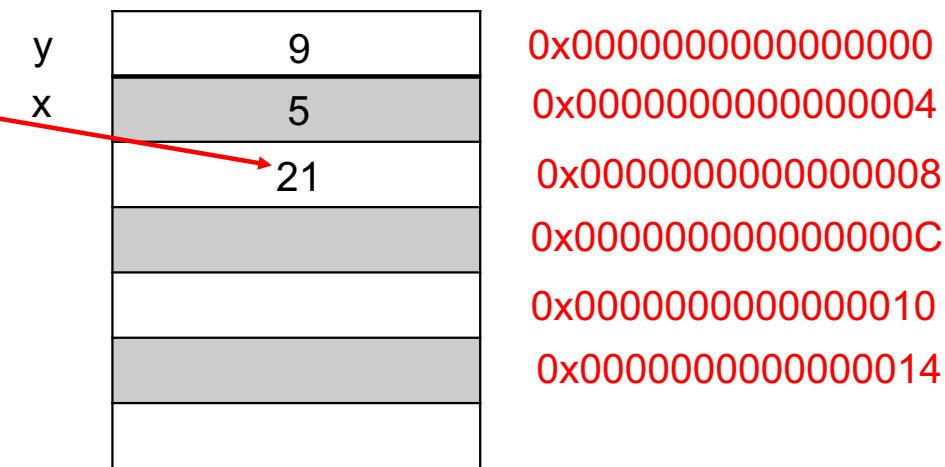
Synchronization in LEGv8

- Example 1: atomic swap

again: LDXR x10, [x20,#0]

STXR x23,x9,[x20] //Store x23 content into
mem location pointed by
x20 (AND) x9 <= status

X20	0x0000...008
X23	21
X10	7
X9	0x0000...000



Synchronization in LEGv8

- Example 1: atomic swap

again: LDXR x10, [x20,#0]

STXR x23,x9,[x20] //x9 = status

CBNZ x9, again //was it successful?

X20 0x0000...008

X23 21

X10 7

X9 0



HW reports that during the LDXR and STXR, nothing changed in that mem location (no access).

y	9	0x0000000000000000
x	5	0x0000000000000004
	21	0x0000000000000008
		0x000000000000000C
		0x0000000000000010
		0x0000000000000014

Synchronization in LEGv8

- Example 1: atomic swap

again: LDXR X10, [X20,#0]

STXR X23,X9,[X20,#0] // X9 = status

CBNZ X9, again // was it successful?

ADD X23,XZR,X10 // X23 <= loaded value

X20 0x0000...008

X23 7

X10 7

X9 0

y	9	0x0000000000000000
x	5	0x0000000000000004
	21	0x0000000000000008
		0x000000000000000C
		0x0000000000000010
		0x0000000000000014



Synchronization in LEGv8

- Example 2: lock (1 = locked , 0 = unlocked)

```
        ADDI X11,XZR,#1      // get ready to lock  
again: LDXR X10,[X20,#0] // read lock  
          CBNZ X10, again    // check if it is 0 yet  
          STXR X11, X9, [X20] // attempt to lock  
          CBNZ X9,again     // try again if fails  
          . . .             // can access/update  
          STUR XZR, [X20,#0] // free lock
```

X20	0x0000...014
X11	1
X10	0 = locked
X9	0= success

y	9	0x00000000
x	5	0x00000004
		0x00000008
		0x0000000C
		0x00000010
	0	0x00000014



Chapter 3

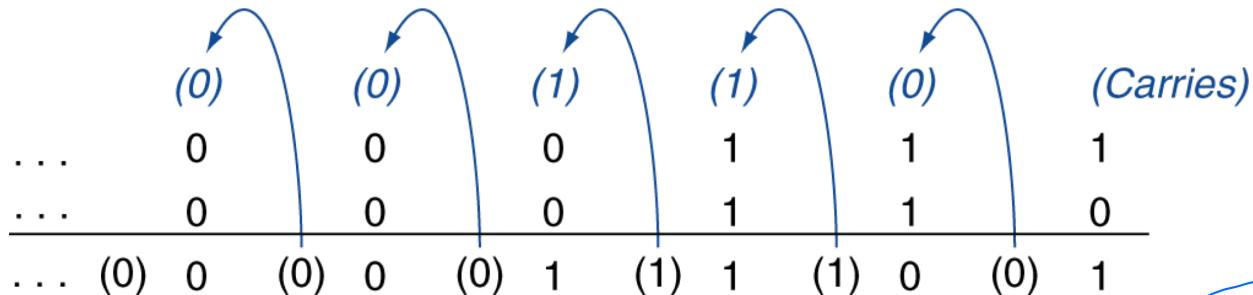
Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
 - Addition and subtraction
 - Multiplication and division
- Floating-point real numbers
 - Representation and operations

Integer Addition/Subtraction

- Example: $7 + 6$



- Overflow if result out of range

- Adding +ve and –ve operands?

- No overflow

- Adding two +ve operands?

- Overflow if result sign is 1 in MSB

- Adding two –ve operands?

- Overflow if result sign is 0 in MSB.

Copy

$$\begin{aligned}
 0+1 &= 0 \\
 1+1 &= 0 \\
 0+0 &= 0
 \end{aligned}$$

Integer Addition/Subtraction

- For subtraction, add negation of second operand
- Example: $7 - 6 = 7 + (-6)$

$$\begin{array}{r} +7: \quad 0000\ 0000 \dots 0000\ 0111 \\ -6: \quad 1111\ 1111 \dots 1111\ 1010 \\ \hline +1: \quad 0000\ 0000 \dots 0000\ 0001 \end{array}$$

2's complement of 6
→ add it.

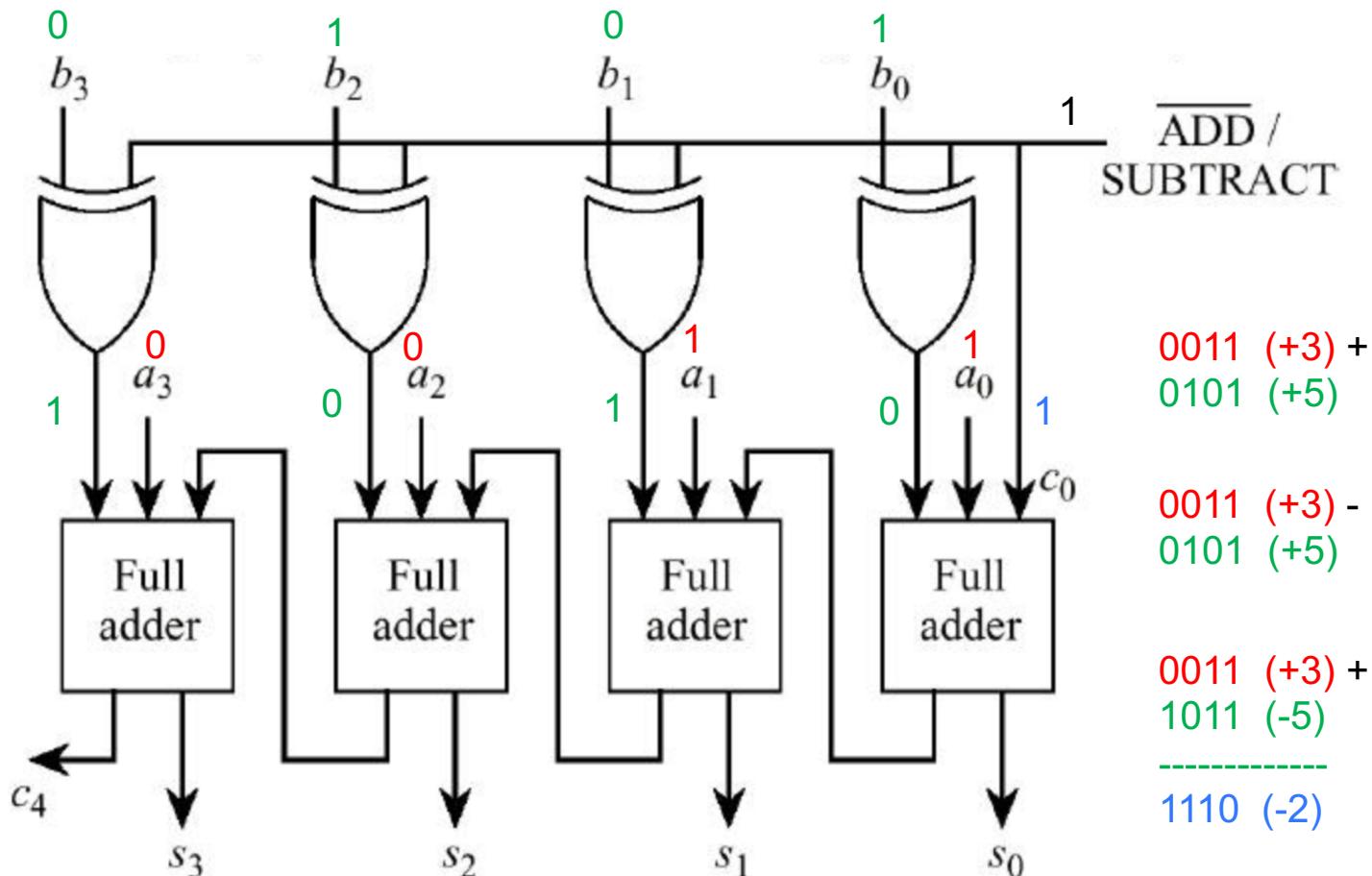
- Overflow if result out of range
 - Subtracting two +ve or two –ve operands
 - No overflow
 - Subtracting +ve from –ve operand
 - Overflow if result sign is 0
 - Subtracting –ve from +ve operand
 - Overflow if result sign is 1



Overflow Conditions for Addition and Subtraction

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Integer Add/Sub Circuit



2's complement:-

① Sign bits will be zero

② Add 1 to the
rightmost bit.

isse (+ve) No. (-ve)

Bar jata hai.

Changing the Sign in 2's Complement

Sign+Magnitude:

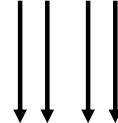
$$+5 = 0101$$

Change 1 bit

$$-5 = 1101$$

2's Complement:

$$+5 = 0101$$



Invert

$$\begin{array}{r} 1010 \\ +1 \\ \hline \end{array}$$

Increment

$$-5 = 1011$$

Easier Hand Method (example 1)

Step 2: Copy the inverse of the remaining bits.

$$\begin{array}{rcl} +4 & = & 0100 \\ & & \downarrow \quad \downarrow \\ -4 & = & 1100 \end{array}$$

- ↳ Jab tak 1 nahi aata chalte Raho Right to Left.
- ↳ Jaise hi copy kro uska everything same & uska aage ka inverse.

Step 1: Copy the bits from right to left, through and including the first 1.

Easier Hand Method (example 2)

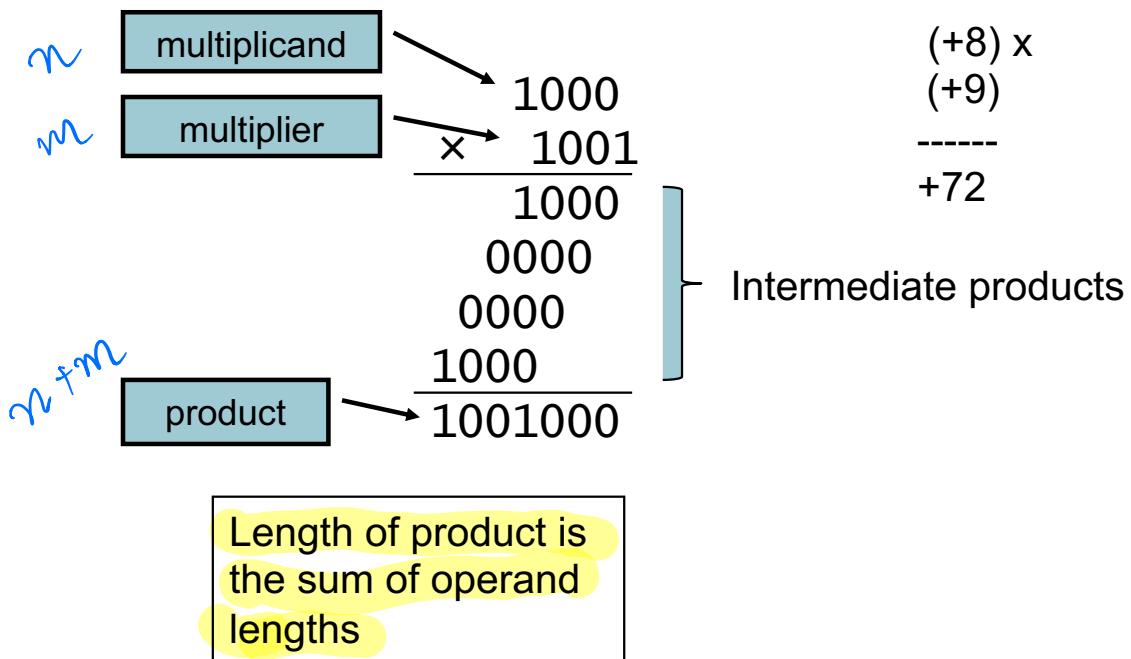
Step 2: Copy the inverse of the remaining bits.

$$\begin{array}{rcl} +5 & = & 0101 \\ & & \downarrow \quad \downarrow \\ -5 & = & 1011 \end{array}$$

Step 1: Copy the bits from right to left, through and including the first 1.

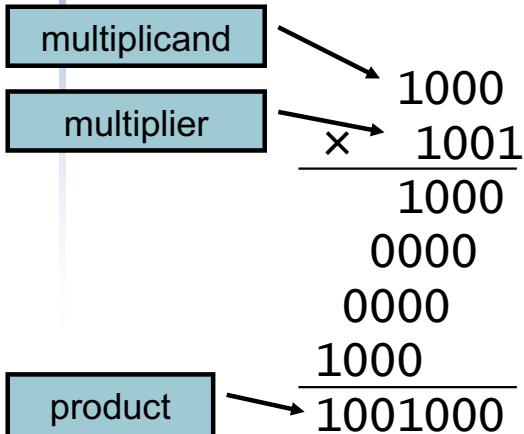
Integer Multiplication

- Start with long-multiplication approach

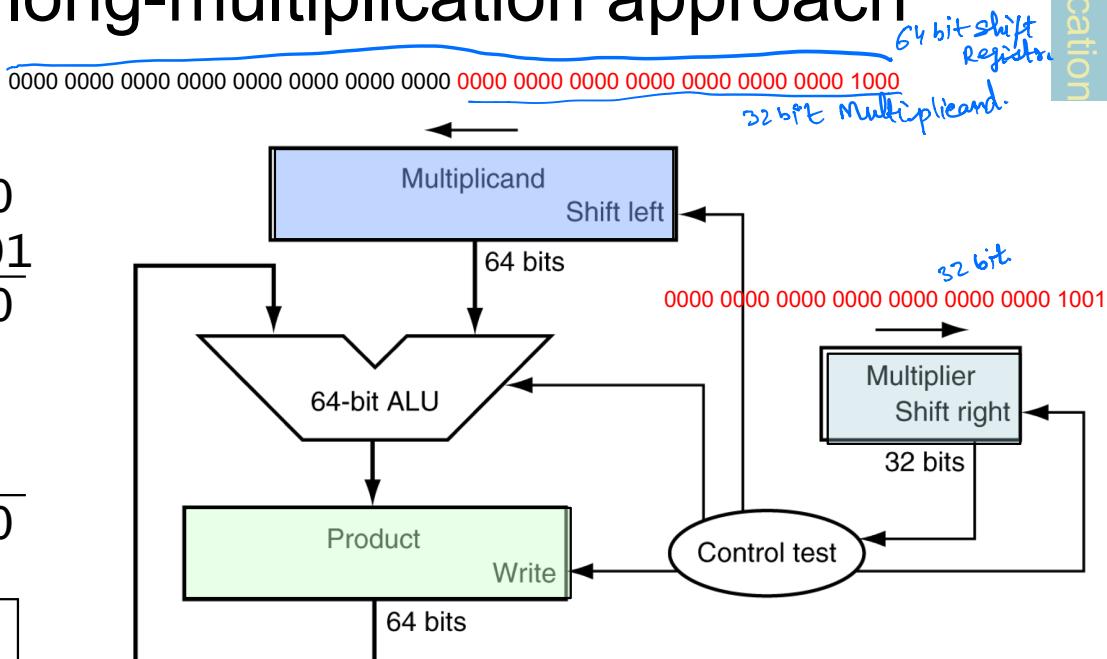


Integer Multiplication

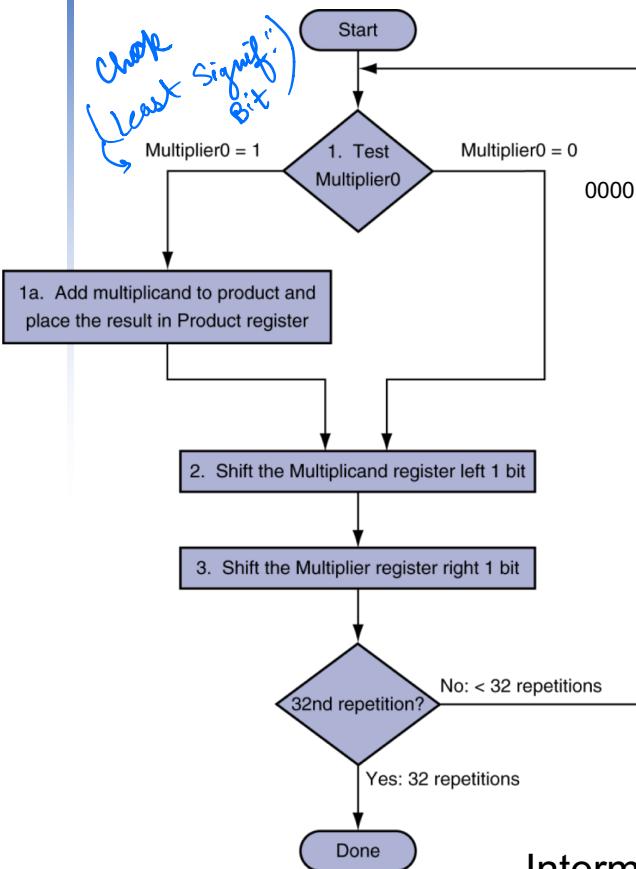
- Start with long-multiplication approach



Length of product is the sum of operand lengths

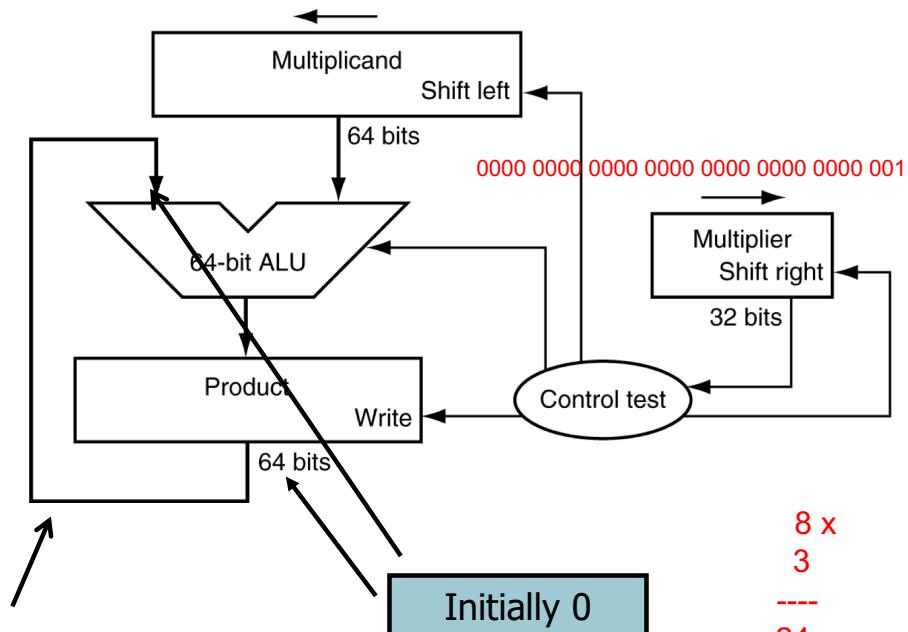


Multiplication Hardware

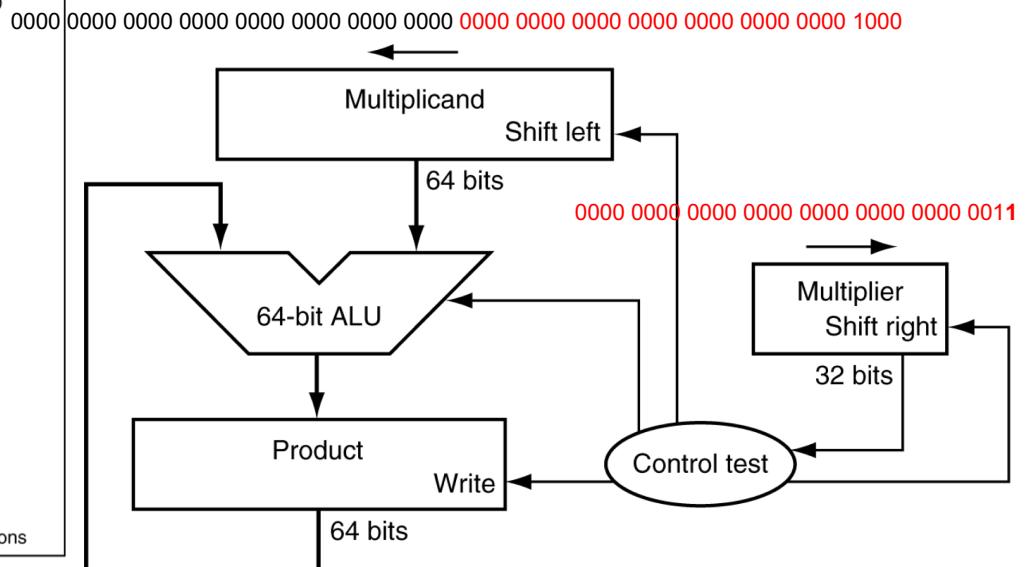
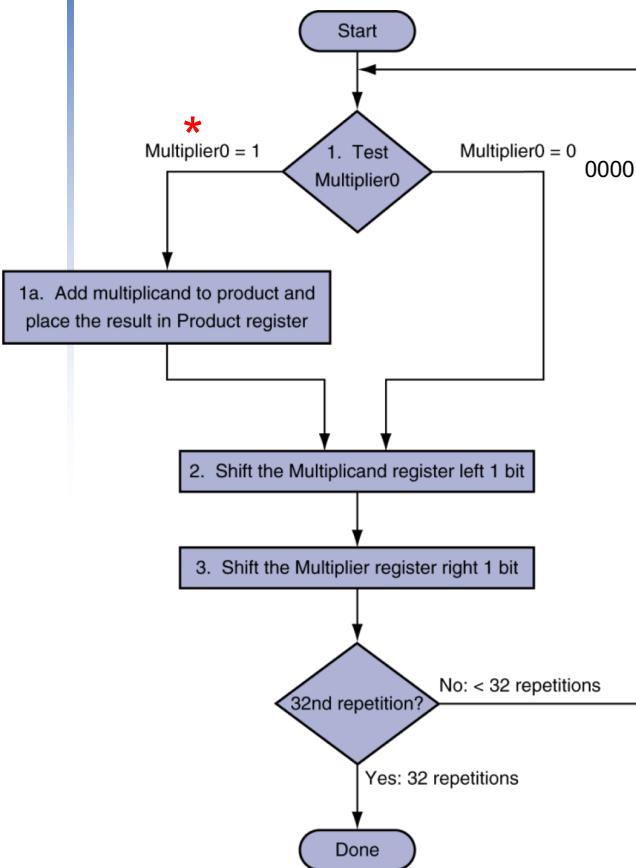


Intermediate products

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1000

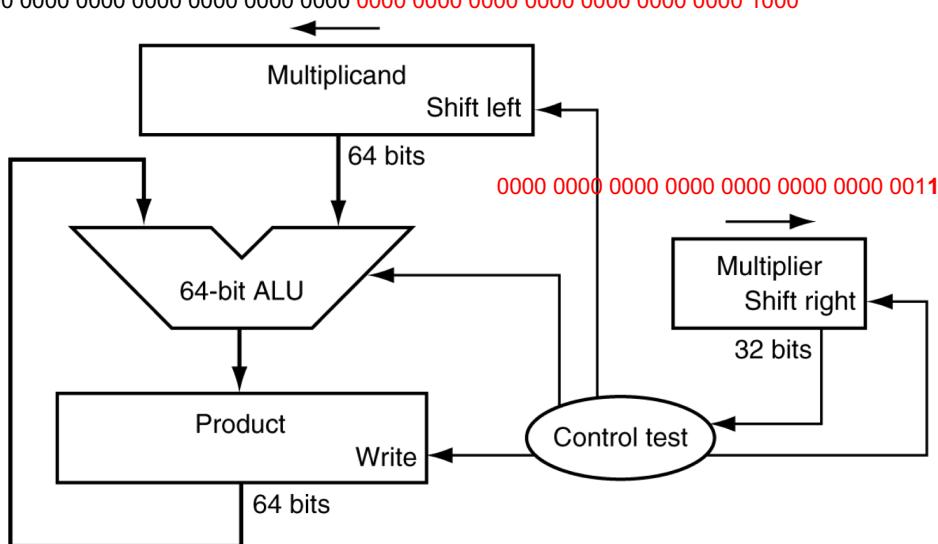
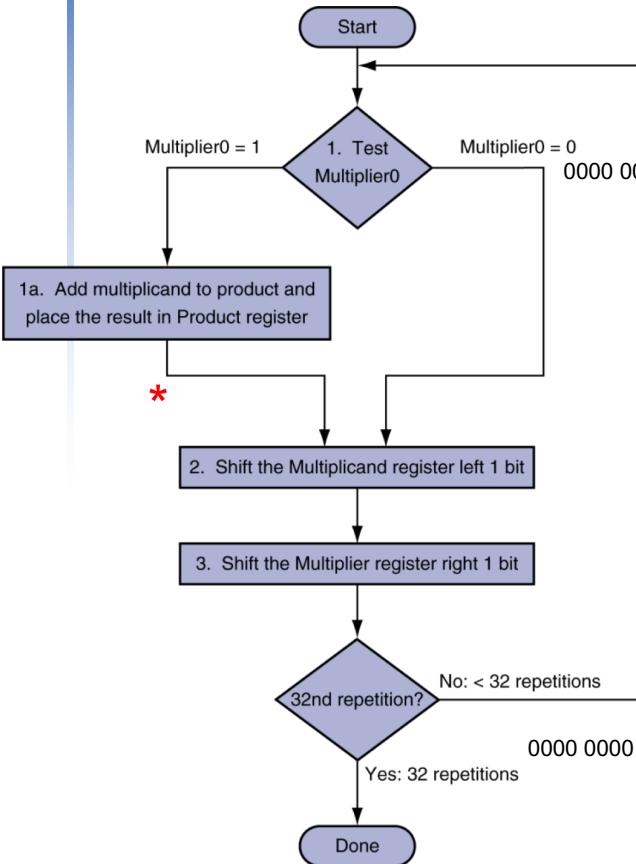


Multiplication Hardware



$$\begin{array}{r} 8 \times \\ 3 \\ \hline 24 \end{array}$$

Multiplication Hardware



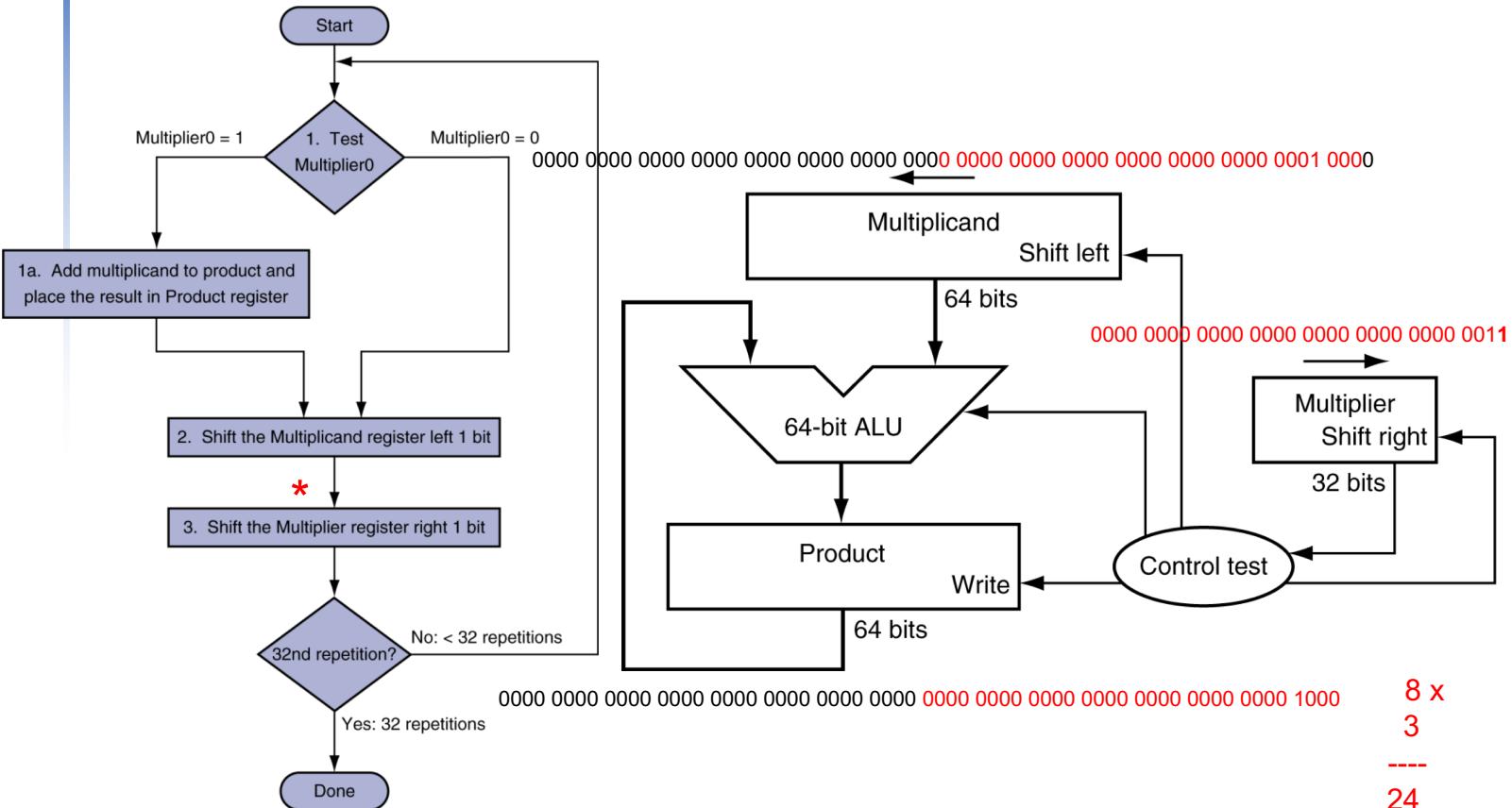
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1000

8 x

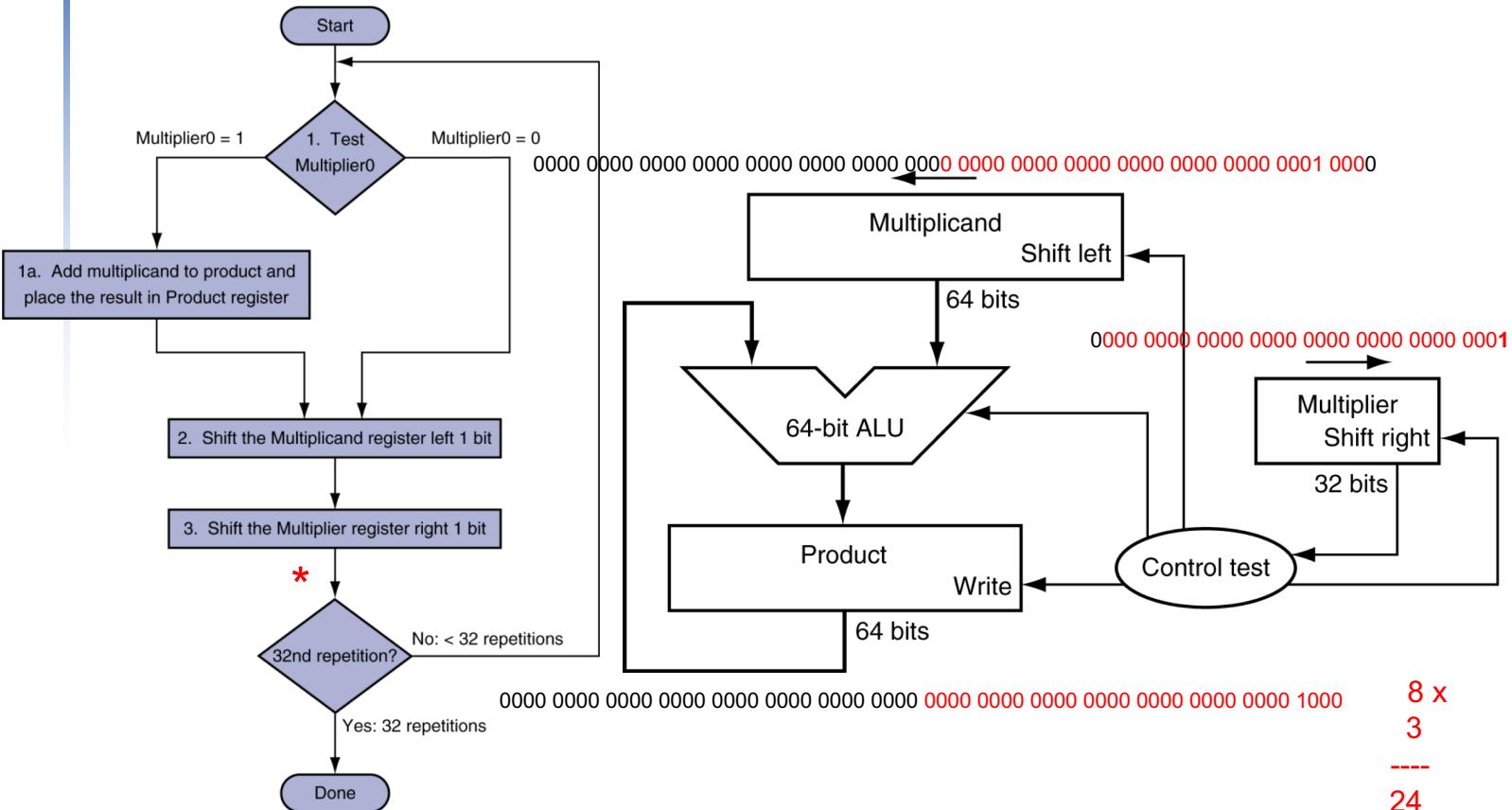
3

24

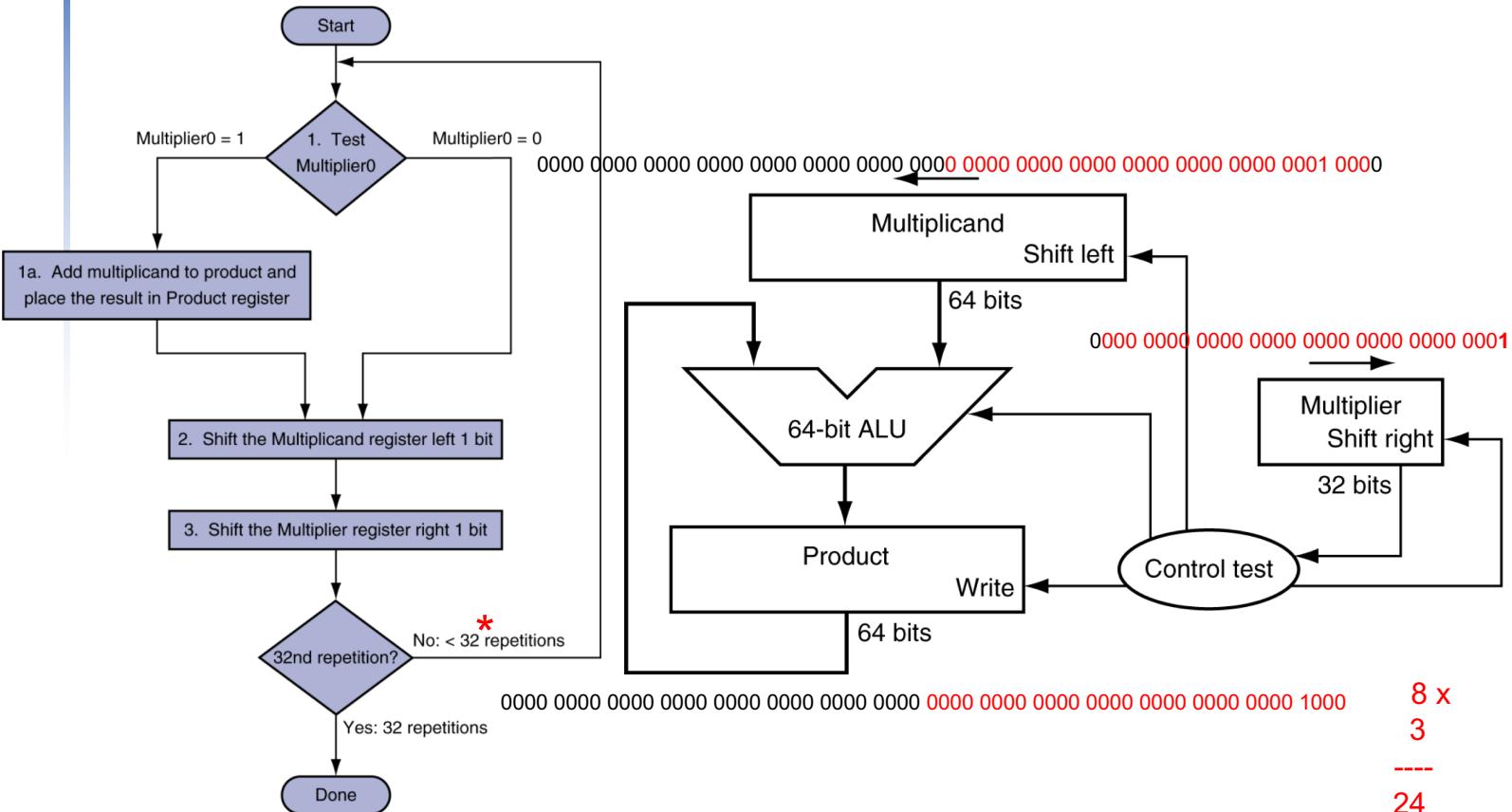
Multiplication Hardware



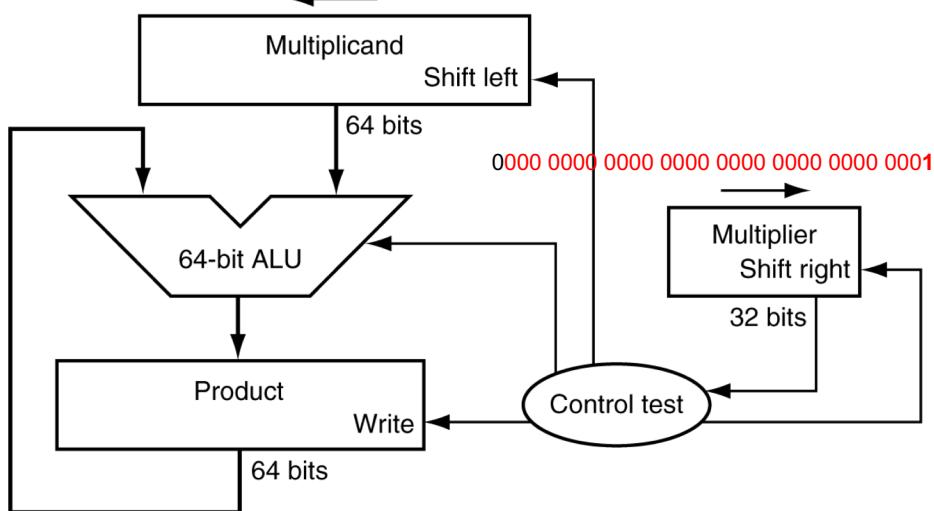
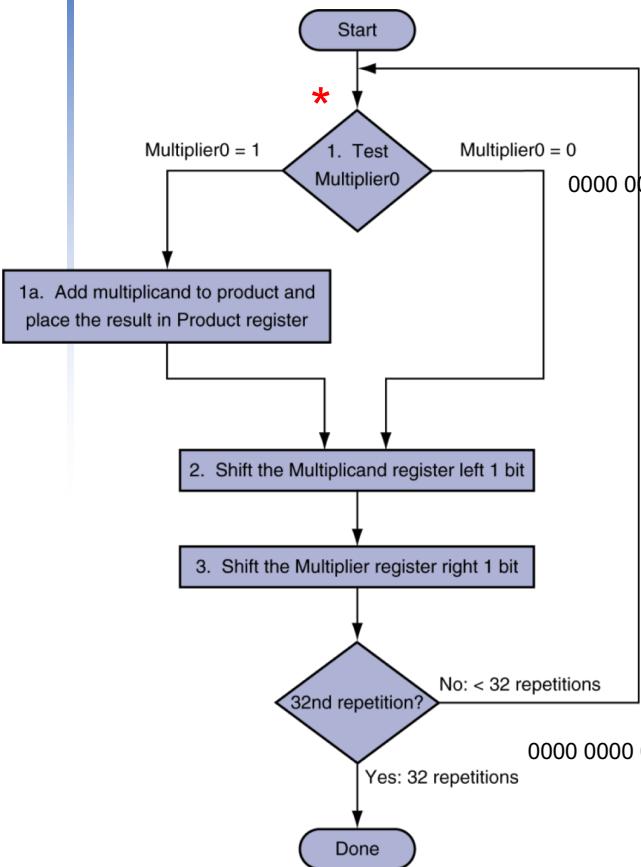
Multiplication Hardware



Multiplication Hardware



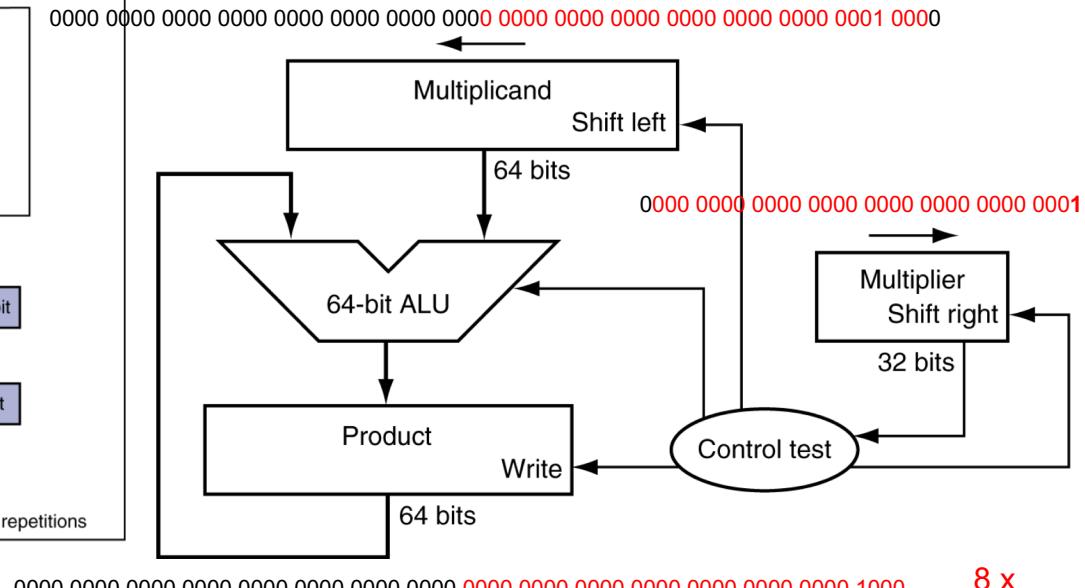
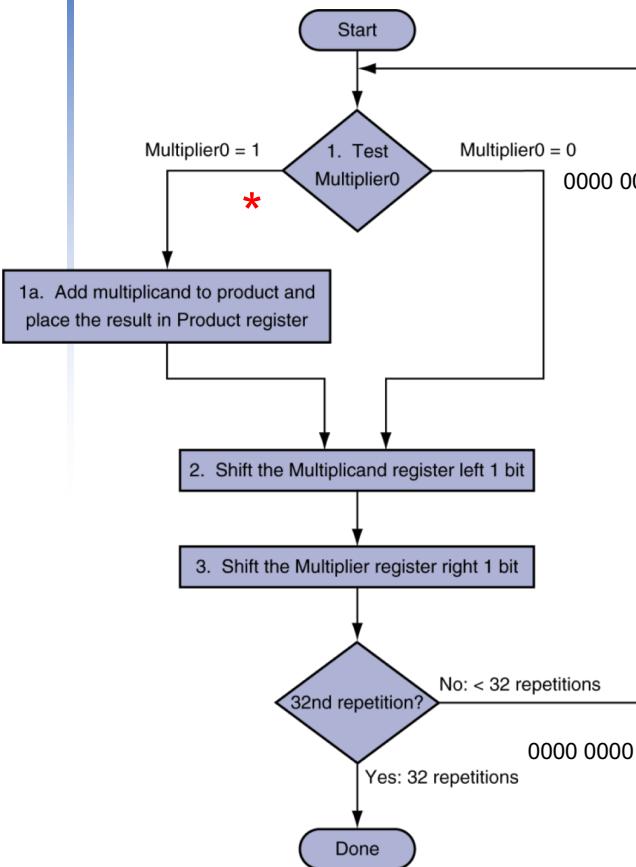
Multiplication Hardware



0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 0000

$$\begin{array}{r} 8 \times \\ 3 \\ \hline 24 \end{array}$$

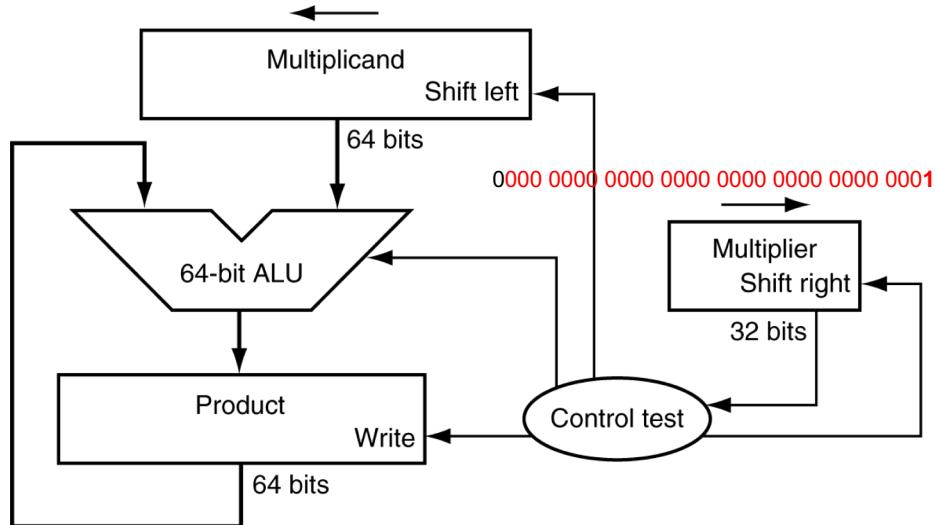
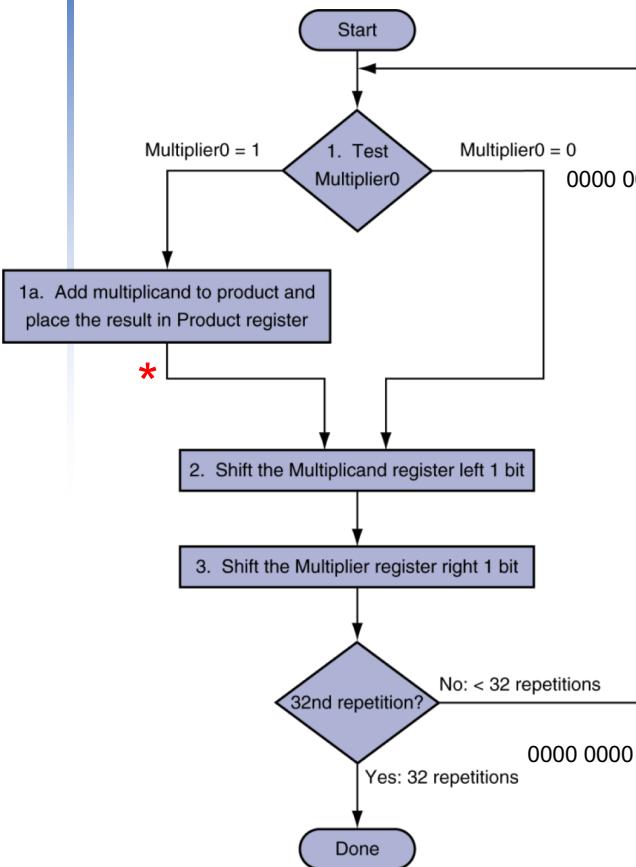
Multiplication Hardware



$$\begin{array}{r}
 8 \times \\
 3 \\
 \hline
 24
 \end{array}$$

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 0000
 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001

Multiplication Hardware



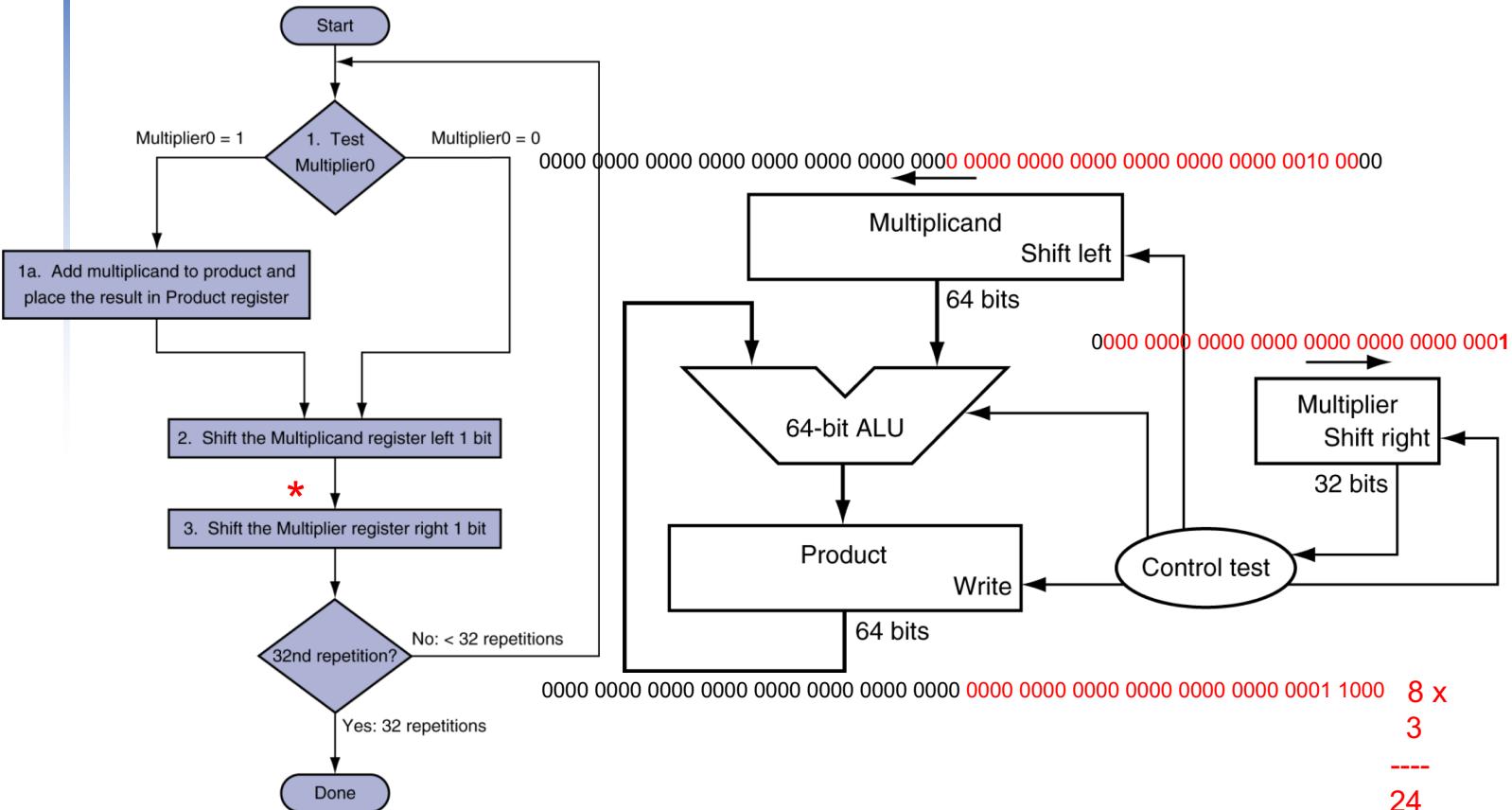
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 1000

