# Chapter 4

The Processor

Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two LEGv8 implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: LDUR, STUR
  - Arithmetic/logical: ADD, SUB, AND, & ORR
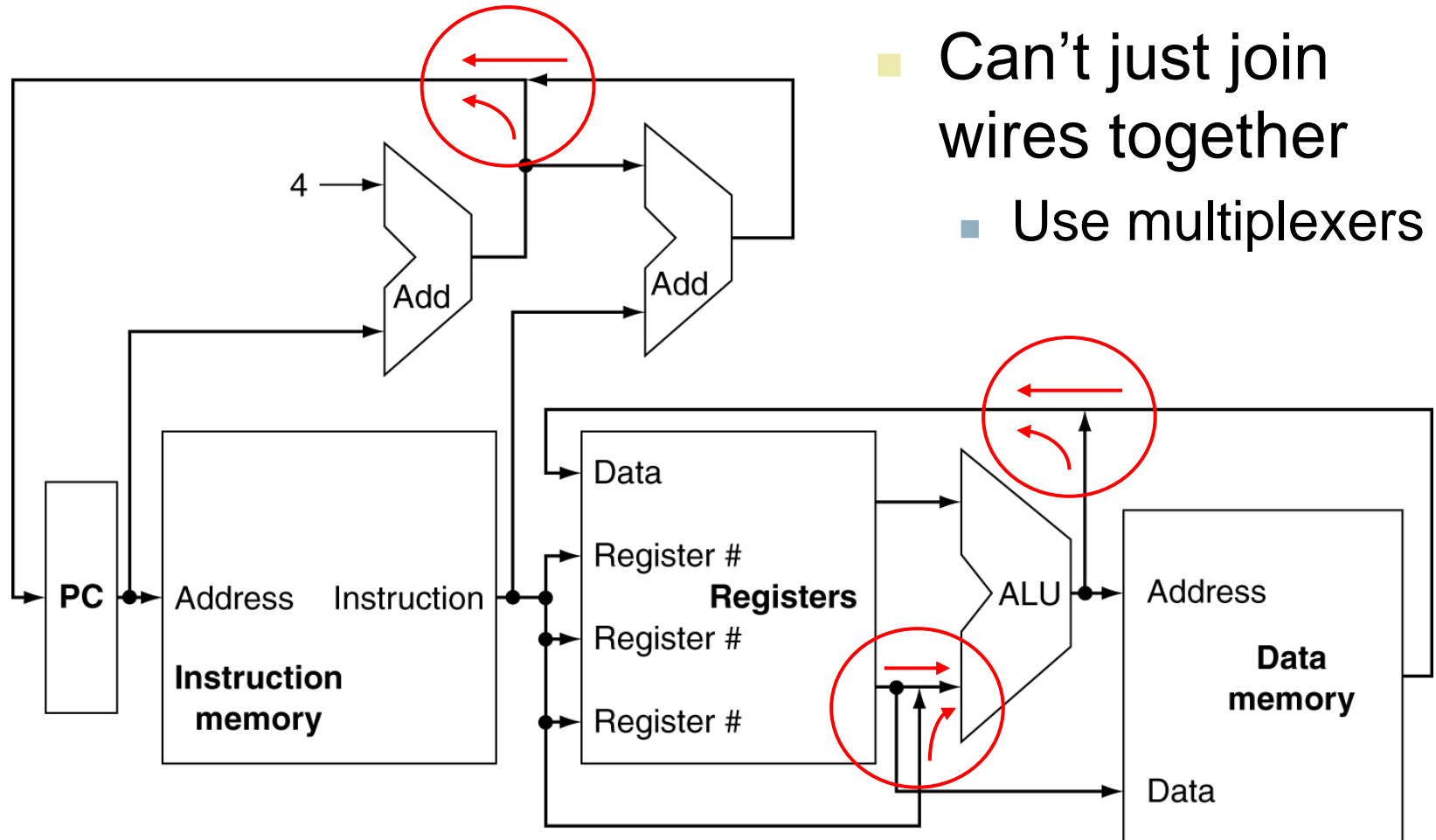 - Control transfer: CBZ, B