$8 \times$

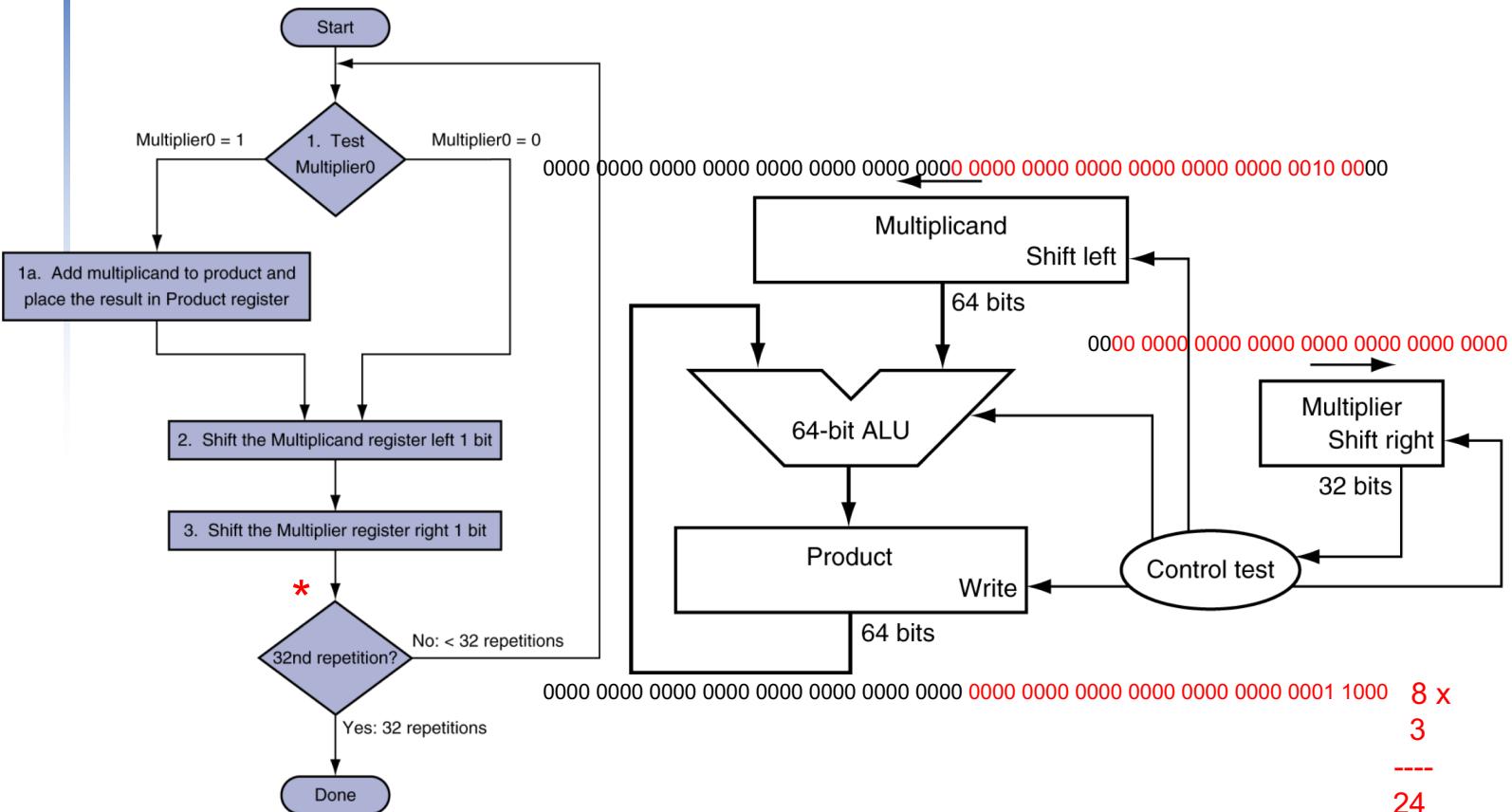
3

24

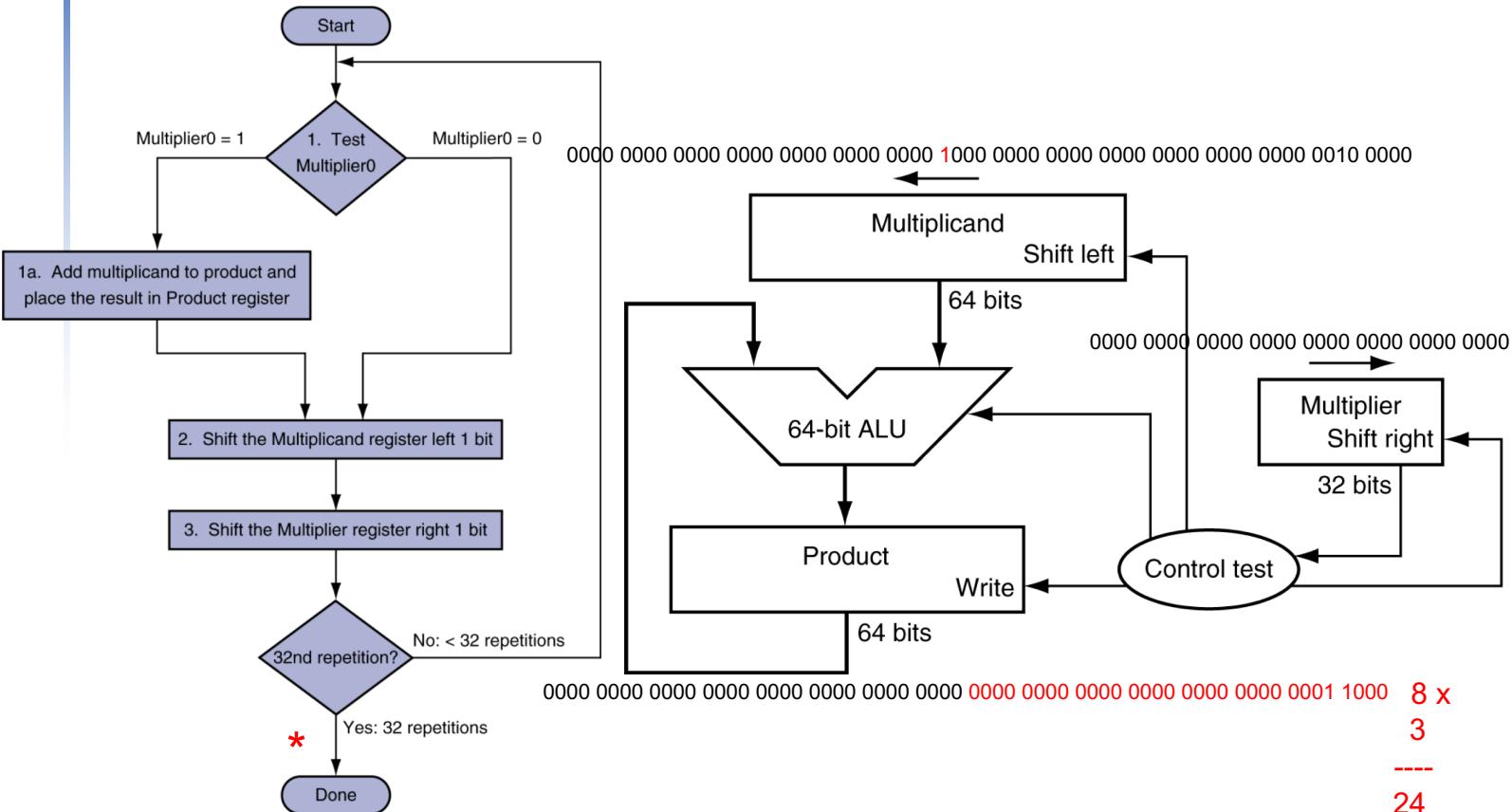
Multiplication Hardware



Multiplication Hardware



Multiplication Hardware



Multiplication Hardware

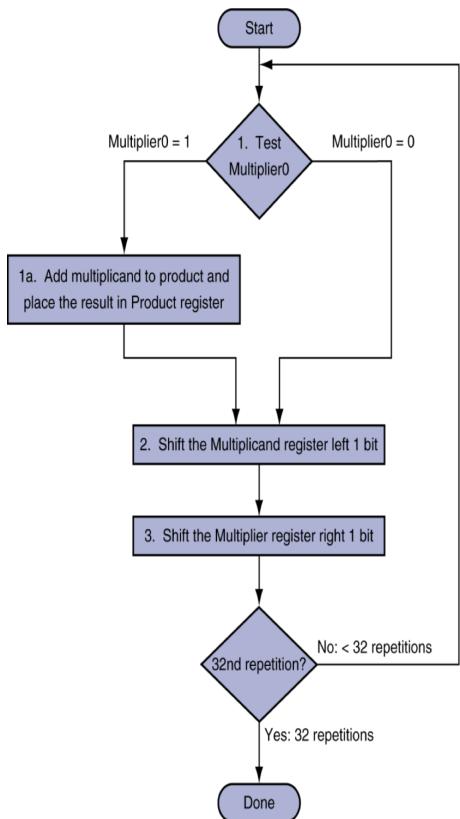


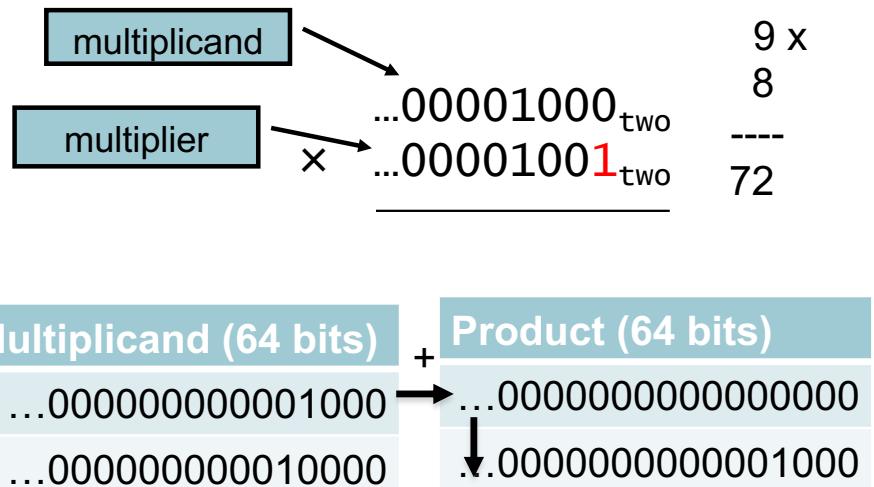
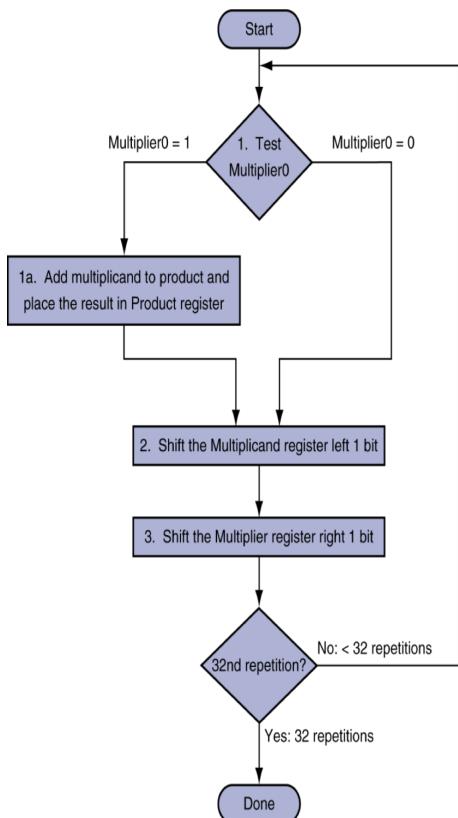
Diagram illustrating the multiplication of two binary numbers:

multiplicand ...00001000_{two}
multiplier x ...00001001_{two}

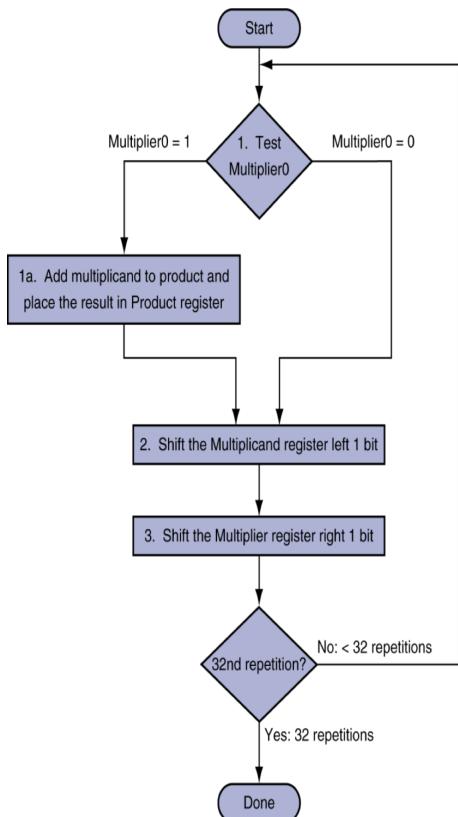
72

Multiplicand (64 bits)	Product (64 bits)
...0000000000001000	...0000000000000000

Multiplication Hardware



Multiplication Hardware



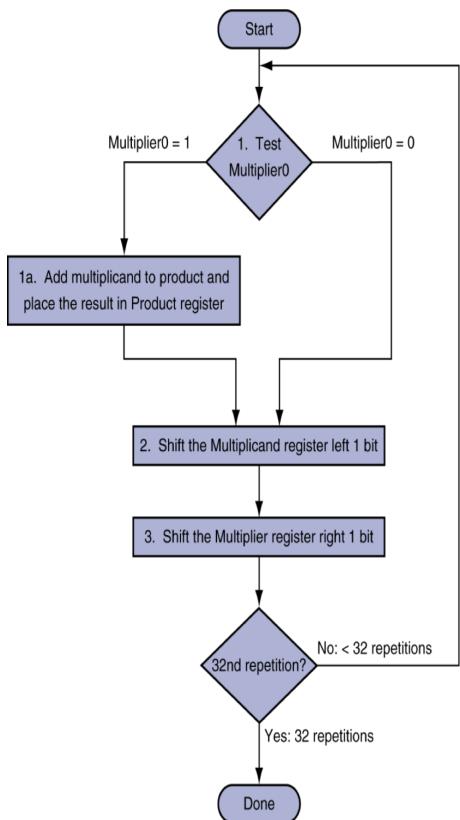
multiplicand

multiplier

$$\begin{array}{r} \dots00001000_2 \\ \times \dots00001001_2 \\ \hline 72 \end{array}$$

Multiplicand (64 bits)	Product (64 bits)
...00000000001000	...0000000000000000
...000000000010000	...00000000000000001000
...000000000100000	...00000000000000001000

Multiplication Hardware



multiplicand

multiplier

$$\begin{array}{r} 9 \times \\ 8 \\ \hline \end{array}$$

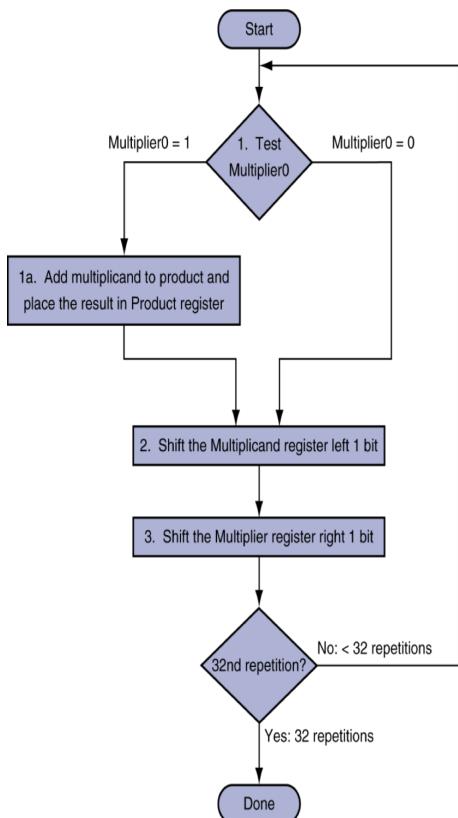
$\dots00001000_{\text{two}}$

$\times \dots00001001_{\text{two}}$

72

Multiplicand (64 bits)	Product (64 bits)
...00000000001000	...0000000000000000
...000000000010000	...00000000000000001000
...000000000100000	...00000000000000001000
...000000001000000	...00000000000000001000

Multiplication Hardware



multiplicand

multiplier

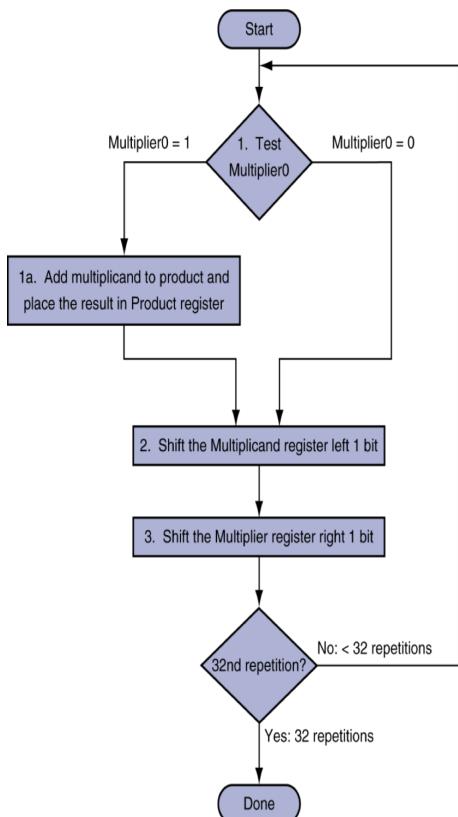
$$\begin{array}{r} 9 \times \\ 8 \\ \hline \end{array}$$

$$\begin{array}{r} \dots00001000_2 \\ \times \dots00001001_2 \\ \hline \end{array}$$

$$72$$

Multiplicand (64 bits)	Product (64 bits)
...00000000001000	...0000000000000000
...000000000010000	...00000000000000001000
...000000000100000	...00000000000000001000
...000000001000000	...00000000000000001000

Multiplication Hardware



A binary multiplication diagram showing 9×8 . The multiplicand is $\dots00001000_2$ and the multiplier is $\dots00001001_2$. The result is $\underline{\dots00001000_2 \dots00001001_2} = 72$.

Multiplicand (64 bits)	Product (64 bits)
$\dots00000000001000$	$\dots0000000000000000$
$\dots000000000010000$	$\dots00000000000000001000$
$\dots000000000100000$	$\dots00000000000000001000$
$\dots000000001000000$	$\dots00000000000000001000$
	$\begin{array}{r} + \\ \dots00000000000000001000 \\ \hline \dots0000000001001000 \end{array}$

LEGv8 Multiplication

- Three multiply instructions:
 - MUL: multiply
 - Two 64-bit registers can be multiplied to produce a 64-bit result: Gives the lower 64 bits of the product
 - SMULH: signed multiply returning high half
 - Gives the upper 64 bits of the 128-bit product, assuming the operands are signed
 - UMULH: unsigned multiply returning high half
 - Gives the upper 64 bits of the 128-bit product, assuming the operands are unsigned

LEGv8 Multiplication

MUL Wd, Wn, Wm ; 32-bit general registers

MUL Xd, Xn, Xm ; 64-bit general registers

Where:

Wd

Is the 32-bit name of the general-purpose destination register.

Wn

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm

Is the 32-bit name of the second general-purpose source register holding the multiplier.

Xd

Is the 64-bit name of the general-purpose destination register.

Xn

Is the 64-bit name of the first general-purpose source register holding the multiplicand.

Xm

Is the 64-bit name of the second general-purpose source register holding the multiplier.

Operation

Rd = Rn * Rm, where R is either W or X.

LEGv8 Multiplication

SMULH

Signed Multiply High.

Syntax

SMULH Xd , Xn , Xm

Where:

Xd

Is the 64-bit name of the general-purpose destination register.

Xn

Is the 64-bit name of the first general-purpose source register holding the multiplicand.

Xm

Is the 64-bit name of the second general-purpose source register holding the multiplier.

Operation

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

$Xd = \text{bits}\langle 127:64 \rangle \text{ of } Xn * Xm.$

LEGv8 Multiplication

UMULH

Unsigned Multiply High.

Syntax

UMULH Xd , Xn , Xm

Where:

Xd

Is the 64-bit name of the general-purpose destination register.

Xn

Is the 64-bit name of the first general-purpose source register holding the multiplicand.

Xm

Is the 64-bit name of the second general-purpose source register holding the multiplier.

Operation

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

$Xd = \text{bits}\langle 127:64 \rangle \text{ of } Xn * Xm.$



LEGv8 Multiplication

Example:

Real result of the two unsigned numbers

0xFFFF_FFFF_FFFF_FFFF ($2^{64} - 1$)

multiplied by

0x2 (2)

Result can not be represented by 64 bits and
will overflows to the higher 64 half

Result =

0x0000_0000_0000_0010_0000_0000_99D5_5C00

Assumptions:

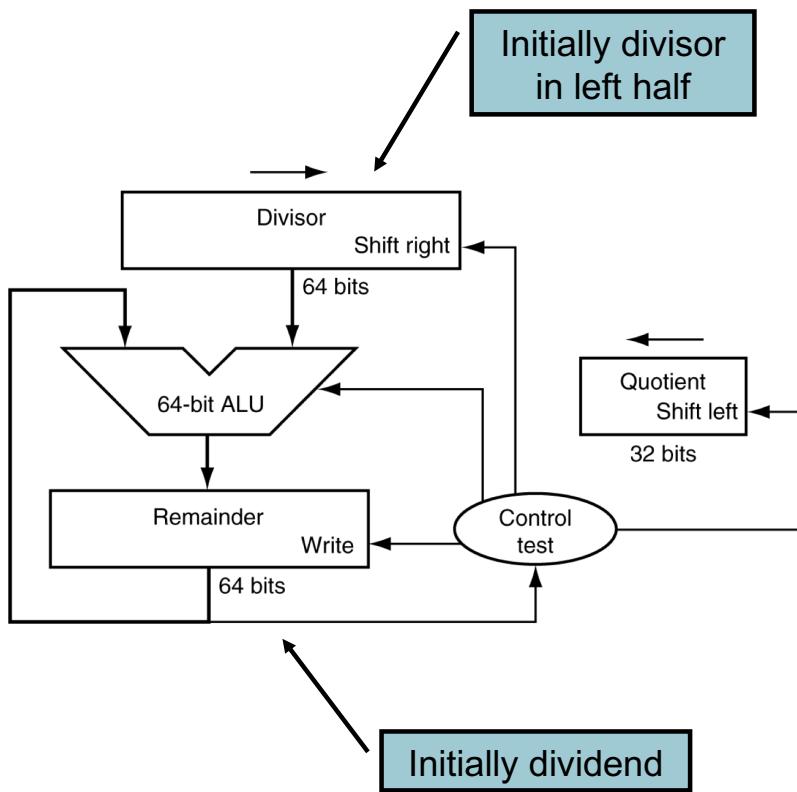
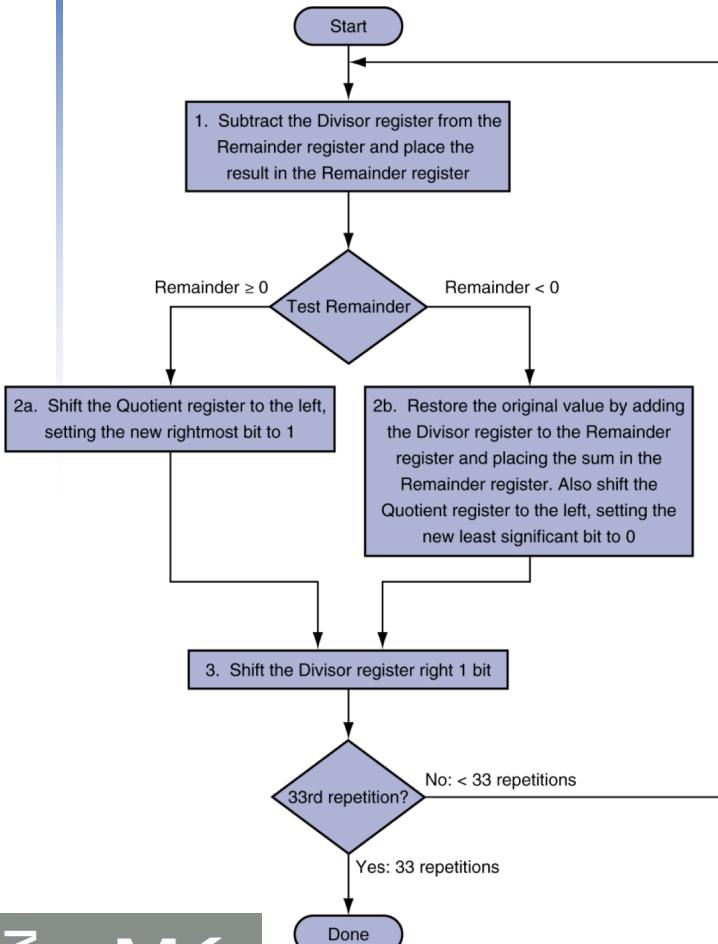
X0 = #0xFFFFFFFFFFFFFFF

X1 = #0x2

```
MUL X2, X0, X1 // X2 = 0x0000000099D55C00
UMULH X3, X0, X1 // X3 = 0x0000000000000010
```



Division Hardware



LEGv8 Division

- Two instructions:
 - SDIV (signed divide)
 - UDIV (unsigned divide)
- Examples:
 - UDIV W0, W1, W2 // $W0 = W1 / W2$ (unsigned, 32-bit divide)
 - SDIV X0, X1, X2 // $X0 = X1 / X2$ (signed, 64-bit divide)

Floating Point (recall)

- Representation for non-integral numbers
 - Including very small and very large numbers
- Like scientific notation

■ -2.34×10^{56}

normalized

■ $+0.002 \times 10^{-4}$

not normalized

■ $+987.02 \times 10^9$

ER & No. of leading zeros.
it should have a point.

- In binary

■ $\pm 1.xxxxxxx_2 \times 2^{yyyy}$

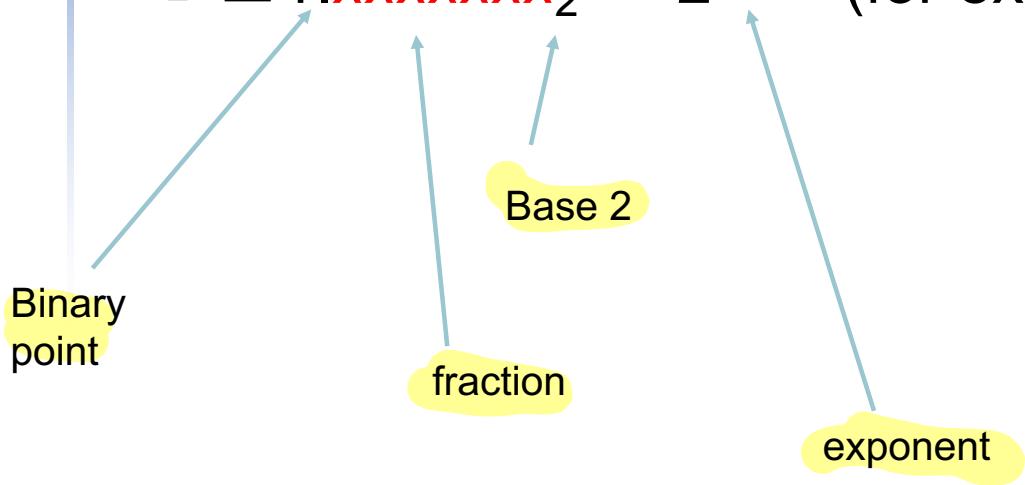
Normalized

- Types in C: float (single-precision FP) and double (double-precision FP)

Floating Point (recall)

- In binary

- $\pm 1.xxxxxxx_2 \times 2^{yyy}$ (for example, $1.0_2 \times 2^{-1}$)



A tradeoff between precision and range.

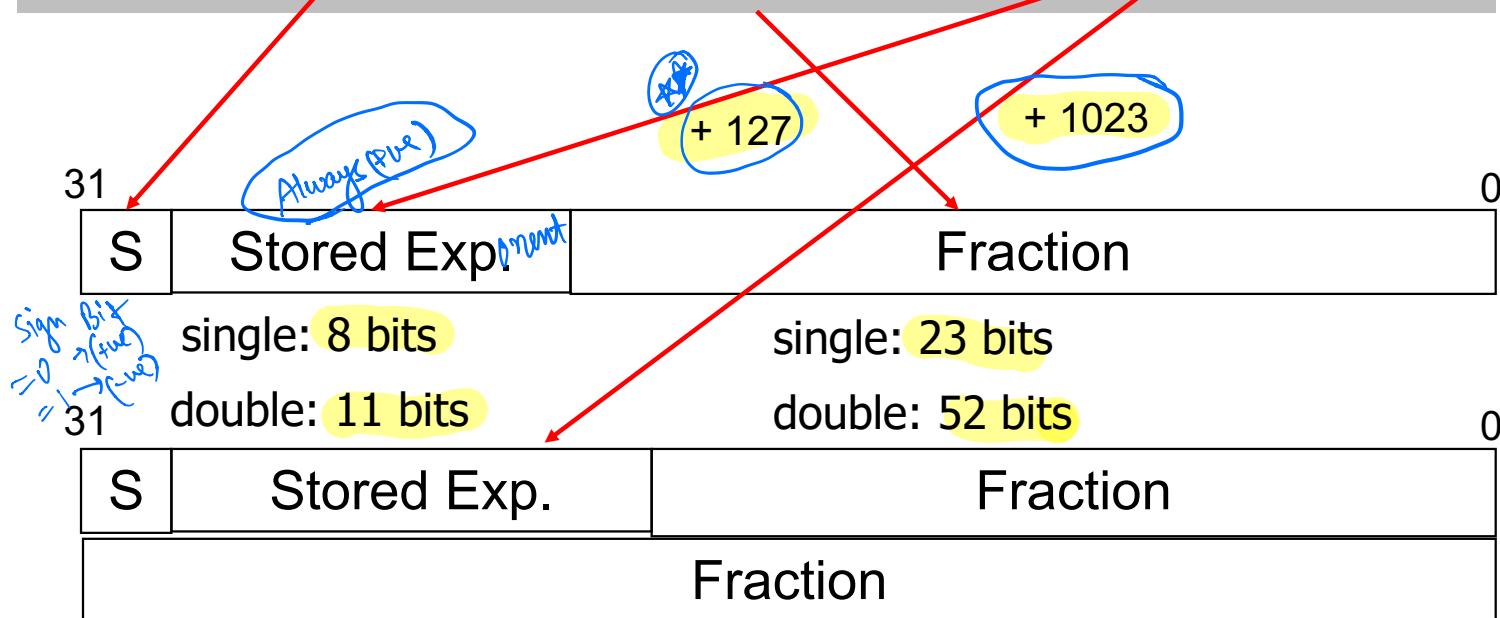
Floating Point Standard (Recall)

- Defined by **IEEE Standard 754 -1985**
- Developed in response to divergence of representations
 - Portability issues for scientific code
- Now almost universally adopted
- Two representations
 - Single precision (32-bit representation) *(In class.)*
 - Double precision (64-bit representation) *X*



IEEE Floating-Point Format (recall)

$$x = (-1)^s \times (1 + \text{Fraction}) \times 2^{(\text{Actual Exponent})}$$



Floating-Point Example

What is the IEEE 754 single precision representation of the binary number 111111.01×2^0 ?

Normalize the binary number first:

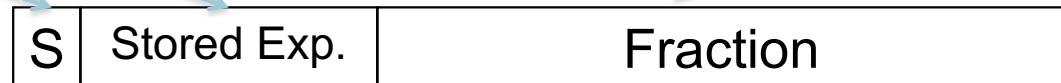
=> move binary point 5 to the left $1.\textcolor{green}{1111101} \times 2^5$ (sign positive => S = **0**)

Exponent (stored) = Actual Exponent (5) + bias

=> Stored Exp. = $5 + 127 = \textcolor{red}{132}$

Final bit pattern:

0 **1000 0100 1111 1010** 0000 0000 0000 000



Floating-Point Example

What is the IEEE 754 single precision representation of the decimal number +97.25?

Recall: Decimal to Binary Conversion (Whole Part: Repeated Division)

Convert 97_{10} to base 2

$97 \div 2 \rightarrow$	quotient = 48,	remainder = 1 (LSB)
$48 \div 2 \rightarrow$	quotient = 24,	remainder = 0.
$24 \div 2 \rightarrow$	quotient = 12,	remainder = 0.
$12 \div 2 \rightarrow$	quotient = 6,	remainder = 0.
$6 \div 2 \rightarrow$	quotient = 3,	remainder = 0.
$3 \div 2 \rightarrow$	quotient = 1,	remainder = 1.
$1 \div 2 \rightarrow$	quotient = 0 (Stop)	remainder = 1 (MSB)

Result =
2

Recall: Decimal to Binary Conversion (Fractional Part: Repeated Multiplication)

Example: 0.25_{10}

$.25 \times 2 \rightarrow 0.5$ (fractional part = .5, whole part = 0)

$.5 \times 2 \rightarrow 1.0$ (fractional part = .0 (stop), whole part = 1)

.0

.01

Result = .01₂

Floating-Point Example

What is the IEEE 754 single precision representation of the decimal number 97.25?

01100001.01_2

$01100001.01_2 \times 2^0$

$01.10000101_2 \times 2^6$

Final bit pattern:

0 **1000 0101 1000 0101** 0000 0000 0000 000

More Floating-Point Examples

S(1) Exp+127(8) (implied 1).Significand(23)

2.000	0	10000000	(1).00000000000000000000000000000000
1.000	0	01111111	(1).00000000000000000000000000000000
0.750	0	01111110	(1).10000000000000000000000000000000
0.500	0	01111110	(1).00000000000000000000000000000000
0.000	0	00000000	(0).00000000000000000000000000000000
-0.500	1	01111110	(1).00000000000000000000000000000000
-0.750	1	01111110	(1).10000000000000000000000000000000
-1.000	1	01111111	(1).00000000000000000000000000000000
-2.000	1	10000000	(1).00000000000000000000000000000000

More Floating-Point Examples

Representing decimal 2.0 in IEEE 754 single-precision format:

S (1) Exp+127 (8u) (implied 1) . Significand (23)

0 10000000 (1) . 0000000000000000000000000000000

$2.0_{10} \Rightarrow 10_2$ (a +ve number so S = 0)

Then normalize it $\rightarrow 1.0 \times 2^1$

Actual exponent = 1 so the stored exponent in single precision is calculated as:

$1+127 = 128_{10}$ or 10000000_2

More Floating-Point Examples

Representing decimal 0.75 in IEEE 754 single-precision format:

S (1) Exp+127 (8u) (implied 1) .Significand(23)

0 01111110 (1) .1000000000000000000000000000000

$0.75 \Rightarrow 0.11_2$ (a +ve number so S = 0)

Normalize it $\rightarrow 1.1 \times 2^{-1}$

Actual exponent = -1 so the stored exponent in single precision is calculated as:

$-1 + 127 = 126_{10}$ or 01111110_2



More Floating-Point Examples

S (1) Exp+127 (8u) (implied 1) .Significand(23)

-0.500 1 01111110 (1).0000000000000000000000000

$-0.50_{10} \rightarrow -0.1_2$ (a -ve number so S = 1)

Normalize it $\rightarrow -1.0 \times 2^{-1}$

Actual exponent = -1 so the stored exponent in single precision is calculated as:

$$-1 + 127 = 126_{10} \text{ or } 01111110_2$$

Single-precision Floating-point Representation of Zero and One

S (1) Exp+127 (8) (implied 1).Significand (23)

0.000 0 00000000 (1).0000000000000000000000000000000

Stored exponent = 127 (bias) + Actual exponent

0 = 127 + Actual Exponent

Actual Exponent = -127

1.0×2^{-127} does not represent 1.

Another question that comes to mind is this:

If we always have an implied 1 how do we represent a 0 in single-precision IEEE 754 format?

Single-precision Floating-point Representation of Zero and One

S(1) Exp+127(8) (implied 0).Significand(23)

0.000 0 00000000 → (0).00000000000000000000000000000000

Implied 0, if and only if, both stored exponent and the fractional part are zero.

S(1) Exp+127(8) (implied 1).Significand(23)

0.000 0 01111111 (1).00000000000000000000000000000000

$1.000_{10} \rightarrow 0001_2$ then normalize it as 1.0×2^0

Actual exponent = 0 so the stored storage in single precision becomes 0+127 = 127 or 01111111

FP Instructions in LEGv8

- Separate FP registers
 - 32 single-precision: S0, ..., S31
 - 32 double-precision: D0, ..., D31
- FP instructions operate only on FP registers
 - Programs generally don't do integer ops on FP data, or vice versa
- FP load and store instructions
 - LDURS, LDURD
 - STURS, STURD



FP Instructions in LEGv8

- Single-precision floating-point arithmetic
 - FADDS, FSUBS, FMULS, FDIVS
 - e.g., FADDS S2, S4, S6
- Double-precision arithmetic
 - FADDD, FSUBD, FMULD, FDIVD
 - e.g., FADDD D2, D4, D6
- Single- and double-precision comparison
 - FCMPS, FCMPD
 - Sets or clears FP condition-code bits
- Branch on FP condition code true or false
 - B.cond



FP Example: ° F to ° C

- C code:

```
float f2c (float fahr) {  
    return ((5.0/9.0)*(fahr - 32.0));  
}
```

- `fahr` in S0, result in S0, and the starting address of constants in memory is in X27

- Compiled LEGv8 code:

f2c:

```
LDURS S16, [X27,const5]      // S16 = 5.0 (5.0 in memory)  
LDURS S18, [X27,const9]      // S18 = 9.0 (9.0 in memory)  
FDIVS S16, S16, S18          // S16 = 5.0 / 9.0  
LDURS S18, [X27,const32]     // S18 = 32.0  
FSUBS S18, S0, S18           // S18 = fahr - 32.0  
FMULS S0, S16, S18           // S0 = (5/9)*(fahr - 32.0)  
BR LR                         // return
```



Concluding Remarks

- Bits have no inherent meaning
 - Interpretation depends on the instructions applied
- Computer representations of numbers
 - Finite range and precision
 - Need to account for this in programs

Concluding Remarks

- ISAs support arithmetic
 - Signed and unsigned integers
 - Floating-point approximation to reals
- Bounded range and precision
 - Operations can overflow

Ch - 4 (Part 1)

Start from Week 5

✓ edit.

A. Performance of a comp." depends on:-

Affected By

Algo, ISA & Compiler ① Instruction Count (IC)
CPU H/w ② Clock Cycles Per Instr." (CPI)
 ③ Clock Cycle Time (T_c)

$$\text{Execution Time (sec/program)} = \frac{\text{Instructions} \times \text{Clock Cycles}}{\text{Program Instn.}} \times \frac{\text{Seconds}}{\text{clock cycle}}$$

B. 2 LEGV8 Implementations :-

① Simplified Version (Single Cycle)

- Not Practical
- Not Efficient.

→ 1 Instn." / cycle. Datapath

② Realistic pipelined Version

- Multiple Instr." Executed at the same time.
- ∴ ↑ing Throughput

Simple ISA Subset uses :-

- Memory Reference \rightarrow LDUR, STUR
- Arithmetic/Logical \rightarrow ADD, SUB, AND, OR.
- Control Transfer \rightarrow CBNZ, B.

* We have our instructions in the Text field.

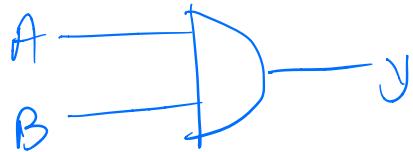
for Every Instruction :- (few steps that are same)

- ① Set the PC (Program Counter) to the memory location where the 1st instruction is going to be fetched.
- ② Based on the instr., what it wants, we fetch & load the operands.

Cont'd from Pg 6 of Slides.

Gates \Rightarrow

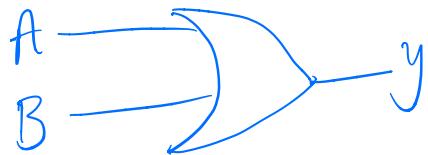
① AND \rightarrow



$$Y = A \cdot B \quad \boxed{(\text{multiply})}$$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

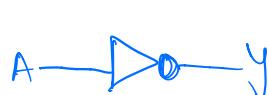
② OR \rightarrow



$$Y = A + B \quad \boxed{(\text{Add})}$$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

③ Not \rightarrow



$$Y = \bar{A} \quad (\text{Invert})$$

A	Y
0	1
1	0

④ XOR \rightarrow

$$\begin{aligned} A \oplus B \\ \overline{A \cdot B} = Y \end{aligned}$$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0



Chapter 4

The Processor

Introduction

- The performance of a computer is determined by three key factors:
 1. Instruction Count (IC) → Determined by algorithm, ISA and Compiler
 2. Clock Cycles per Instruction (CPI)
 3. Clock Cycle Time (T_c) → Determined by CPU H/W

$$\text{Execution Time (seconds/program)} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

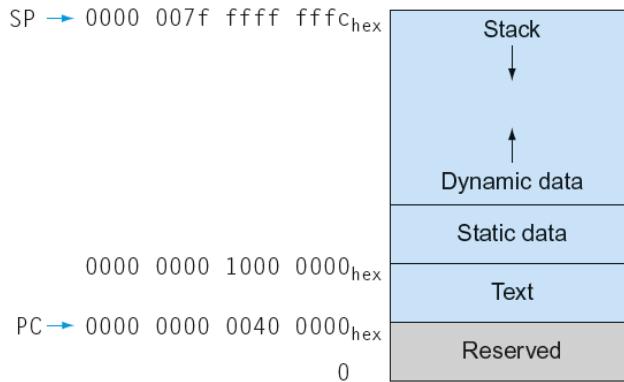
- Chapter 2 & 3
 - The compiler, algorithm and the instruction set architecture determine the instruction count required for a given program
- Chapter 4 :
 - How implementation of the processor determines both clock cycle time and the CPI.

Introduction

- We will examine two LEGv8 implementations
 - A simplified version (single-cycle datapath)
 - One instruction per cycle
 - Every instruction begins execution on one clock edge and complete execution on the next clock edge
 - Not efficient (why?)
 - A more realistic pipelined version
 - Designing instruction set for pipelining
 - Pipelined datapath and control
 - Pipeline hazards
- Simple ISA subset, shows most aspects
 - Memory reference: LDUR, STUR
 - Arithmetic/logical: ADD, SUB, AND, OR
 - Control transfer: CBNZ, B

Instruction Execution

- PC → instruction in memory, and fetch it
- Register numbers → register file, read registers

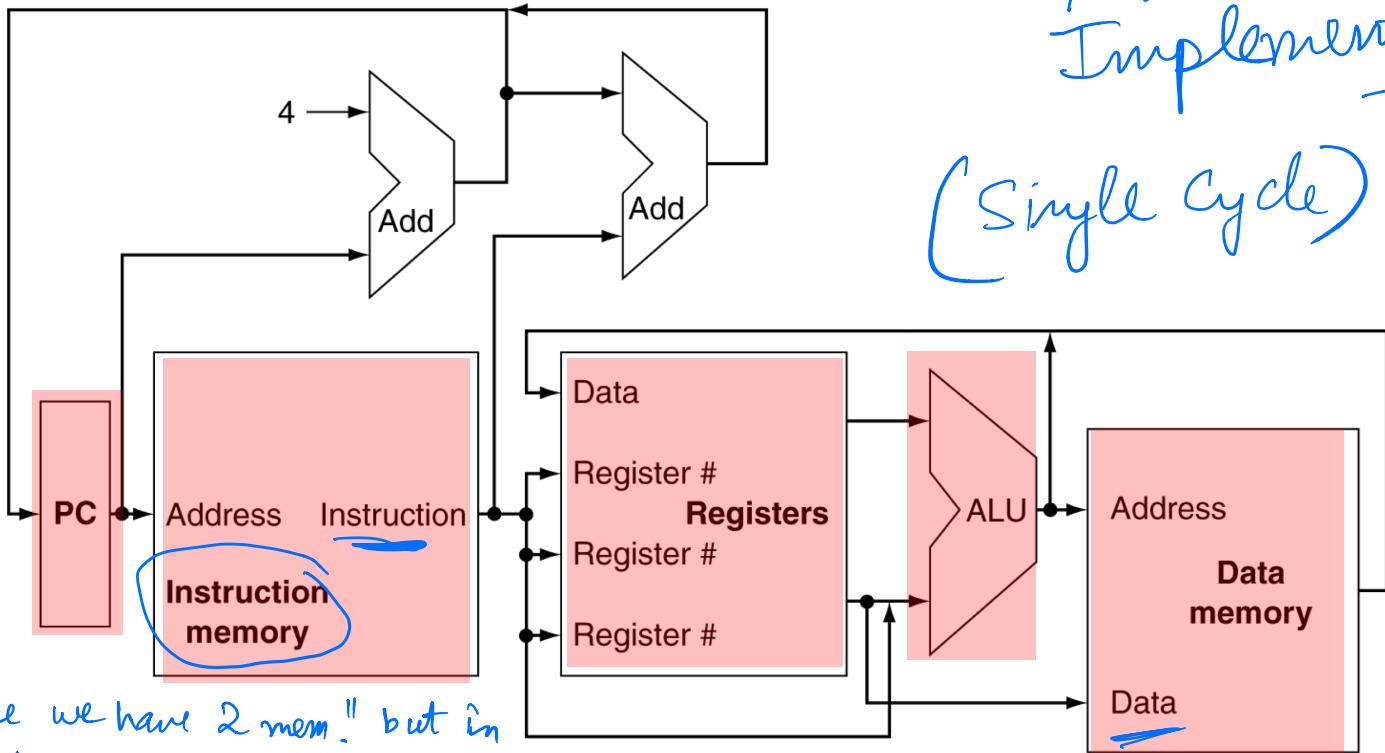


Instruction Execution

- Depending on instruction type
 - Use ALU to:
 - Perform Arithmetic result (**ADD X0, X1, X2**)
 - Calculate the memory address for load/store using base and offset addresses (**LDUR X0, [X2, #4]**)
 - Calculate the branch target address (**CBZ X0, exit**)
 - Calculate **PC + target address** or **PC + 4**
 - For branches after comparison or simply go to the next instruction address (**PC + 4**) in unconditional **B**

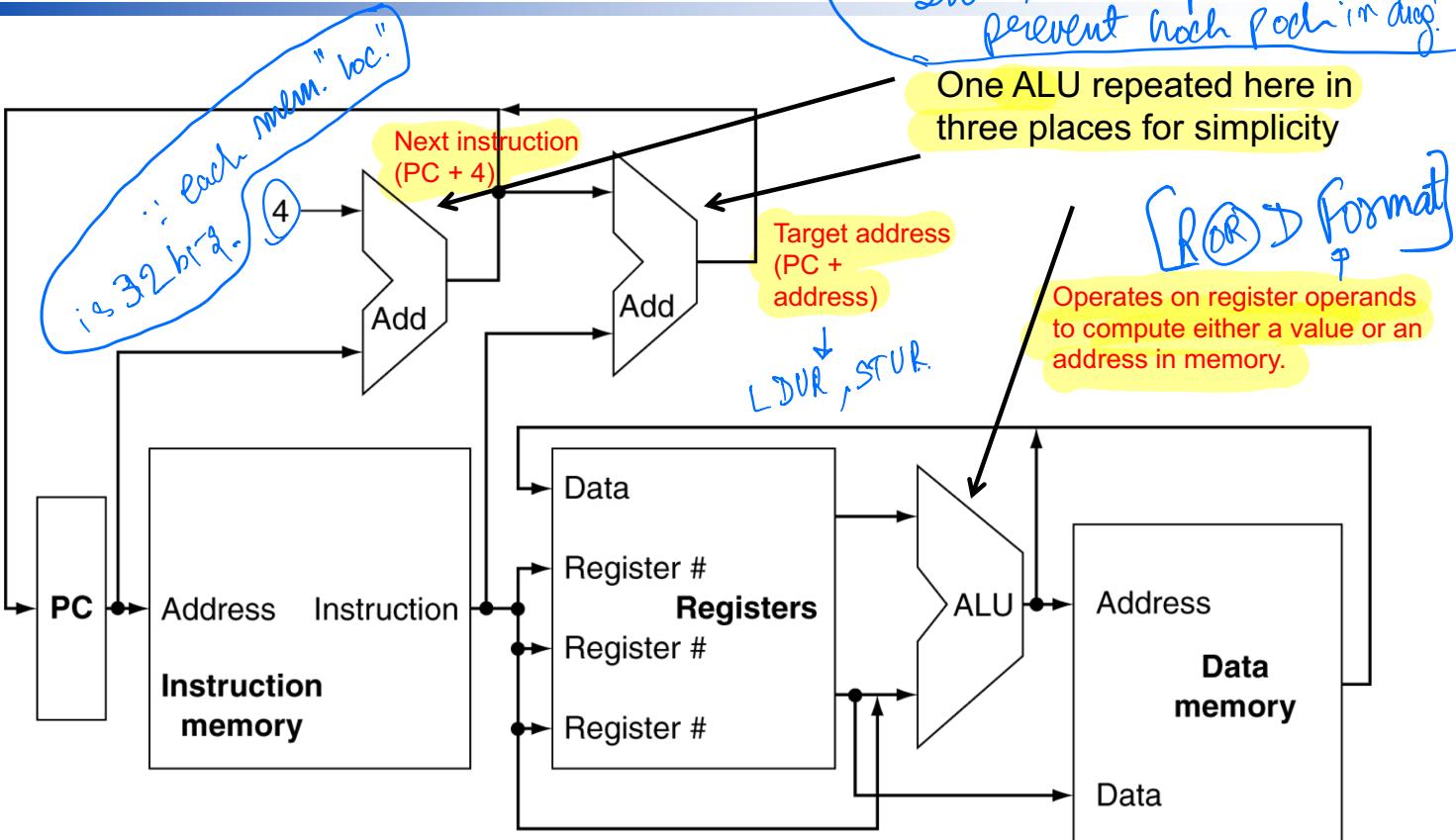
CPU Overview

Leg v8
Simplified Processor
Implementation - n
(Single Cycle)

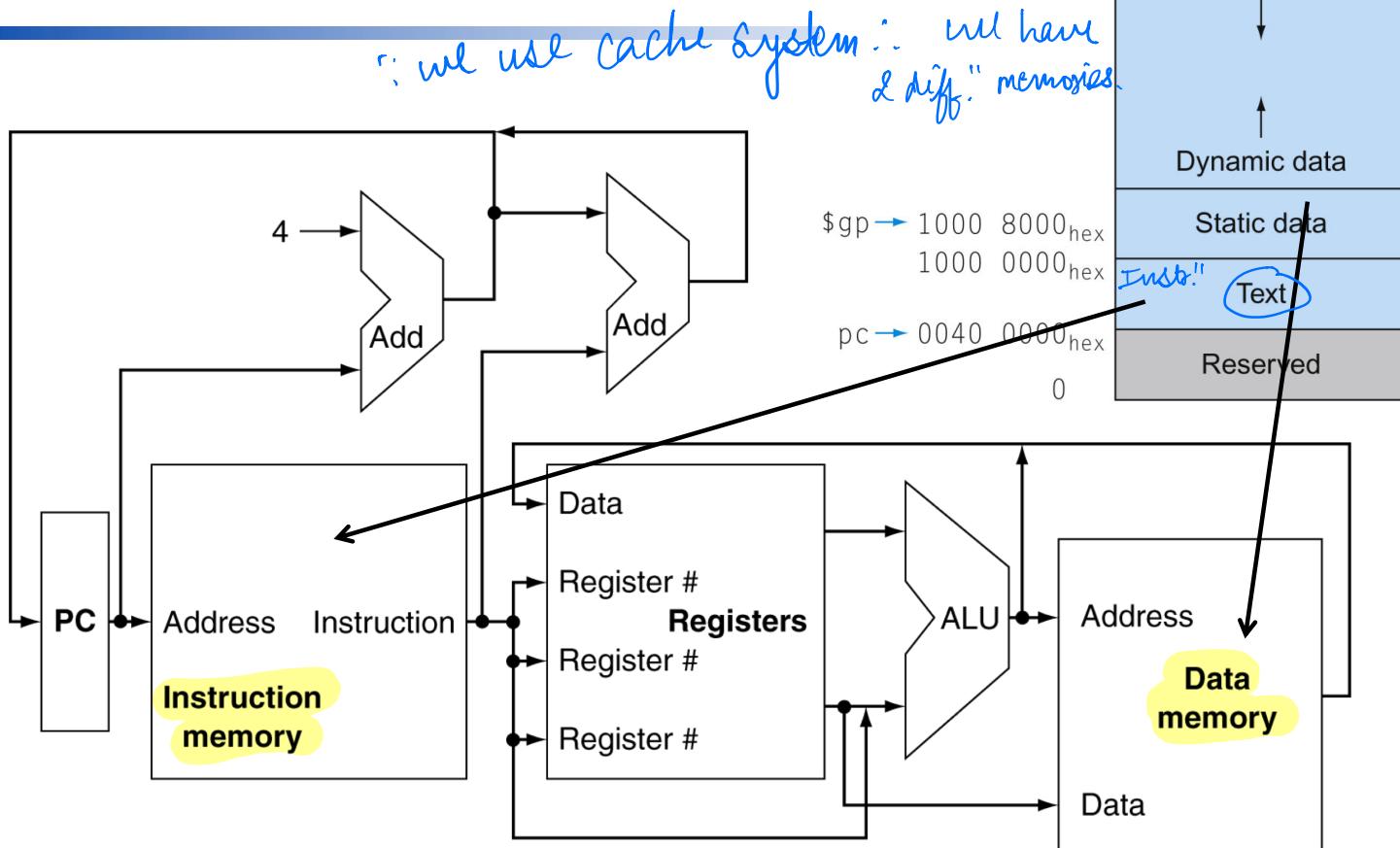


Here we have 2 mem. " but in Reality we " 1 mem."