Instruction Execution

- PC $\rightarrow$ instruction memory, fetch instruction
- Register numbers $\rightarrow$ register file, read registers
- Depending on instruction class
    - Use ALU to calculate
        - Arithmetic result
        - Memory address for load/store
        - Branch target address
    - Access data memory for load/store
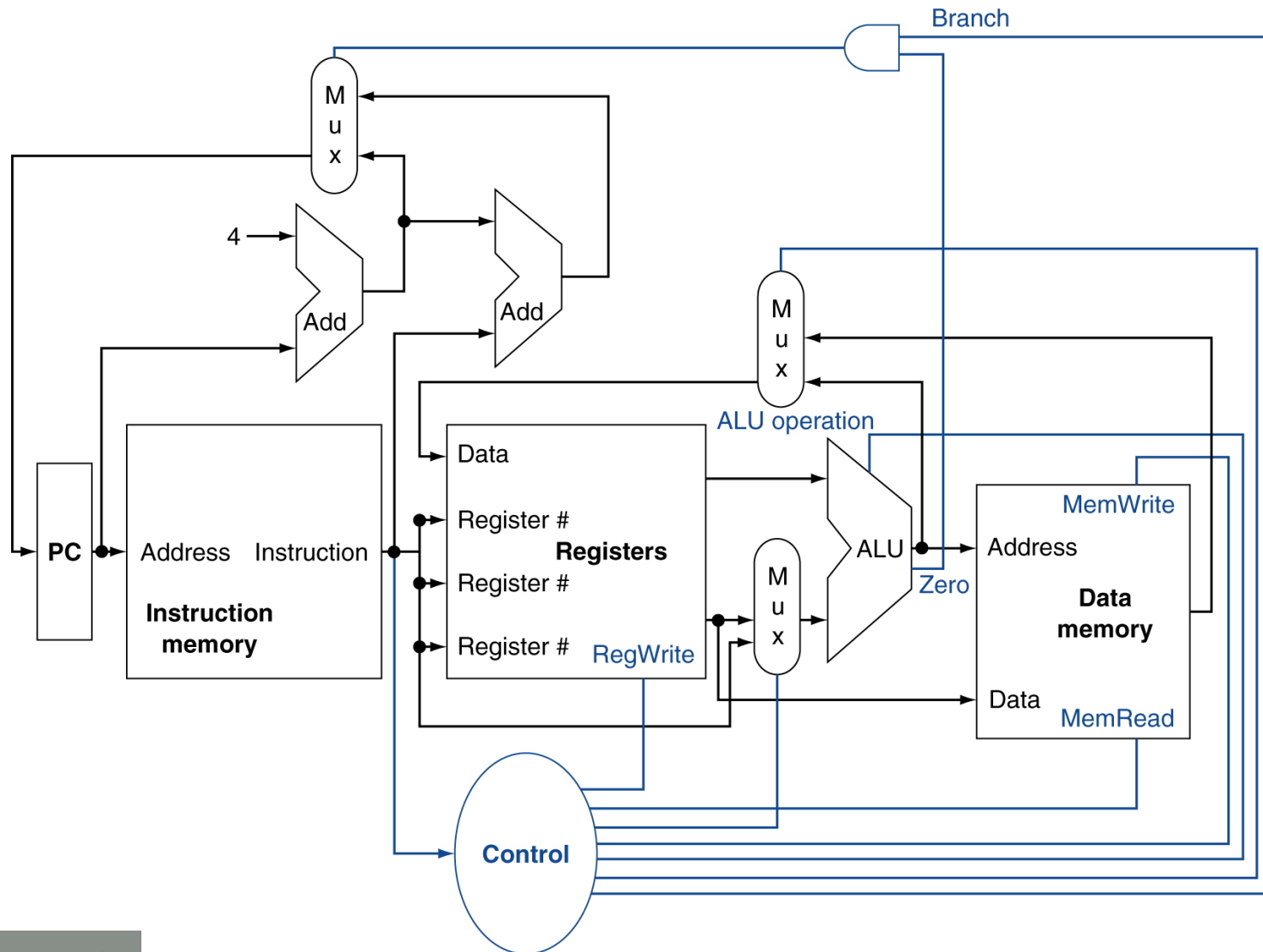   - PC $\leftarrow$ target address or PC + 4