CPU Overview

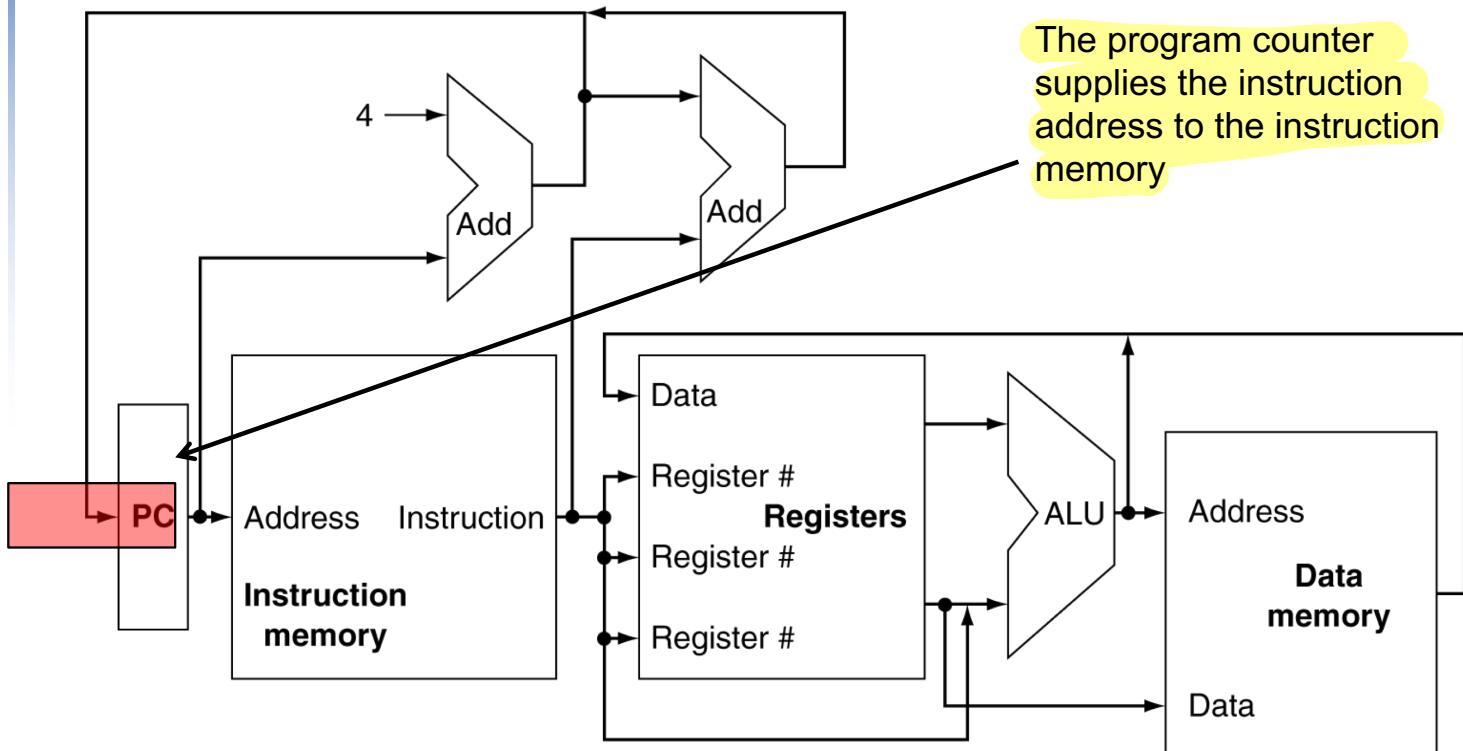


CPU Overview



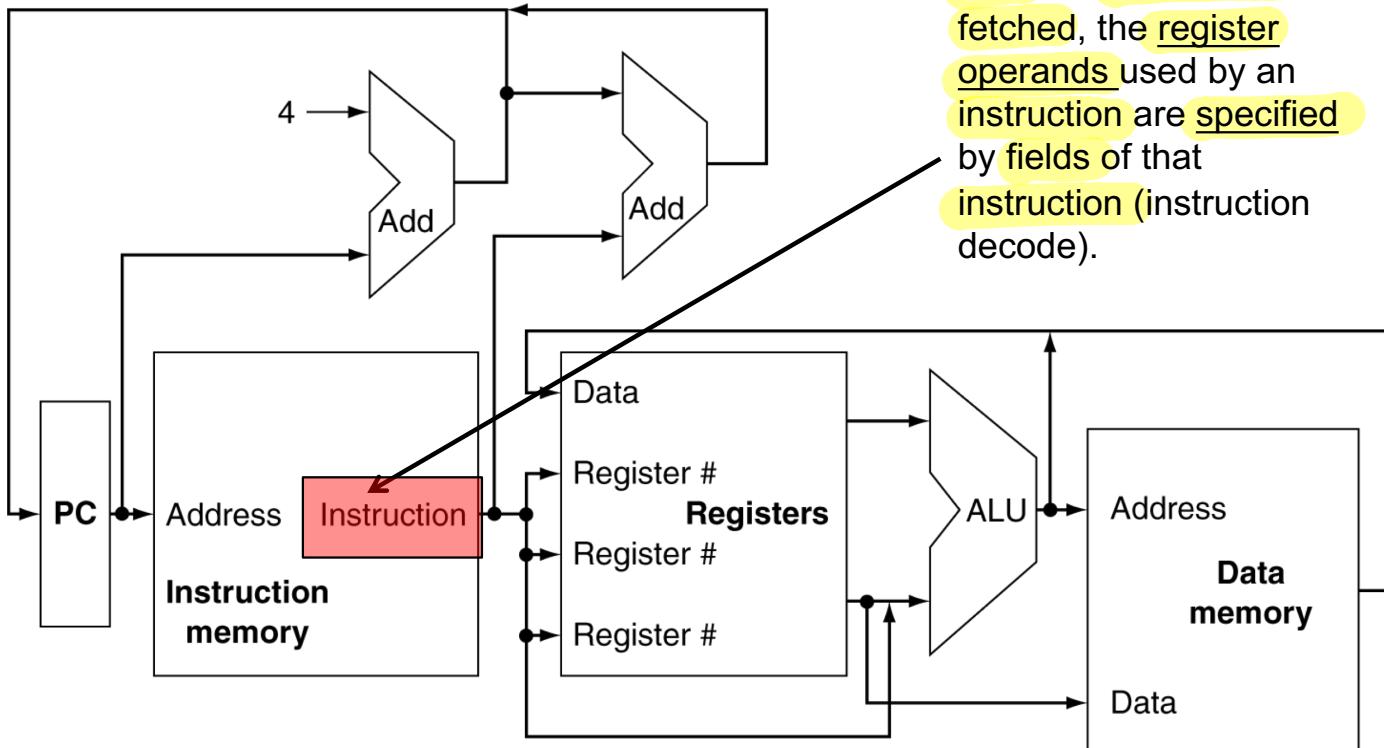
CPU Overview

① PC stores the address
of next instructions



CPU Overview

② Fields of the Instruction
(Registers Involved)

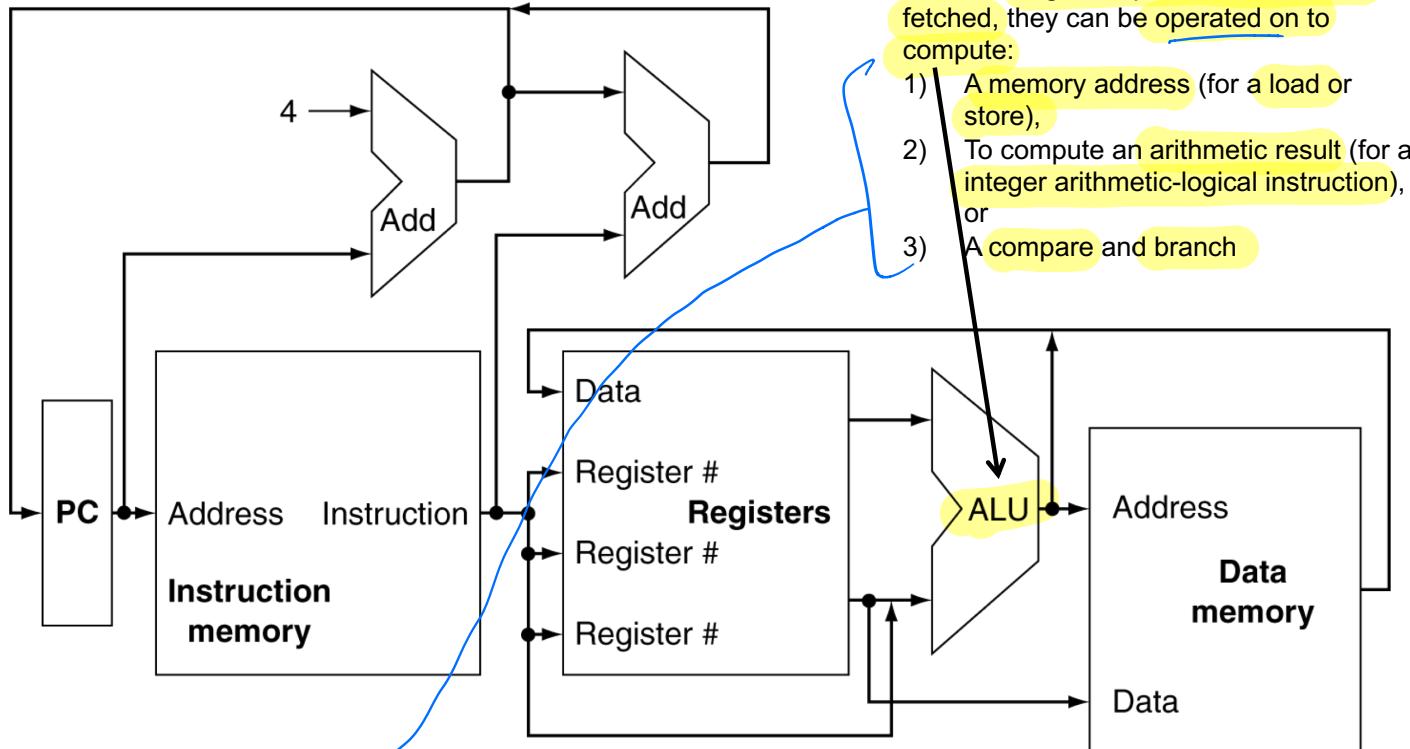


LDUR x9, [x22, #64]

Registers Involved.

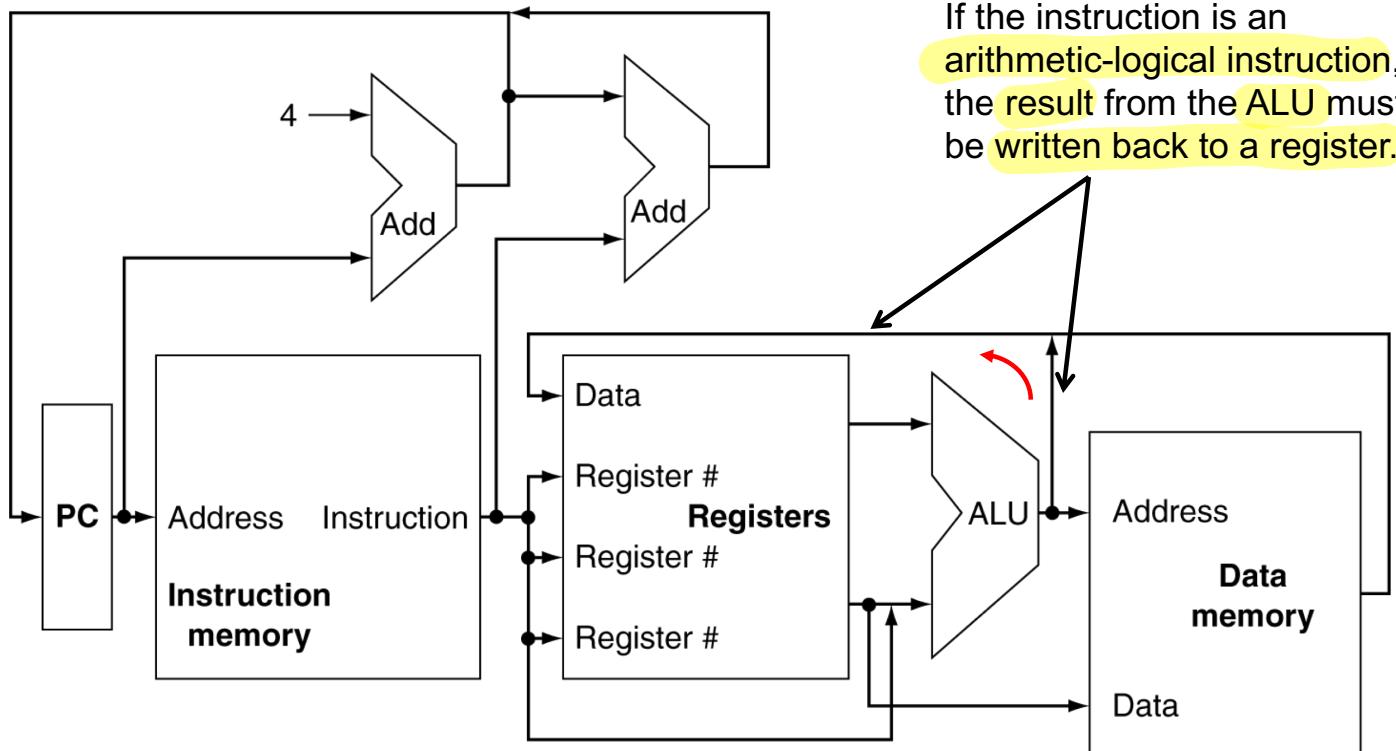
CPU Overview

③ ALU used to compute.



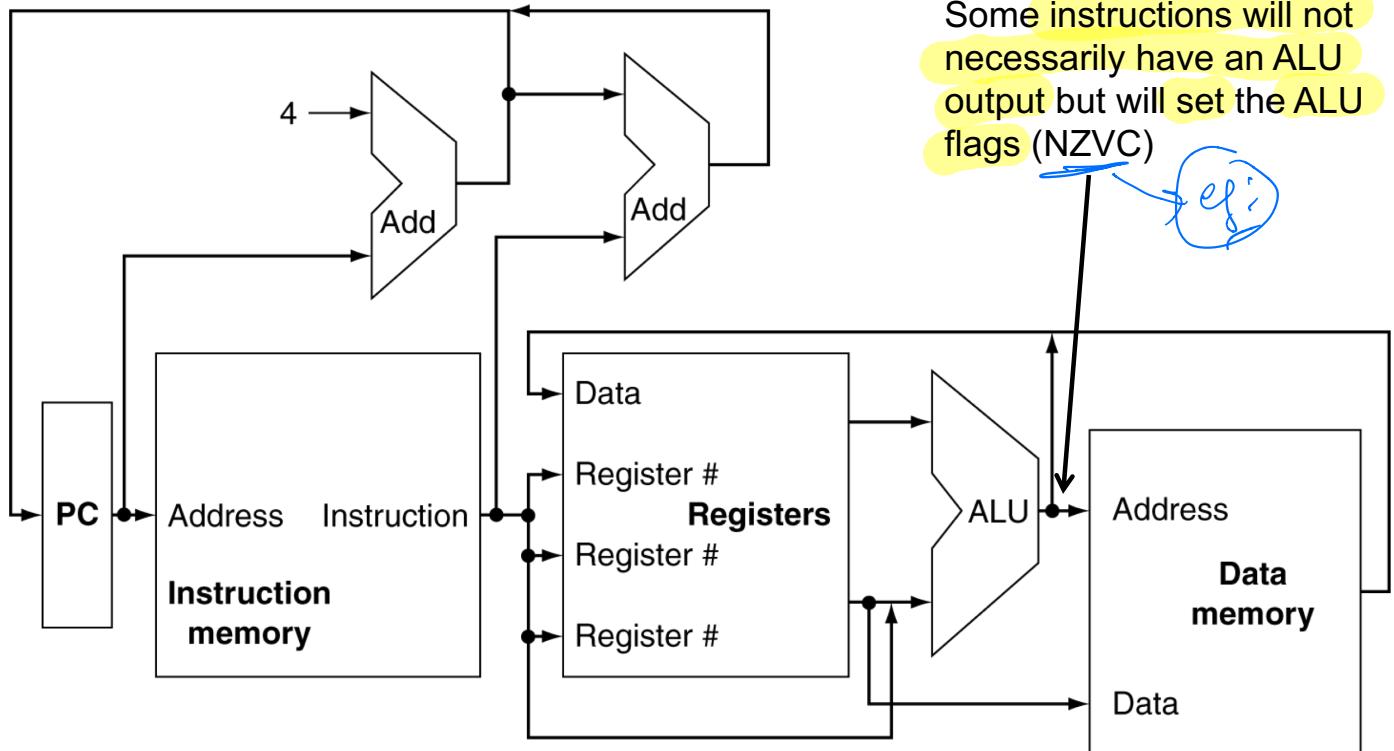
- 1) LDUR X9, [X22, #64]
2) ADD X0, X1, X2
3) CBZ X0, L1

CPU Overview



ADD x_0, x_1, x_2
OR x_9, x_{10}, x_{11}

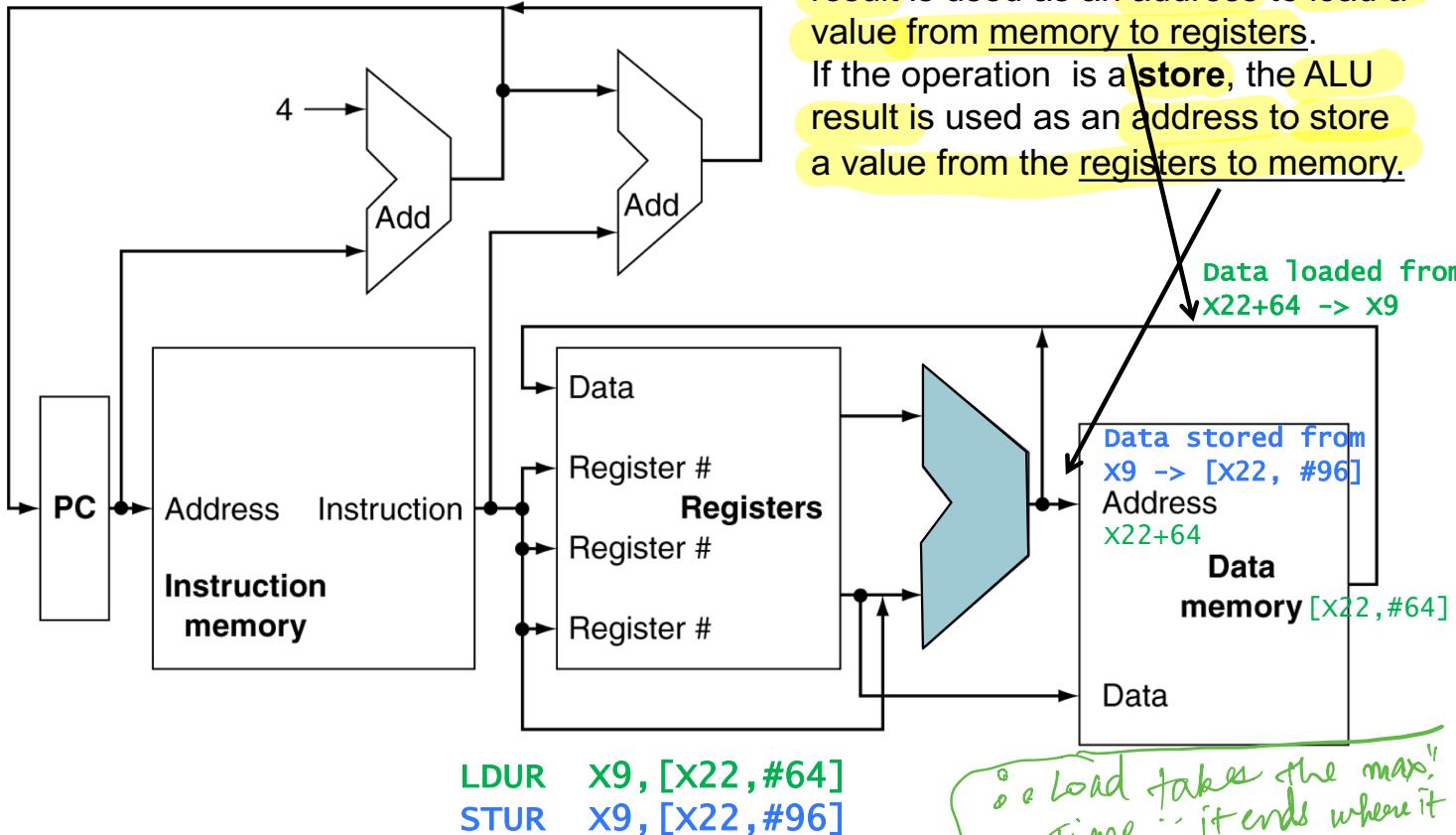
CPU Overview



CMP X9, X10

CPU Overview

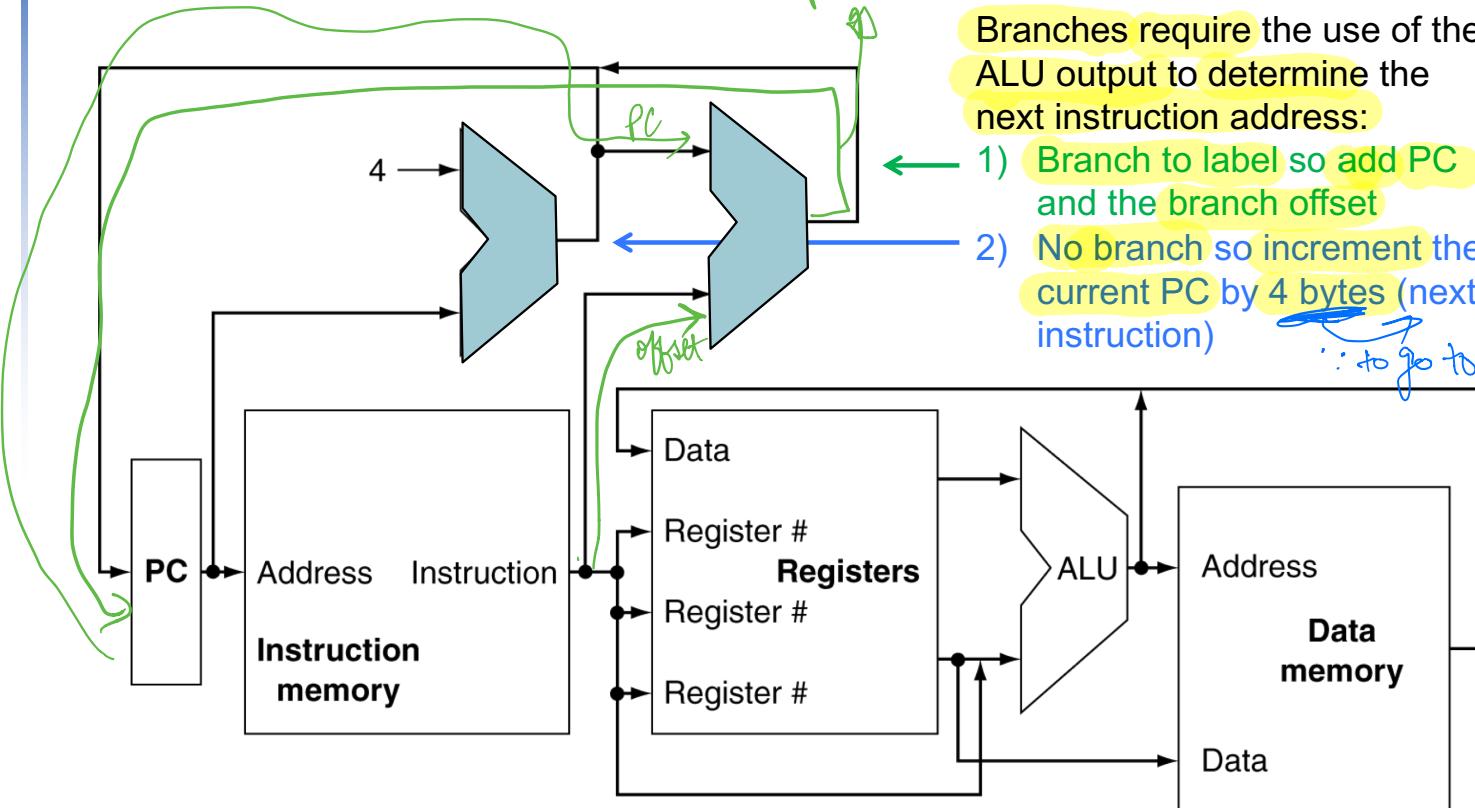
LOAD & STORE ↗



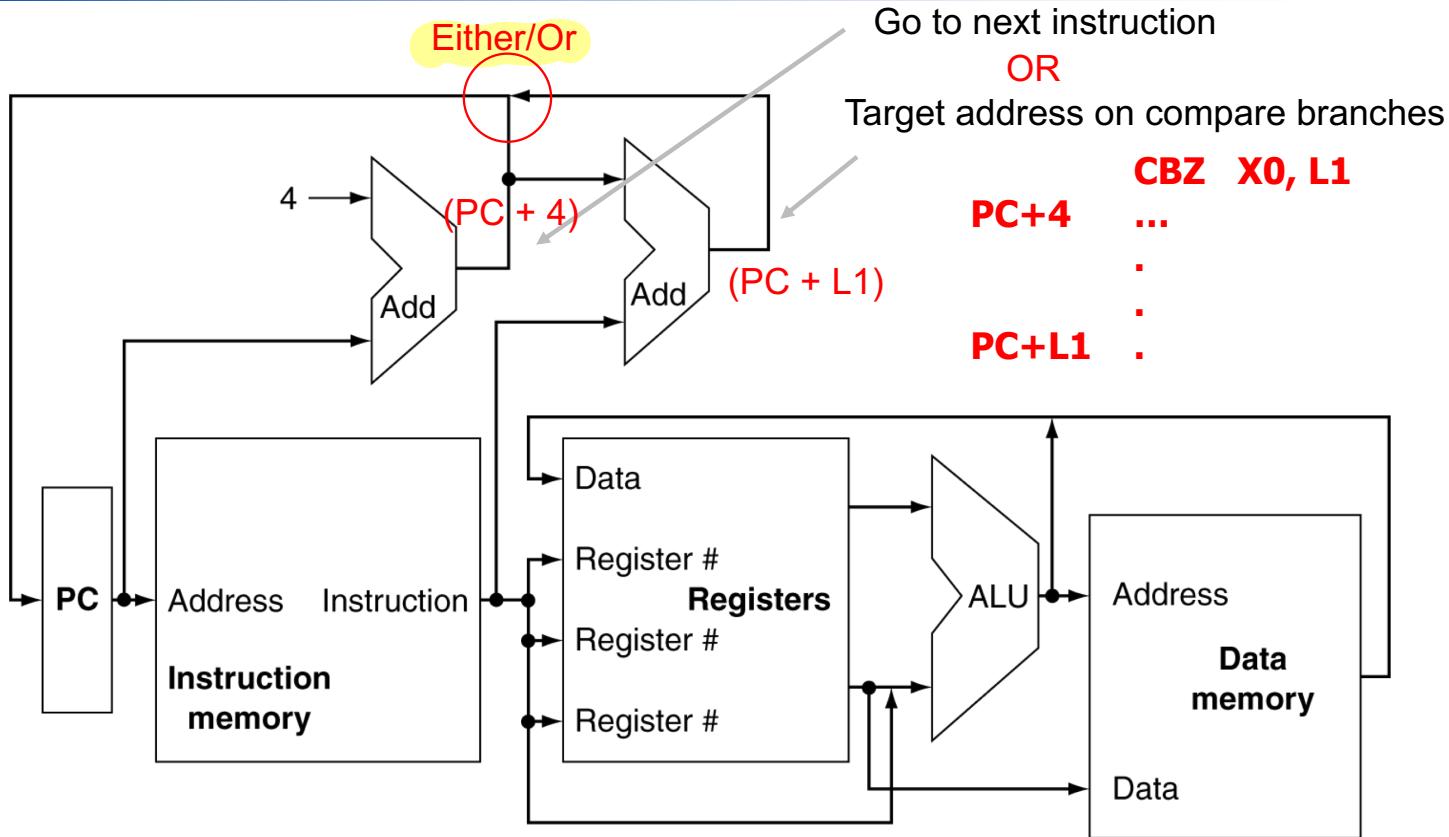
CPU Overview

Branch \oplus →

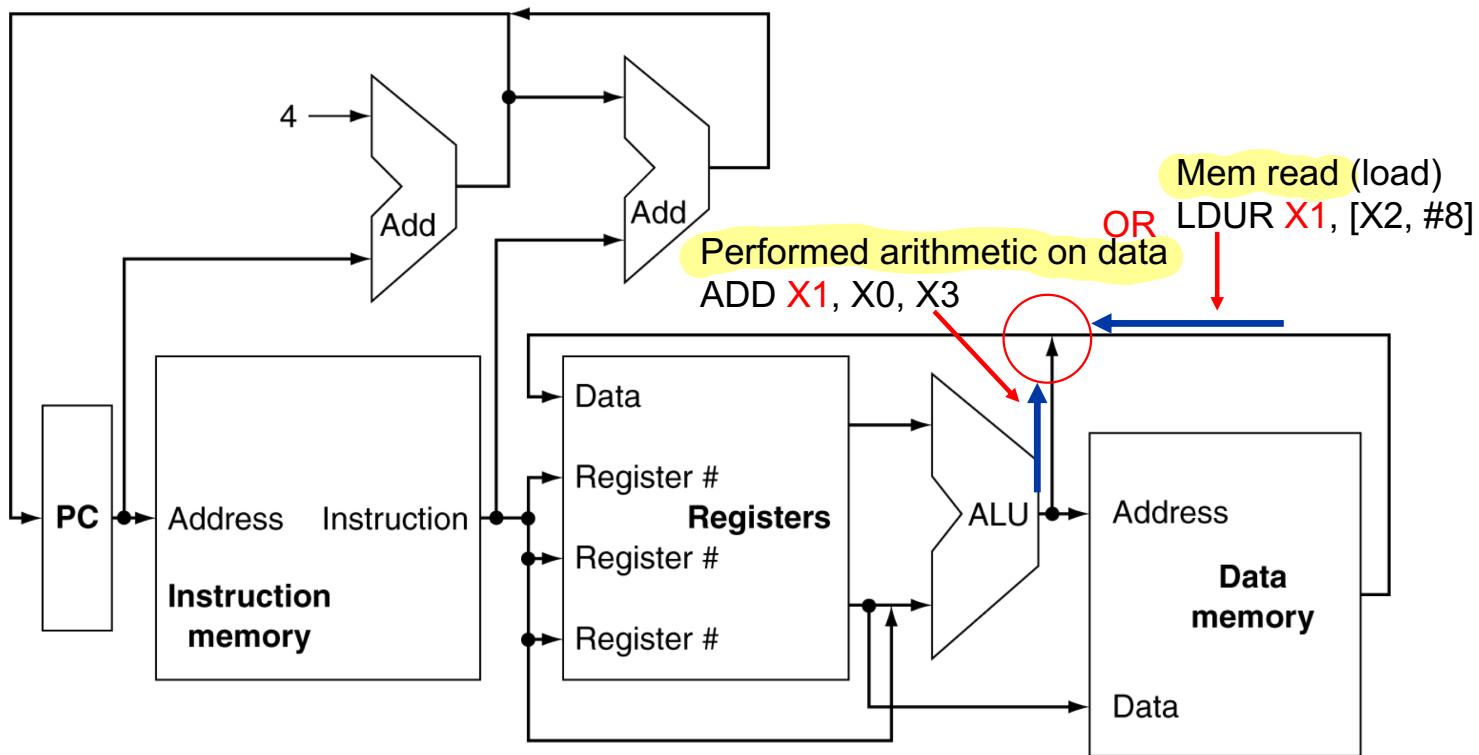
PC + offset.



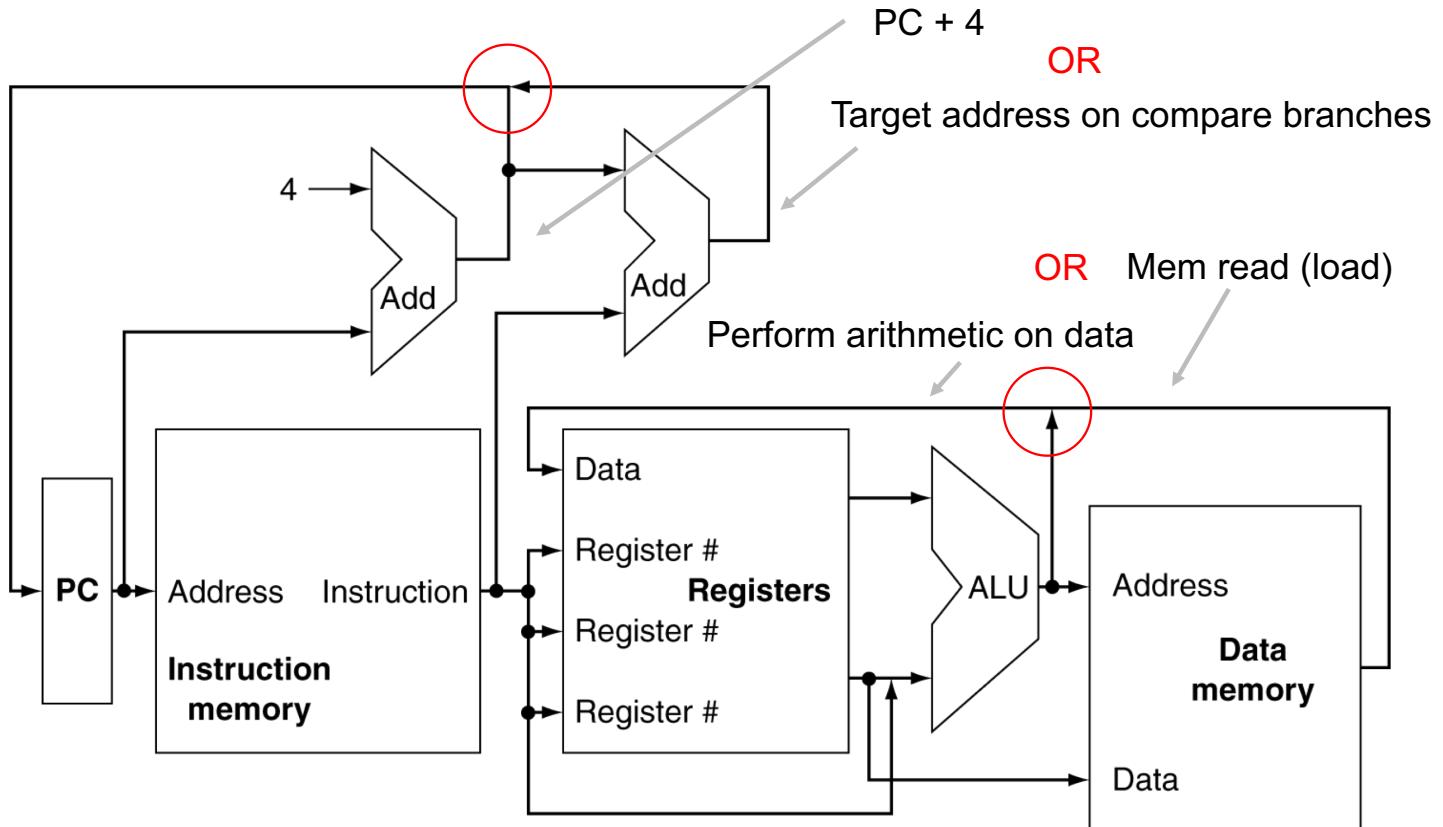
CPU Overview



CPU Overview



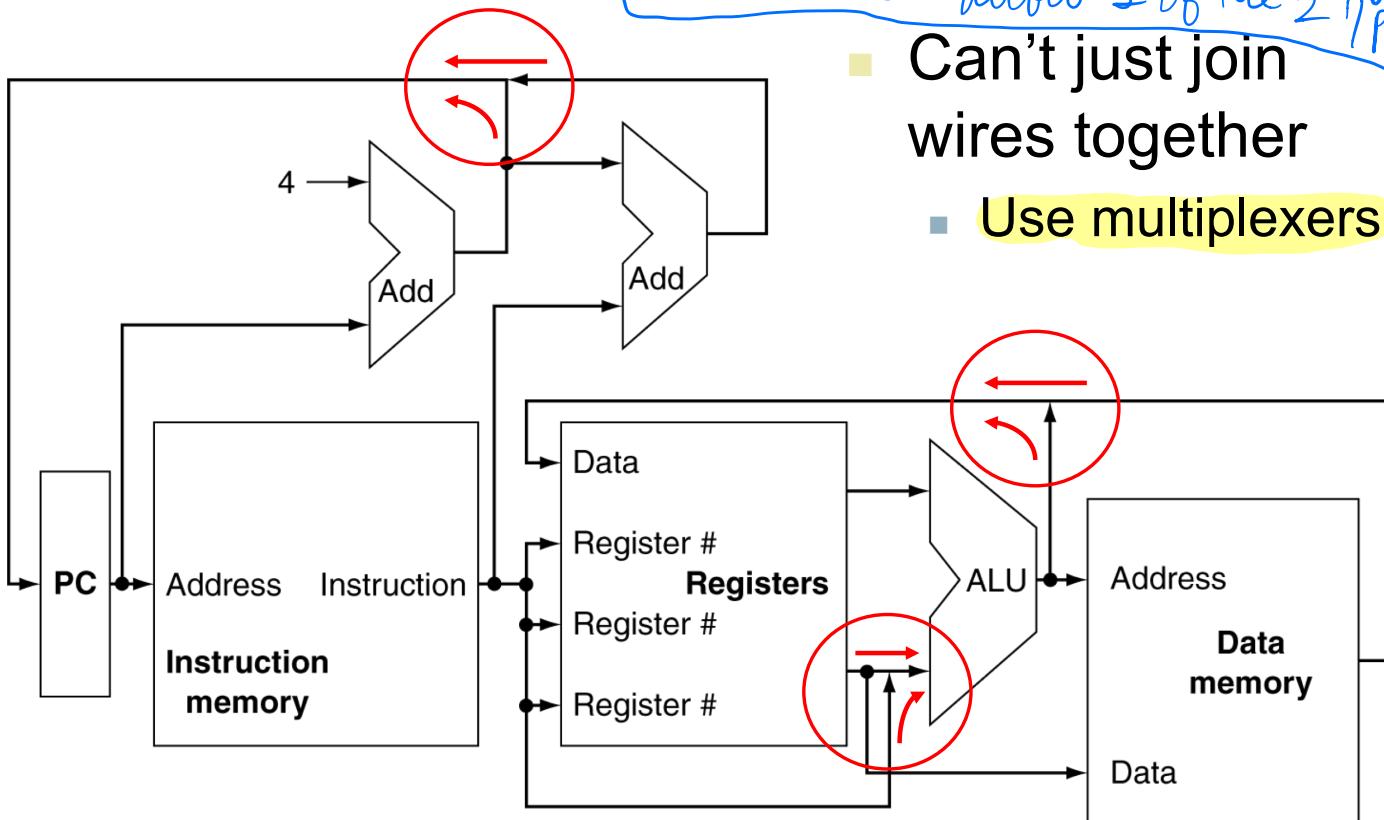
CPU Overview



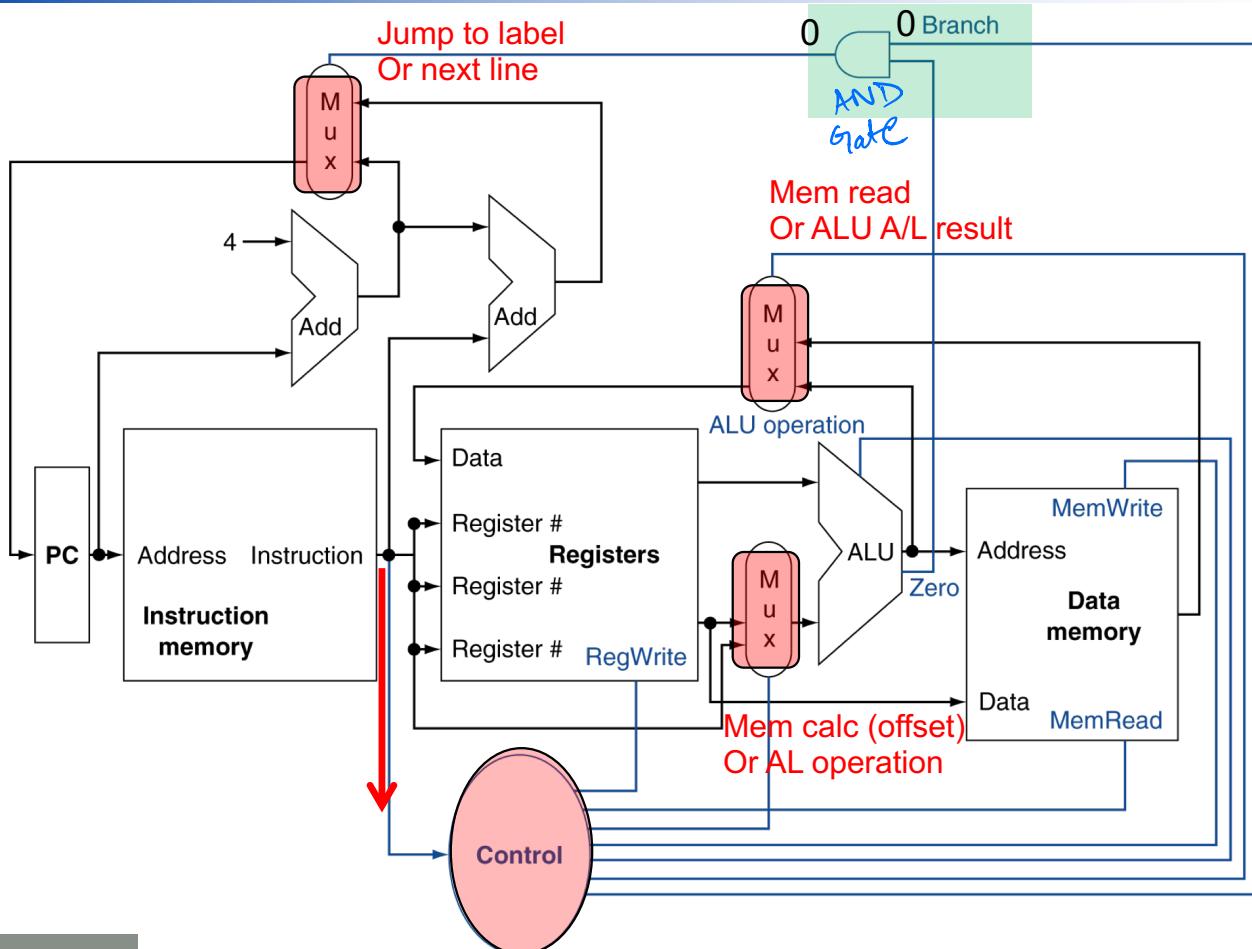
Multiplexers

It has a control line $cmy\ IN$
& depending on 0/1 it will only
allow 1 of the 2 i/p's.

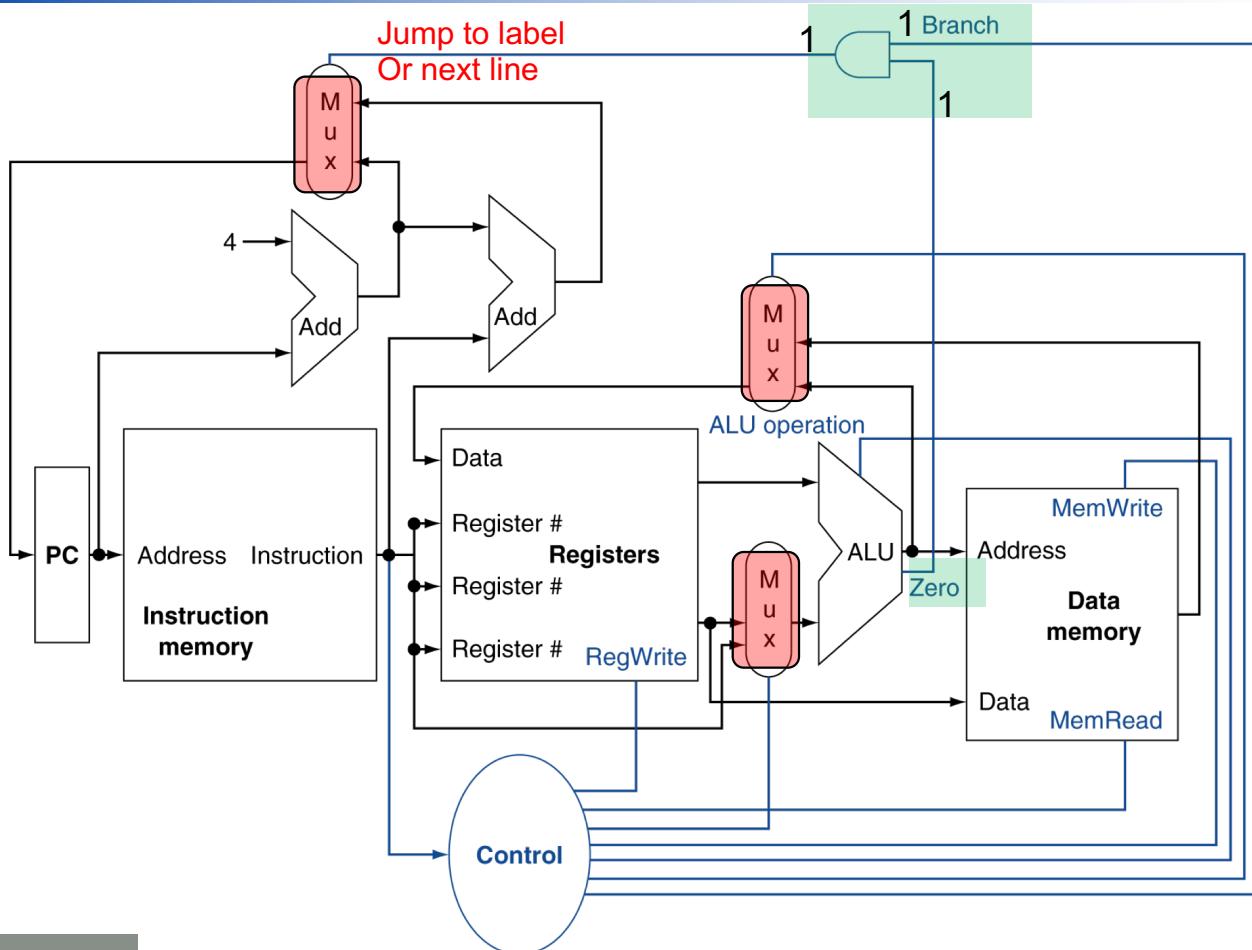
- Can't just join wires together
 - Use multiplexers



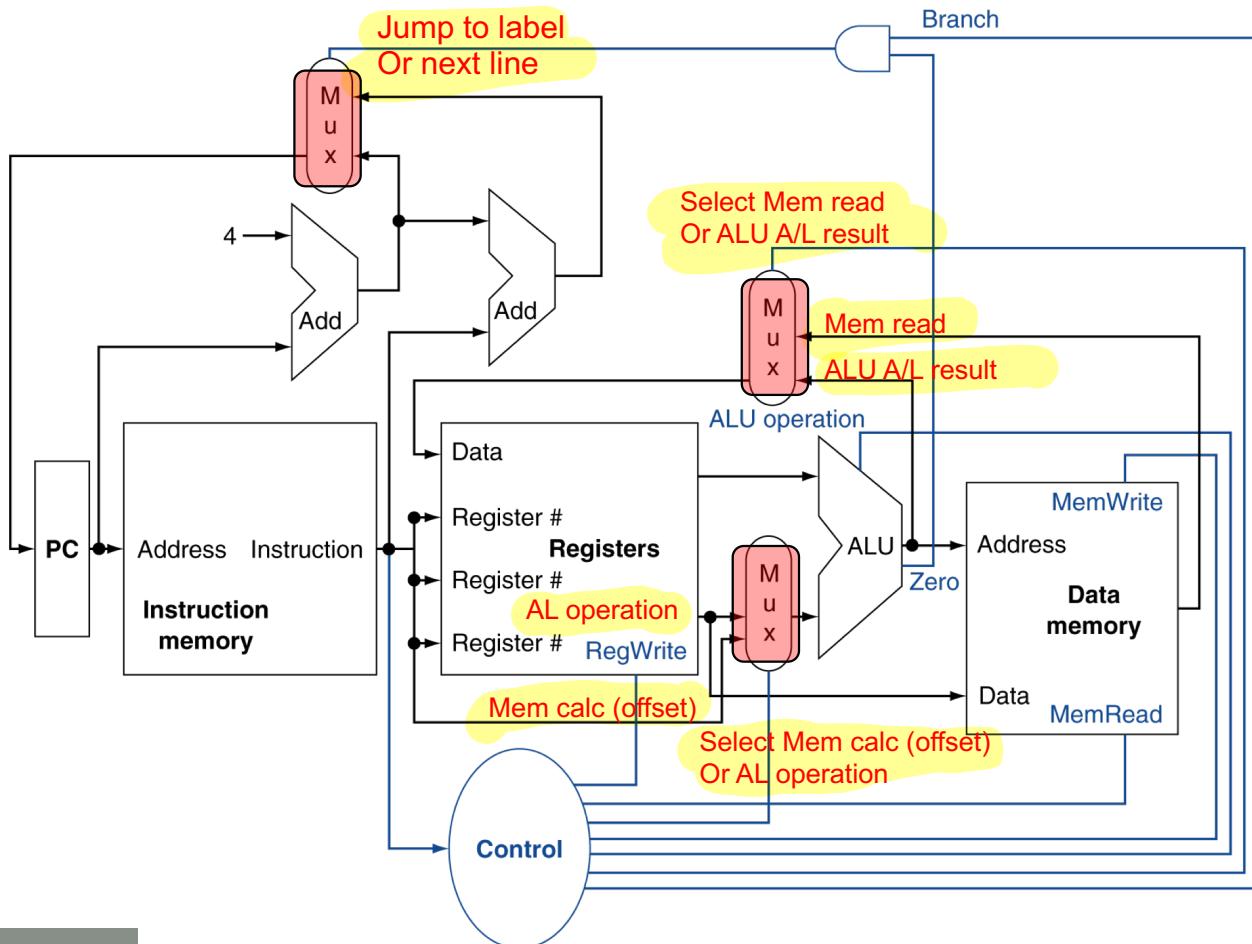
Control



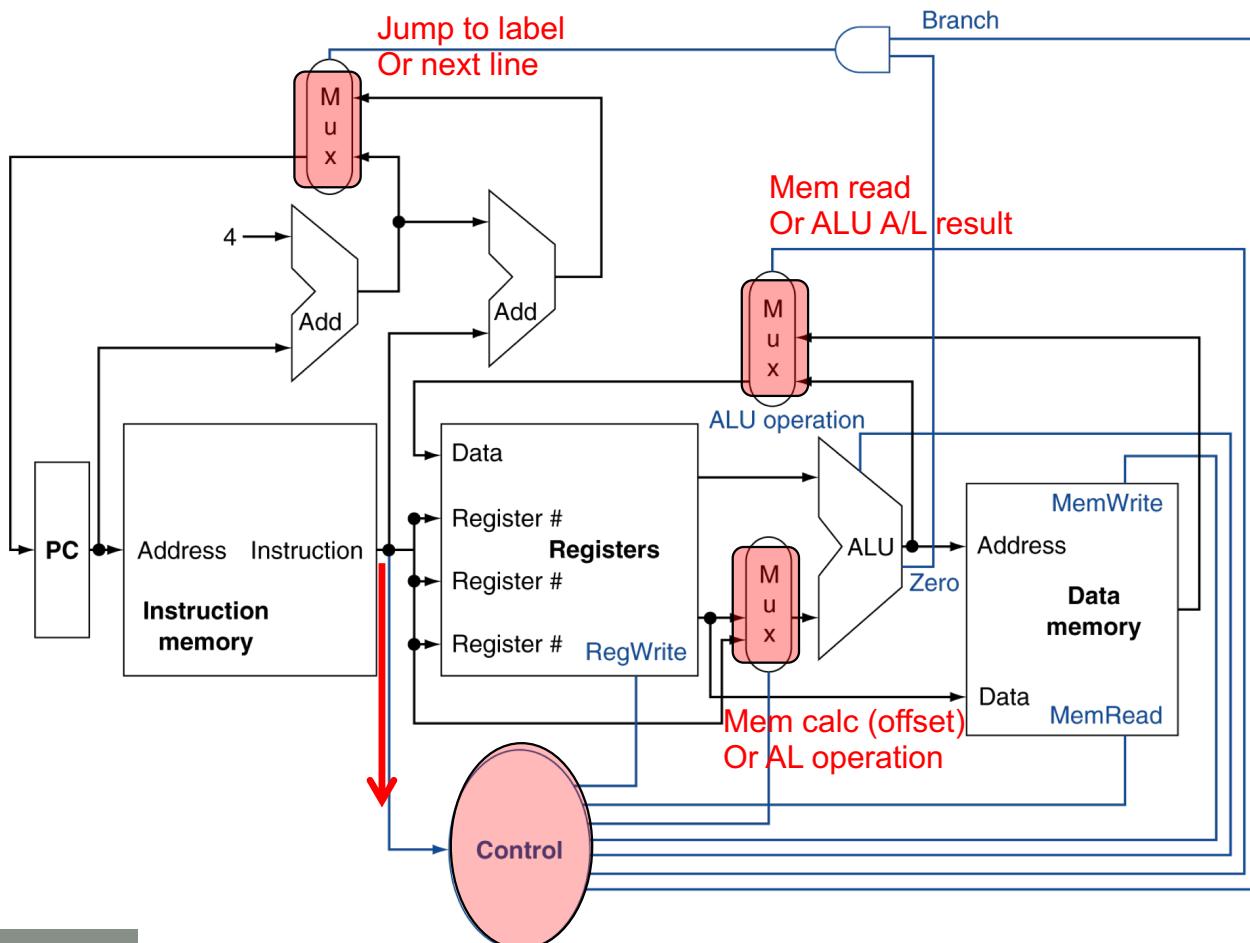
Control



Control



Control



Building a Datapath

- Datapath
 - Elements that process data and addresses in the CPU
 - Registers, ALUs, mux's, memories, ...
- We will build a LEGv8 datapath incrementally
 - Refining the overview design

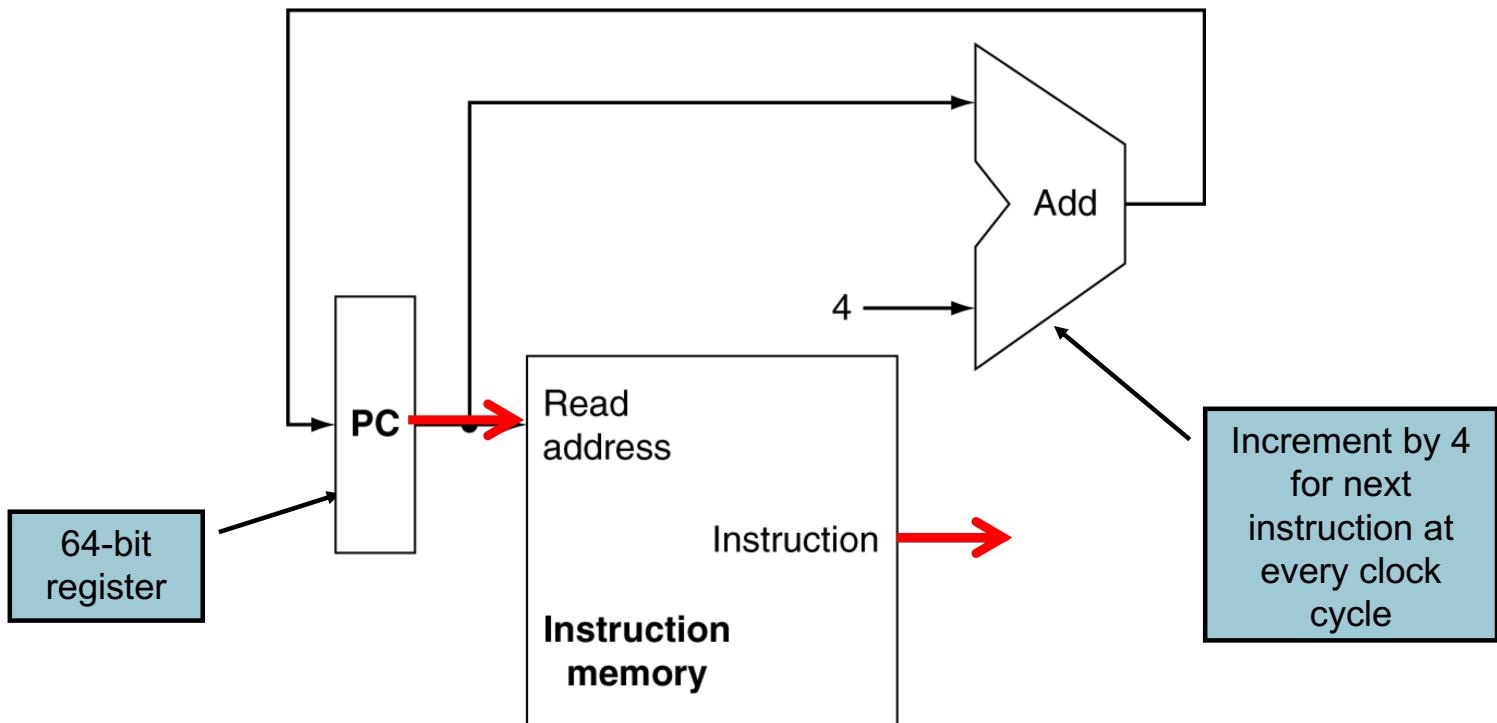
LEGv8 Instruction Formats

LEGv8

Name	Format	Example					Comments
ADD	R	1112	3	0	2	1	ADD X1, X2, X3
SUB	R	1624	3	0	2	1	SUB X1, X2, X3
ADDI	I	580	100			2	1
SUBI	I	836	100			2	1
LDUR	D	1986	100		0	2	1
STUR	D	1984	100		0	2	1

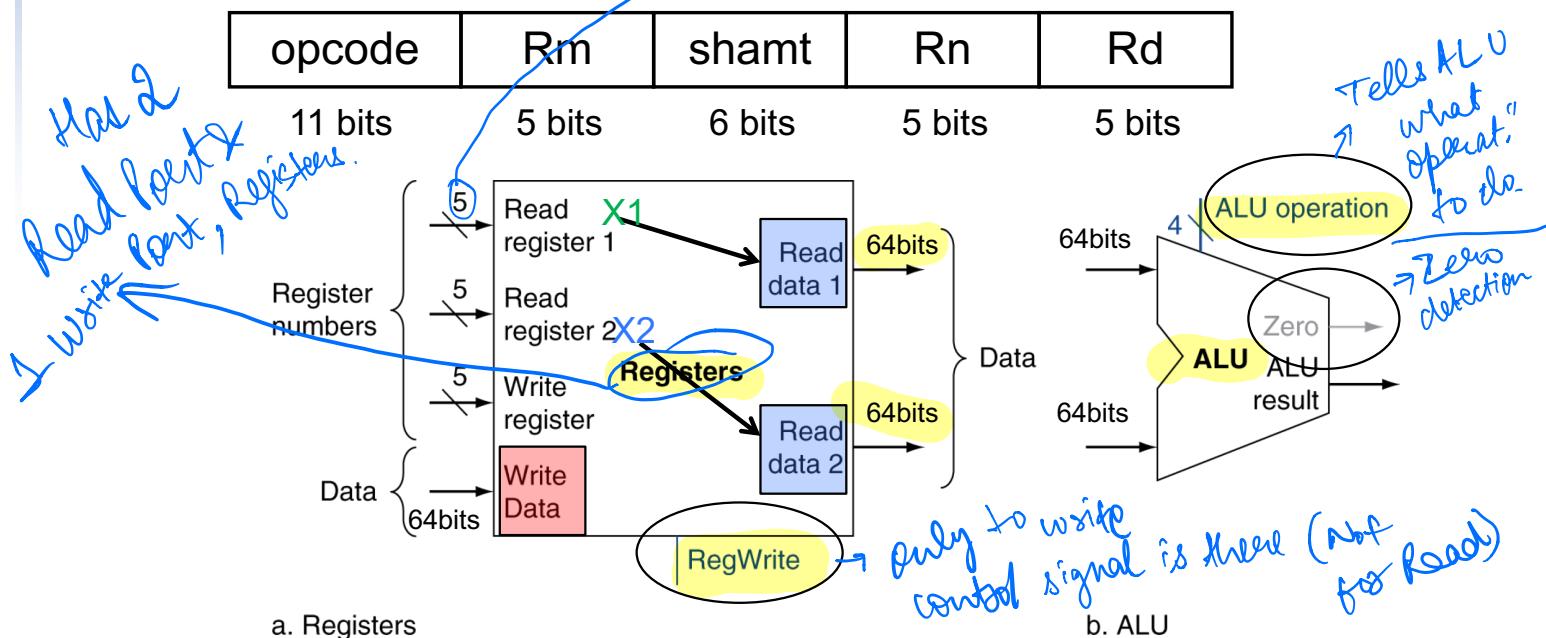
Name	Fields						Comments
Field size	6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	All LEGv8 instructions are 32 bits long
R-format	R	opcode	Rm	shamt	Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd
D-format	D	opcode	address	op2	Rn	Rt	Data transfer format
B-format	B	opcode	address				Unconditional Branch format
CB-format	CB	opcode	address			Rt	Conditional Branch format
IW-format	IW	opcode	immediate			Rd	Wide Immediate format

Instruction Fetch



R-Format Instructions

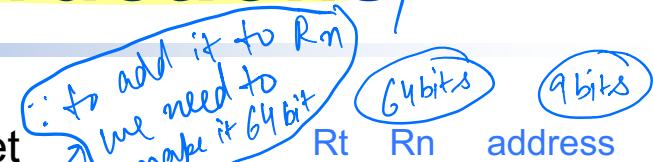
1. Read two register operands ADD X_0, X_1, X_2
 - assume those registers are loaded with data from memory already
2. Perform arithmetic/logical operation
3. Write register result



Load/Store Instructions

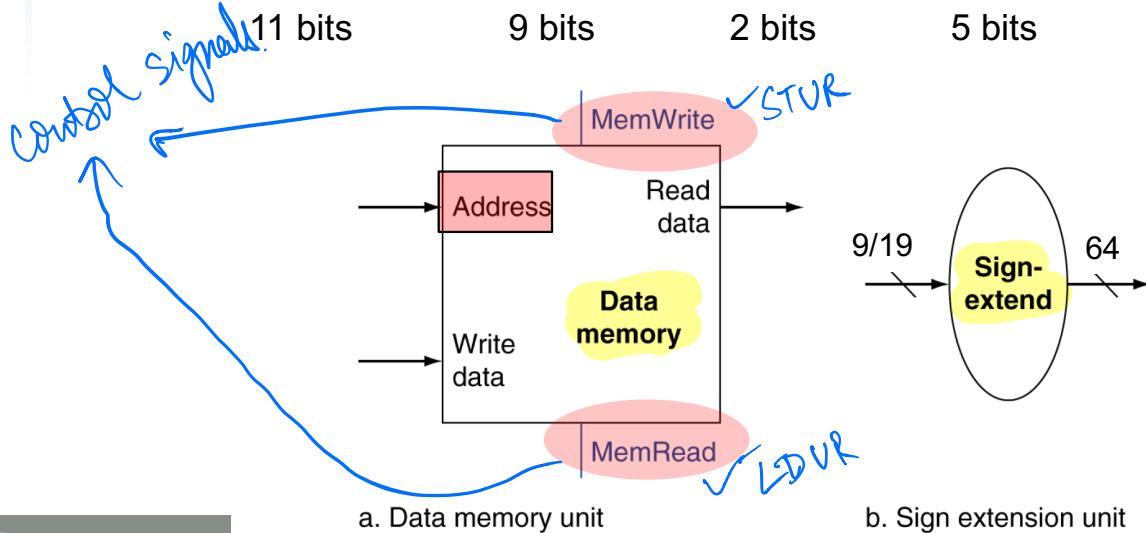
D format.

- Read register operands
- Calculate address using 9-bit offset
 - Use ALU, but first sign-extend offset
- LDUR: Read memory and update register
- STUR: Write register value to memory



LDUR X1, [X2, offset_value]
STUR X1, [X2, offset_value]

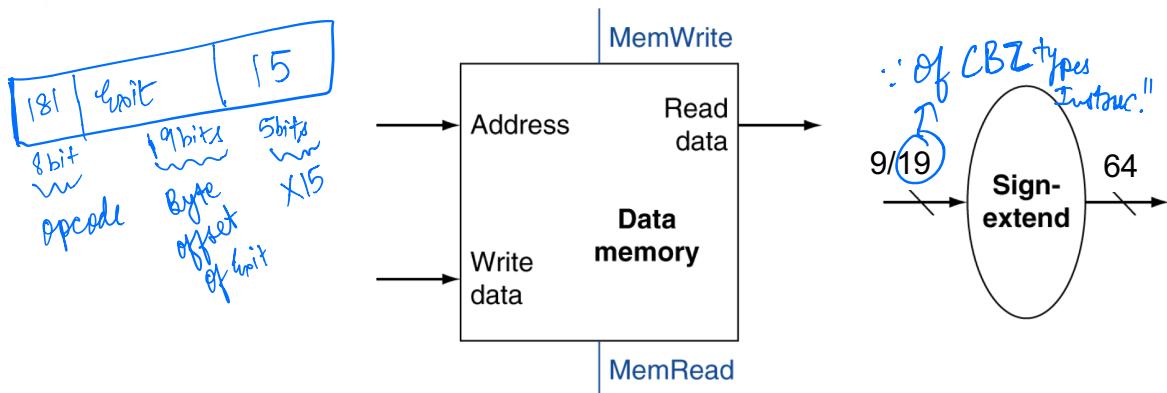
opcode	address	op2	Rn	Rt
--------	---------	-----	----	----



Load/Store Instructions

- Read register operands
- Calculate address using 9-bit offset
 - Use ALU, but first sign-extend offset
- LDUR: Read memory and update register
- STUR: Write register value to memory

Rt Rn address
LDUR X1, [X2, offset_value]
STUR X1, [X2, offset_value]
CBZ X15, Exit



a. Data memory unit

b. Sign extension unit

Conditional Branch Instruction

- Compare or subtract operands
 - Read register operands.
 - Use ALU: Subtract and check ~~destination register~~ *Zero O/P.*
- Branch
 - Either,
 - Calculate target address,
 - Or use PC + 4

SUB X0, X1, X2

CBZ X0, offset

Next instruction at PC + 4

.

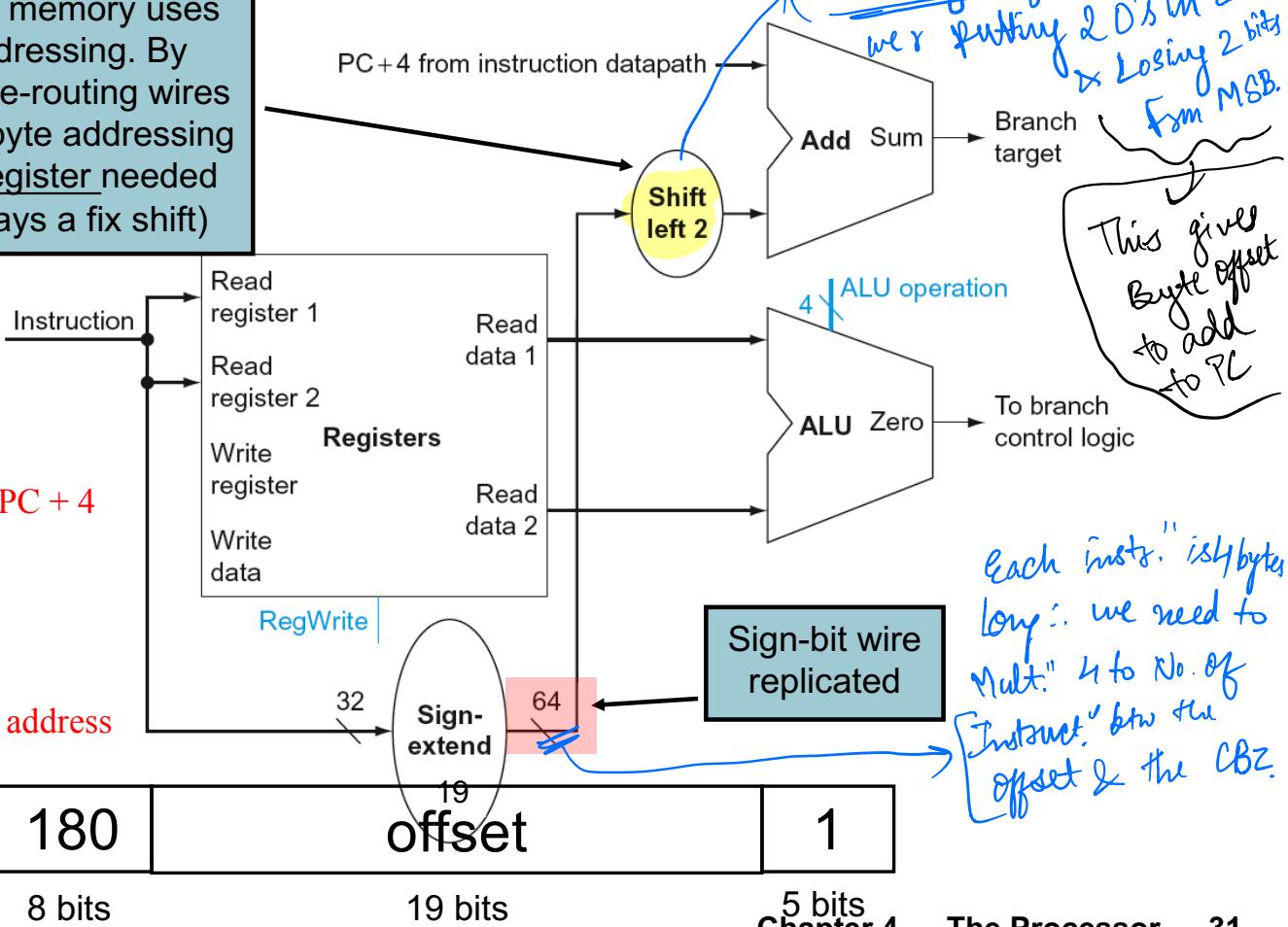
.

.

Branch target address (PC + offset)

Conditional Branch Instruction

Instruction memory uses word addressing. By physically re-routing wires convert to byte addressing (no shift register needed since always a fix shift)



Conditional Branch Instruction

Instruction Memory

	Byte 3	Byte 2	Byte 1	Byte 0	
Instruction 1					0x00000000 (0)
Instruction 2					0x00000004 (4)
Instruction 3					0x00000008 (8)
Instruction 4					0x0000000C (12)
Instruction 5					0x00000010 (16)
Instruction 6					0x00000014 (20)

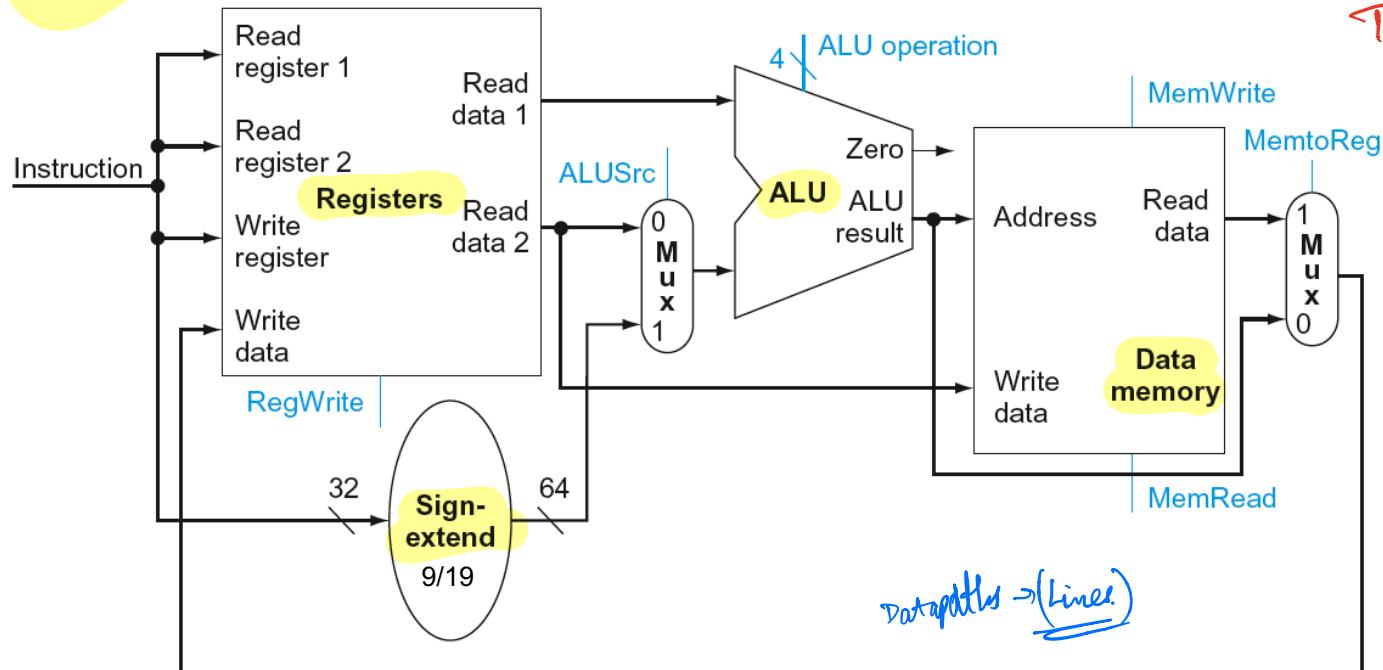
Instruction Offset = 4



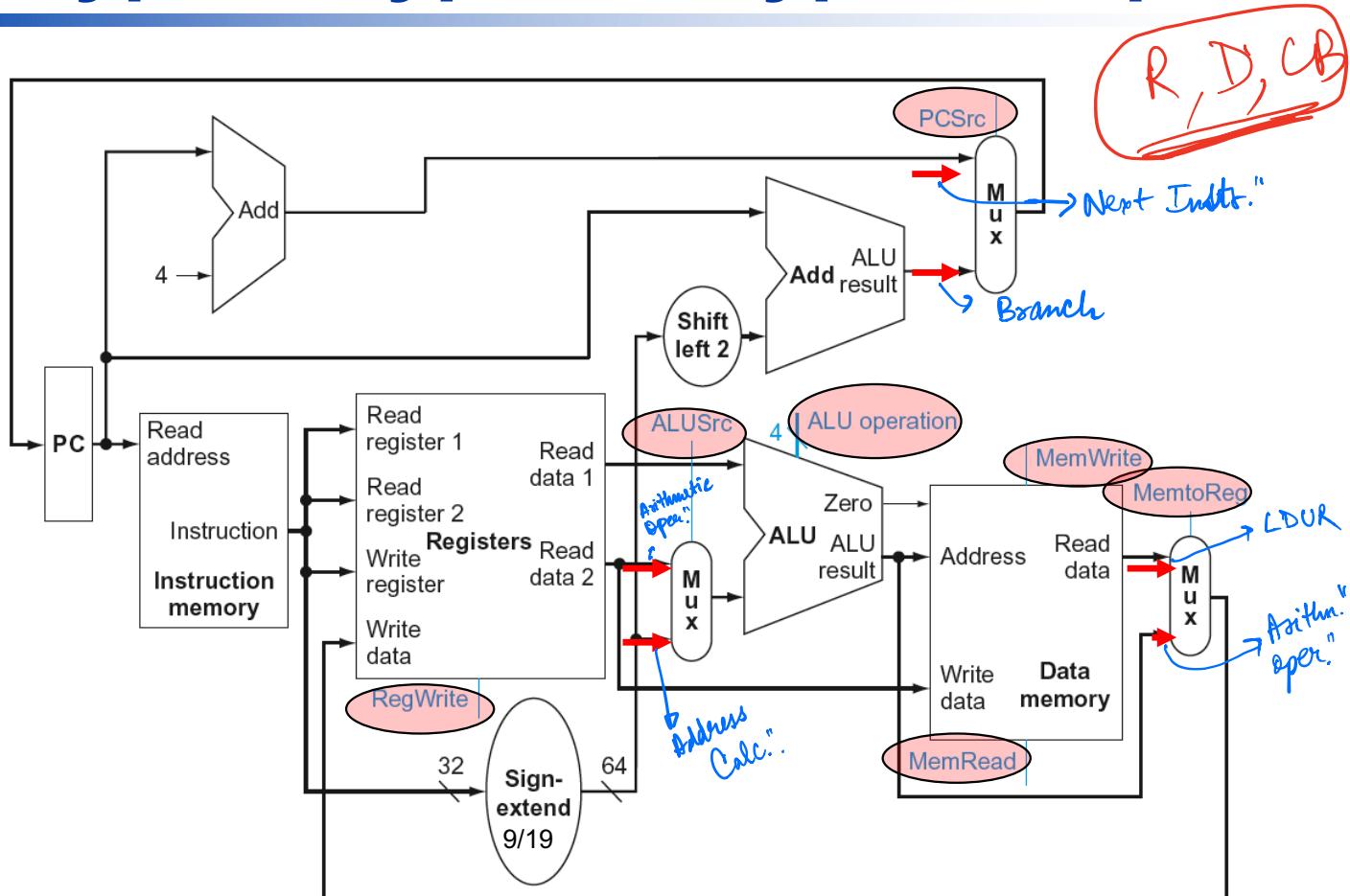
Composing the Elements

- First-cut data path makes an attempt to do an instruction in one clock cycle
 - Each datapath element can only do one function at a time
 - Hence, we need separate instruction and data memories (using separate data and instruction caches for example)
- Use multiplexers where alternate data sources are used for different instructions

R-Type/D-Type(Idr/str) Datapath



R-Type/D-Type/CB-Type Datapath



ALU Control

(4 bit value)

- ALU used for
 - D-type Load/Store: Function = add
 - CB-type CBZ: Function = subtract
 - R-type: Function depends on opcode

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	pass input b
1100	NOR

ALU Control

- Assume 2-bit ALUOp control lines derived from opcode and used as an input to the ALU control code generator.

opcode	ALUOp	Operation	11 bit. Opcode field	ALU function	ALU control
LDUR	00	load register	XXXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXXX	pass input b	0111
R-type <i>In this we have to see opcode.</i>	10	add	10001011000	add	0010
		subtract	11001011000	subtract	0110
		AND	10001010000	AND	0000
		ORR	10101010000	OR	0001

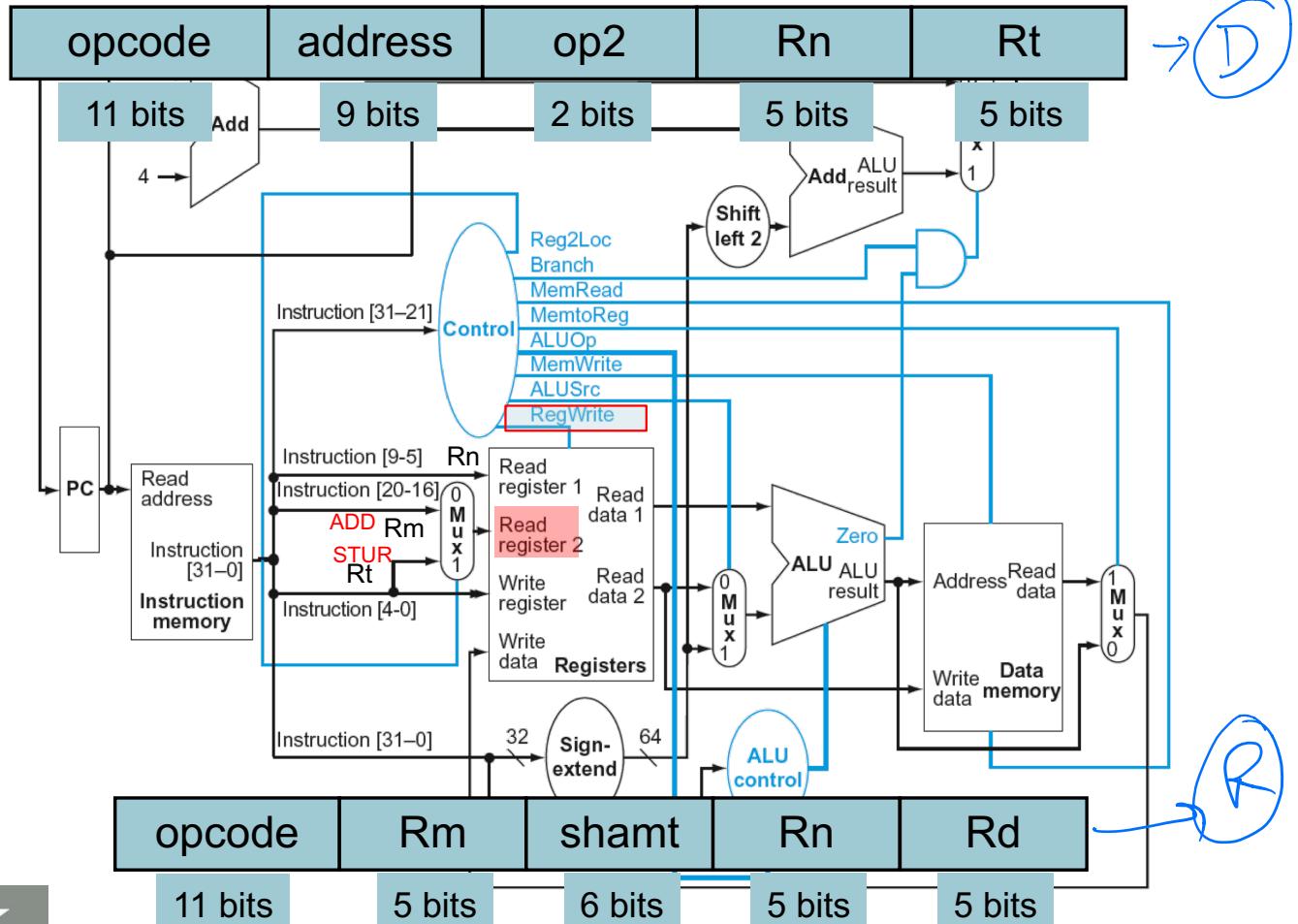
The Main Control Unit

- Control signals derived from instruction

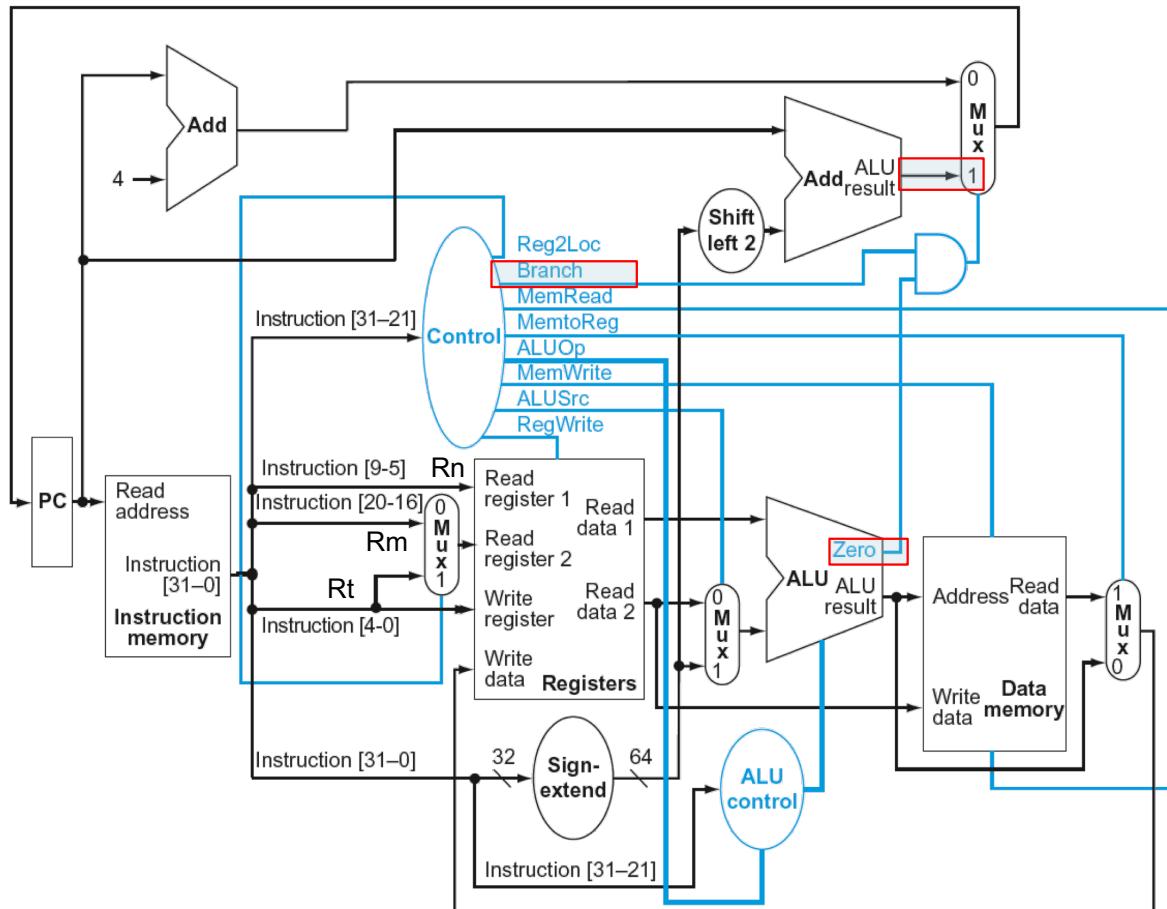
Field	opcode	Rm	shamt	Rn	Rd
Bit positions	31:21	20:16	15:10	9:5	4:0
a. R-type instruction	<i>Loc. of</i>				
Field	1986 or 1984	address	0	Rn	Rt
Bit positions	31:21	20:12	11:10	9:5	4:0
b. Load or store instruction	9-bit offset address				
Field	180	address		Rt	
Bit positions	31: 24	23:5			4:0
c. Conditional branch instruction	19-bit offset branch address				

Datapath With Control

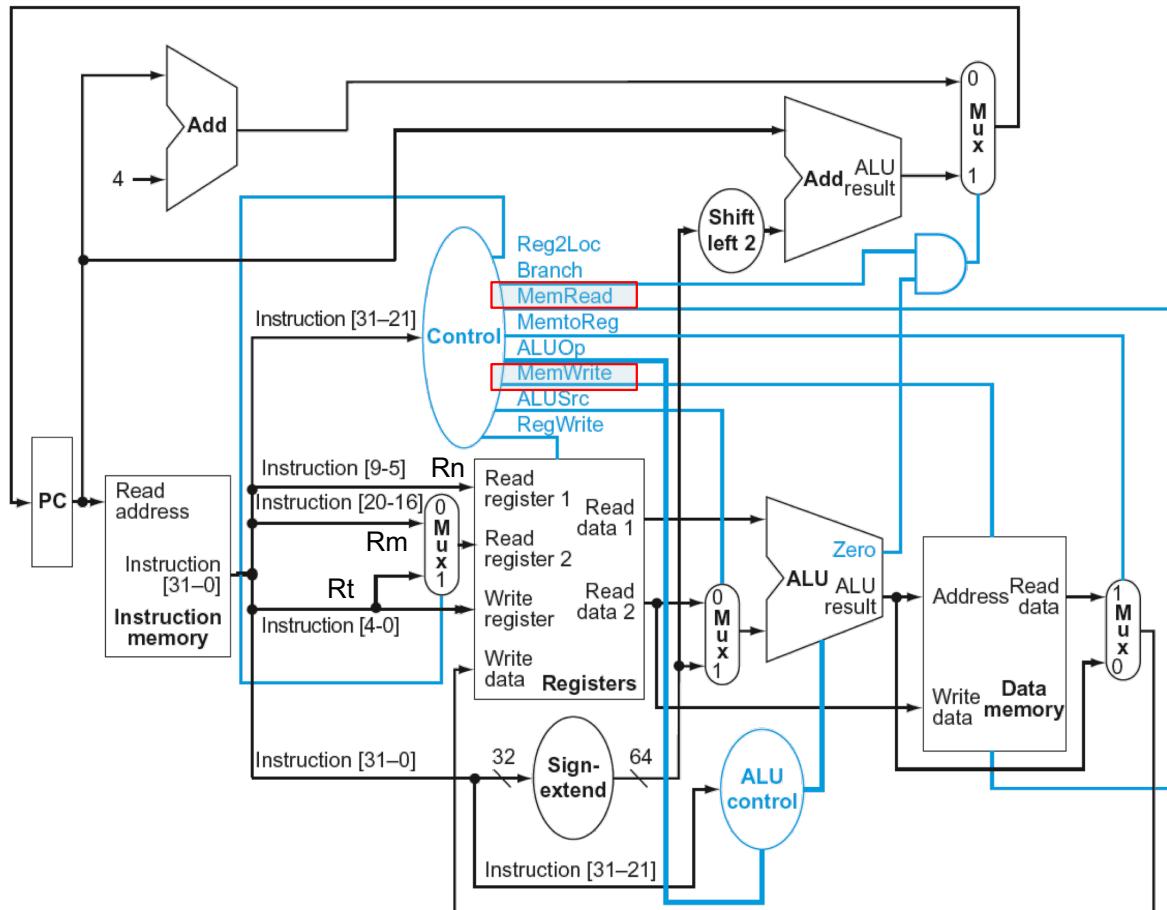
R, D, CB.



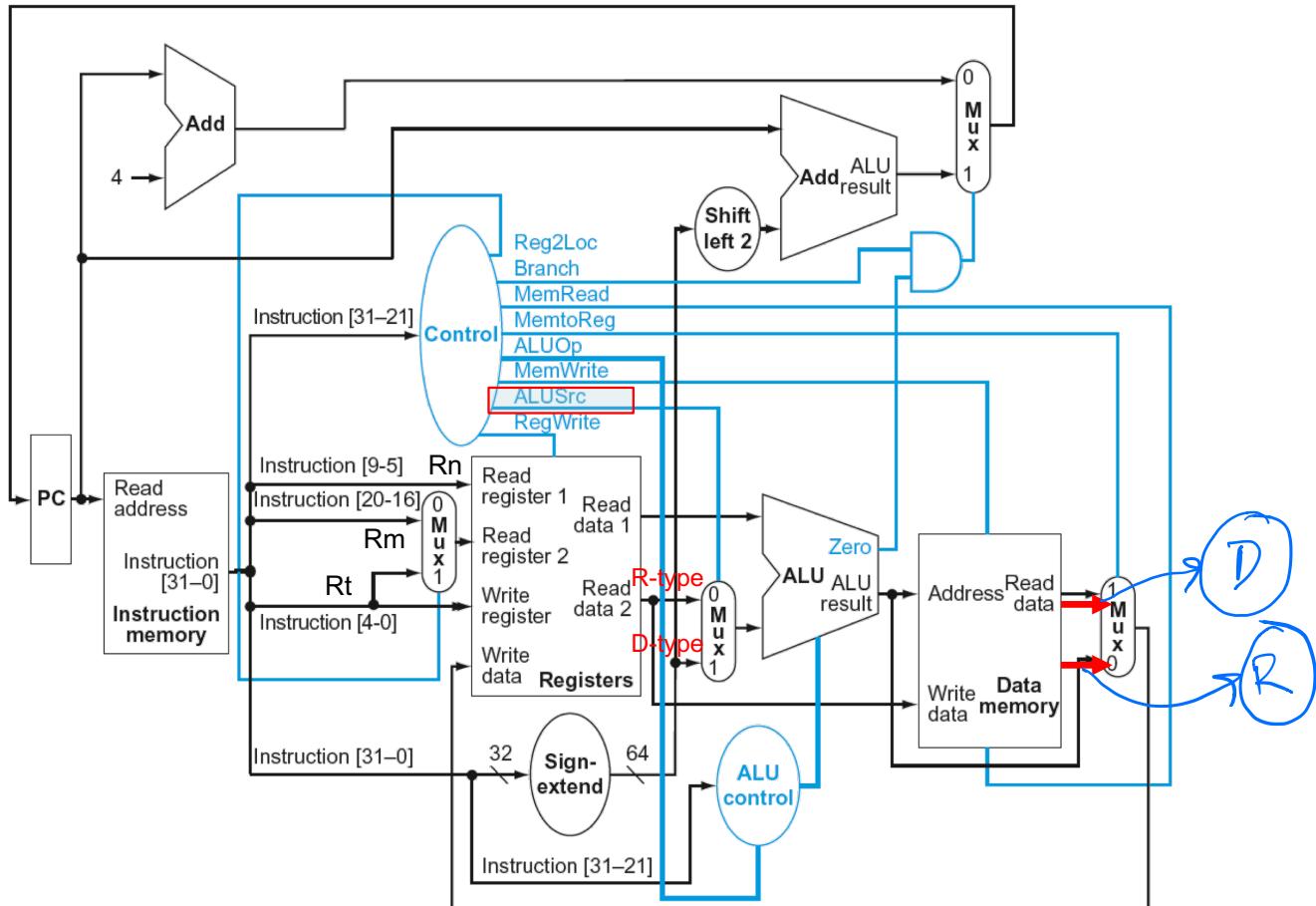
Datapath With Control



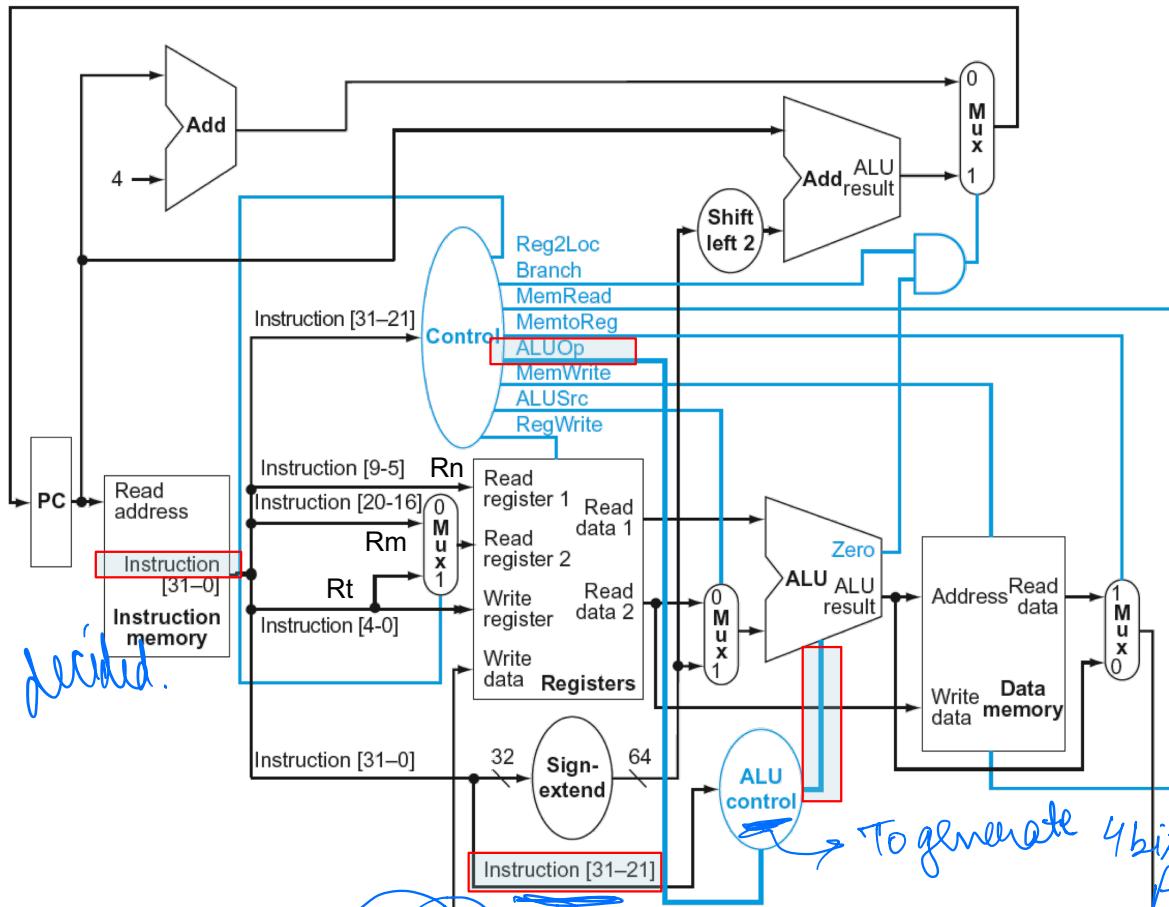
Datapath With Control



Datapath With Control



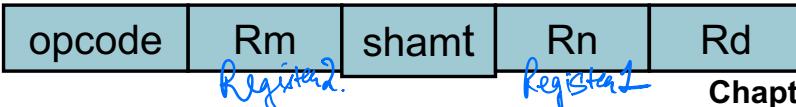
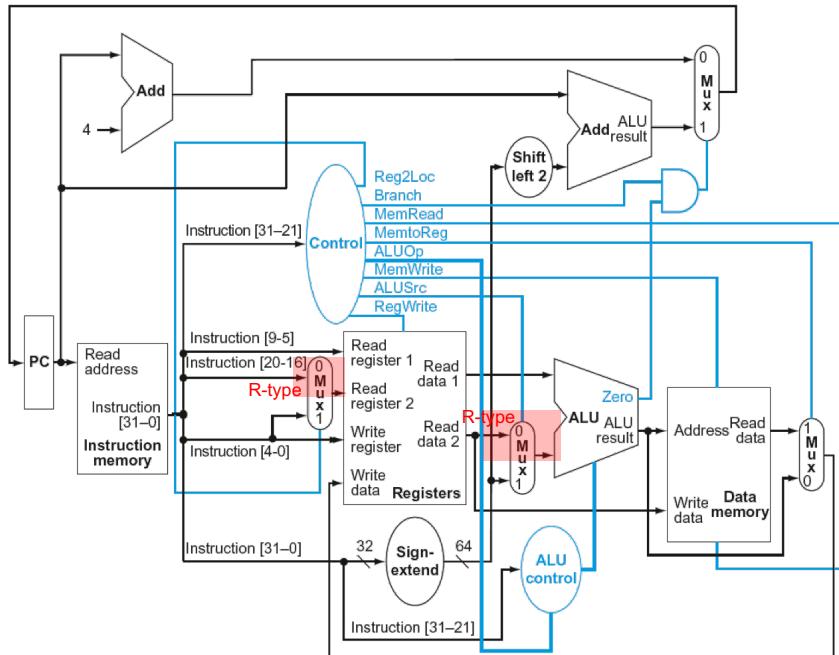
Datapath With Control



Opcode Determines Control Lines

JMP

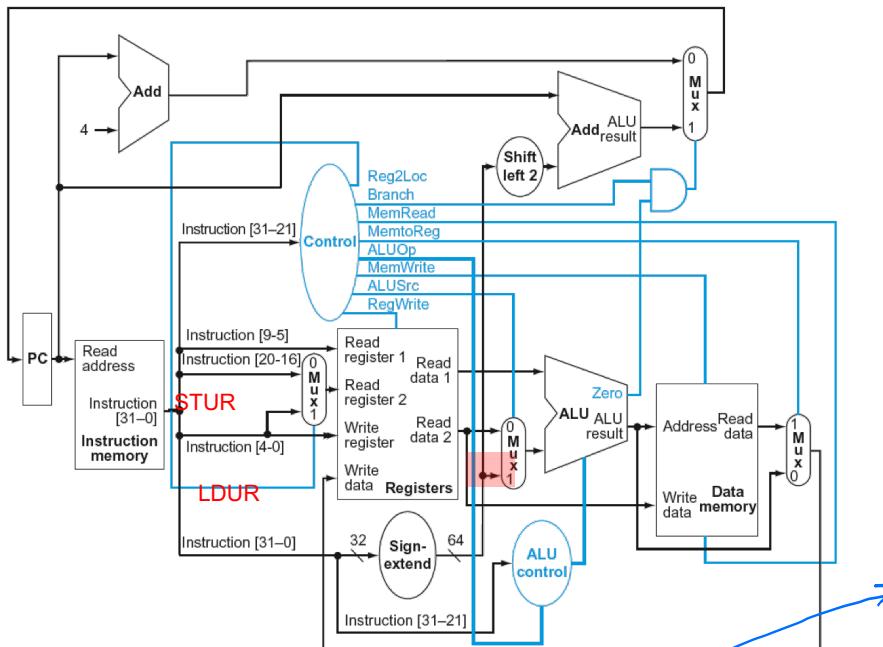
Instruction	Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	0	0	0	1	0	0	0	1	0
LDUR	X	1	1	1	1	0	0	0	0
STUR	1	1	X	0	0	1	0	0	0
CBZ	1	0	X	0	0	0	1	0	1



Opcode Determines Control Lines

Instruction	Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	0	0	0	1	0	0	0	1	0
LDUR	X	1	1	1	1	0	0	0	0
STUR	1	1	X	0	0	1	0	0	0
CBZ	1	0	X	0	0	0	1	0	1

Don't care condition.



LDUR $x9, [x22, \#64]$

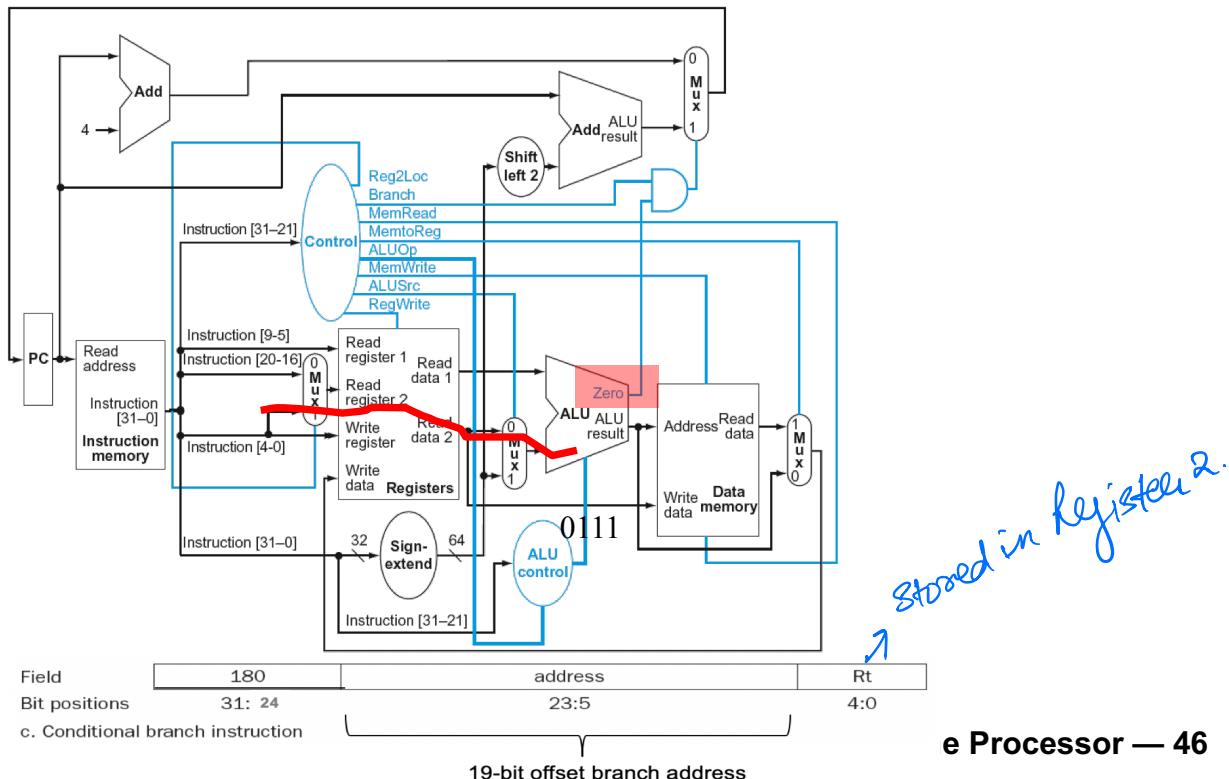
STUR $x9, [x22, \#64]$

Rt. Rn address

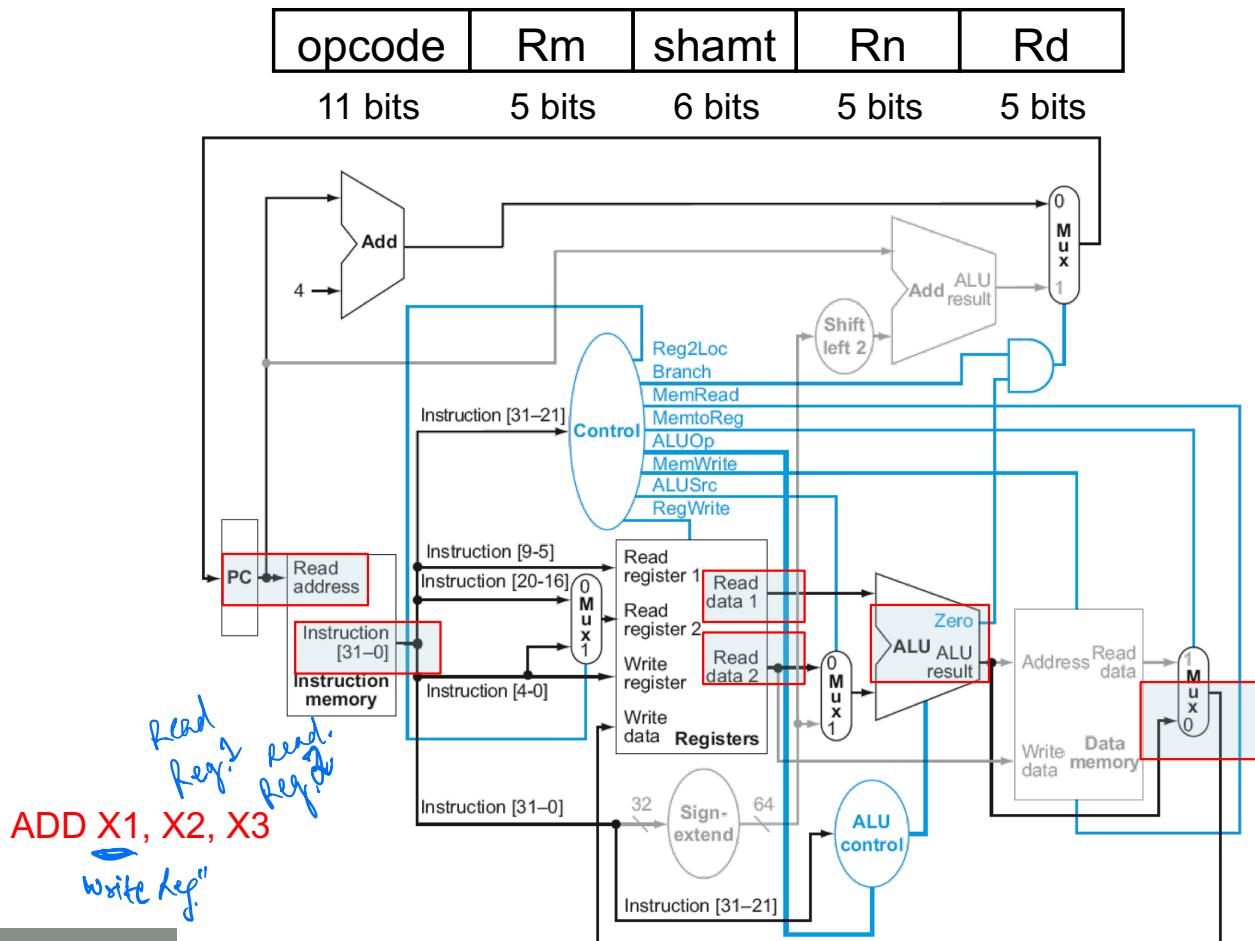
ignore in D.

Opcode Determines Control Lines

Instruction	Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	0	0	0	1	0	0	0	1	0
LDUR	X	1	1	1	1	0	0	0	0
STUR	1	1	X	0	0	1	0	0	0
CBZ	1	0	X	0	0	0	1	0	1

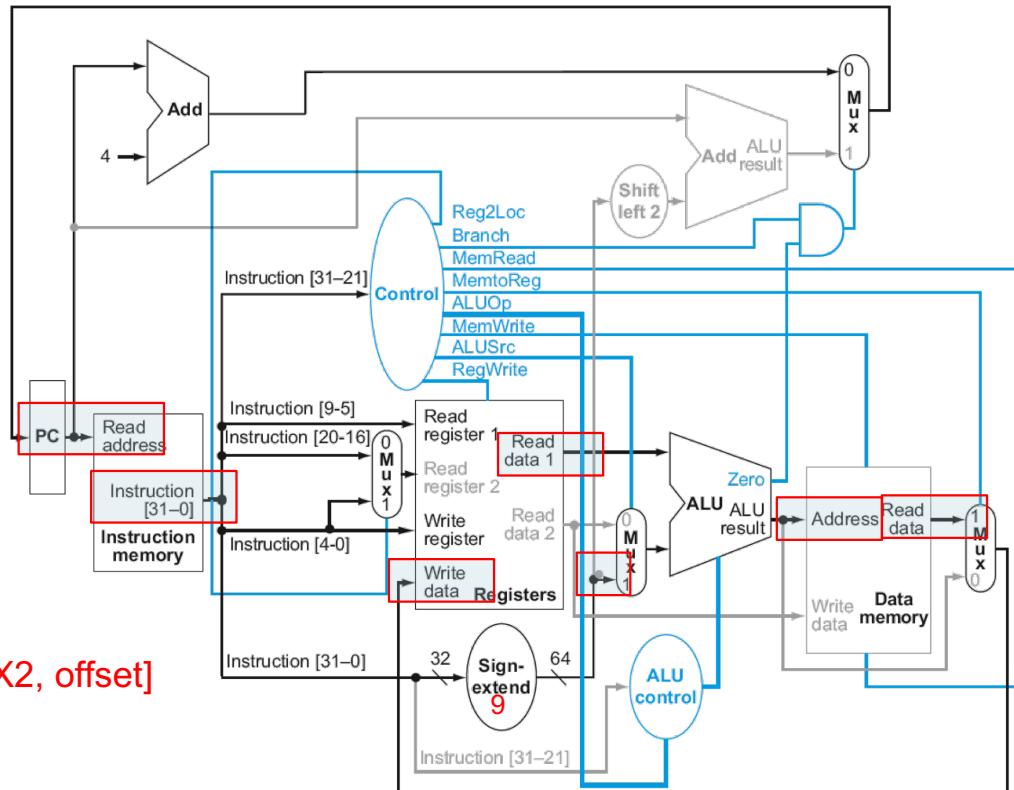


R-Type Instruction



D-Type (Load Instruction)

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits



LDUR X1, [X2, offset]

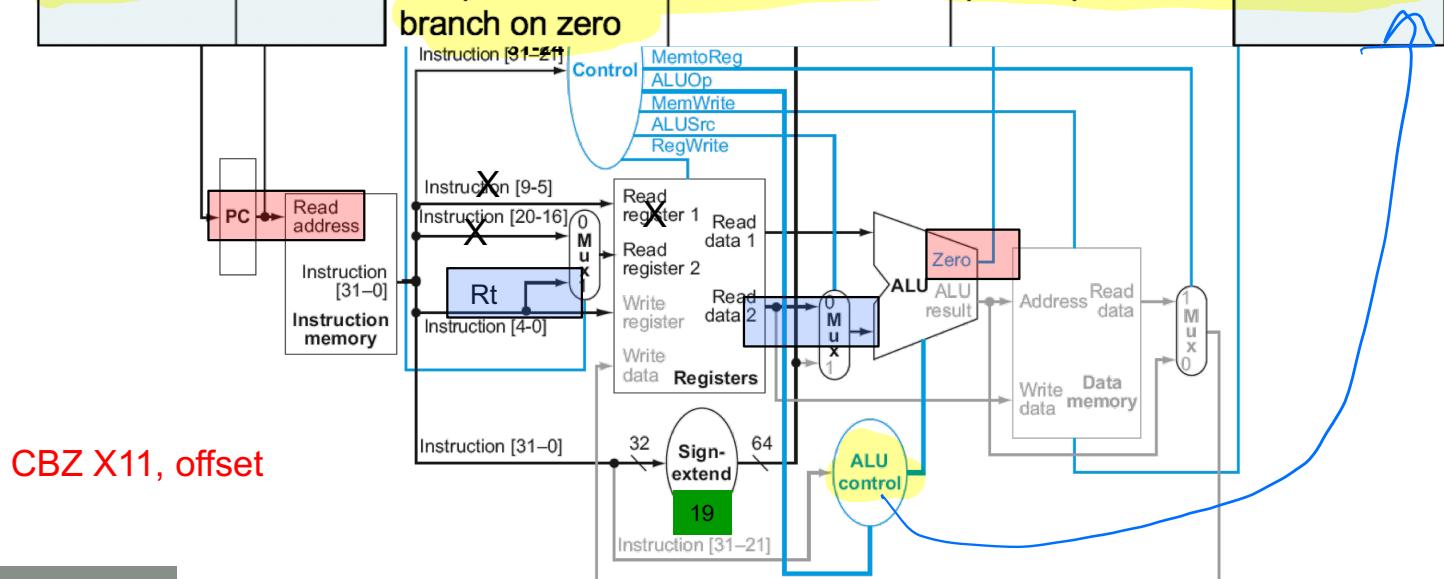
CBZ Instruction

180

Exit

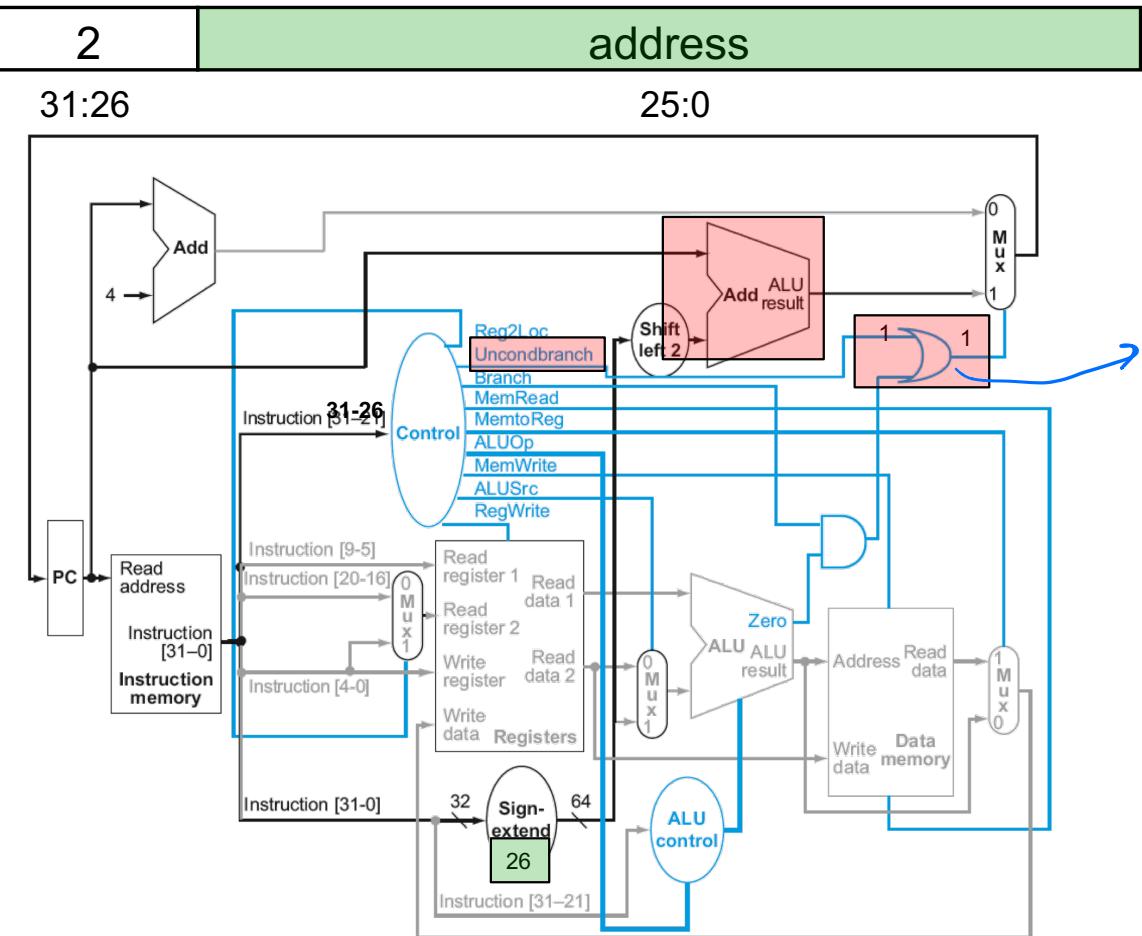
11

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
LDUR	00	load register	XXXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXXX	pass input b	0111

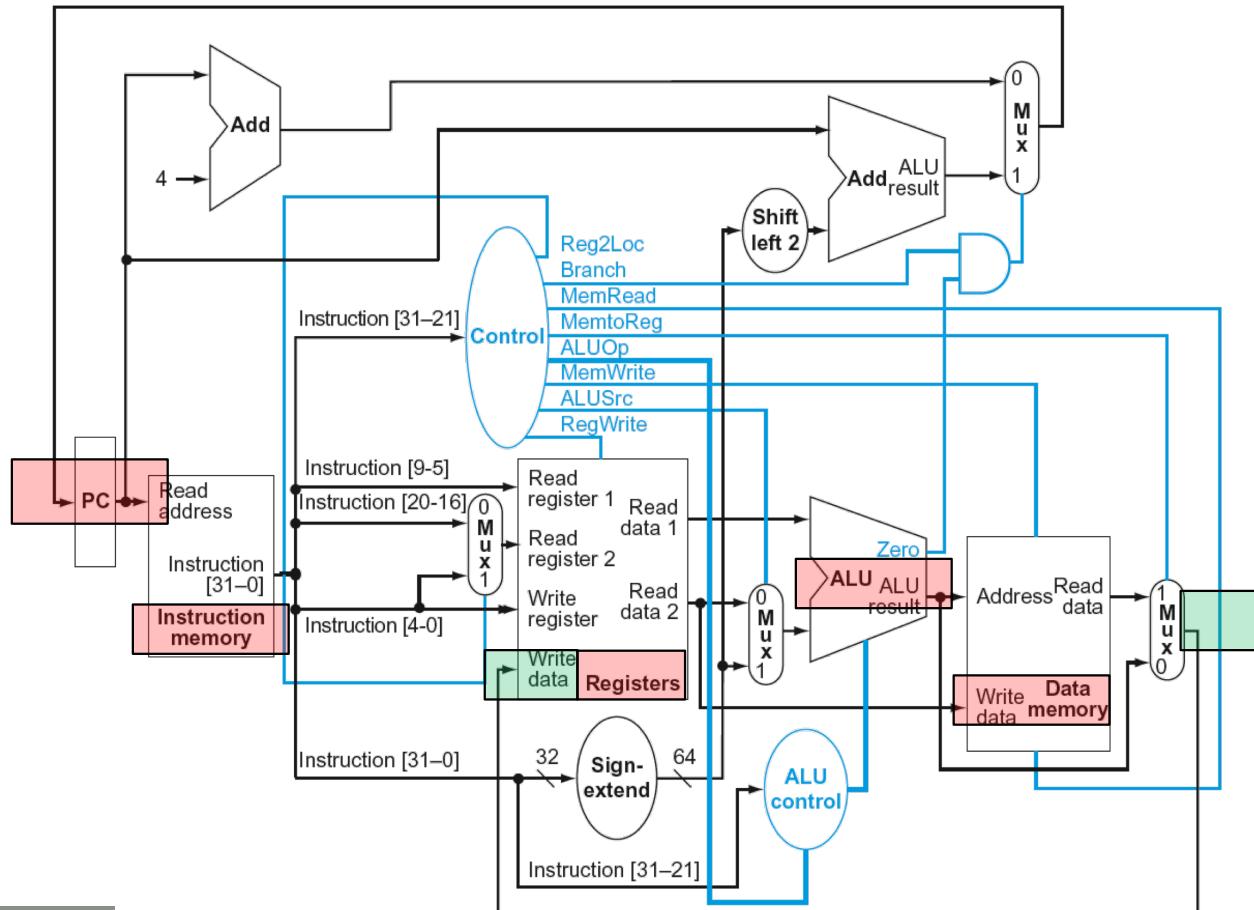


Datapath With B-type Added

unconditional Branch.

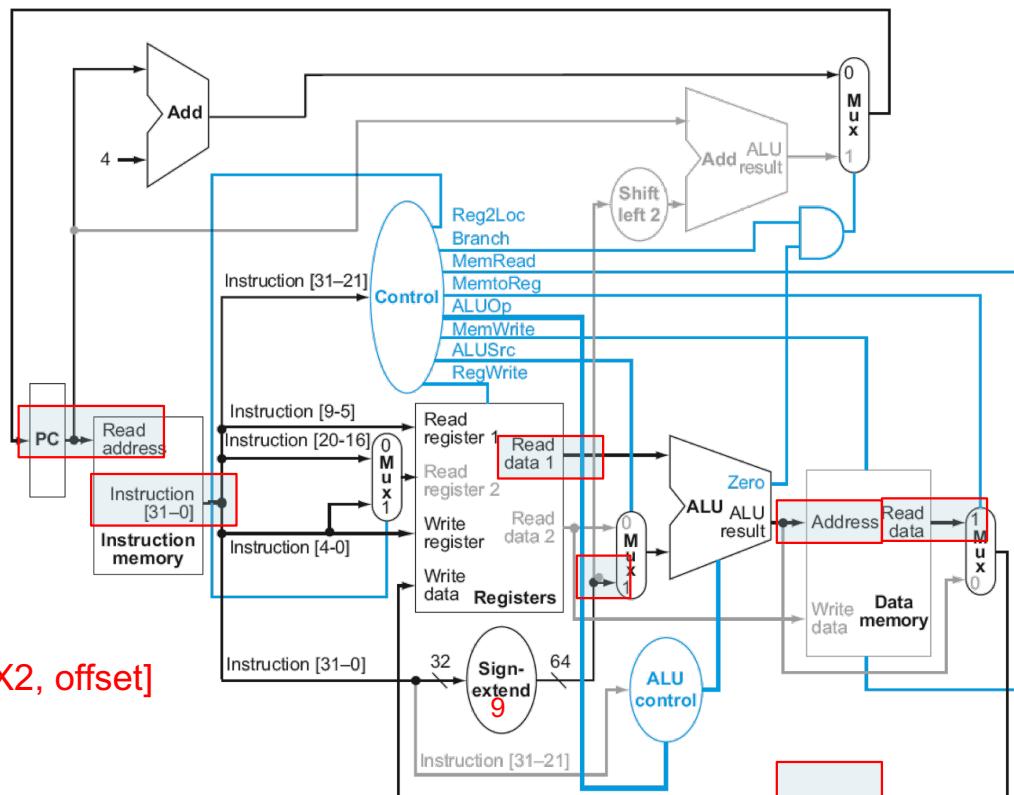


Performance Issues



D-Type (Load Instruction)

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits



LDUR X1, [X2, offset]

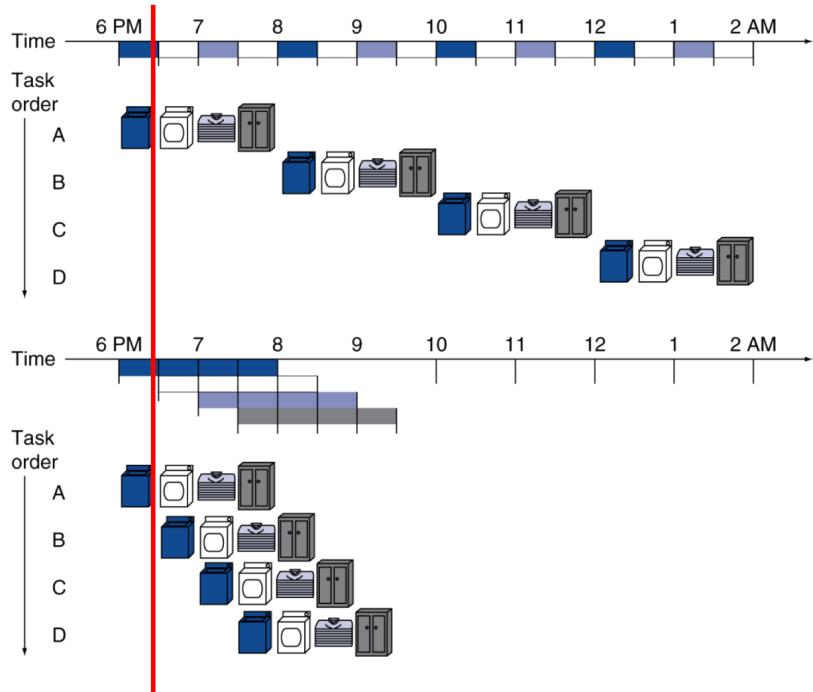
Performance Issues

- Longest delay determines clock period
 - Critical path: load instruction
 - Instruction fetch → read register file → ALU → read data from memory → write to register file
- Not feasible to vary period for different instructions
- Violates design principle
 - Making the common case fast
- We will improve performance by pipelining

OK
Throughput.

Pipelining Analogy

- Pipelined laundry: overlapping execution
 - Parallelism improves performance



LEGv8 Pipeline

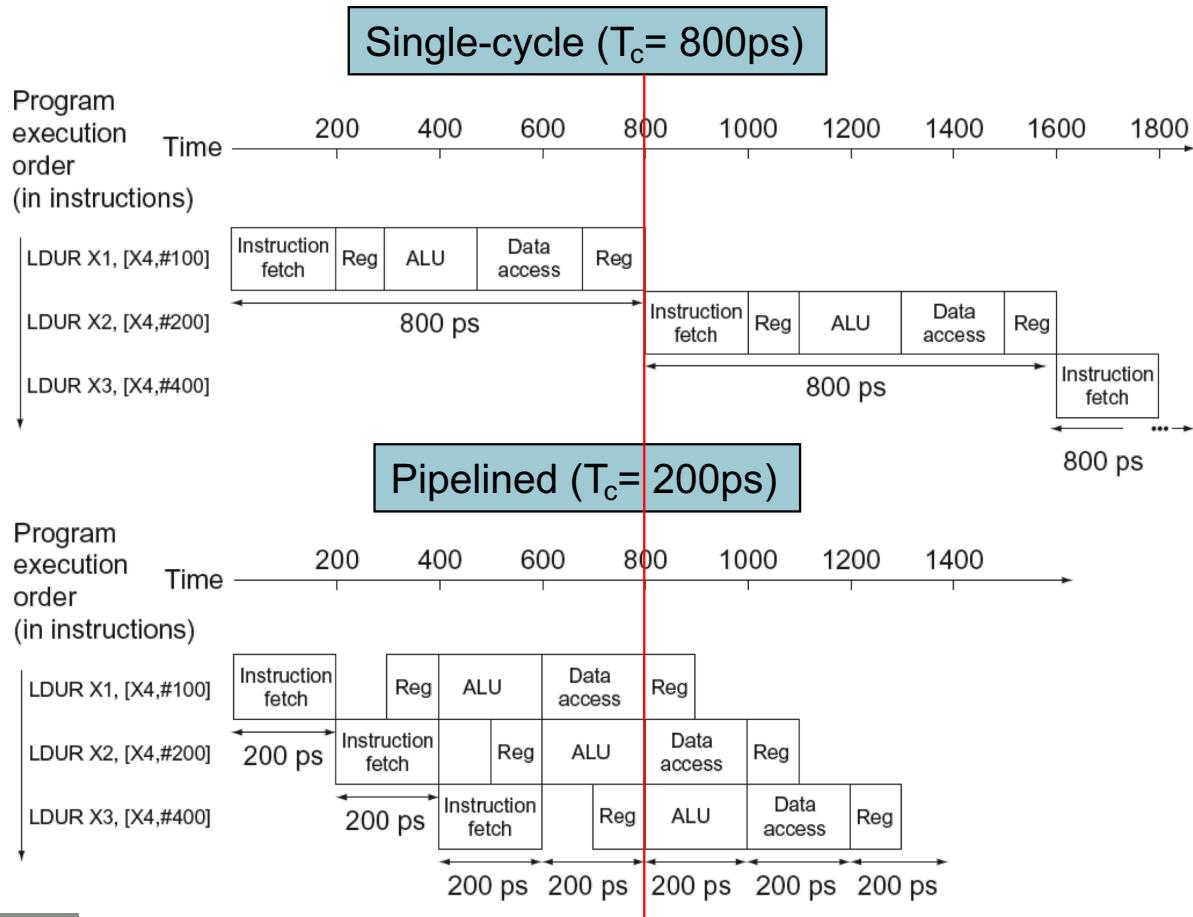
- Five stages, one step per stage
 1. IF: Instruction fetch from memory
 2. ID: Instruction decode & register read
 3. EX: Execute operation or calculate address
 4. MEM: Access memory operand
 5. WB: Write result back to register

Pipeline Performance

- For these typical LEGv8 instructions:
 - Let's assume time taken for each stage is as follows:
 - 100ps for a register read or register write
 - 200ps for other stages

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
LDUR	200ps	100 ps	200ps	200ps	100 ps	800ps
STUR	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
CBZ	200ps	100 ps	200ps			500ps

Pipeline Performance



Pipeline Performance

ps \rightarrow Pico Second.

Single-cycle ($T_c = 800\text{ps}$)

clock period.

So in the single-cycle we need to stretch the clock period to 800ps to accommodate the LDUR instruction:

Clock Rate = $(1 / \text{Clock Period}) = (1 / 800 \text{ ps}) = [1 / (800 \times 10^{-12})] = 1.25 \text{ GHz}$
(or 1.25 billions cycles per second.)

Pipelined ($T_c = 200\text{ps}$)

Versus in the pipelined design our clock period can be about 200 ps to accommodate the same load instruction:

Clock Rate = $(1 / \text{Clock Period}) = (1 / 200 \text{ ps}) = [1 / (200 \times 10^{-12})] = 5 \text{ GHz}$ (or 5 billions cycles per second.)



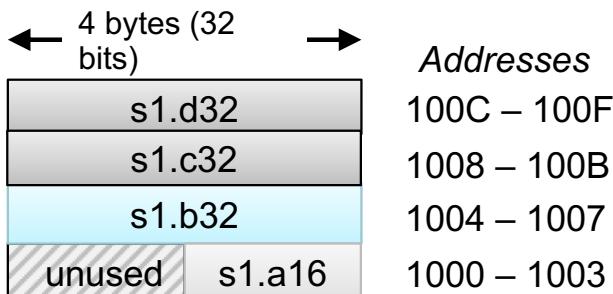
Pipelining and ISA Design

- LEGv8 ISA designed for pipelining
 - All instructions are 32-bits
 - Easier to fetch and decode in one cycle
 - Few and regular instruction formats
 - Can decode and read registers in one step

Name	Fields							Comments
Field size	6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	All LEGv8 instructions are 32 bits long	
R-format	R	opcode	Rm	shamt	Rn	Rd	Arithmetic instruction format	
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format
B-format	B	opcode	address					Unconditional Branch format
CB-format	CB	opcode	address				Rt	Conditional Branch format
IW-format	IW	opcode	immediate				Rd	Wide Immediate format

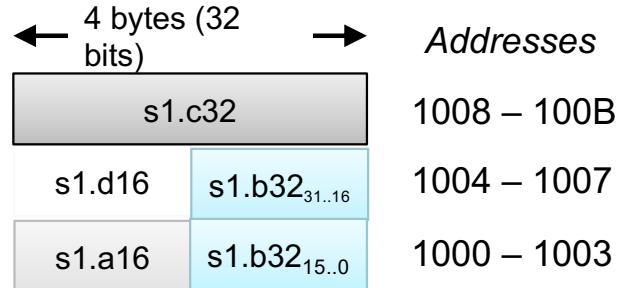
Pipelining and ISA Design

- Alignment of memory operands in LEGv8
 - Memory access takes only one cycle



Optimized for speed (default)

Cursive note: The entire `s1.b32` instruction is at 1004 address in memory and therefore, can be accessed in one clock cycle.



Optimized to conserve memory

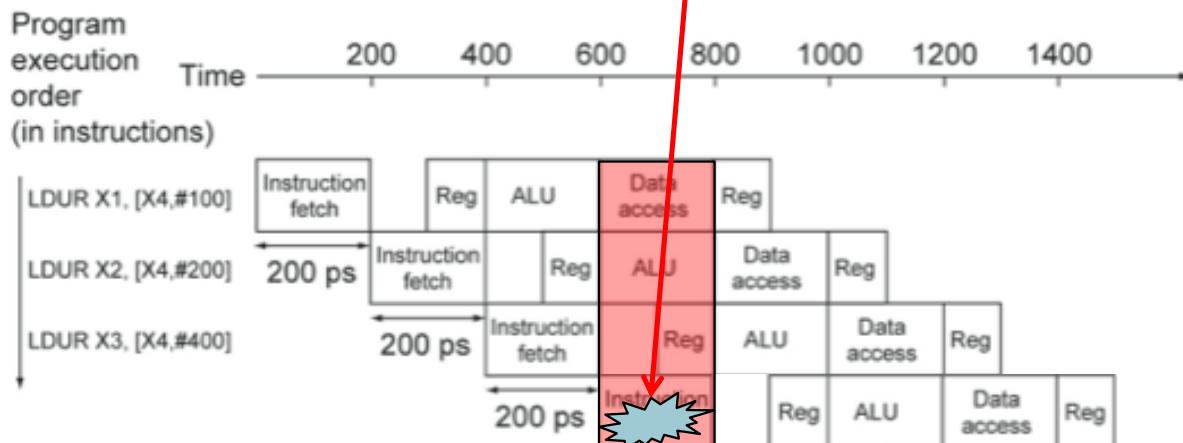
First half of the `s1.b32` is at 1000 and the second half is at 1004 memory location so requires two memory accesses.

Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
 - A required resource is busy *e.g. memory*.
- Data hazard
 - Need to wait for previous instruction to complete its data read/write
- Control hazard
 - Deciding on control action depends on previous instruction

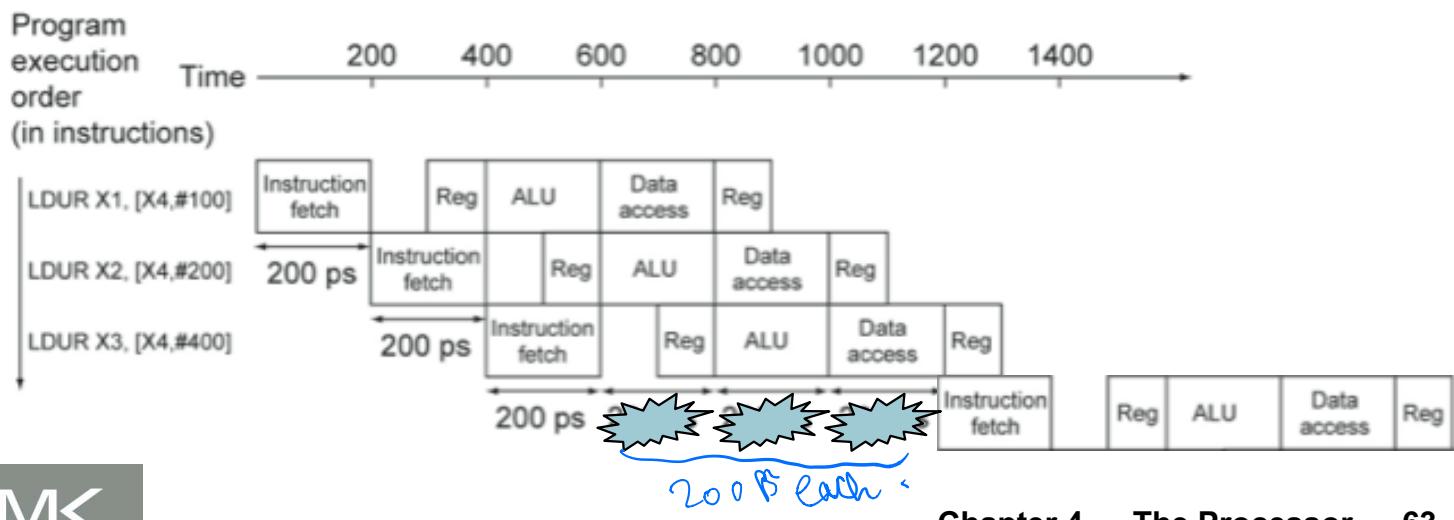
Structure Hazards

- Conflict for use of a resource
- In LEGv8 pipeline with a single memory
 - Load requires instruction/data memory access
 - Instruction fetch would have to *stall* for that cycle
 - Would cause a pipeline “bubble”



Structure Hazards

- Conflict for use of a resource
- In LEGv8 pipeline with a single memory
 - Load requires instruction/data memory access
 - Instruction fetch would have to *stall* for 600 ps
 - Would cause three pipeline “bubbles”

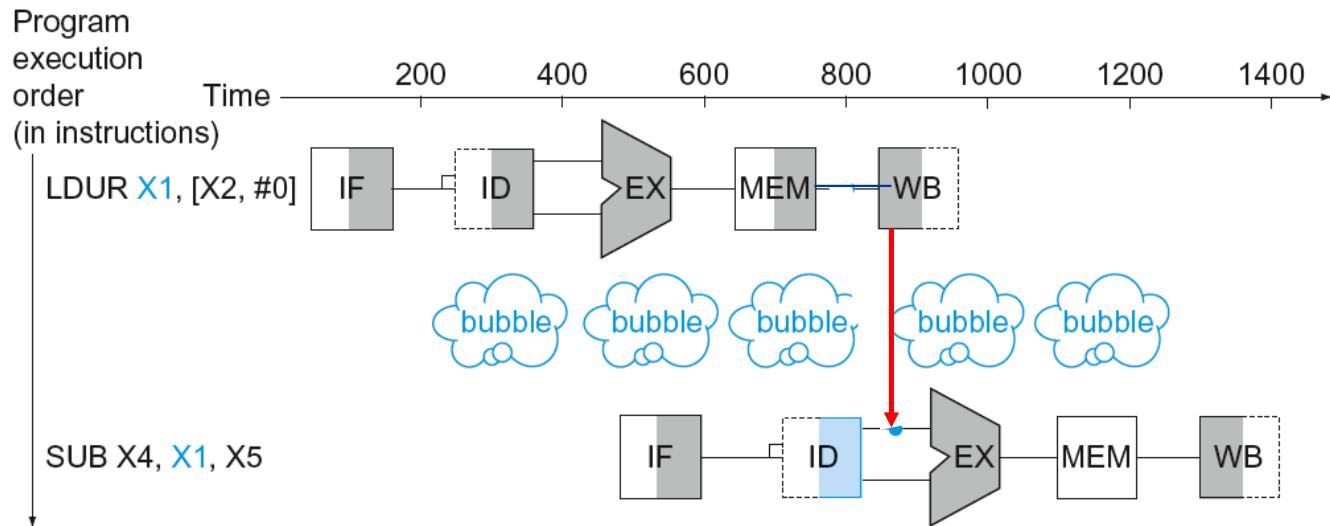


Structure Hazards

- Conflict for use of a resource
- In LEGv8 pipeline with a single memory
 - Load/store requires data access
 - Instruction fetch would have to *stall* for that cycle
 - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
 - Or separate instruction/data caches

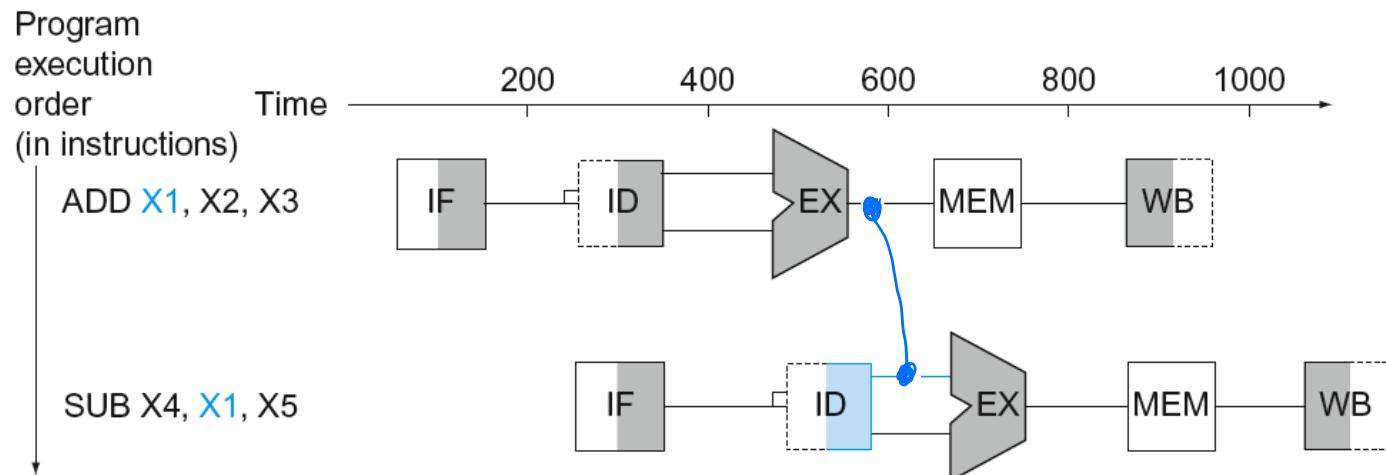
Data Hazards

- An instruction depends on completion of data access by a previous instruction
 - LDUR $x1$, [$x2$, #0]
 - SUB $x4$, $x1$, $x5$



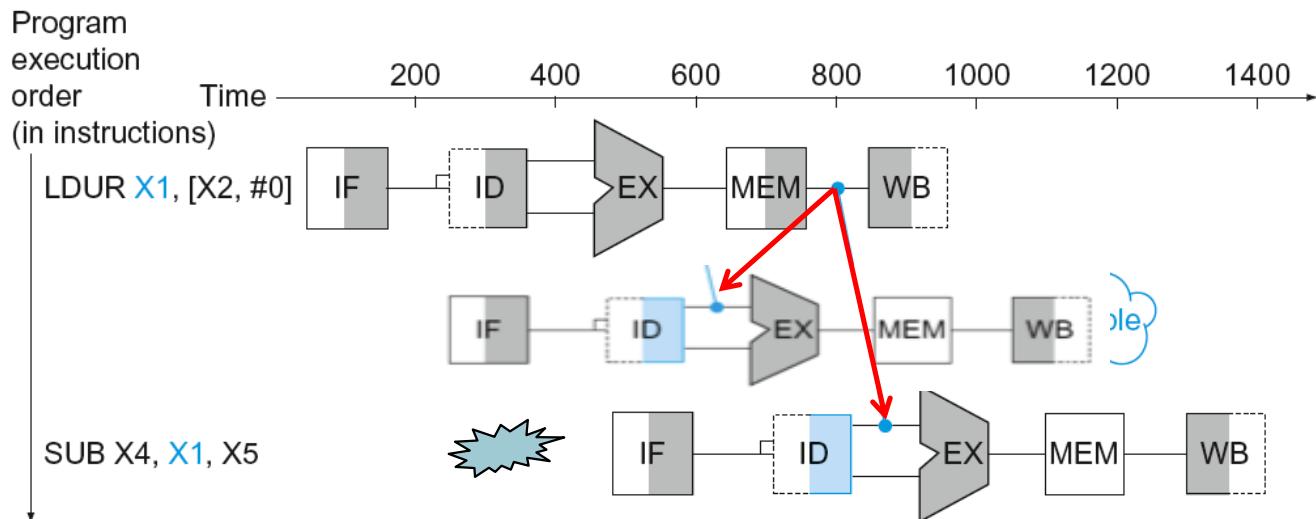
Forwarding (aka Bypassing)

- Use result when it is computed
 - Don't wait for it to be stored in a register
 - Requires extra connections in the datapath



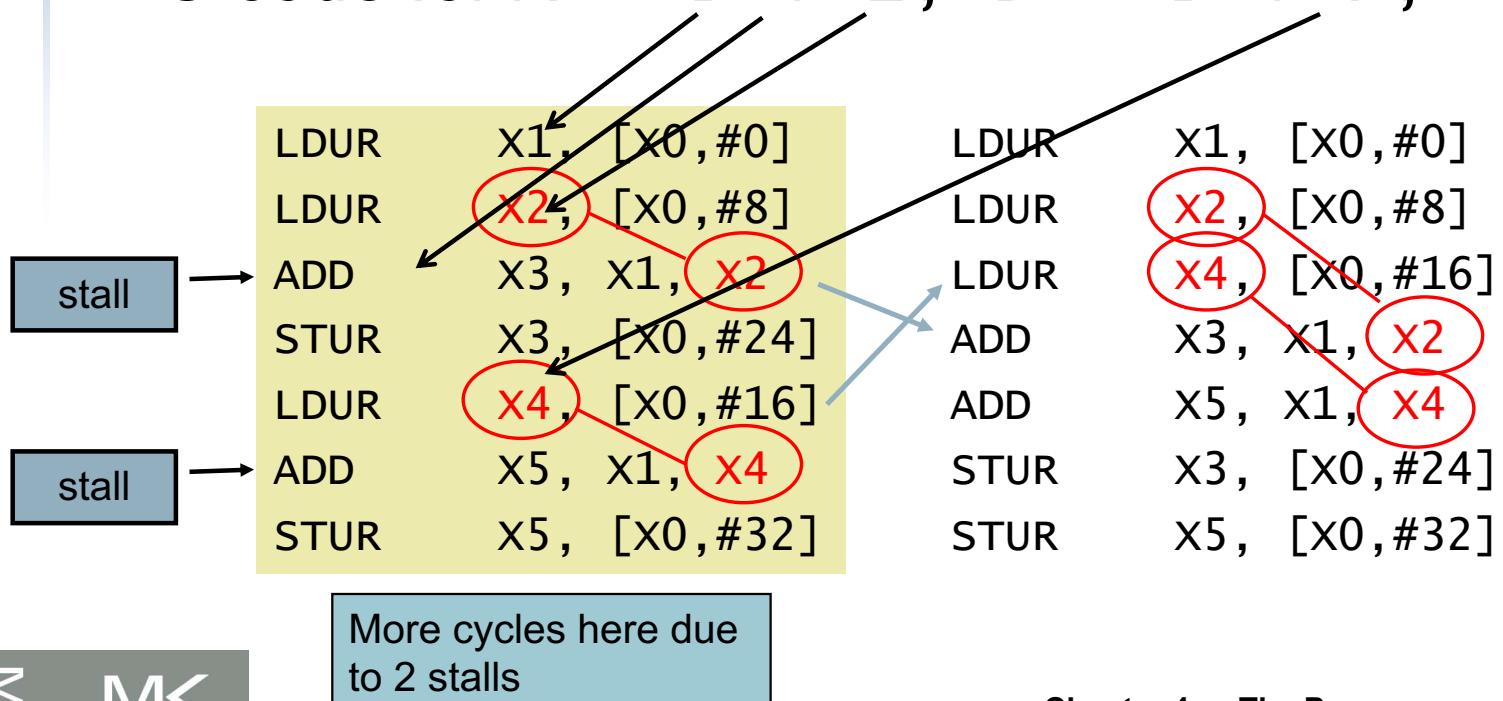
Load-Use Data Hazard

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



Code Scheduling to Avoid Stalls

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





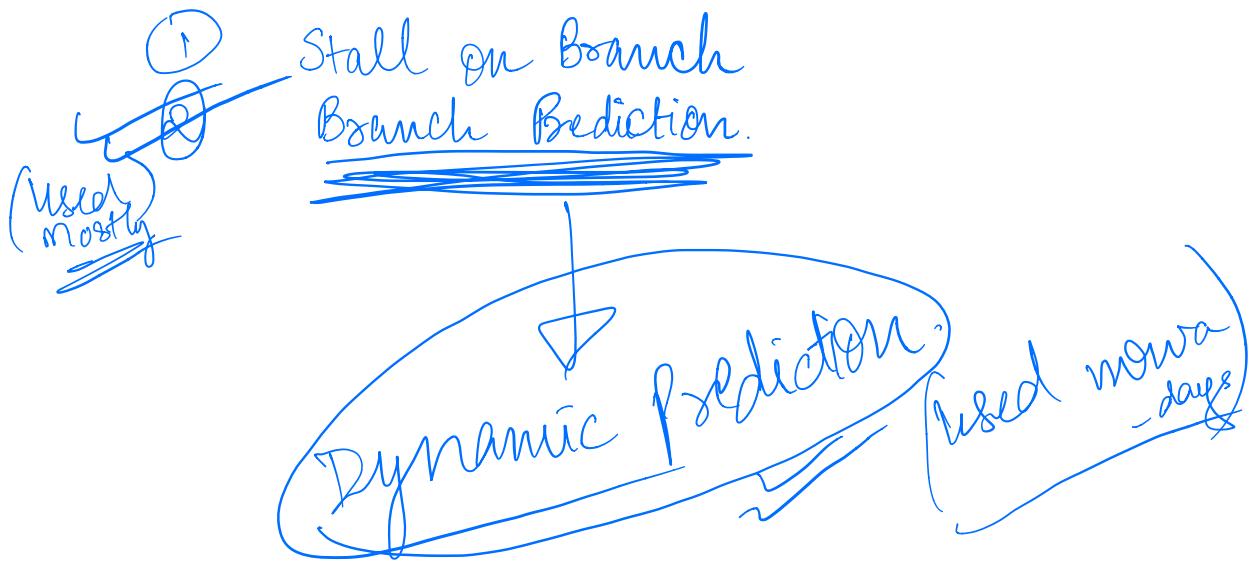
Chapter 4

The Processor – Part 2

Control Hazards

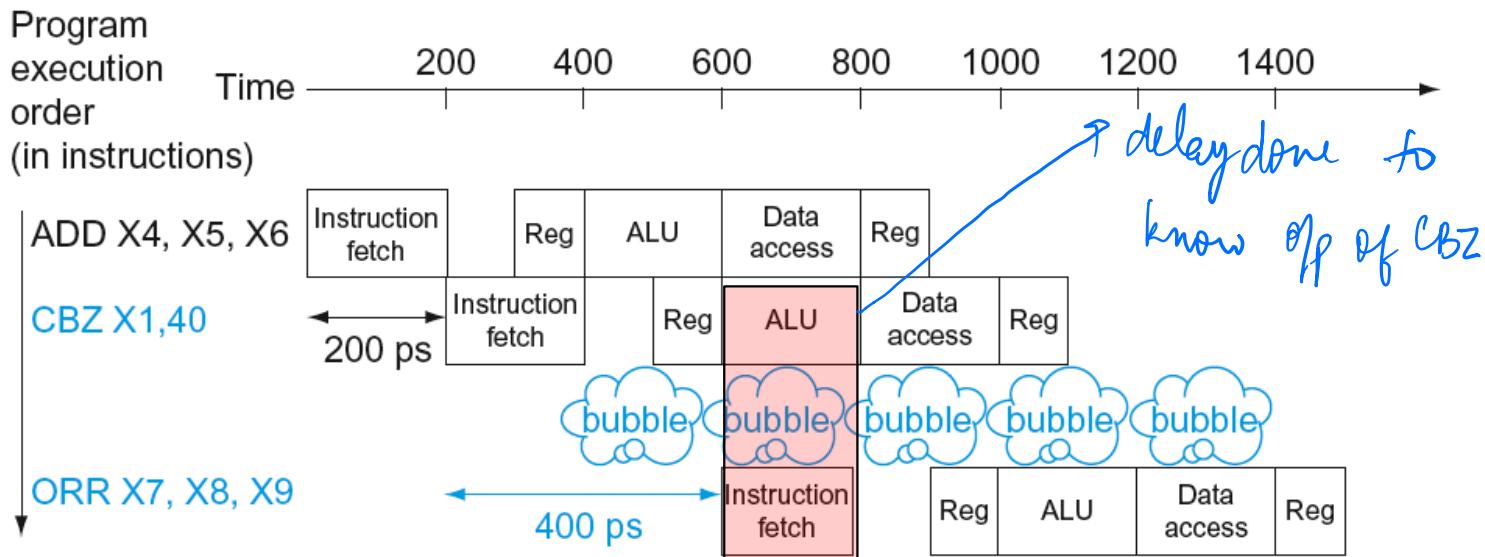
- Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - In LEGv8 pipeline
 - Need to compare registers and compute target early in the pipeline
 - Add hardware to find the outcome in earlier stages or use prediction
- PC+4 : SUB X1, X2, X3
CBZ X1, 40
LDUR X3, [X0, #0]
- PC+40 : ORR X7, X8, X9

2 ways to eliminate Control Hazards:-



Stall on Branch

- Wait until branch outcome determined before fetching next instruction



Stall on Branch

- Longer pipelines can't readily determine branch outcome early
 - Stall penalty becomes unacceptable
- Predict outcome of branch
- Only stall if prediction is wrong
- 2nd option*

Branch Prediction

- Predict outcome of branch
 - Only stall if prediction is wrong
- In LEGv8 pipeline
 - Predict branches are not taken
 - Fetch instruction after branch instruction , with no delay

We assume that
branches & not
to be taken. x ; go to
the next inst. immediately.

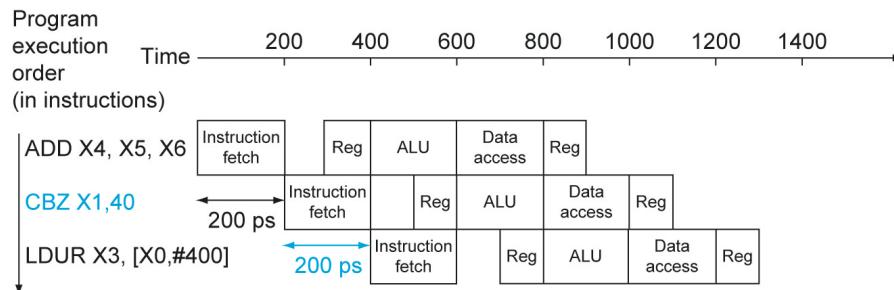
CBZ X1, 40

PC+4 : LDUR X3, [X0, #400]

PC+40 : ORR X7, X8, X9

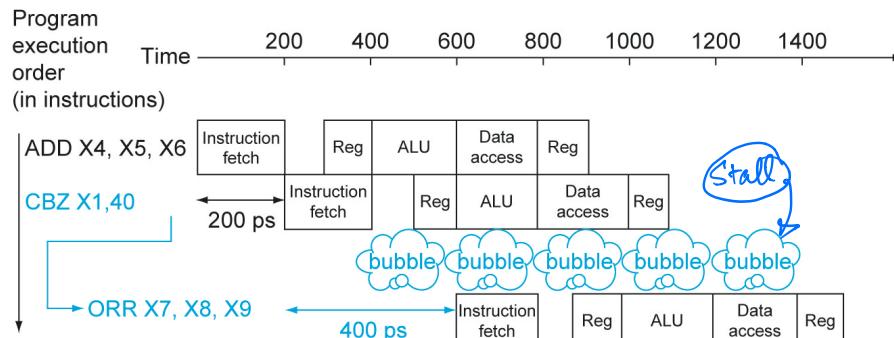
Branch Prediction

- Predict branches not taken



- What if the prediction is wrong?

↳ then discard the instr. we fetched & fetch the instr. at target address.



More-Realistic Branch Prediction

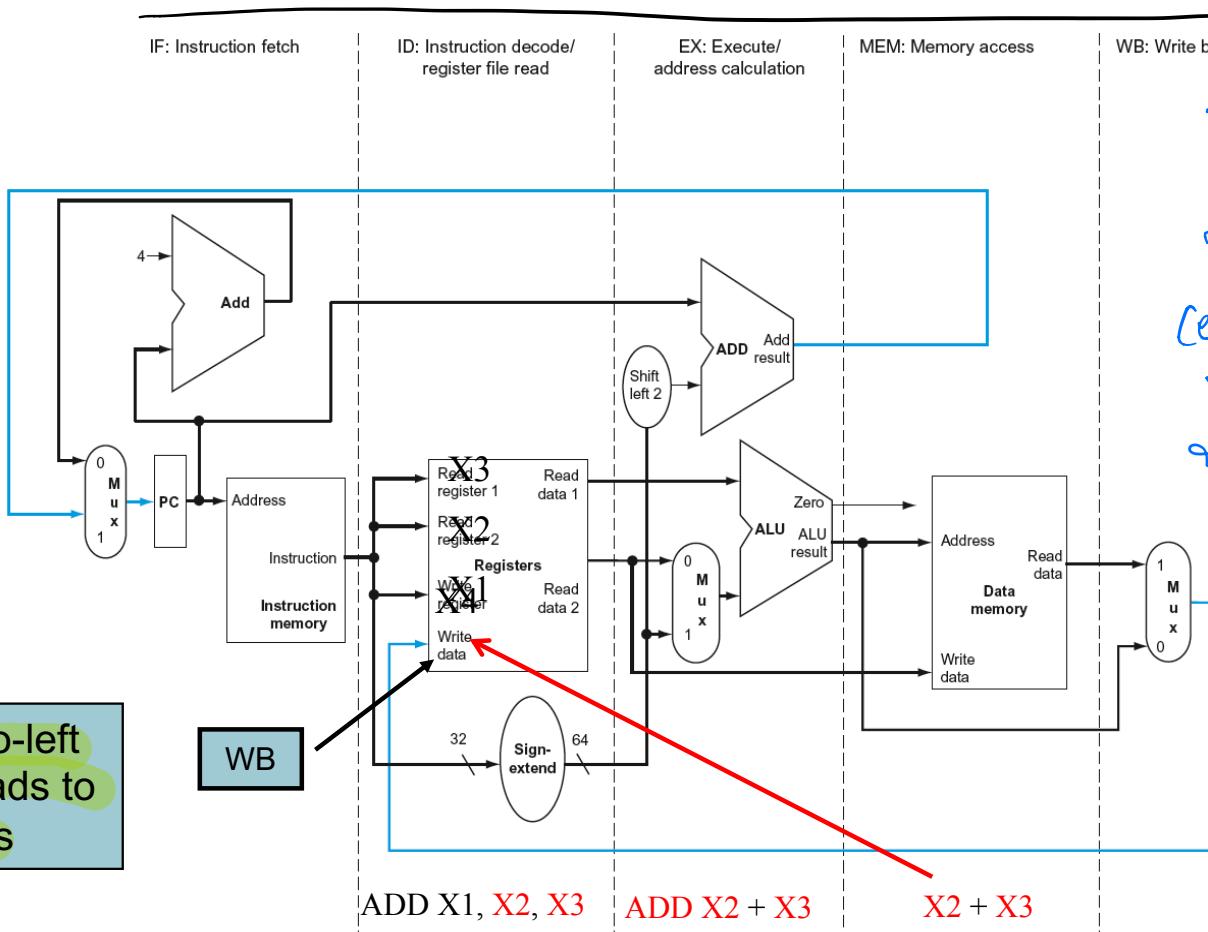
- Dynamic branch prediction

- Hardware measures actual branch behavior
- Assume future behavior will continue the trend

Pipeline Summary (So Far)

- Pipelining improves performance by increasing instruction throughput
 - Subject to hazards
 - Structure, data, control
 - Instruction set design affects complexity of pipeline implementation
- Same latency is same for all "instr."*

LEGv8 Pipelined Datapath



Right-to-left
flow leads to
hazards

ADD X4, X5, X6

ADD X1, X2, X3

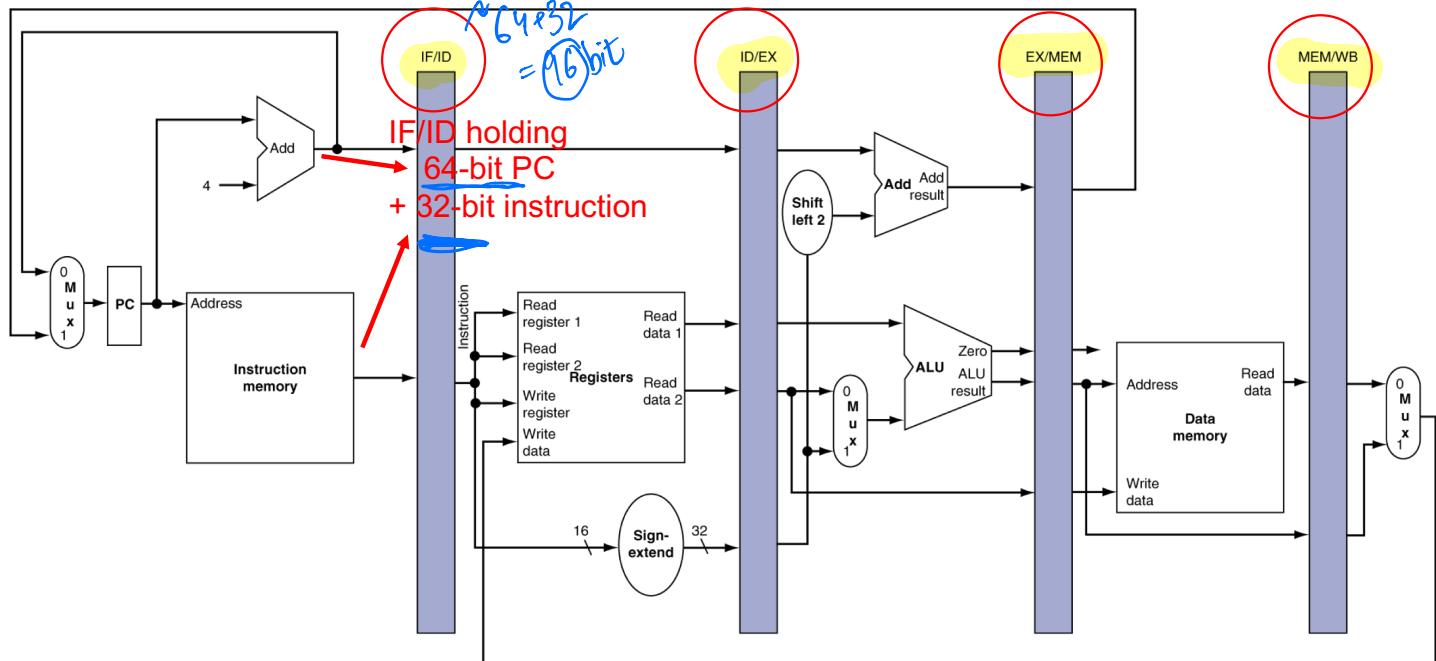
Chapter 4 — The Processor — 9

~~5 Stages:-~~

- ① IF (Inst. fetch)
- ② ID (" Decode
Register file read.)
- ③ EX (Execute OR Address Cal.)
- ④ MEM (Memory Access).
- ⑤ WB (Write Back).

Pipeline registers

- Need registers between stages
 - To hold information produced in previous cycle

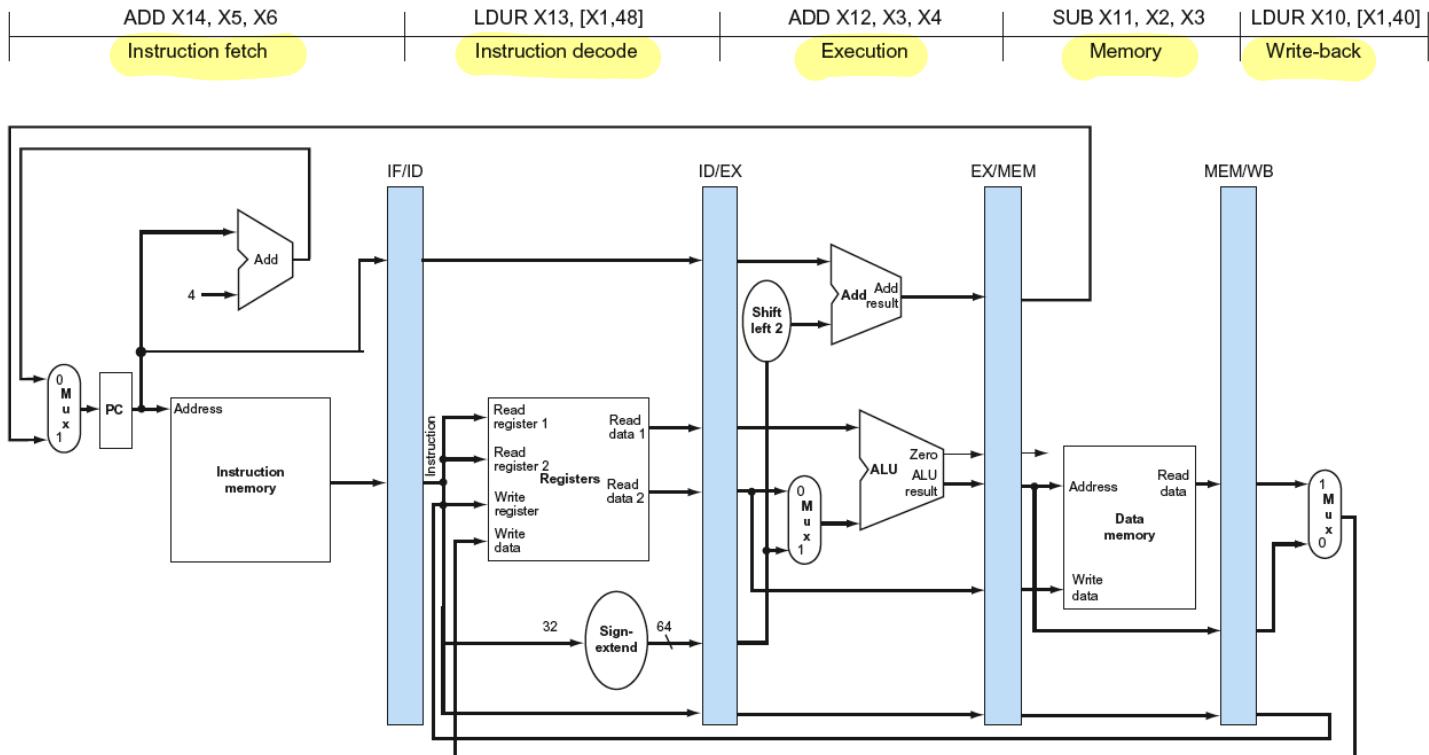


Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
 - “Single-clock-cycle” pipeline diagram
 - Shows pipeline usage (snapshot) in a single cycle
 - Highlight resources used
 - c.f. “multi-clock-cycle” diagram
 - Graph of operation/instructions over time

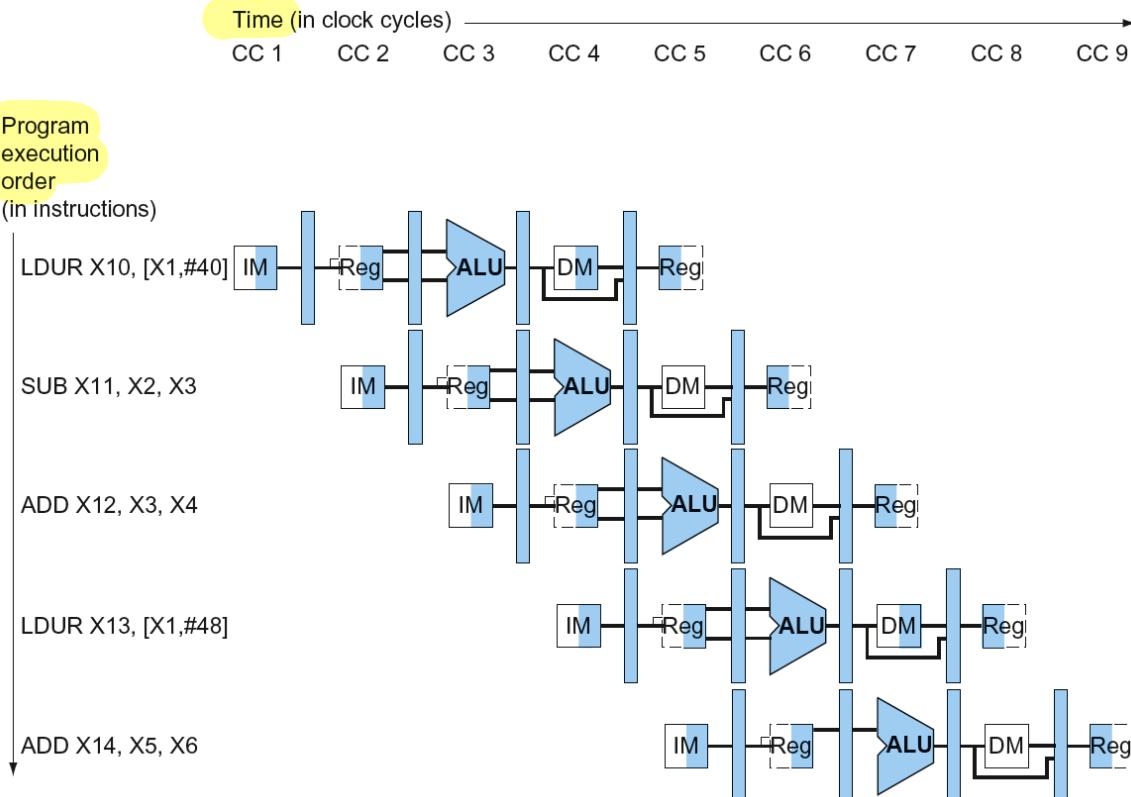
Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle



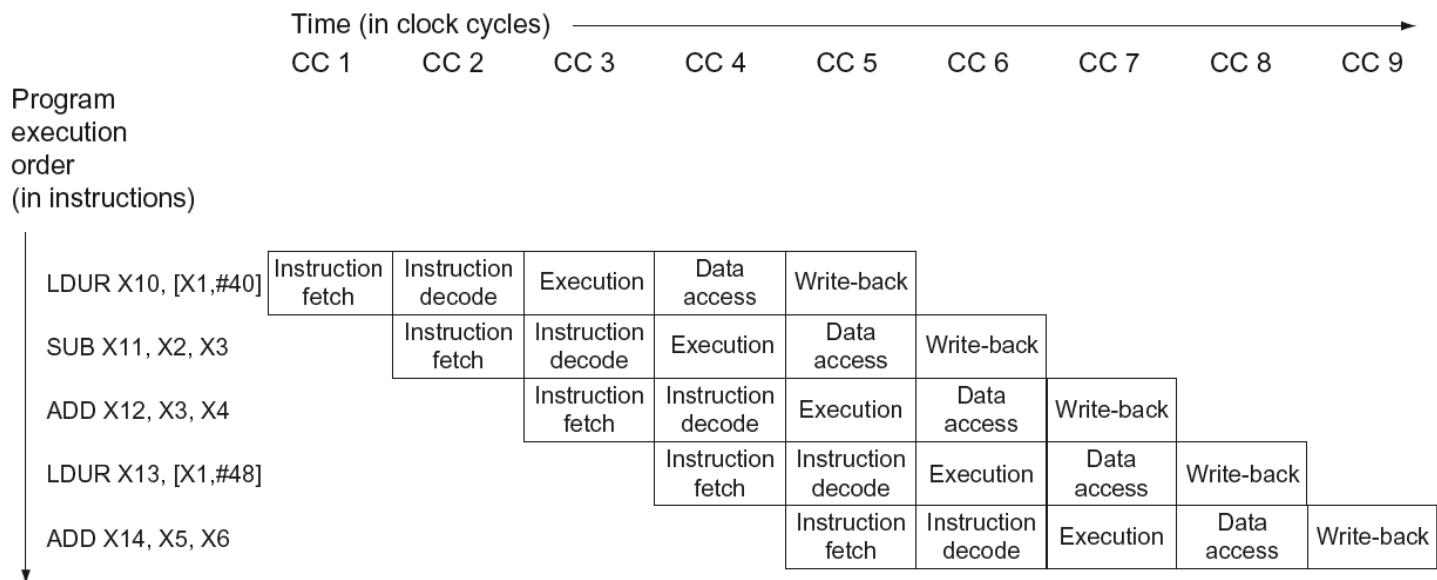
Multi-Cycle Pipeline Diagram

- Form showing resource usage



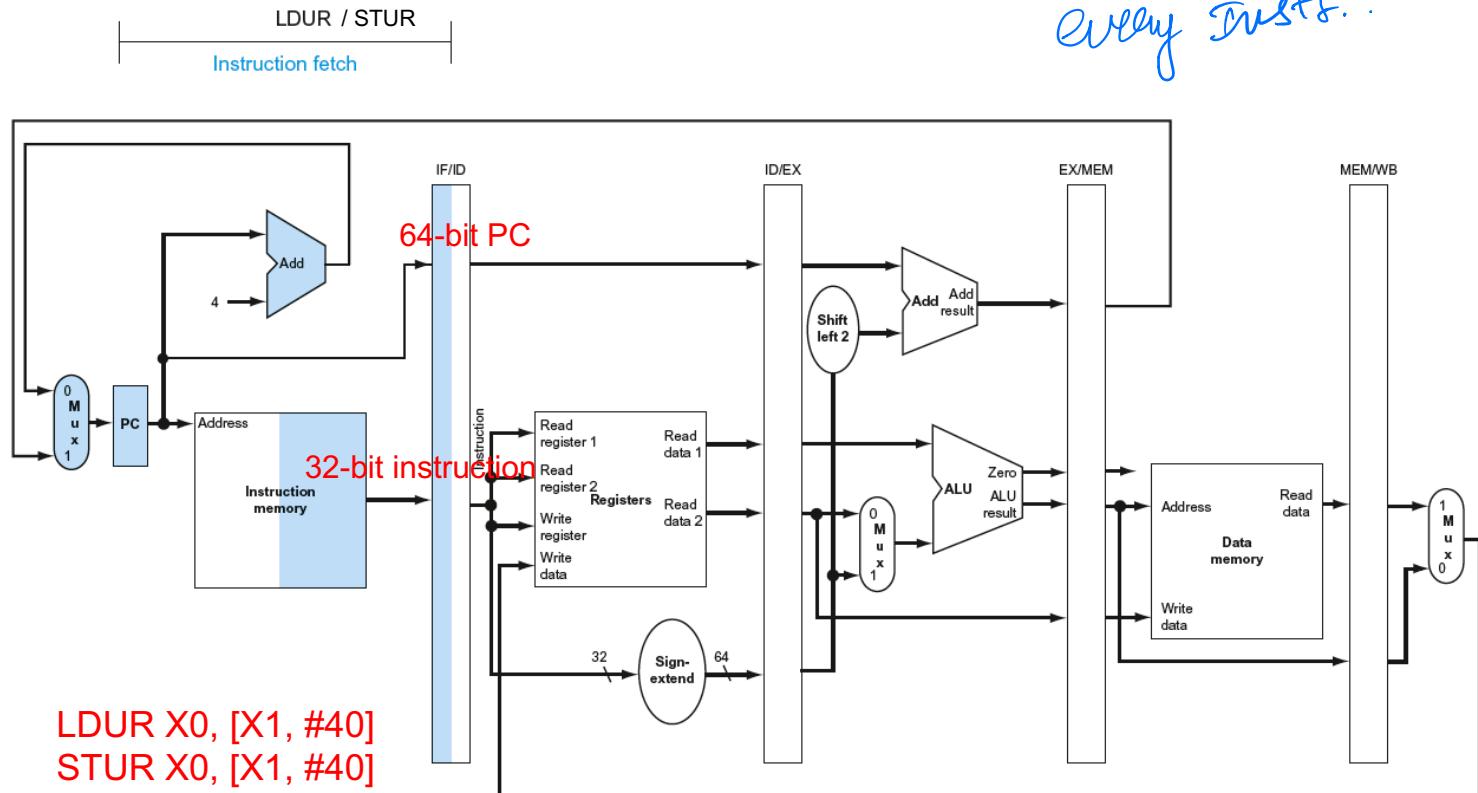
Multi-Cycle Pipeline Diagram

■ Traditional form

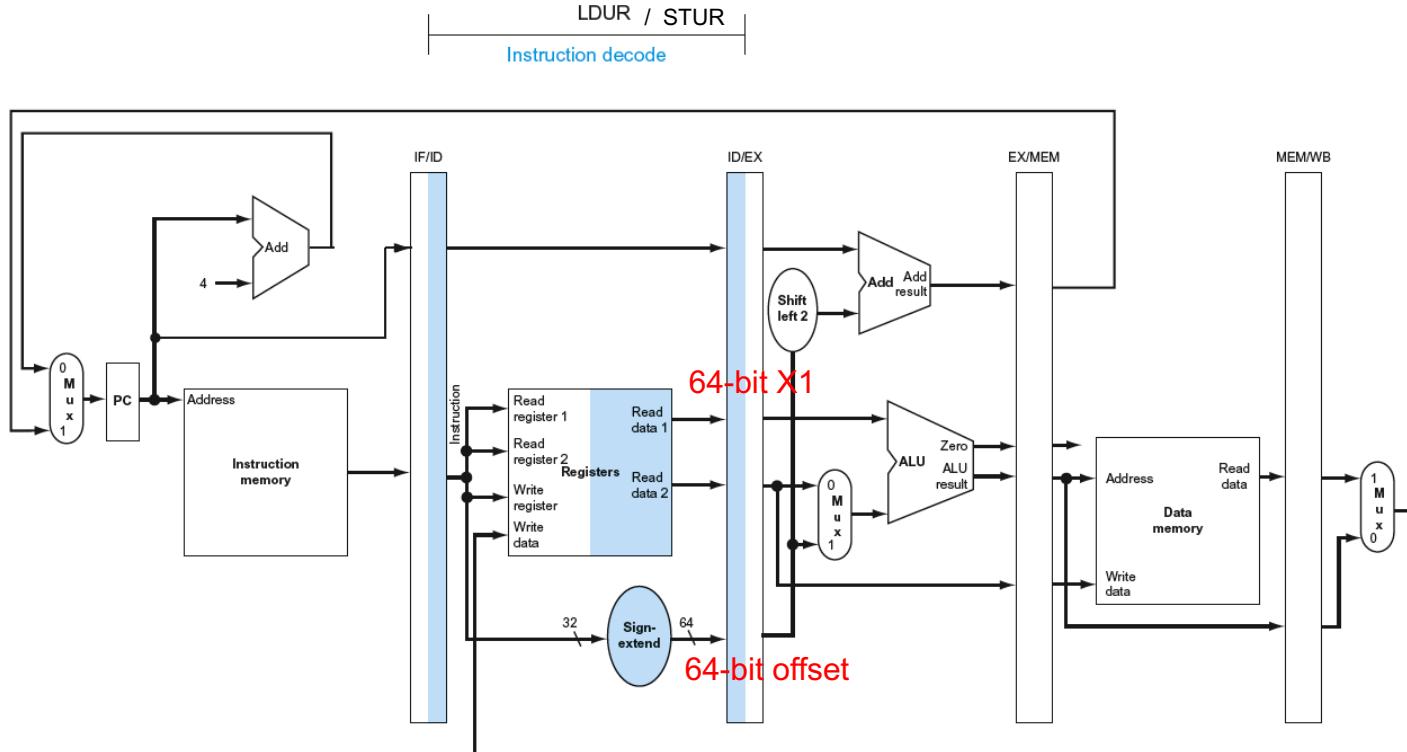


IF for Load, Store, ...

If is passed by every Insts.

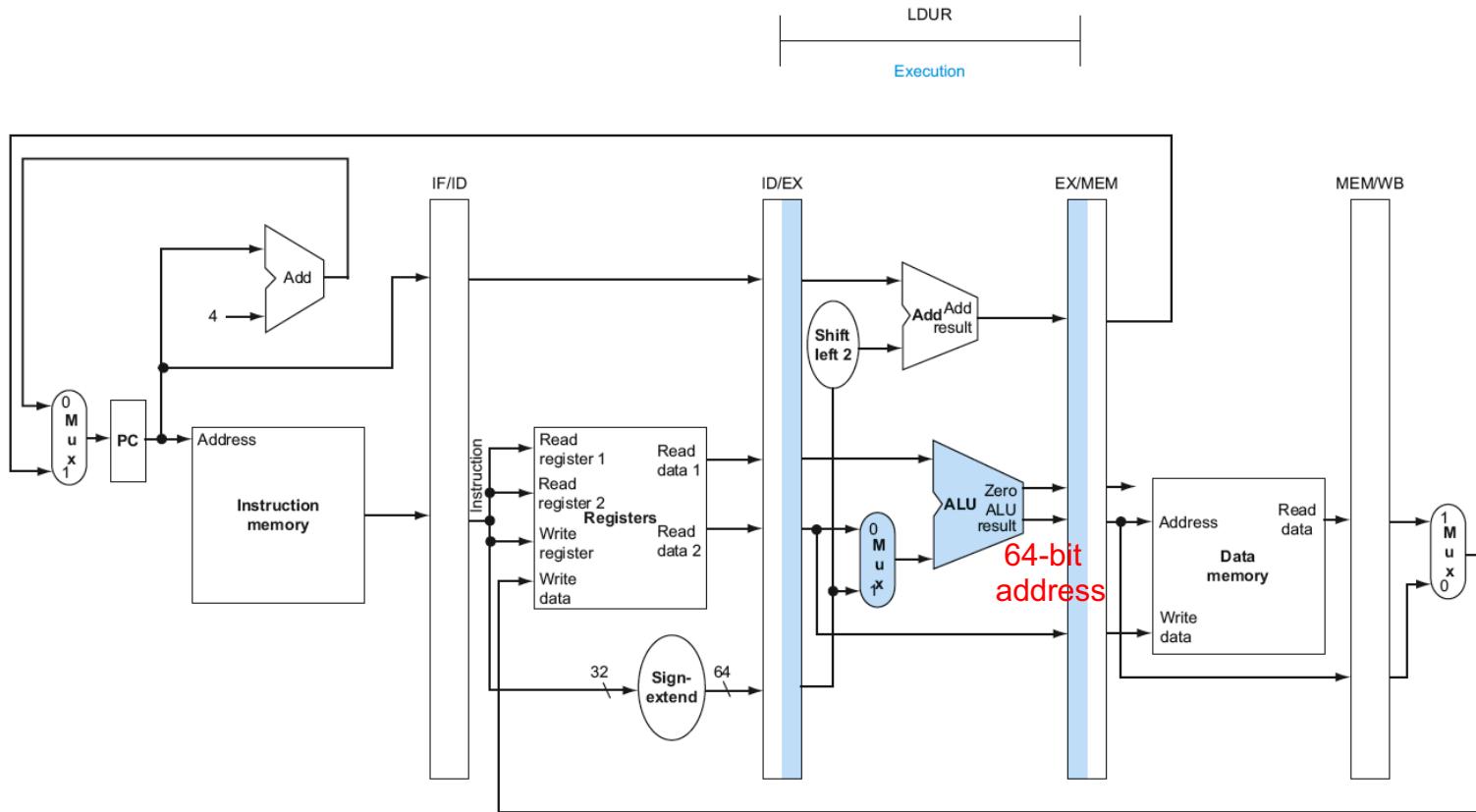


ID for Load, Store, ...



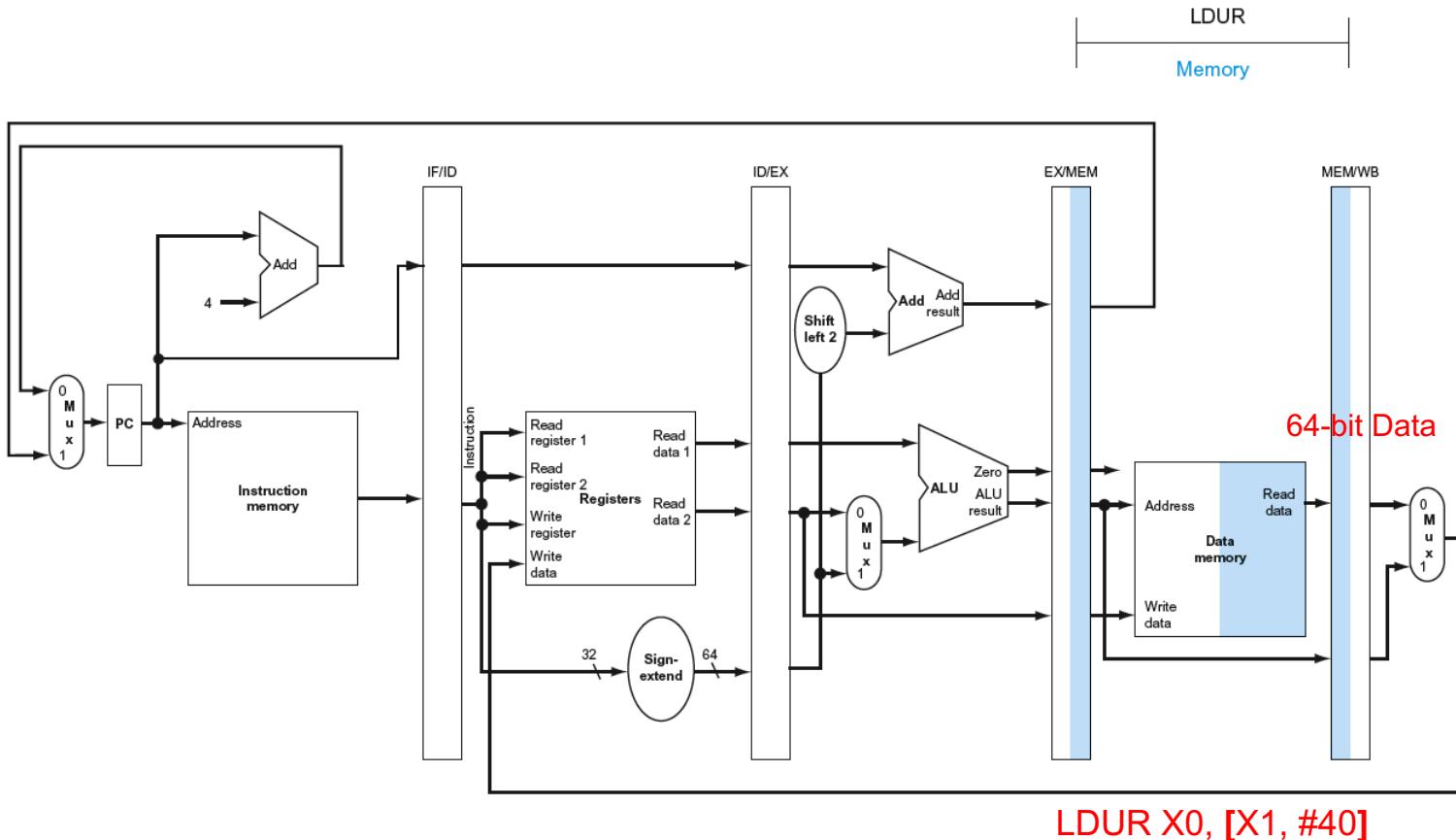
LDUR X0, [X1, #40]
STUR X0, [X1, #40]

EX for Load

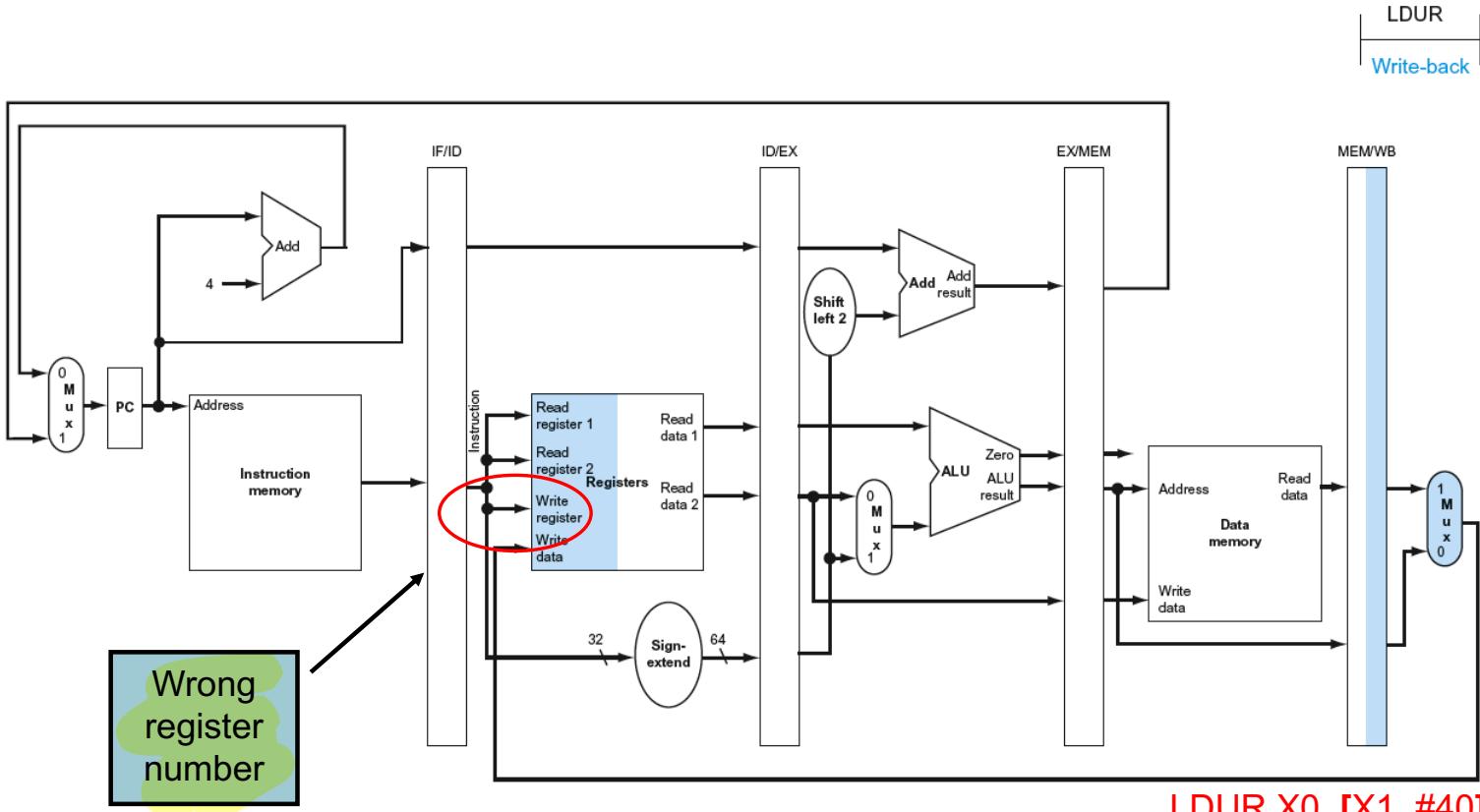


LDUR X0, [X1, #40]

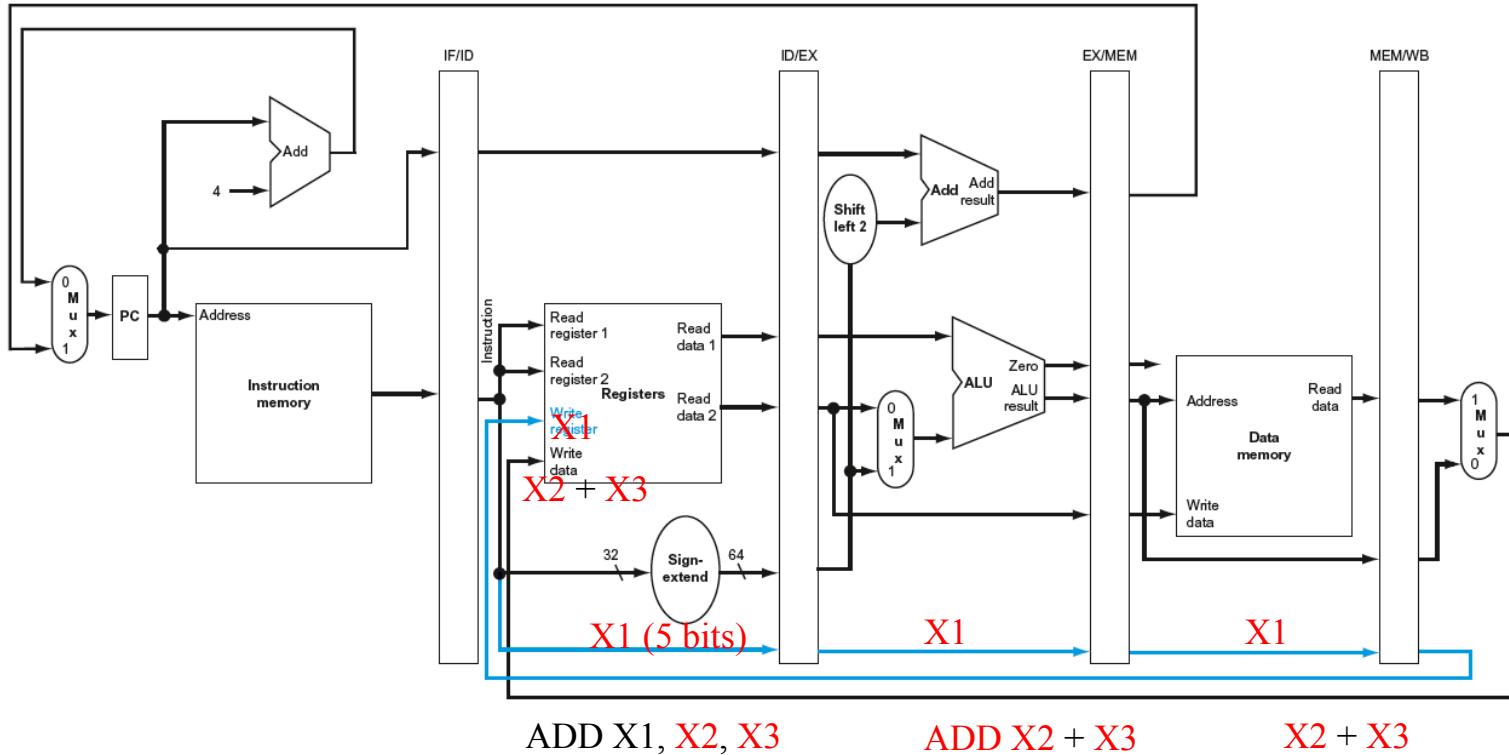
MEM for Load



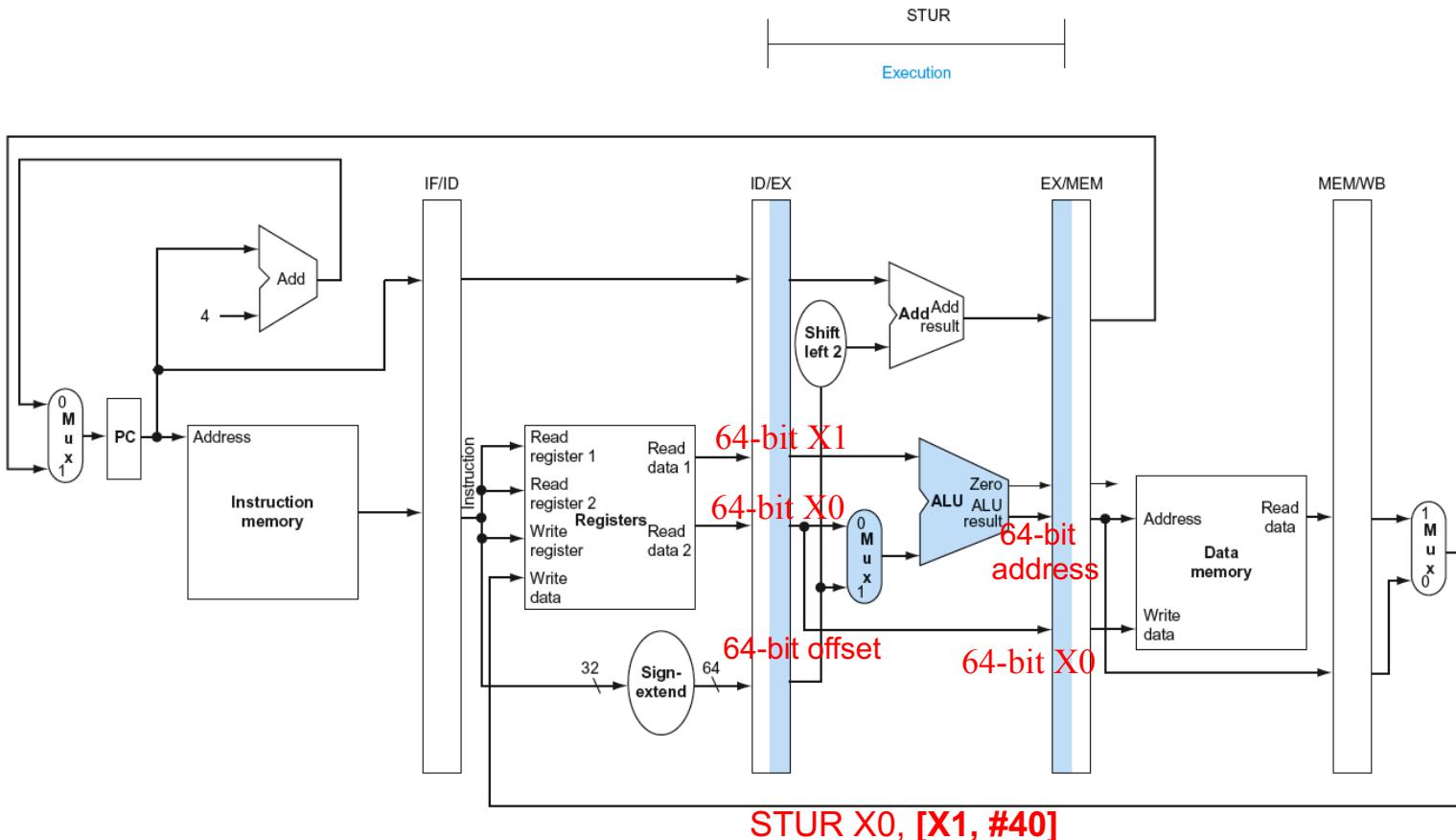
WB for Load



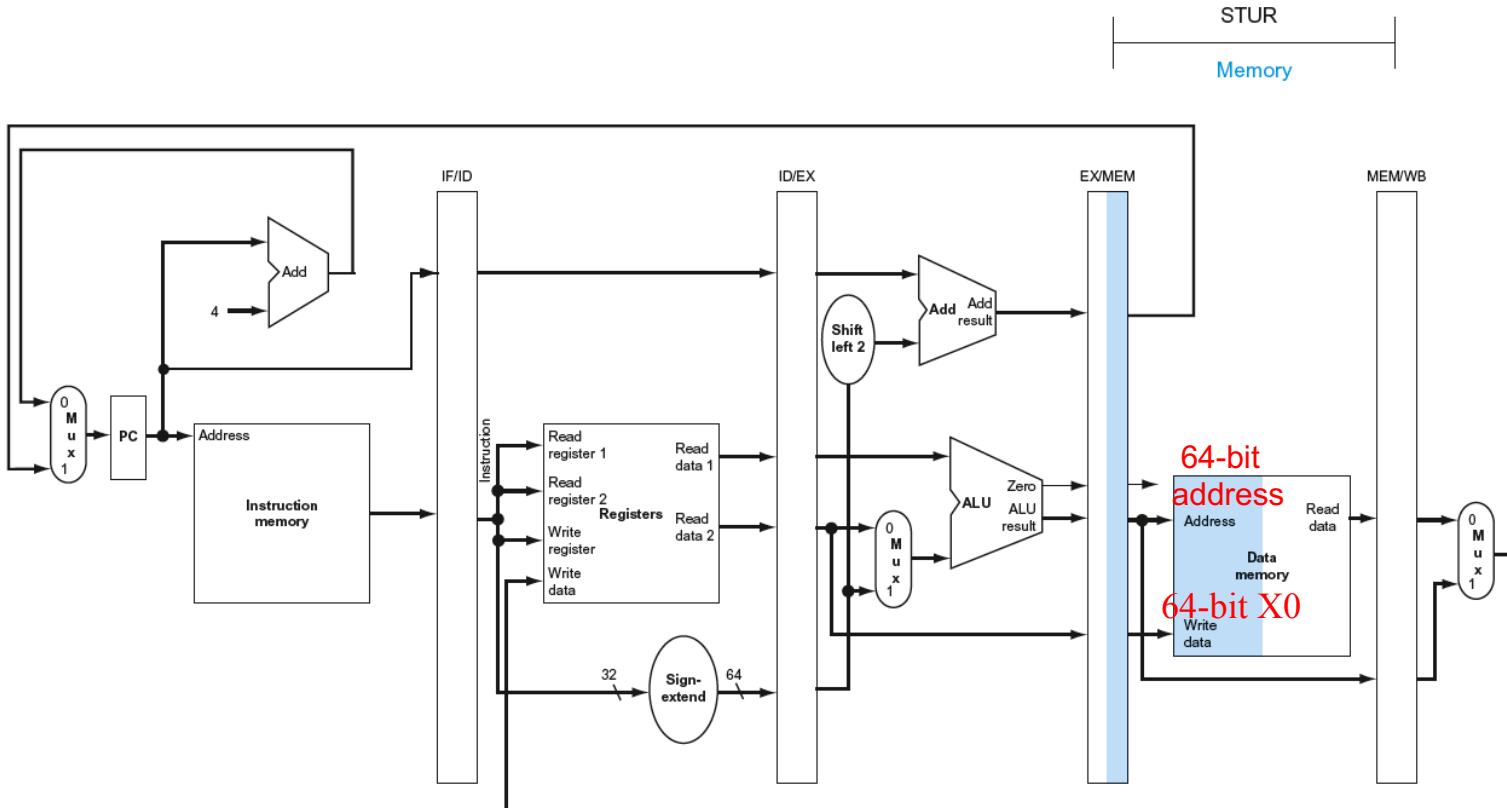
Corrected Datapath for Load



EX for Store



MEM for Store

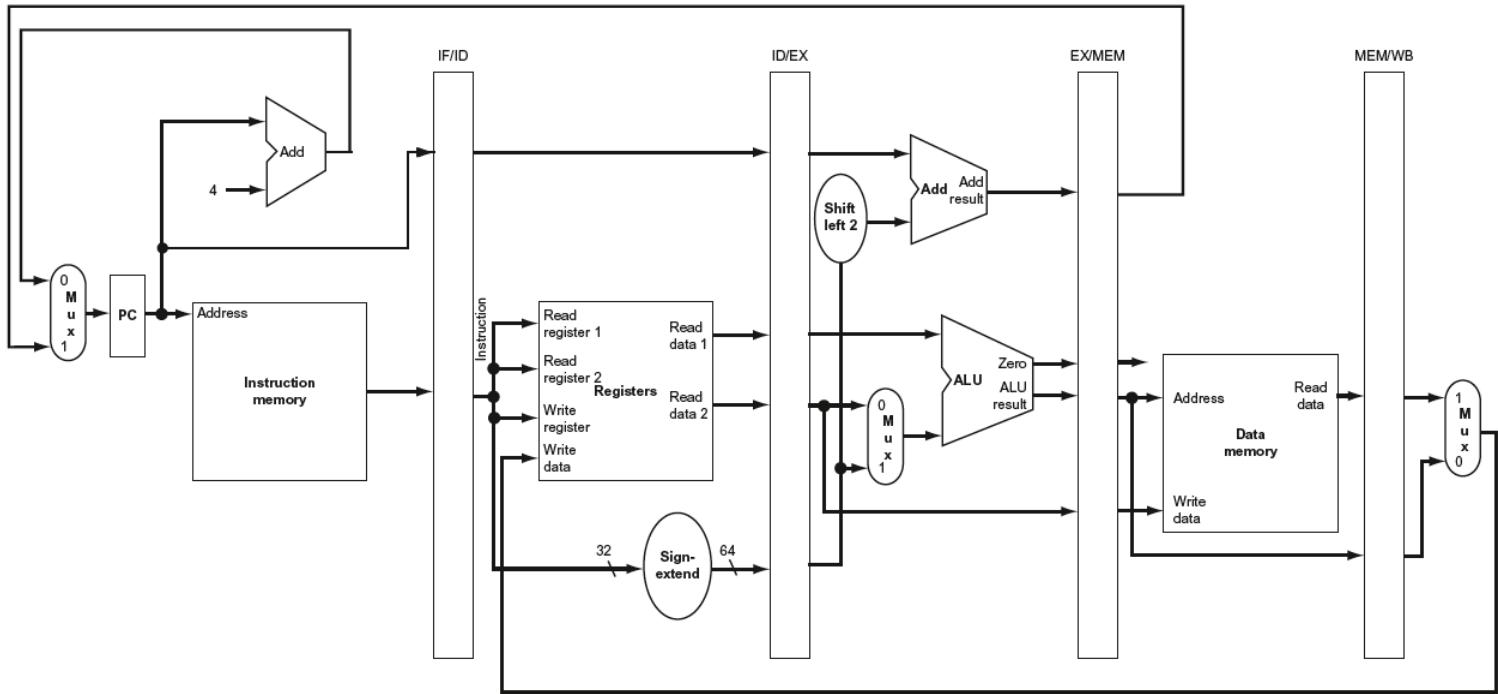


STUR X0, [X1, #40]

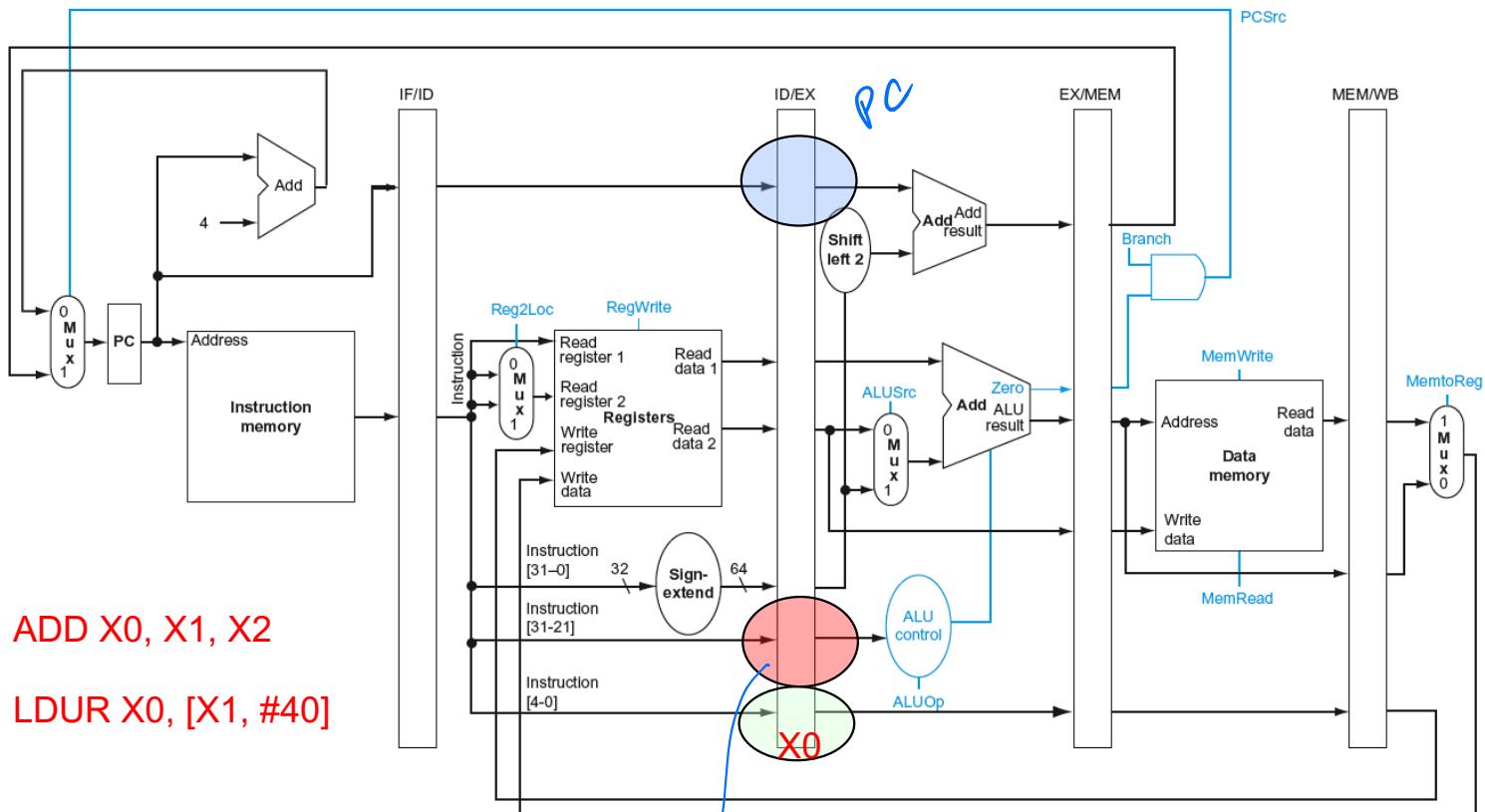
WB for Store

STUR

Write-back



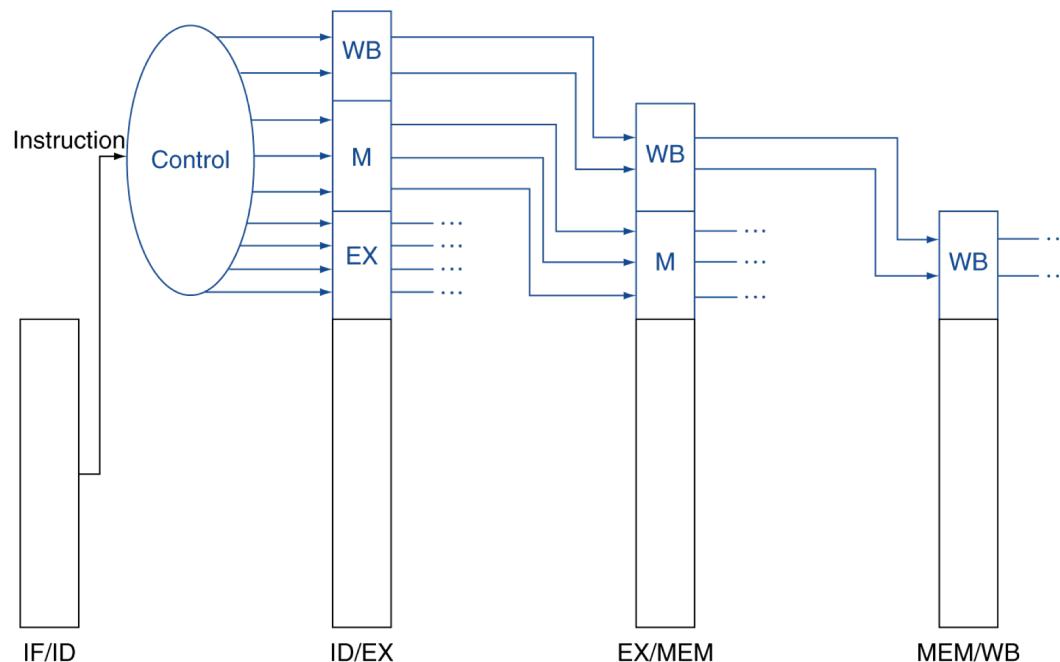
Pipelined Control (Simplified)



opcode.

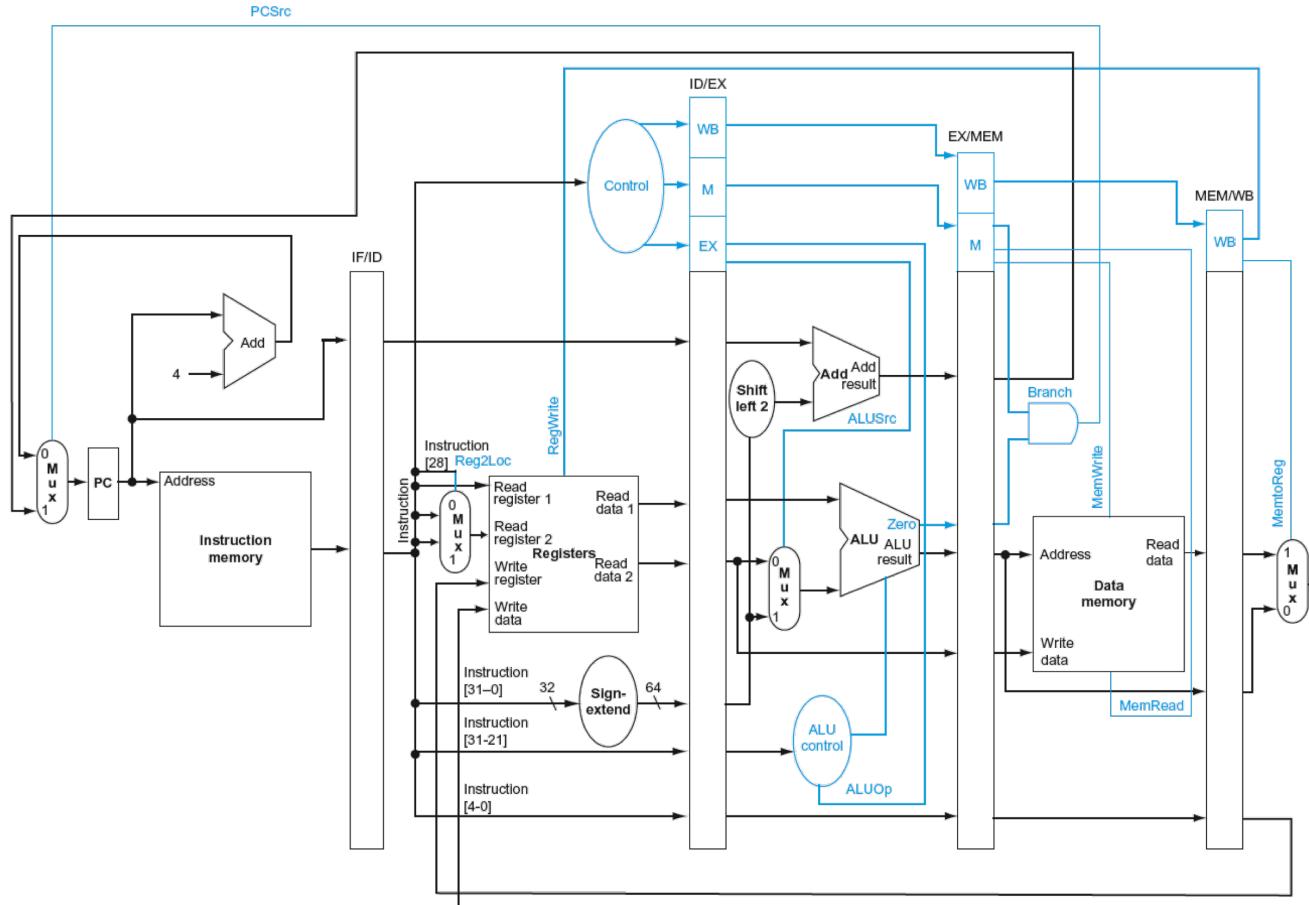
Pipelined Control

- Control signals derived from instruction
 - As in single-cycle implementation

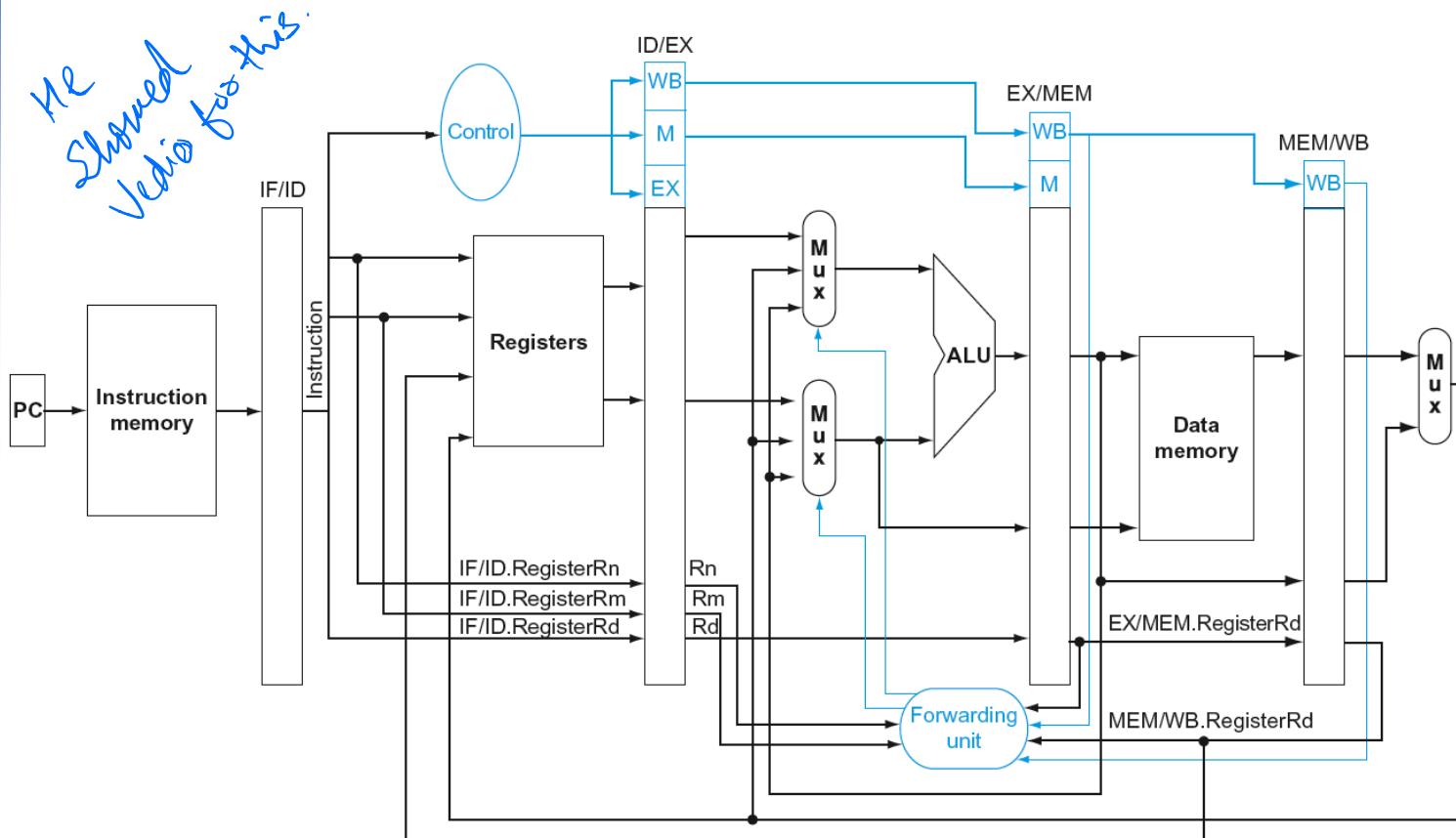


Pipelined Control

FINAL Ding!



Datapath with Forwarding



Fallacies

- Pipelining is easy (!)
 - The basic idea is easy
 - The devil is in the details
 - e.g., detecting data hazards
- Pipelining is independent of technology
 - So why haven't we always done pipelining?
 - More transistors make more advanced techniques feasible
 - Pipeline-related ISA design needs to take account of technology trends
 - e.g., predicated instructions

Pitfalls

- Poor ISA design can make pipelining harder
 - e.g., complex instruction sets (VAX, IA-32)
 - Significant overhead to make pipelining work
 - IA-32 micro-op approach
 - e.g., complex addressing modes
 - Register update side effects, memory indirection
 - e.g., delayed branches
 - Advanced pipelines have long delay slots

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
 - More instructions completed per second
 - Latency for each instruction not reduced
- Hazards: structural, data, control



Chapter 5 – Part 1

Large and Fast Memory: Exploiting Memory Hierarchy

Ideal Memory

- Ideal memory for computer programmers:
 - Large (almost no limit)
 - Fast
 - Inexpensive
- But in reality we can not have all three at the same time.
 - Choose 2 out of three, maybe.
- So we must somehow create an illusion of an affordable and large memory that can be accessed as fast as possible.

Principle of Locality

Temporal
Spatial.

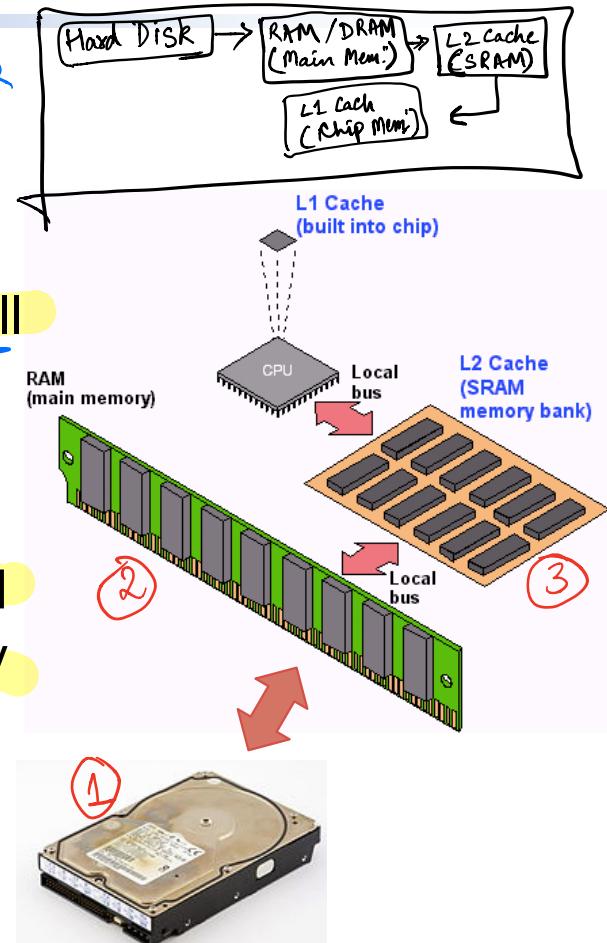
- Programs access a small proportion of their address space at any time
 - Temporal locality
 - Items accessed recently are likely to be accessed again soon
 - e.g., **instructions** in a loop, induction variables → e.g. var "i" in for loop.
 - Spatial locality
 - Items near those accessed recently are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

	<i>Spatial</i>	<i>Temporal</i>
<i>Data</i>	arrays	loop counters
<i>Code</i>	no branch/jump	loop

Taking Advantage of Locality

Memory hierarchy

- Store everything on disk
- Copy recently accessed (and nearby) items from disk to small DRAM memory
 - Main memory
- Copy more recently accessed (and nearby) items from DRAM to even smaller SRAM memory (off-chip L2 Cache)
- Copy even more recently accessed items to L1 Cache
 - memory attached to CPU



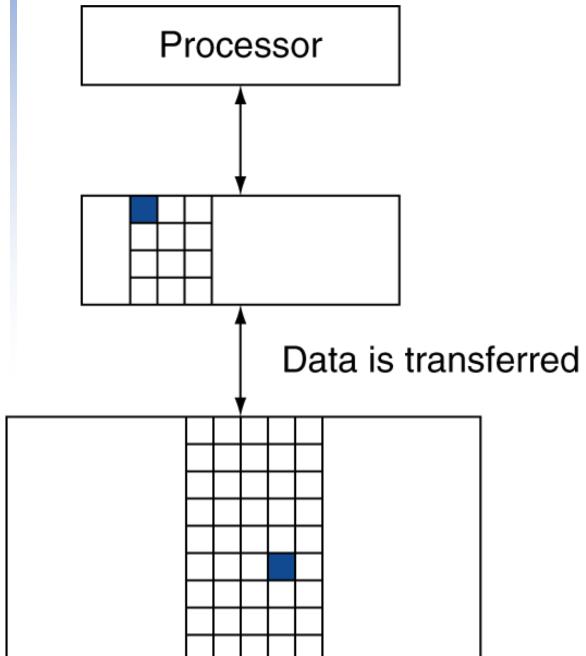
DRAM

SRAM

- we have to Refresh data frequently.
- Dynamic RAM
- Static RAM

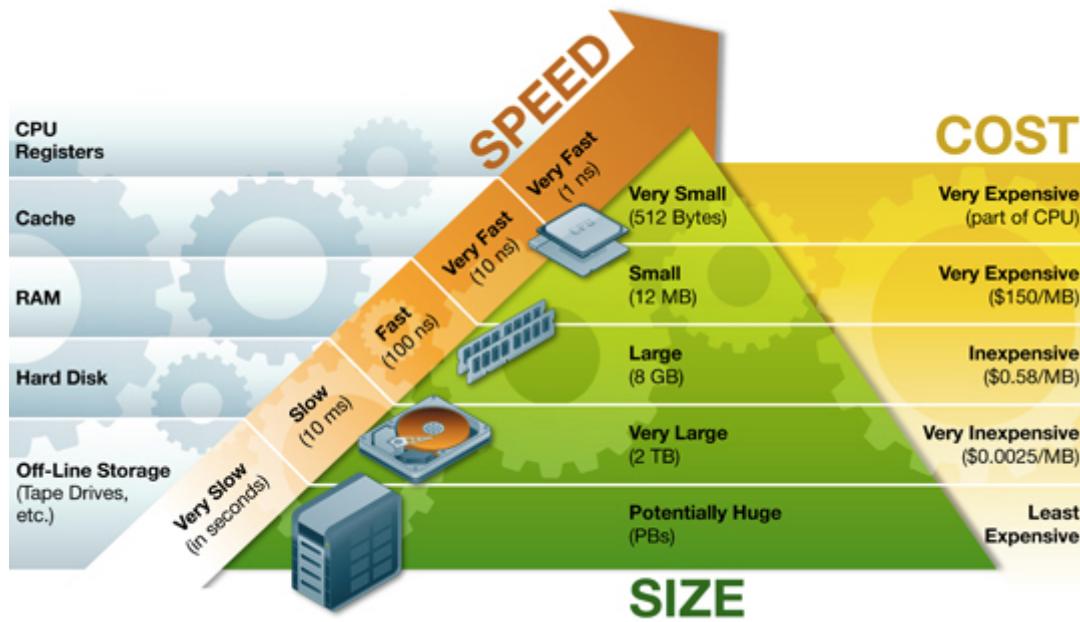
Memory Hierarchy Levels

Within each lvl the
g unit of info. present or not present
-t.



- Block (aka line): unit of copying
 - May be multiple words, or more
- If accessed data is present in upper level
 - Hit: access satisfied by upper level
 - Hit ratio: hits/accesses
- If accessed (requested) data is absent
 - Miss: block copied from lower level
 - Time taken: miss penalty
 - Miss ratio: misses/accesses
 - $\text{Miss ratio} = 1 - \text{hit ratio}$

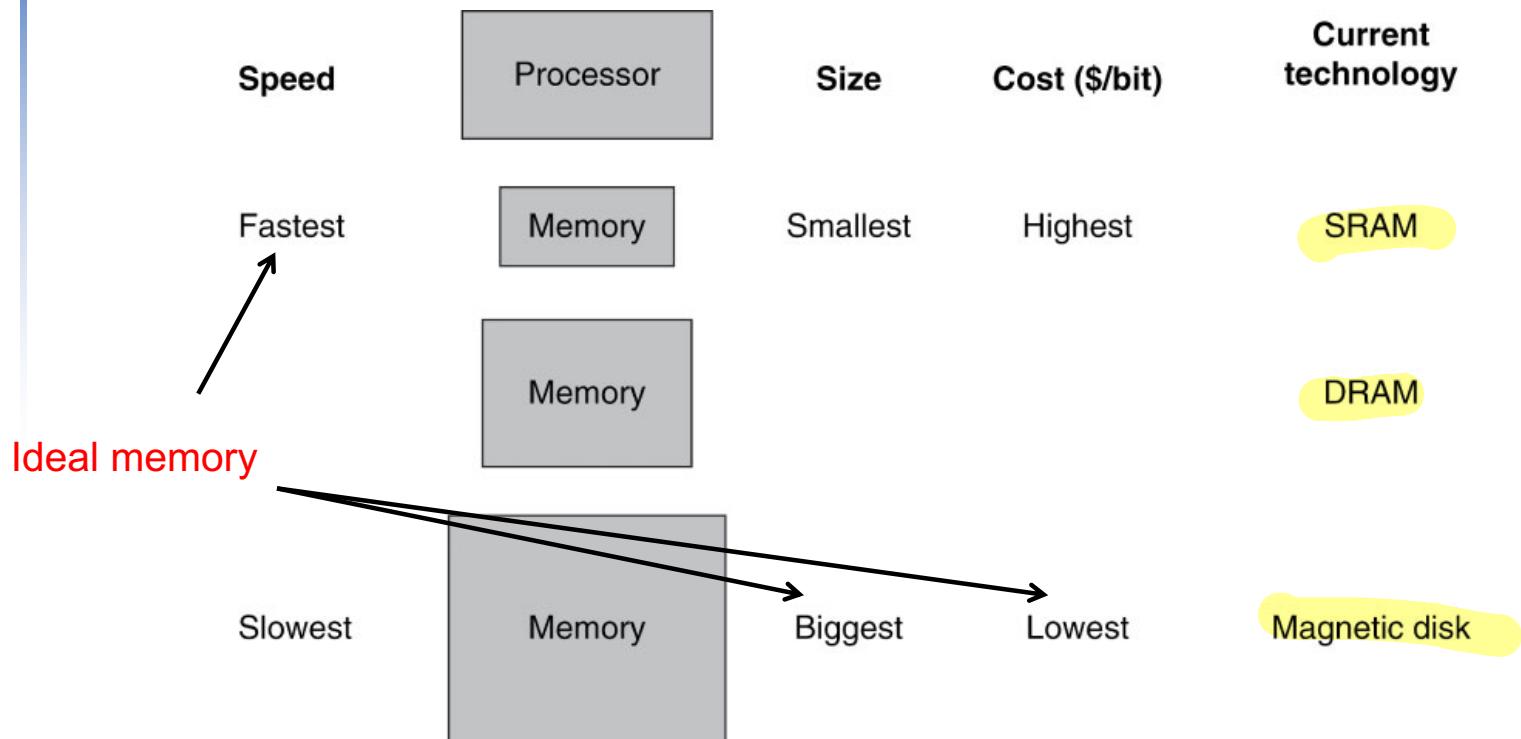
Memory Hierarchy Levels



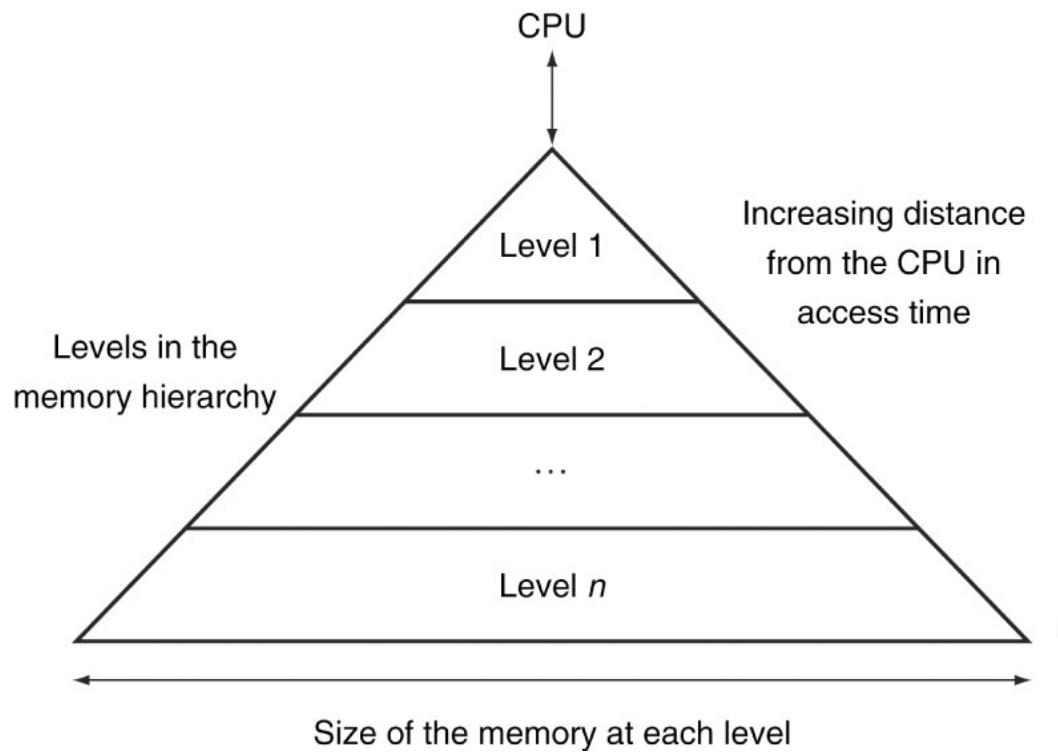
Ideal memory

- Access time of SRAM
- Capacity and cost/GB of disk

Memory Hierarchy Levels



Memory Hierarchy Levels

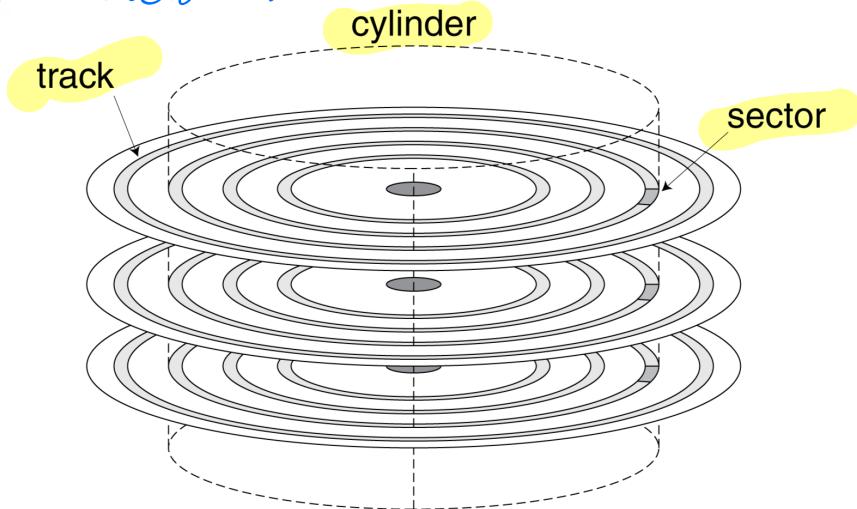


Disk Storage

- Nonvolatile, rotating magnetic storage



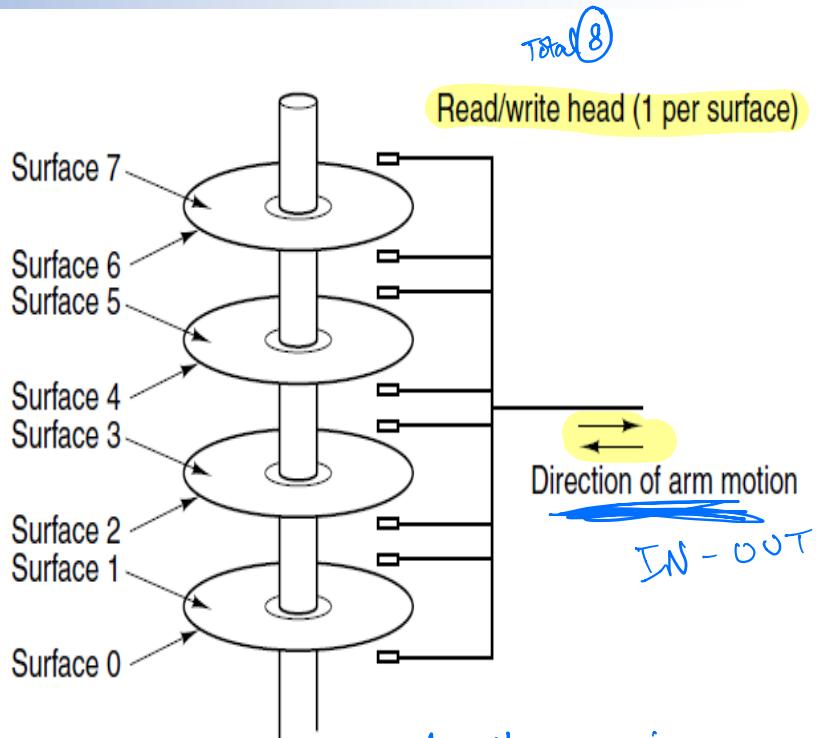
Few concentric cylinders, Moving arm with a Read/Write Tip.
Each cylinder consists of tracks on both sides. Each track is broken into sectors & each sector has Sector ID



Disk Storage

- Data stored on surfaces
 - Up to two surfaces per **platter**
 - One or more platters per **disk**
- Data in concentric tracks
 - Tracks broken into **sectors** -
256B–4KB per sector
 - **Cylinder**: corresponding tracks on all surfaces
- Data read and written by heads
 - Actuator moves heads
 - Heads move all together

is Read or
written



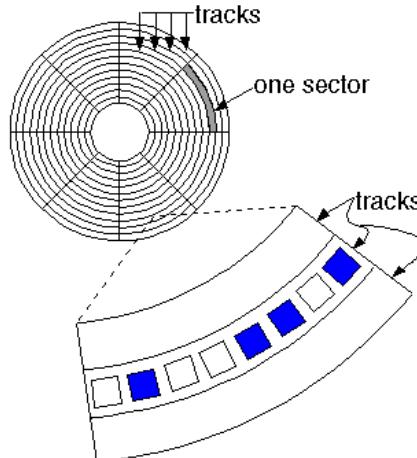
Disk Sectors

- Each sector contains:

1. **Sector ID** (number and location)
2. **Data** → 512 Bytes to 4 KB of data can be stored
3. **Error correcting code (ECC)**
4. **Synchronization fields and gaps**

→ Locate sector on Disc
→ Provide status info about the sector
(e.g.: "good" "defective")

To ensure Data Integrity

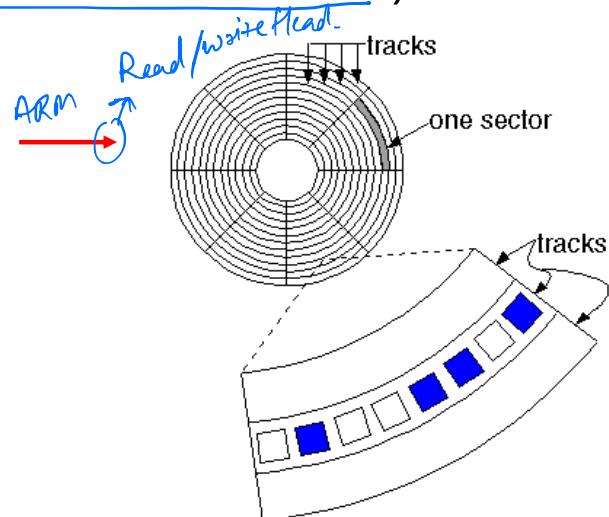


Synchronization +
to separate sectors
→ provide time for controller to process additional data

Disk Sector Access

what all is there in
a sector? ↴

- Access to a sector involves
 - Queuing delay if other accesses are pending
 - Seek: move the heads to proper track
 - Rotational latency (wait for the desired sector to rotate under the read/write head)
 - Data transfer (time to transfer a block of bits)
 - Controller overhead



Disk Sector Access Example

- Given
 - 512B sector, ^{size of each sector} 15,000 rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk \rightarrow queuing time = 0.
- Average read time for one sector
 - 4ms (seek time) \rightarrow (Revolution Per Sec.)
A.V. Rotational latency
 - + $[1/2 * 1/(15,000/60)] = 2\text{ms}$ (avg. rotational latency) \rightarrow Time taken to make 1 revolution.
 - + $[512\text{B} / 100\text{MB/s}] = 0.005\text{ms}$ (transfer time)
 - + 0.2ms (controller delay)
 - = **6.2ms** (average read time)
- If we can reduce average seek time to 1ms
 - Average read time = 3.2ms

Disk Performance Issues

- Manufacturers quote average seek time → Problem
- Smart disk controller allocate physical sectors on disk Solution.
- Disk drives include caches

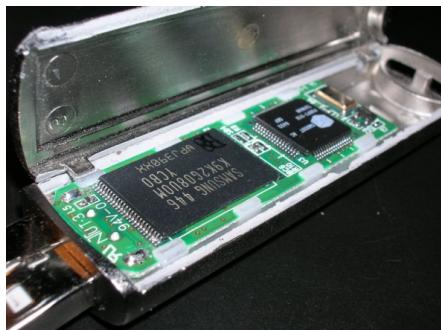
Flash Storage

- faster access time than Disc.
- more expensive " "
- Low Capacity " "

Non-volatile semiconductor storage

- 100 × – 1000 × faster than disk
- Smaller, lower power, more robust
- But more \$/GB (between hard disk and main memory DRAM)

Content Not
Lost when power is down.



Flash Types

→ Does Random Read-Write Access
→ used for "Insts." "Mem." in
Embedded Systems.

- NOR flash: bit cell like a NOR gate

→ Denser than NOR.
→ Cheaper per GB | → used for USB IC,
Media Storage.

- NAND flash: bit cell like a NAND gate

- Flash bits wears out after 1000's of accesses

→ To remedy it we do

wear leveling.

→ Remap data to a less used blocks.

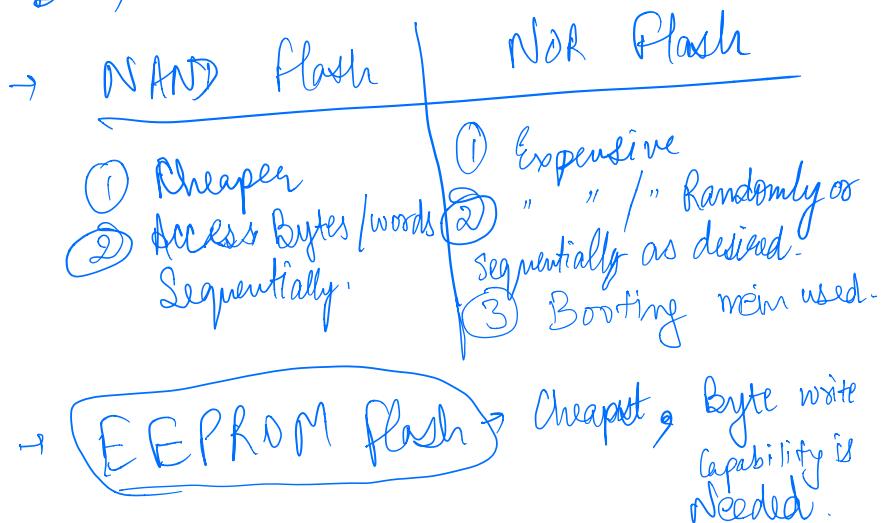
video:

<https://www.youtube.com/watch?v=ELI3abwYQ90>

From Vedio

- Flash is completely erased by placing all bits to '1'.
- Flash is Non-Volatile
- Flash's data to write ~~is~~ ^{or} easily change

But, erase mein time lagta hai.



Cost per Bit Comparison

Specialty Memories (NVRAMs, etc.)	\$\$
EEPROM	\$.82 ~ \$20 per Megabit
SRAM	\$.47 ~ \$1.25 per Megabit
NOR Flash	\$.003 ~ \$.015 per Megabit
DRAM	\$1 ~ \$10 per Gigabit
NAND Flash	\$.04 ~ \$.75 per Gigabit

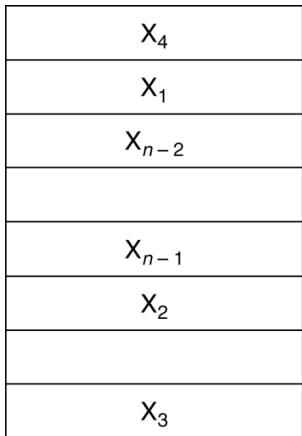
Internet Distributor Pricing (1Ku) @ 'best fit' densities for each technology

Cache Memory

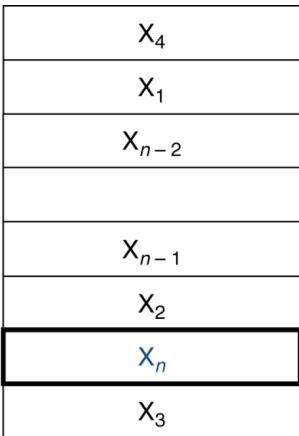
faster Access Time
Smaller in capacity
More expensive.

Cache memory

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



a. Before the reference to X_n



b. After the reference to X_n

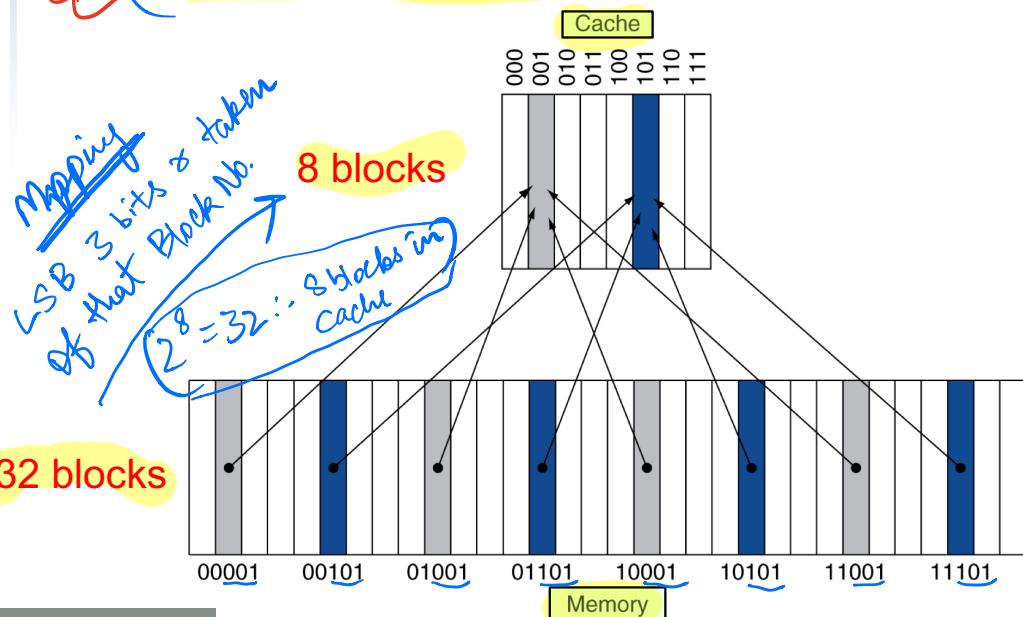
- How do we know if the data is present in cache?
- Where do we look?
- How do we search the cache for that item?

Direct Mapped Cache

Mod → %
gives remainder
after division.

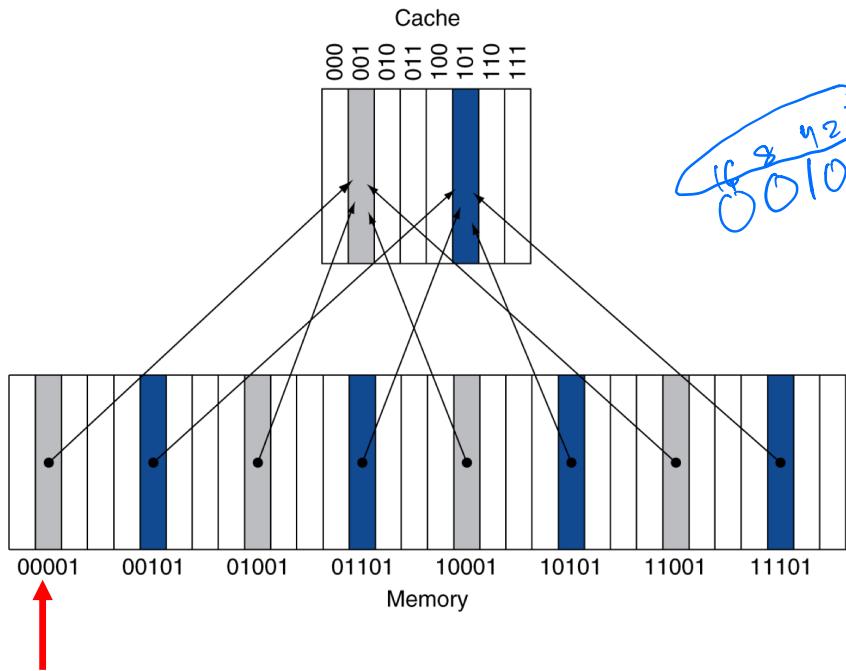
- A 32-block memory and a 8-block cache system
- Cache location determined by address in mem
- Direct mapped: only one choice

(Block address in mem) mod (# of blocks in cache) will lead you to the block address in cache



- # of blocks in cache is a power of 2
- Use 3 low-order address bits as cache index

Direct Mapped Cache



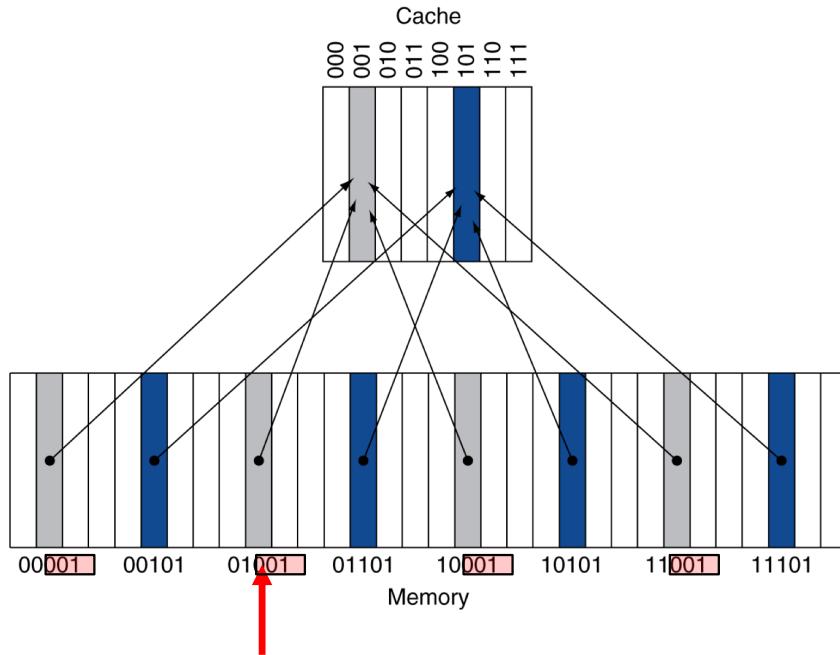
Direct mapping of 32 memory words into 8 cache words

Mem Cache

1 (Mod 8) 1
5 (Mod 8) 5

Formula left

Direct Mapped Cache

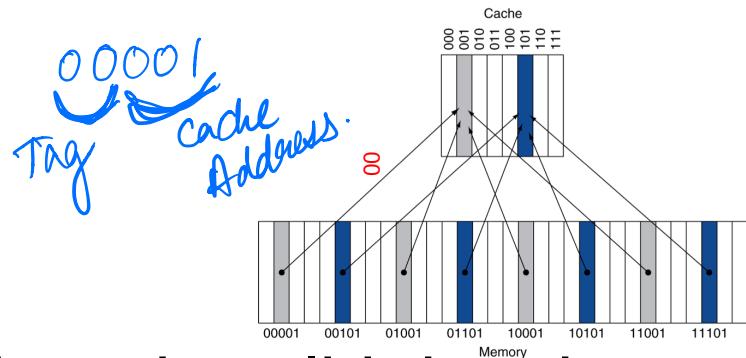


Direct mapping of 32 memory words into 8 cache words

Mem	Cache
1 (Mod 8)	1
5 (Mod 8)	5
9 (mod 8)	1
13 (mod 8)	5
17 (mod 8)	1
21 (mod 8)	5
25 (mod 8)	1
29 (mod 8)	5

Tags and Valid Bits

- How do we know which particular memory content is stored in a cache block?
 - Store also the high-order memory address bits with the data
 - Called the tag



- How can we tell if there is valid data in a cache block?
 - Valid bit: 1 = present, 0 = not present
 - Initially 0

Cache Example

- 8-block cache, 1 word/block, direct mapped
- Initial state ↗
• No Data
• Valid Bit (V) = 0 / N .
↗
• No Tag

8-block Cache System

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

Cache Example

When needed
Not there
in Cache mem

Mem word address	Mem binary address	Hit/miss	Cache block
22	10 110	Miss	110

8-block
Cache
System

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

Cache Example

Mem word address	Mem Binary address	Hit/miss	Cache block
26	11 010	Miss	010

8-block
Cache
System

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



Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11010	Hit	010

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

8-block Cache System

Cache Example

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

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

8-block
Cache
System

Cache Example

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

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

8-block
Cache
System



Cache Example

Replace Data.

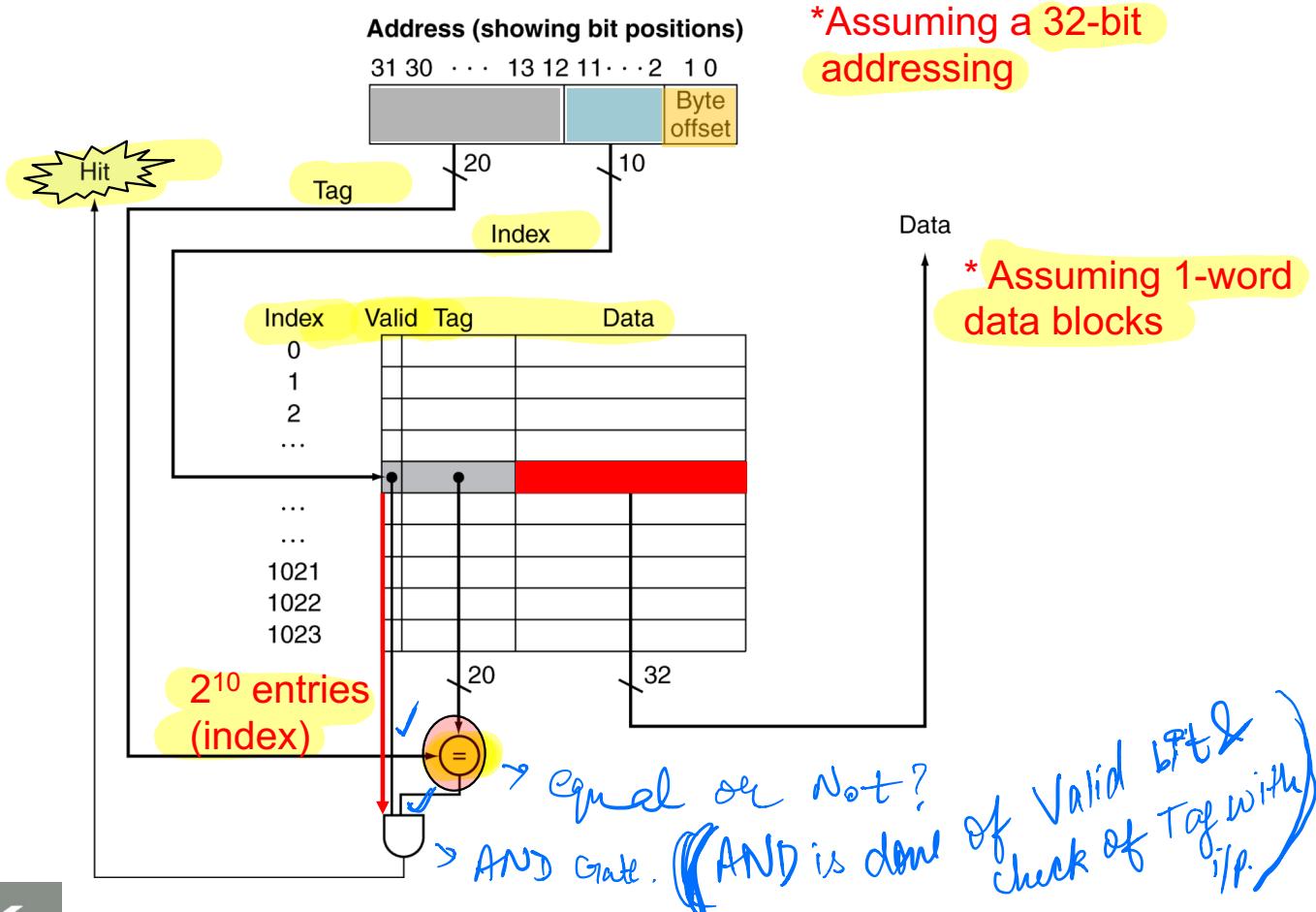
Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

8-block
Cache
System

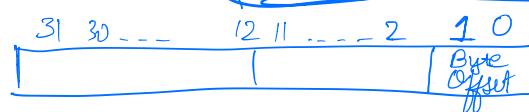
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	10 <i>(It was 11 see earlier pic)</i>	Mem[10010] <i>→ this was Replaced.</i>
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		



Address Subdivision



32 bit Addressing Is Done.



↪ 1st 2 bits are used for Byte offset within the word.

1 word = 32 bits
1 word = 4 bytes.

These next 10 bits are used to select a cache entry.



∴ Total Bytes of Data Possible is:-

$$2^{10} \text{ in } \underline{4KB} \\ = 1024.$$

31 ---- 12

20 bits

Tags. in the Cache.

Larger Block Size

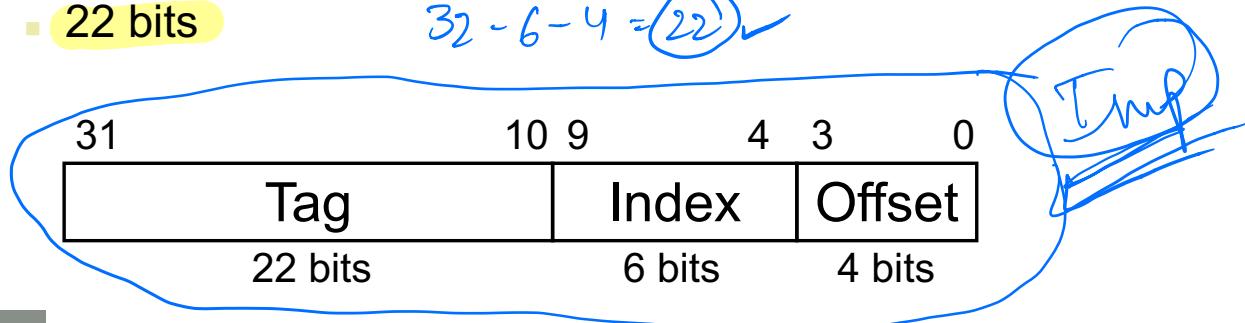
64 blocks \rightarrow 16 bytes per block

- Cache: 64 blocks (or 64 cache entries), 16 bytes per block (assume 32-bit addressing)

- How many bits required for byte offset within a block?
 - $2^{4(\text{bits})} = 16 \text{ words}$ (4 bits.)
- How many bits required to uniquely identify the block index?
 - $64 \text{ (cache entries)} = 2^{6(\text{bits})}$ (6 bits)
- And # of bits left to be used for the tag?

- 22 bits

$$32 - 6 - 4 = 22$$



Block Size Considerations

- Larger blocks should reduce miss rate. Why?
 - Due to spatial locality
- But programs with lots of branches
 - Larger blocks ⇒ lead to fewer blocks in cache
 - More competition ⇒ increased miss penalty
 - Larger blocks ⇒ pollution
- So increased miss penalty
 - Can override benefit of reduced miss rate
 - However, early restart and critical-word-first techniques can help with the increased miss penalty

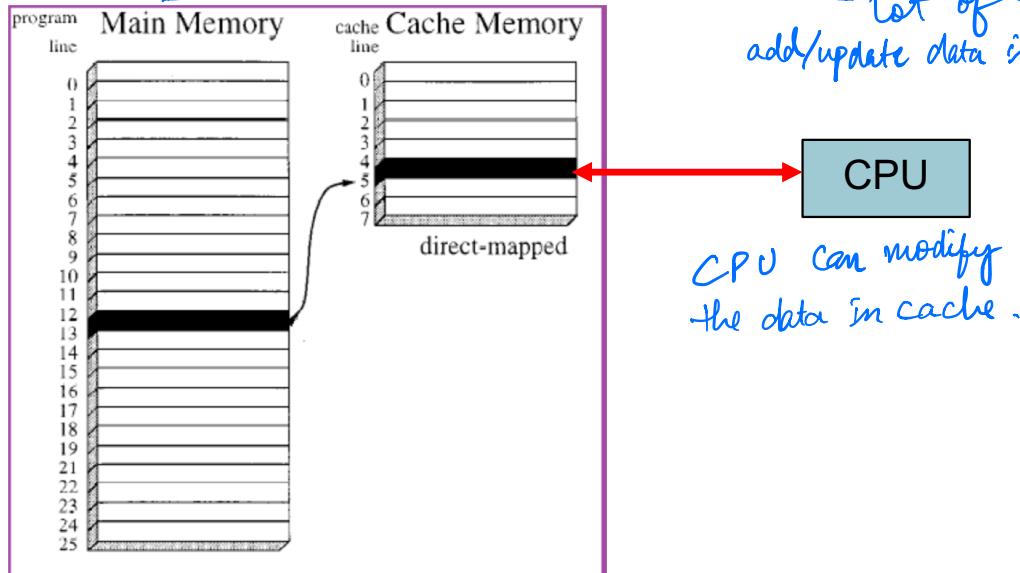
Block Size Considerations

- How restart and critical-word-first can help with the increased miss penalty?

- In both techniques: do not wait for full **large** block to be loaded before restarting CPU
- **Early restart** - As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
Reg. "Block ko start karne kehna. Tab parhne jaie. Requested Word then Start CPU execution & then send the rest of the packet."
- **Critical Word First** - Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block.
Word deko CPU ko, then cont? filling Rest words

Write-Through

- On data-write hit (data to be updated exists in cache):
data is updated in cache but not in main mem!!
 - Could just update the block in cache
 - But then cache and memory would be inconsistent
- Write through: then also update memory
This takes a lot of time to add/update data in mem."

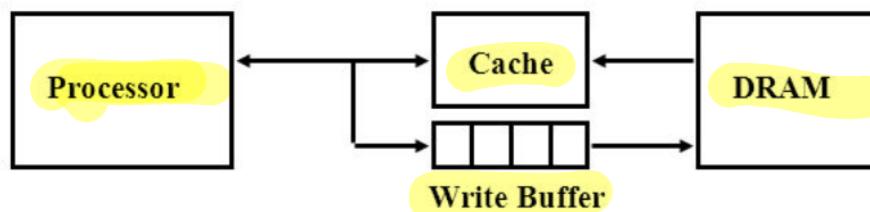


Write-Through

- Memory writes take much longer
 - e.g., if base CPI = 1, and 10% of instructions are stores, and write to memory taking 100 cycles
 - Effective CPI = $1 + 0.1 \times 100 = 11$ (a significant increase in CPI as a result of just 10% memory writes)
- Solution: write buffer
 - Holds data waiting to be written to memory
 - CPU continues immediately
 - Only stalls on write if write buffer is already full

update cache

*Hold kaita hai data Jab tak Voh
full Na ho Jao. If it gets
full it will
write data
into the
mem.*



Write-Back

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

had a diff. mem. data.
Cache & a diff. mem. data.
OP
Block in Cache Table
data is updated.

SPEC

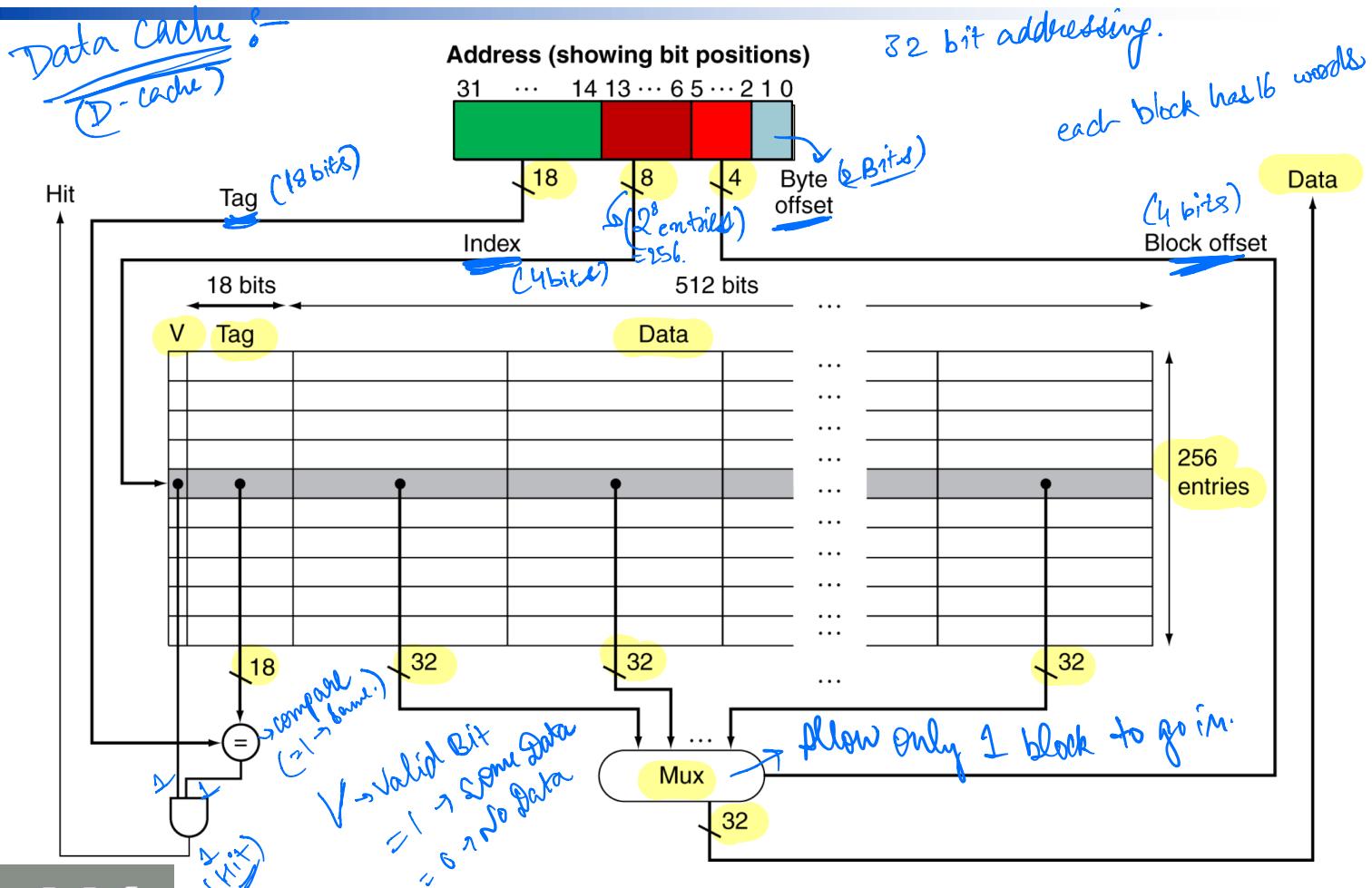
Description	Name	Instruction Count x 10 ⁹	CPI	Clock cycle time (seconds x 10 ⁻⁹)	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	-	-	-	-	-	-	25.7

Example: Intrinsity FastMATH

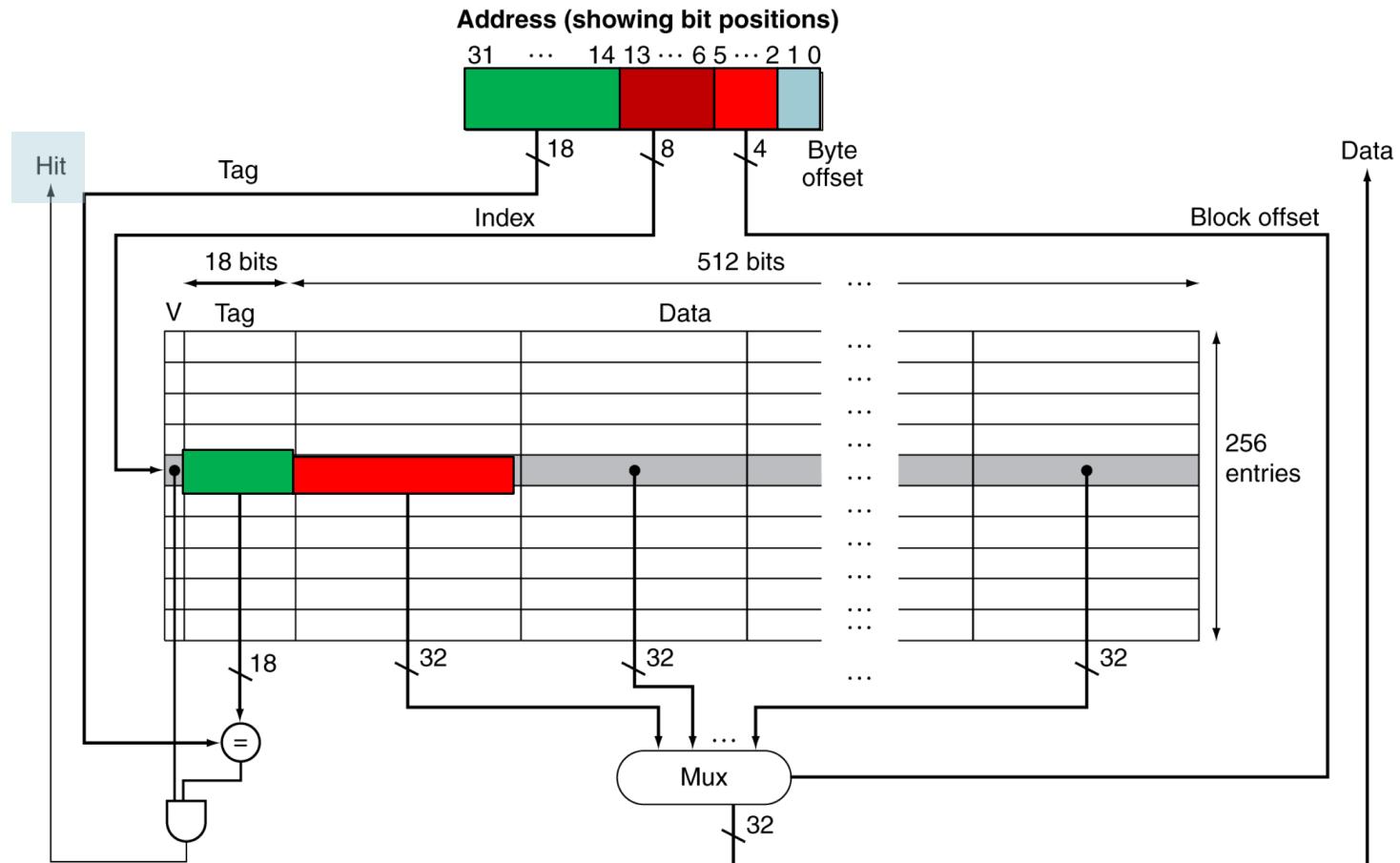
- Embedded MIPS (RISC-based) processor
 - 12-stage pipeline
 - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
 - Each 256 blocks \times 16 words per block (64 bytes per block) = 16KB (total size of each cache):
 - D-cache: write-through or write-back
- SPEC2000 miss rates for:
 - I-cache: 0.4%
 - D-cache: 11.4%
 - Weighted average: 3.2%



Example: Intrinsity FastMATH



Example: Intrinsity FastMATH



Measuring Cache Performance

Components of CPU execution time

Program execution cycles

- Includes cache hit time (about 1 cycle)

Memory stall cycles

- Mainly from cache misses
- This component would be zero for a perfect/ideal cache

① Program execution Cycle
 ② Mem "stall cycle"
 Time taken to grab data from Cache to CPU.

Memory stall cycles

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

Cache Miss es se aaya

Cache Hit

Cache Miss

Cache Performance Example

- Given
 - I-cache miss rate = 2%
 - D-cache miss rate = 4%
 - Miss penalty = 100 cycles
 - Base CPI (ideal/perfect cache) = 2
 - Load & stores are 36% of instructions
- Miss cycles per instruction
 - I-cache: $1(100\% \text{ of instructions use I-cache}) \times 0.02 \text{ (2\% of them miss)} \times 100 \text{ (miss penalty)} = 2$ cycles
 - D-cache: $0.36(36\% \text{ of instructions use D-cache}) \times 0.04 \text{ (4\% of them miss)} \times 100 = 1.44$ cycles
- Actual CPI = 2 + 2 + 1.44 = 5.44 cycles
 - CPU with ideal cache is $5.44/2 = 2.72$ times faster



Average Memory Access Time

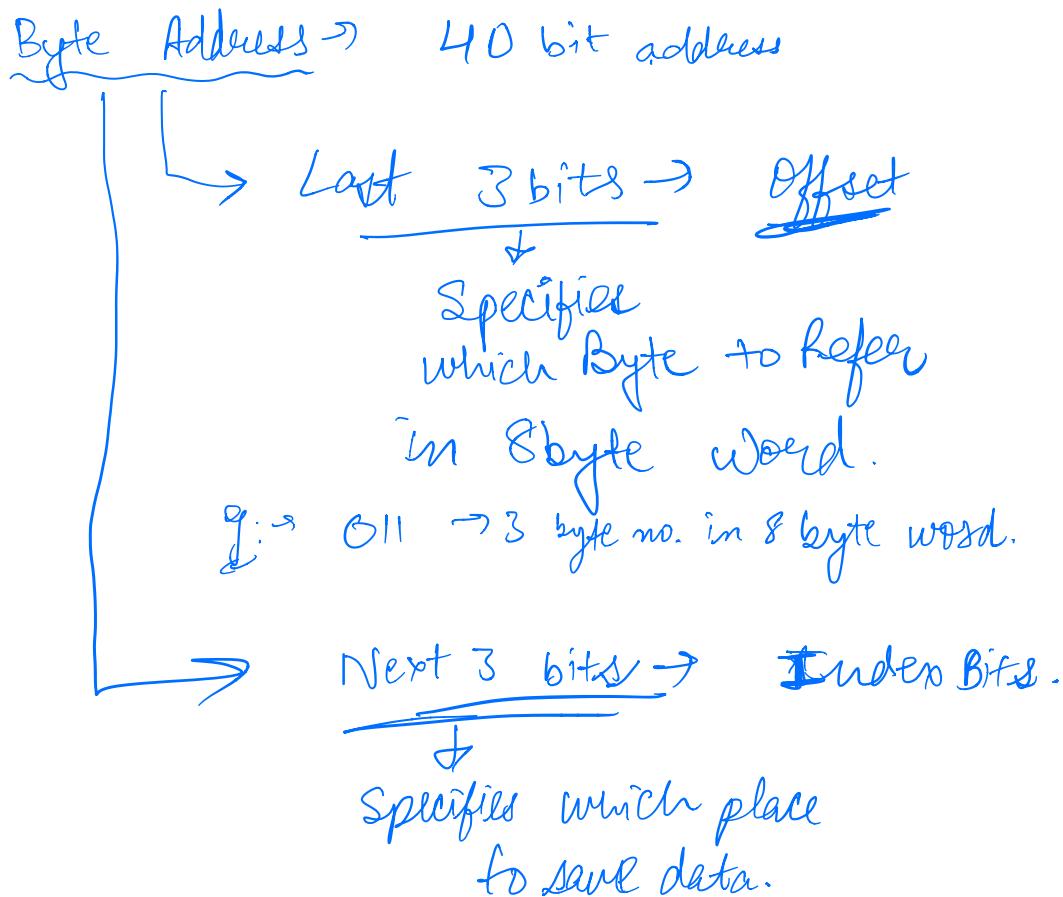
- Hit time is also important for performance
- Average memory access time (AMAT)
 - $\text{AMAT} = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty})$
- Example
 - CPU with 1ns clock,
 - Hit time = 1 cycle,
 - I-cache miss rate = 5%,
 - Miss penalty = 20 cycles,
 - $\text{AMAT} = 1 + 0.05 \times 20 = 2 \text{ cycles}$
 - 2 cycles per instruction fetch (twice the cycles due to miss rate and penalty)

From Vedio :-

Accessing the Cache:-

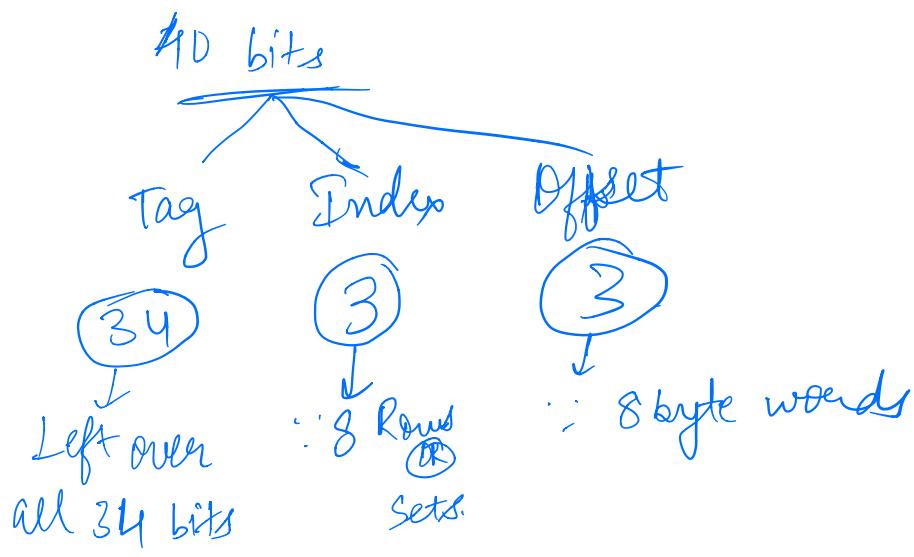
Assume, 64 Byte \rightarrow Cache

\therefore 8 \rightarrow 8 byte words in the Data Array



To distinguish b/w. the addresses that have same location in the cache, we use Tag Array.

Byte Address :-



to distinguish
addresses.

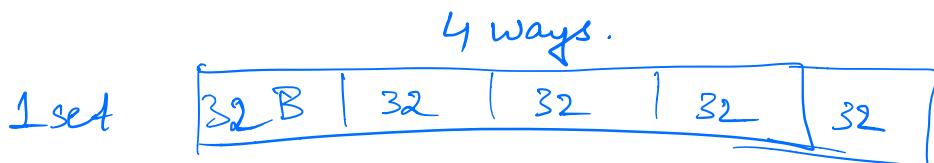
Set Associative Cache \rightarrow Few Confits

Dis Adv. \rightarrow Power wasted
 \because multiple data & tags & Read.

Q 32 KB 4 Way set associative data cache
array with 32 byte line sizes.

- ① How many sets?
- ② " " index bits, offset bits, Tag bits?
- ③ " Cache the tag array is?"

Ans ↗



(5 offset bits req'd to take)
(1 byte out of these 32 bytes.)

$$40 = \underbrace{\text{Tag}}_{2^7} + \underbrace{\text{Index}}_8 + \underbrace{\text{Offset bits}}_5$$

1 set has 4 32 byte blocks. ∴ total = 128 bytes.

$$\text{① No. of sets} = \frac{32 \text{ KB}}{128} = \underline{256}$$

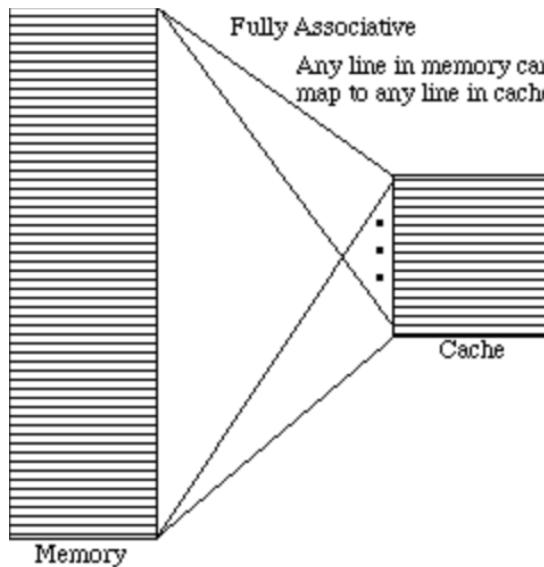
$$\begin{aligned} \text{② } I &\approx 8 \text{ bits} \\ \text{tag} &= 2^7 \text{ bits} \\ \text{offset} &= 5 \text{ bits} \end{aligned}$$

$$\begin{aligned} \text{③ } 2^7 \text{ bits} \times 4 \text{ ways} \times 256 \text{ sets} \\ &= 2^{14} \text{ bits} = 3.375 \text{ KB} \end{aligned}$$

Associative Caches

Fully associative

- Allow a given block to go in any cache entry
- Requires all entries to be searched at once



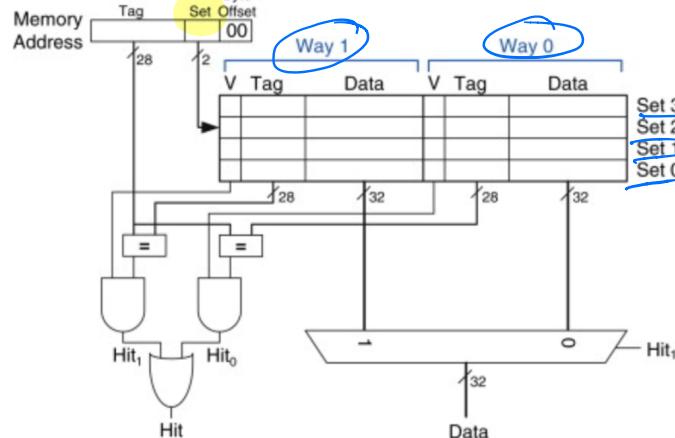
Associative Caches

n-way set associative

- Each entry (set) contains n possibilities to store data
- Block number in memory determines which set
 \checkmark (Block number) mod (# of Sets in cache) \checkmark
- Just search all the possibilities within a set (n)

n Searches
Ratio.

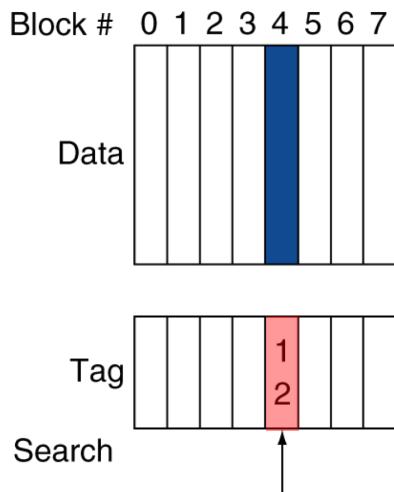
2-way associative cache



Associative Cache Example

34 Types

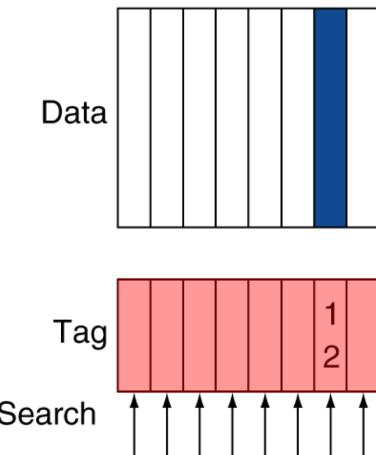
Direct mapped



Set associative

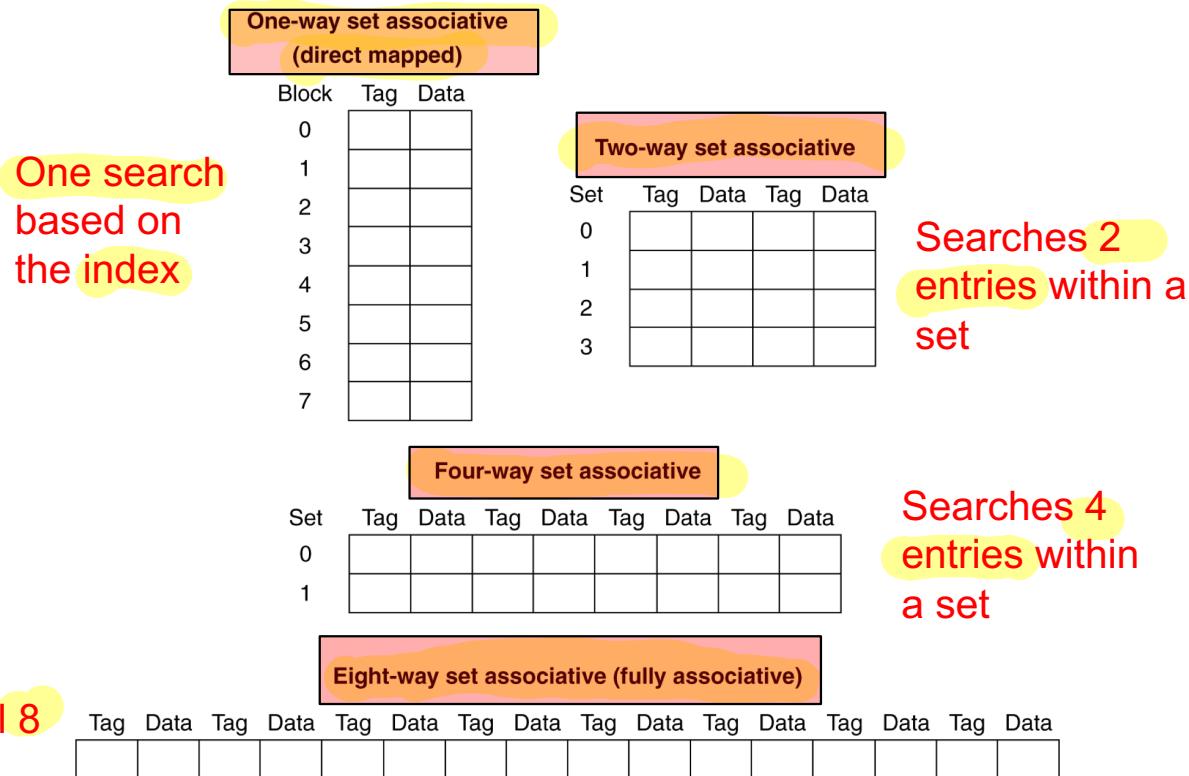


Fully associative



Spectrum of Associativity

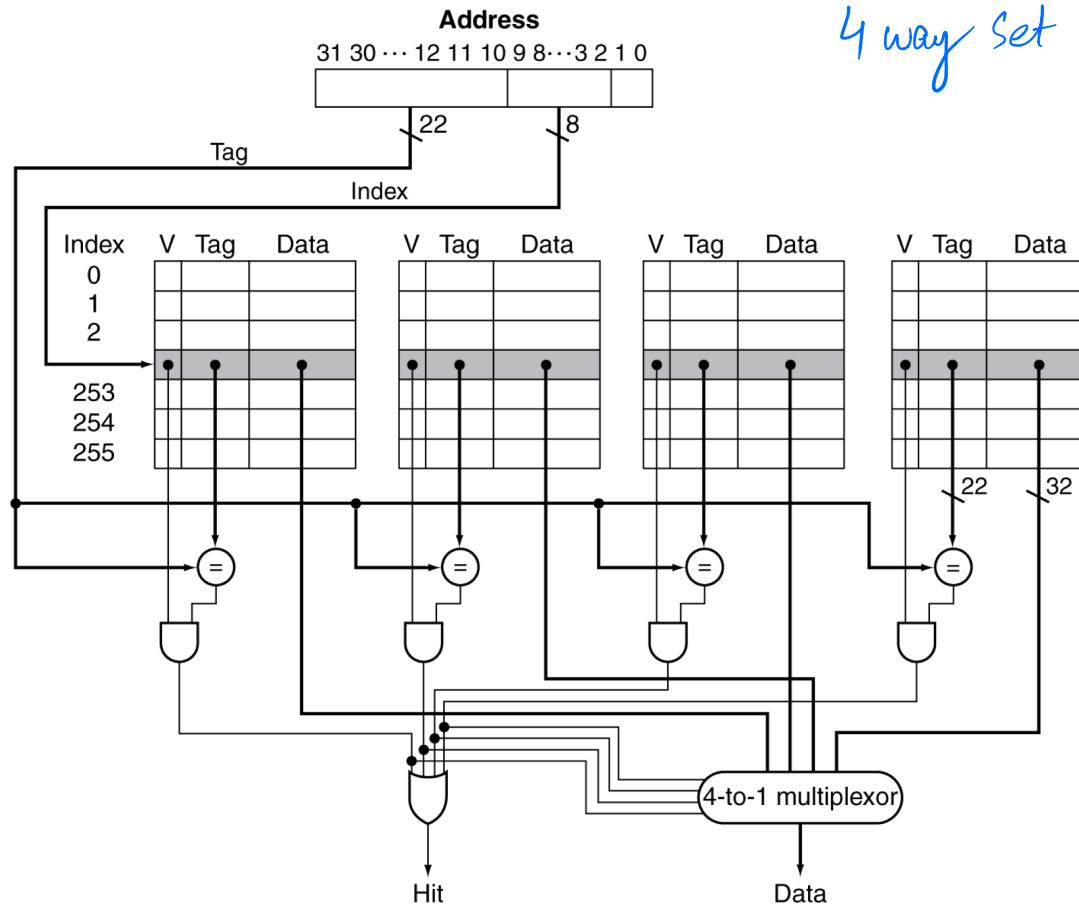
- For a cache size of 8 blocks



How Much Associativity

- Increased associativity decreases miss rate
 - But with diminishing returns
 - Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
 - 1-way: 10.3% (direct map)
 - 2-way: 8.6%
 - 4-way: 8.3%
 - 8-way: 8.1%
- $\frac{\text{Associativity} \propto 1}{\text{Miss Rate}}$

Set Associative Cache Organization



Replacement Policy

- Direct mapped: no choice
- Set associative
 - Prefer non-valid entry
 - Otherwise, choose among entries in the set
 - Least-recently used (LRU)
- Random

Multilevel Caches

Primary cache: attached to CPU

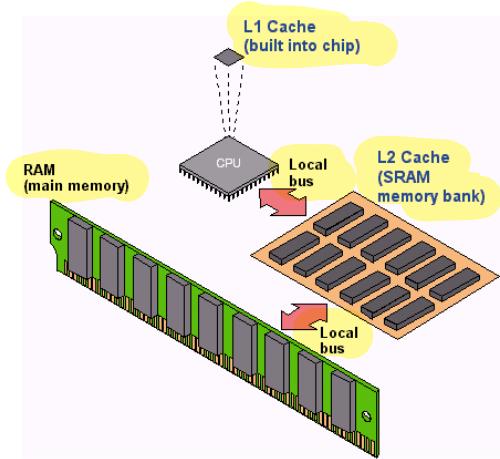
- Small, but very fast

Level-2 cache: services misses from primary cache

- Larger, slower, but still faster than main memory

Main memory: services L-2 cache misses

■ Some high-end systems include L-3 cache



Multilevel Cache Example

- Given
 - CPU base CPI = 1, clock rate = 4GHz
 - Clock period = $1/4000000000 = 0.25 \text{ ns}$
 - Miss rate per 100 instructions = 2%
 - Main memory access time = 100ns
 - Or $100\text{ns}/0.25\text{ns} = 400 \text{ cycles (miss penalty)}$
- With just primary cache
 - Every miss in L1 will cost us 400 cycles
 - Effective CPI = $1 + 0.02 \times 400 = 9$

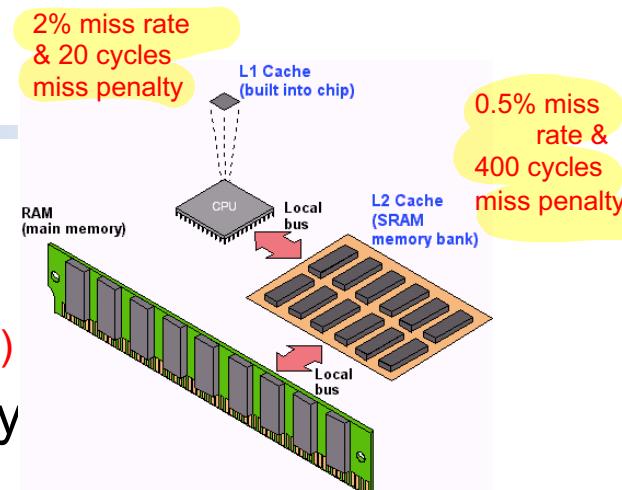
Example (cont.)

Now add L-1 cache

- Access time to L2 = 20 cycles
(a miss in L1 but a hit in L2 costs 20 cycles)
- Global miss rate to main memory
from L2 = 0.5%

(a miss in L2 and going to main memory, costs 400 cycles)

$$\begin{aligned} \text{CPI} = & 1 + (0.02 \times 20 \text{ cycles}) \\ & + (0.005 \times 400 \text{ cycles}) = 3.4 \end{aligned}$$



2% miss rate
& 20 cycles
miss penalty

L1 Cache
(built into chip)

0.5% miss
rate &
400 cycles
miss penalty

Adding L-1, 2% of
misses goes to L2

Now only 0.5% of
misses in L2 has to
go to main memory

$$\text{Performance ratio} = 9/3.4 = 2.6$$



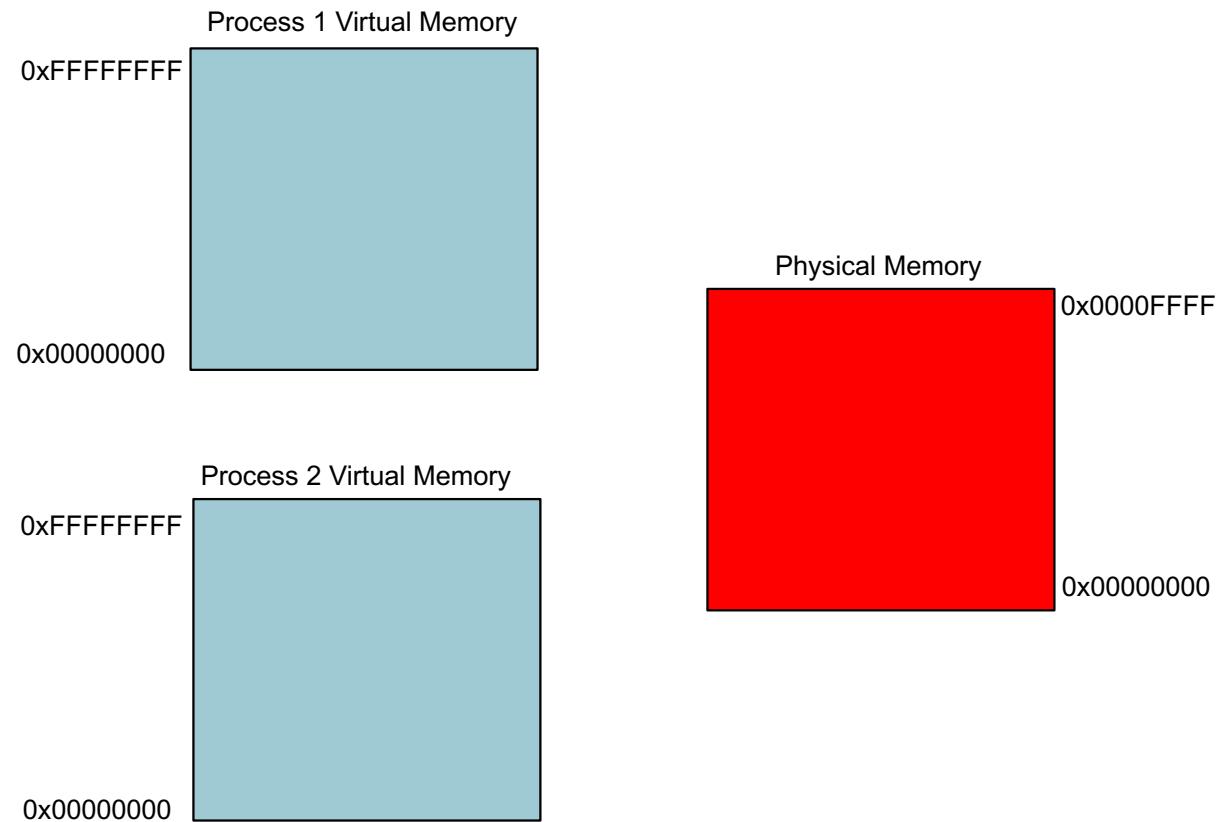
Chapter 5 – Part 2

Large and Fast Memory: Exploiting Memory Hierarchy

Virtual Memory

- There is a need to run programs that are **too large to fit in memory**
- Solution adopted in the 1960s, **split programs into little pieces, called overlays (pages)**
 - Kept on the disk, swapped in and out of main memory
- **Virtual memory** : each program has its own address space, broken up into chunks called **pages**
 - Each page is a continuous range of addresses that are mapped onto physical memory

Virtual Memory



Virtual Memory

- Use main memory as a “cache” for secondary (disk) storage
 - Programs share main memory
 - CPU and OS translate virtual addresses to physical addresses
 - VM “block” is called a page
 - VM translation “miss” is called Page Fault.
- ↳ Each gets a private virtual address space.
holding its frequently used code & data.
- ↳ Protected from other programs (OS).

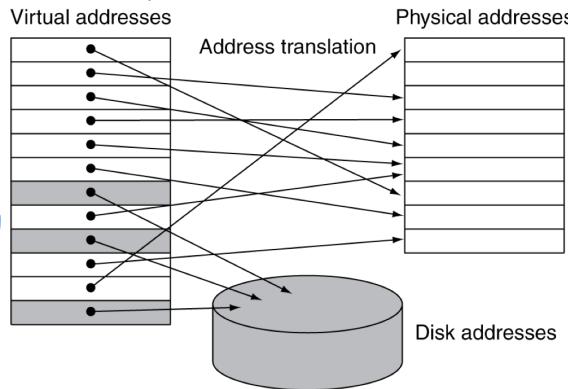
uses: → ① Map Mem." to Disk.

Virtual Memory (VM)

- VM adds a level of indirection between the virtual program addresses (VA) and the physical RAM addresses (PA)
- But there is a bit of a price (performance) to pay for
 - VA to PA translation (some bookkeeping required which impacts the performance)
- Three ways to achieve this VA to PA translation:
 1. VA to PA translation on the fly using a Memory Management Unit (MMU)
 2. Page tables
 3. Fast translation via TLB

MMU
3 ways

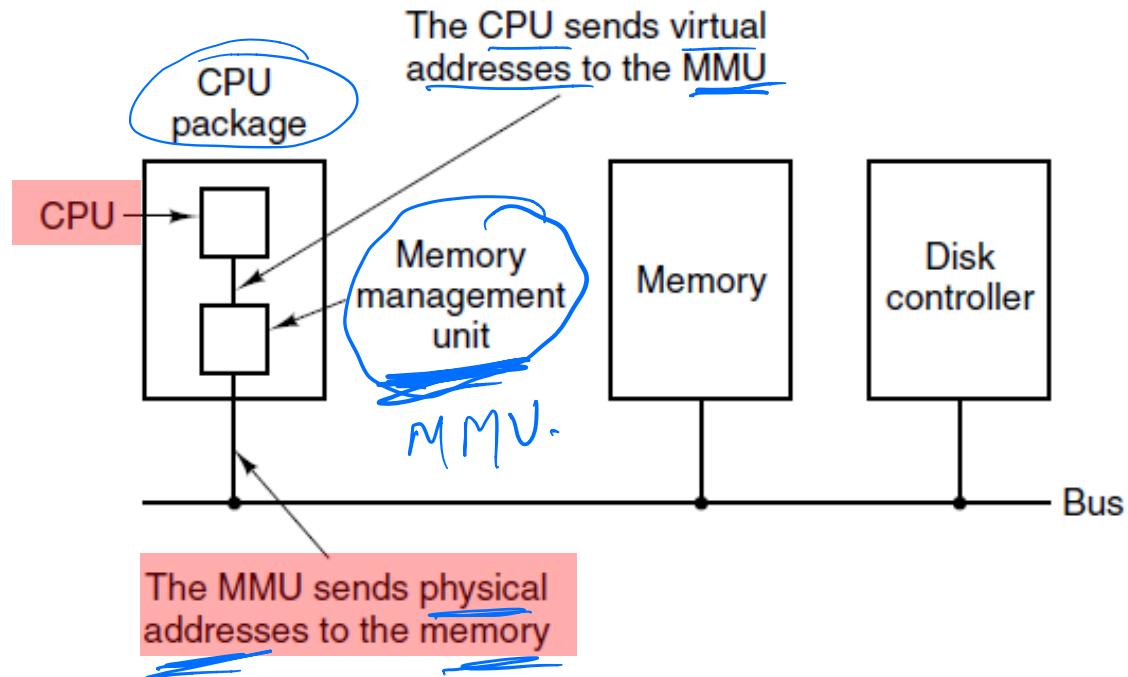
Translation Buffer.
Lookaside



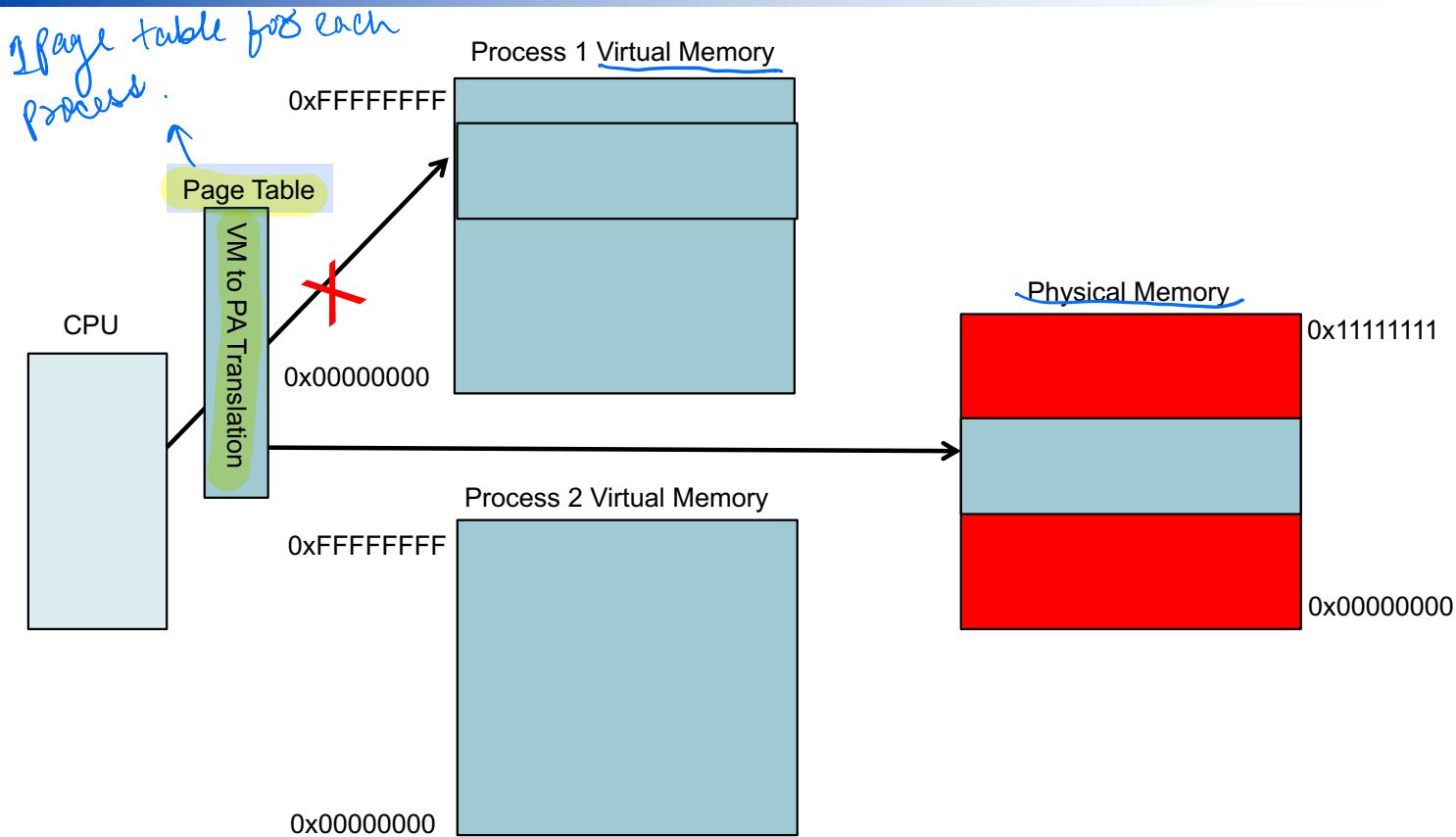
cont." earlier slide:-

- (1) use larger pages (4KB) to reduce the no. of page table entries (PTEs) needed.
- (2) Use a Translation lookaside buffer (TBL) to perform a very fast translation.

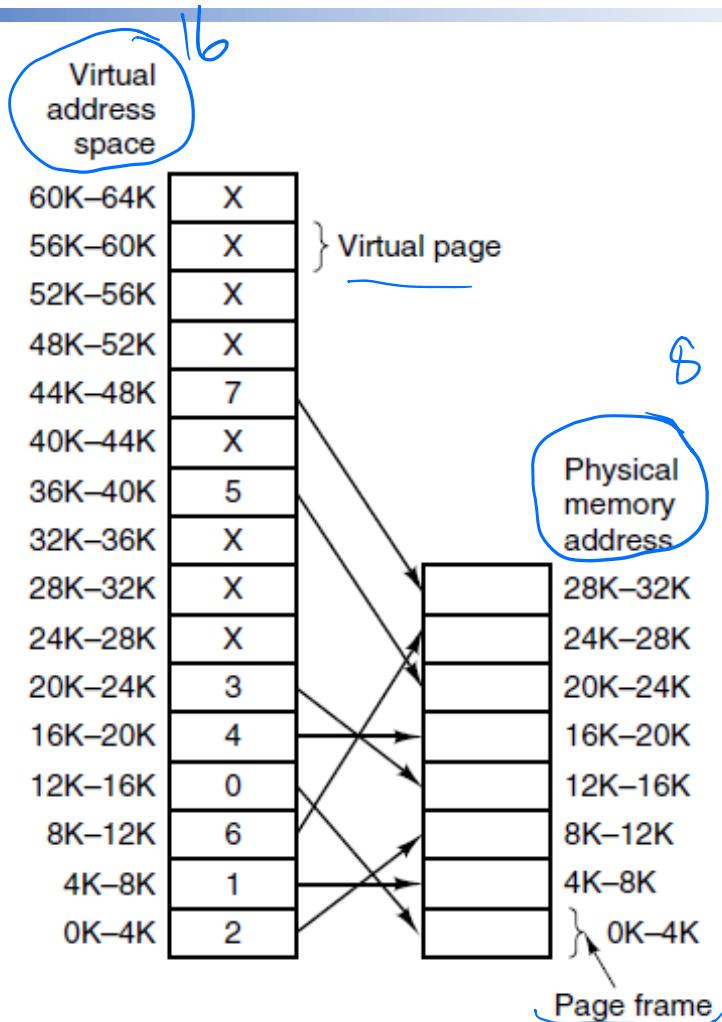
Paging



Paging



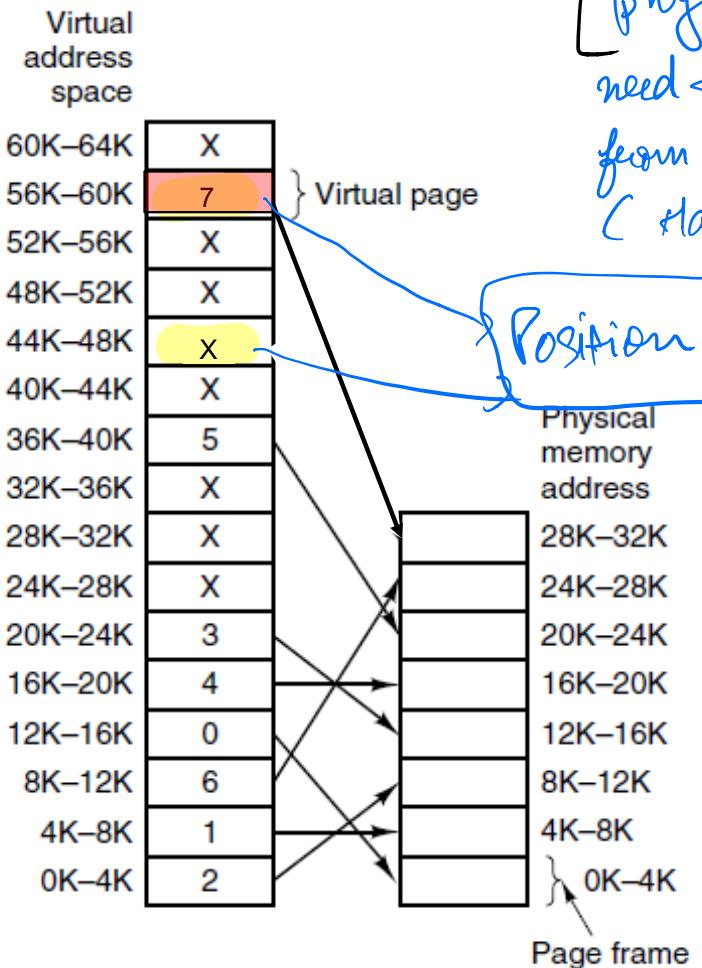
Paging



Paging

In this case, If we use MMU, it causes the CPU to be trapped to OS, this is called **Page fault**.

If instr. "wants an address but its unmapped in physical mem", then we need to bring that address from the lower lvl of mem (hard disk.)

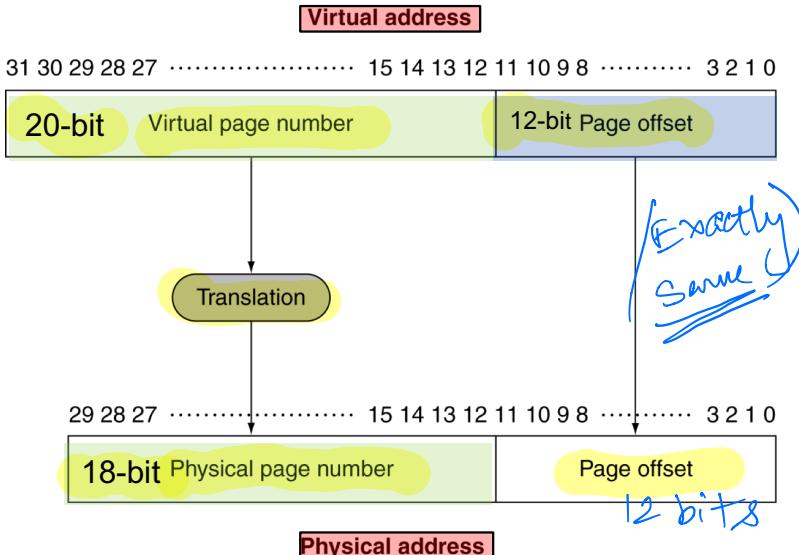
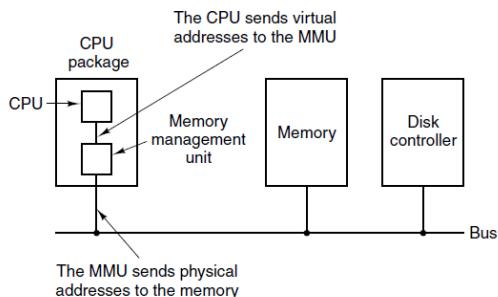
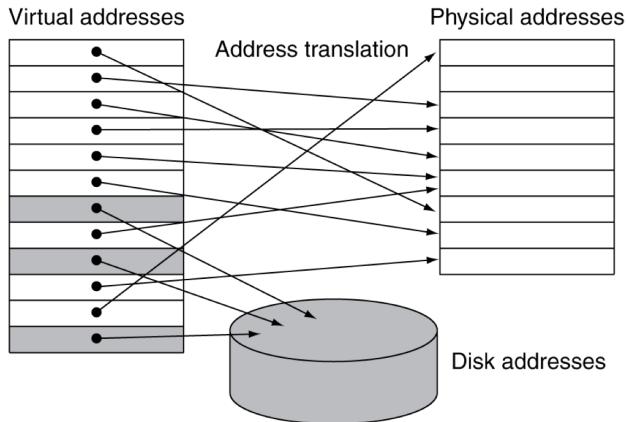


Address Translation

12 bits to uniquely identify them

= 4096 Bytes on the page

Fixed-size pages (e.g., 4K)



Mapping Virtual to Physical Address

4KB-page system

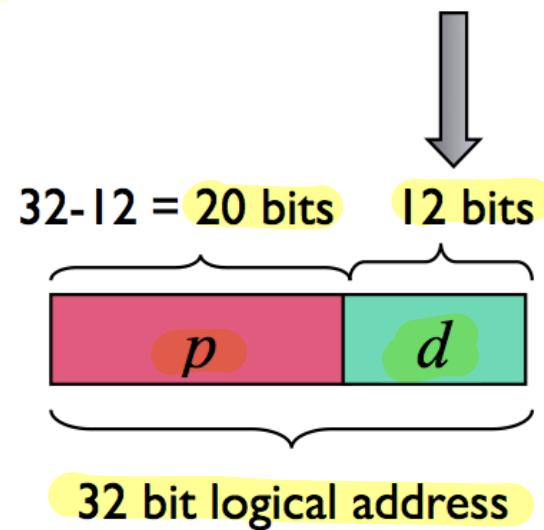
32-bit word addressing

- ♦ Split address from CPU into two pieces
 - Page number (p)
 - Page offset (d)
- ♦ Page number
 - Index into page table
 - Page table contains base address of page in physical memory
- ♦ Page offset
 - Added to base address to get actual physical memory address
- ♦ Page size = 2^d bytes

Example:

- 4 KB (=4096 byte) pages
- 32 bit logical addresses

$$2^d = 4096 \rightarrow d = 12$$



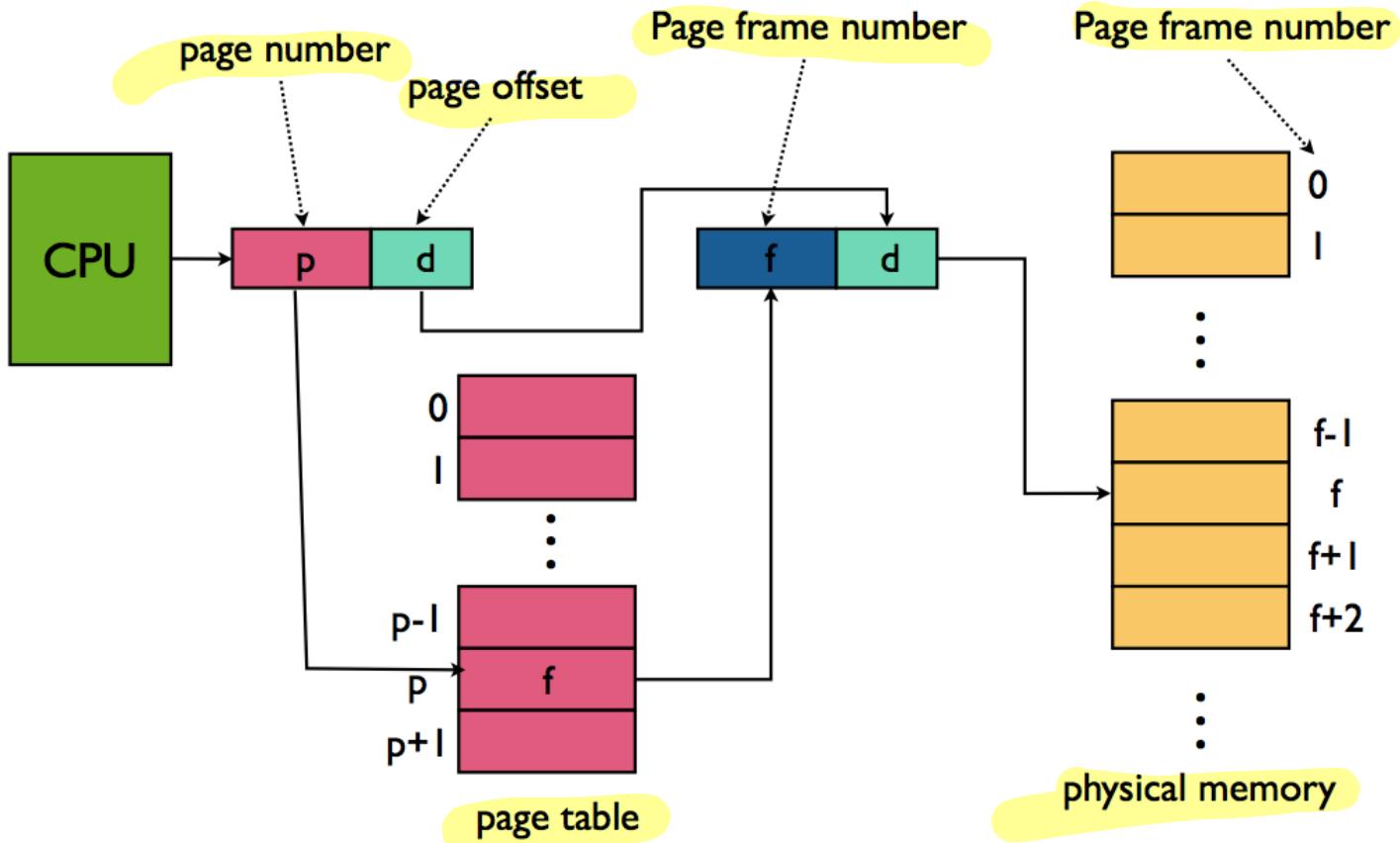
Page Tables

→ Optimizes the translation of a
Virtual Address into physical
address.

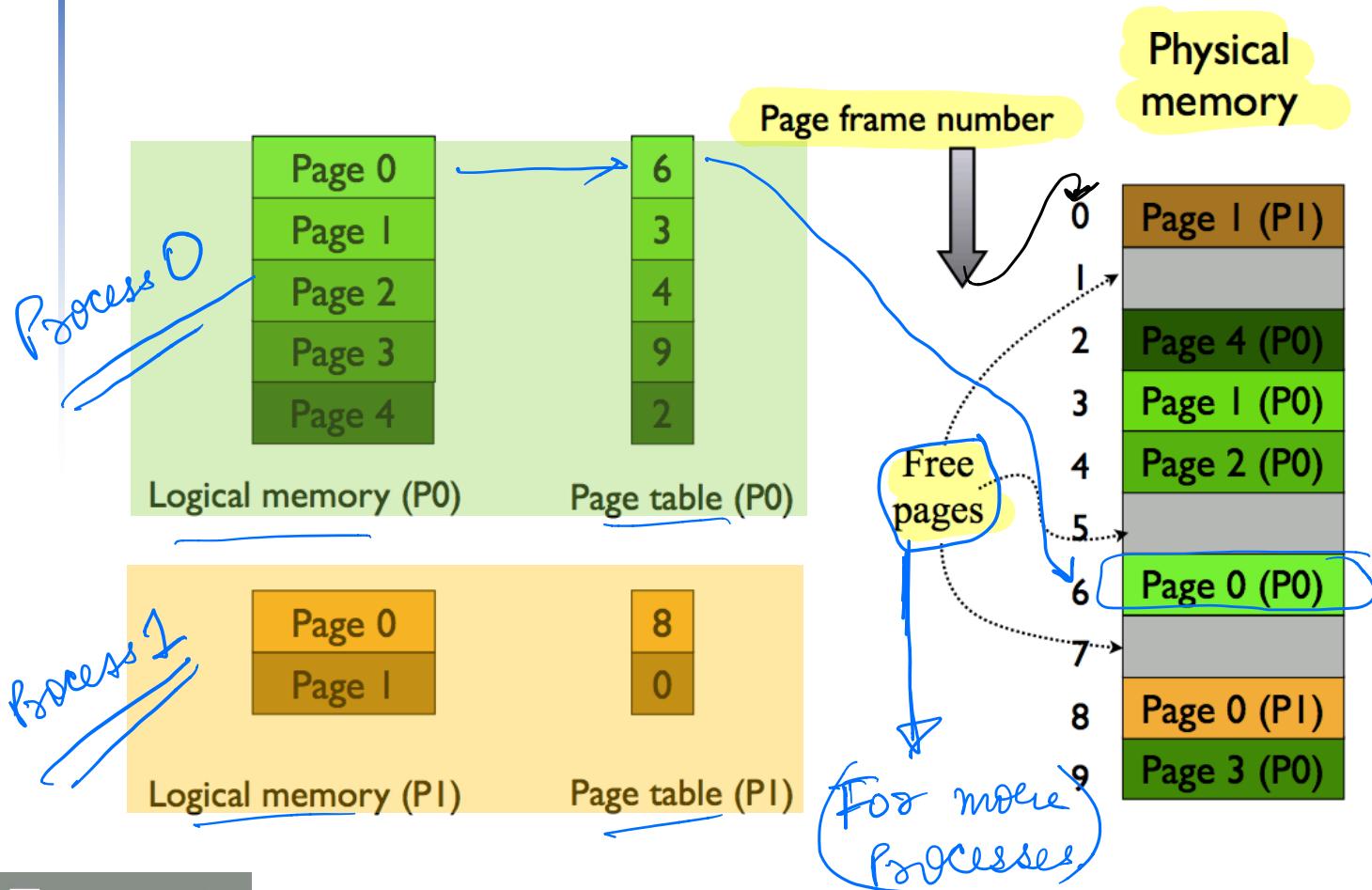
- Stores placement information → physical pg. no.
- If page is present in memory
- If page is not present (page fault)



Translation Using a Page Table



Multiple Processes in Memory



Page Replacement Algorithms

- Optimal algorithm
- Not recently used algorithm
- First-in, first-out (FIFO) algorithm
- Second-chance algorithm
- Clock algorithm
- Least recently used (LRU) algorithm
- Working set algorithm
- WSClock algorithm

Optimal Algorithm

Label each page with no. of "insts" to be executed before we reference the 1st page.

- Best possible but almost impossible to implement
- Almost impossible because the OS has no knowledge on when each page will be referenced next
- Possible solution
 - Run the program on simulator for a specific input data and keep track of those numbers
 - Implement optimal page replacement on the second run based on the info obtained the first time

First-in, First-out (FIFO) Algorithm

- Maintain a linked list of all pages
 - Maintain the order in which they entered memory
- Page at front of list replaced
- Advantage: (really) easy to implement
- Disadvantage: page in memory the longest may be often used
 - This algorithm forces pages out regardless of usage
 - Usage may be helpful in determining which pages to keep

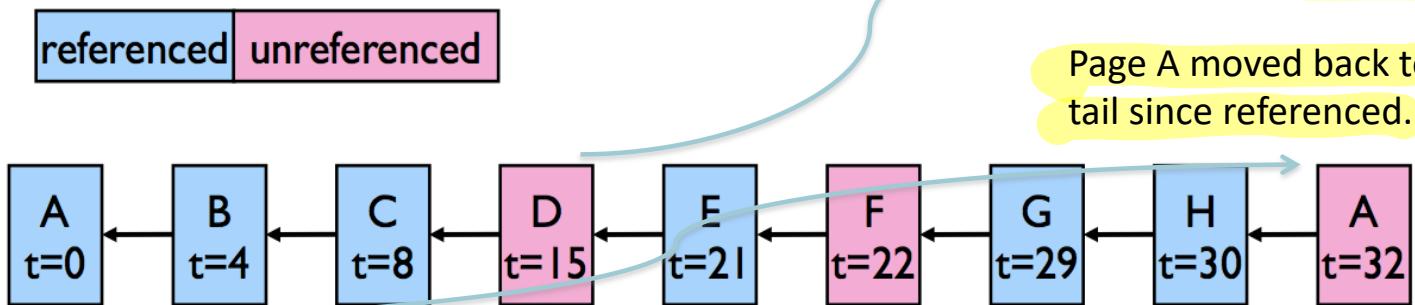
Second-Chance Algorithm

- ♦ **Modify FIFO to avoid throwing out heavily used pages**
 - If reference bit is 0, throw the page out
 - If reference bit is 1
 - Reset the reference bit to 0
 - Move page to the tail of the list
 - Continue search for a free page
- ♦ **Still easy to implement, and better than plain FIFO**

D \rightarrow Throw out.

D thrown out if not referenced when we get to it in the queue.

Page A moved back to the tail since referenced.



Not Recently Used Algorithm

→ Uses R, M
bits

- At page fault, the OS inspects pages and categorizes them based on the current values of their **R** (reference) and **M** (dirty) bits:

Class 0: not referenced, not modified.

Class 1: not referenced, modified.

Class 2: referenced, not modified.

Class 3: referenced, modified.

Simulating LRU in Software

Last Recently Used.

R bits for pages 0-5, clock tick 0		R bits for pages 0-5, clock tick 1		R bits for pages 0-5, clock tick 2		R bits for pages 0-5, clock tick 3		R bits for pages 0-5, clock tick 4	
1	0	1	0	1	1	1	1	0	1
Page	0 1 2 3 4 5	Page	0 1 2 3 4 5	Page	0 1 2 3 4 5	Page	0 1 2 3 4 5	Page	0 1 2 3 4 5
0	10000000	1	11000000	2	11100000	3	11110000	4	01111000
1	00000000	2	10000000	3	11000000	4	01100000	5	10110000
2	10000000	3	01000000	4	00100000	5	00010000	0	10010000
3	00000000	4	00000000	5	10000000	0	01000000	1	00100000
4	10000000	5	11000000	0	01100000	1	10110000	2	01011000
5	10000000	0	01000000	1	10100000	2	01010000	3	00101000
(a)	(b)	(c)	(d)	(e)	<i>Lowest Value :- Kick Out.</i>				

Local versus Global Allocation Policies

Age
A0
A1
A2
A3
A4
A5
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

Youngest

$t = 3$ (oldest in A)

$t = 2$ (oldest of all)

youngest
oldest
youngest-

(a)

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

A0
A1
A2
A3
A4
A5
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(c)

Local versus global page replacement.

- (a) Original configuration. (b) Local page replacement.
(c) Global page replacement.

Multi-level Page Tables

- Problem: page tables can be too large
 - In 4KB page size, with 32 bit logical addresses
 - $2^{12} = 4096$ (bytes), so 12 bits to get to the page byte offset
 - $2^{20} = 1$ million PTEs , so 20 bits to get to the page address
- Solution: use multi-level page tables
 - 1st level page table has pointers to 2nd level page tables
 - 2nd level page table has actual physical page numbers in it
- <http://www.youtube.com/watch?v=Z4kSOv49GNc>

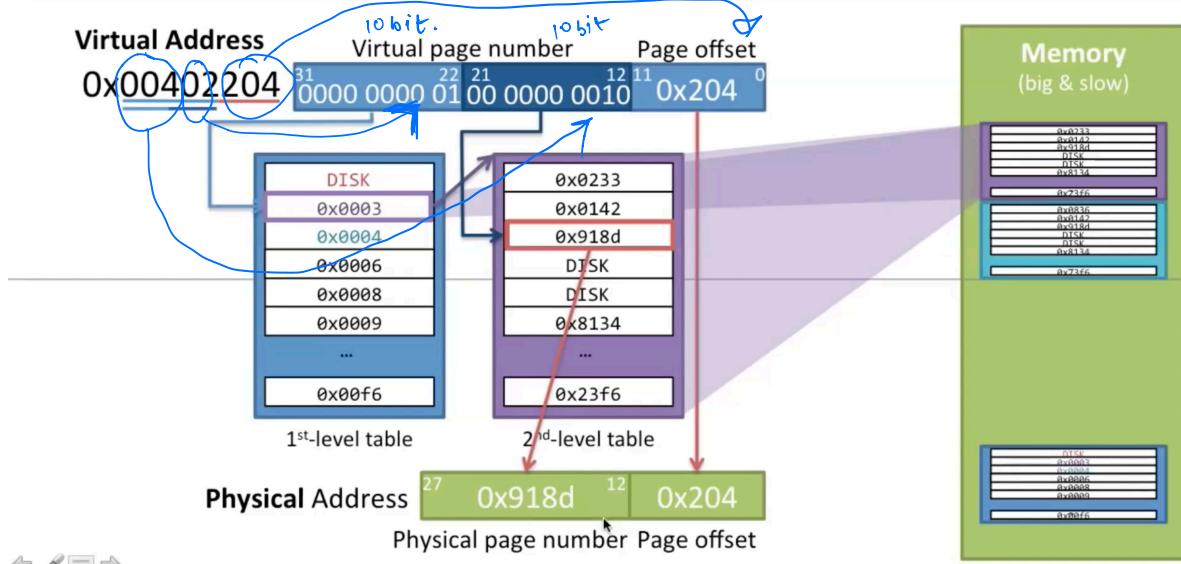
Fast Translation Using a TLB

- A widely implemented scheme for
 - Speeding up paging
 - Handling large virtual address spaces
- Small HW device for mapping virtual address to physical address without going through the page table in memory if possible
- <http://www.youtube.com/watch?v=uyrSn3qbZ8U>
- <http://www.youtube.com/watch?v=95QpHJX55bM>

Concluding Remarks

- Fast memories are small, large memories are slow
 - We really want fast, large memories ☹
 - Caching gives this illusion ☺
- Principle of locality
 - Programs use a small part of their memory space frequently
- Memory hierarchy
 - L1 cache \leftrightarrow L2 cache $\leftrightarrow \dots \leftrightarrow$ DRAM memory \leftrightarrow disk
- Memory system design is critical for multiprocessors

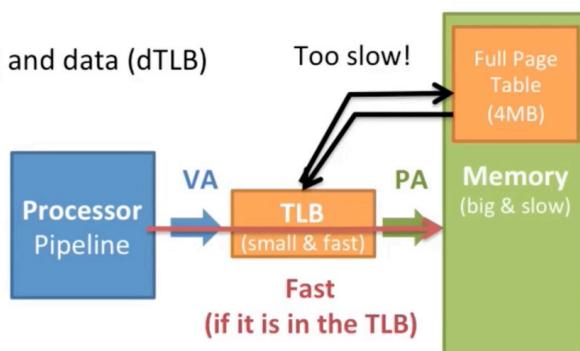
Multi-level page table translation



Making VM fast: the TLB

- To make VM fast we add a special **Page Table cache**: the **Translation Lookaside Buffer (TLB)**
 - Fast: less than 1 cycle (have to do it for every memory access)
 - Very similar to a cache
- To be fast, TLBs must be small:
 - Separate TLBs for instructions (iTLB) and data (dTBL)
 - 64 entries, 4-way (4kB pages)
 - 32 entries, 4-way (2MB pages)
 - (Page Table is 1M entries)

Lots of locality!
Miss rates are typically
only a few percent.



What can happen when we access memory?

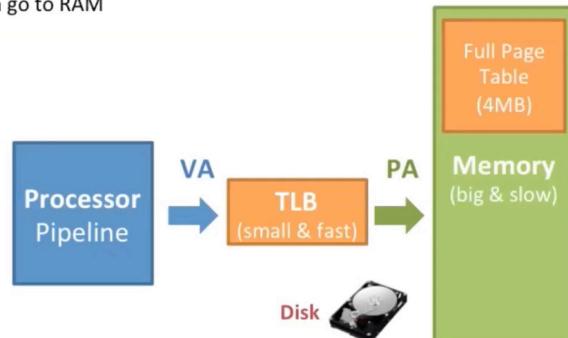
Good: Page in RAM

- PTE in the TLB
 - Excellent
 - <1 cycle to translate, then go to RAM (or cache)
- PTE not in the TLB
 - Poor
 - 20-1000 cycles to load PTE from RAM, then go to RAM

With 1.33 memory accesses per instruction we can't afford 20-1000 cycles very often.

Bad: Page not in RAM

- PTE in the TLB (unlikely)
 - Horrible
 - 1 cycle to know it's on disk
 - ~80M cycles to get it from disk
- PTE not in the TLB
 - (ever so slightly more) horrible
 - 20-1000 cycles to know it's on disk
 - ~80M cycles to get it from disk



Example translation (empty TLB)