CPU Overview
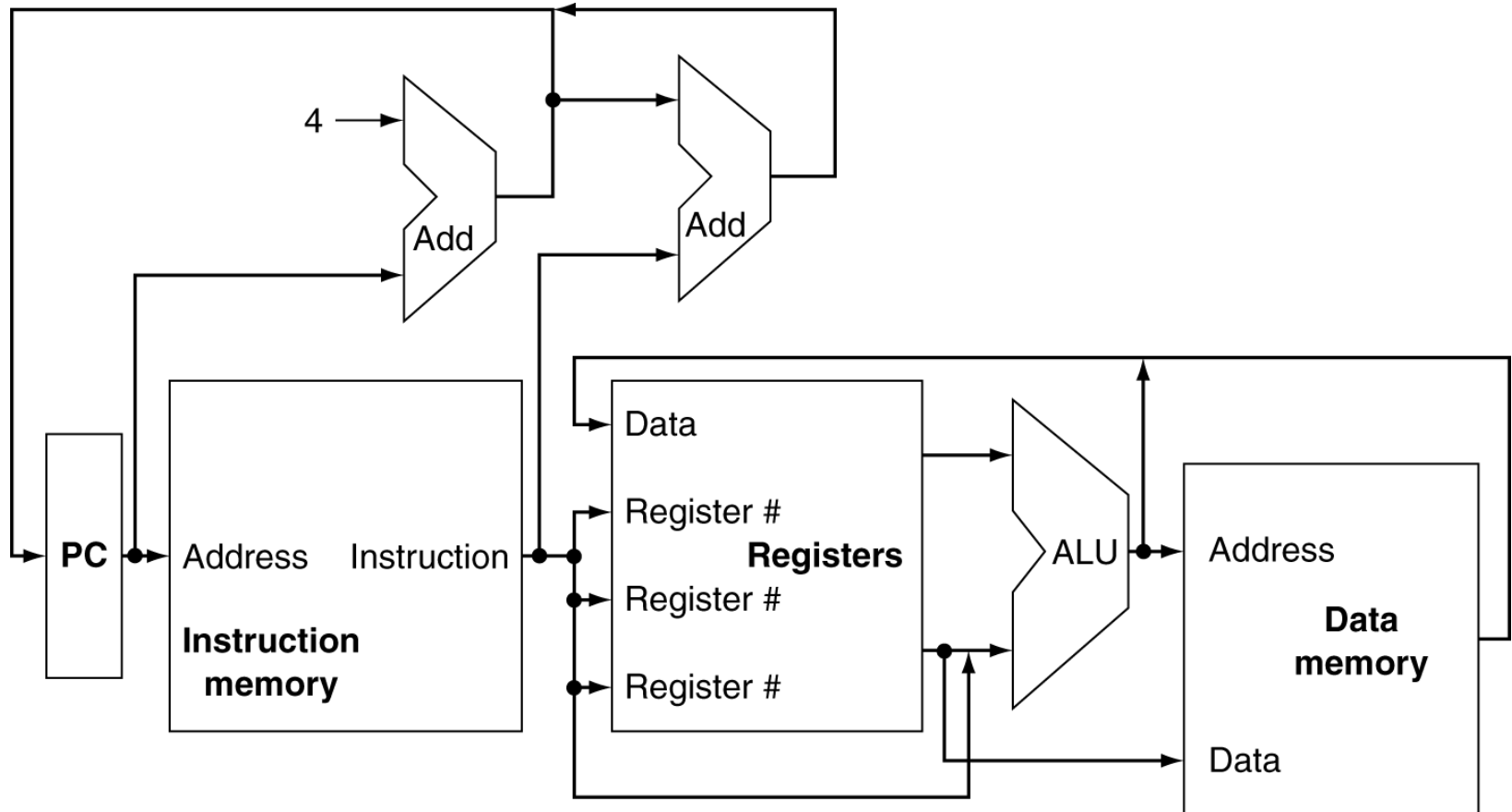
Multiplexers

- Can't just join wires together
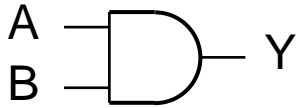 - Use multiplexers

Control

Logic Design Basics

- Information encoded in binary
    - Low voltage = 0, High voltage = 1
    - One wire per bit
    - Multi-bit data encoded on multi-wire buses
- Combinational element
    - Operate on data
    - Output is a function of input
- State (sequential) elements
    - Store information

Combinational Elements

- ## AND-gate
  - ### Y = A & B

A —⊐
B —⊐)— Y

- ## Multiplexer
  - ### Y = S ? I1 : I0

I0 → ⎧M⎫
       ⎨u⎬ → Y
I1 → ⎩x⎭
        |
        S
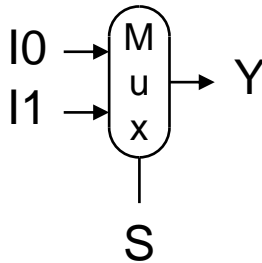
- ## Adder
  - ### Y = A + B

A →
     + → Y
B →
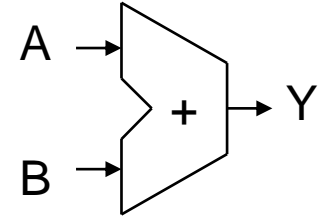
- ## Arithmetic/Logic Unit
  - ### Y = F(A, B)

A →
     ALU → Y
B →
      |
      F
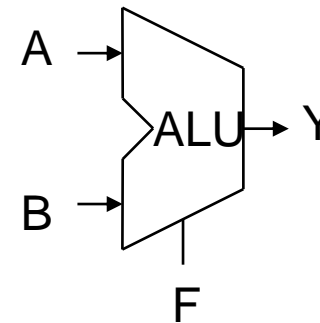
Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1
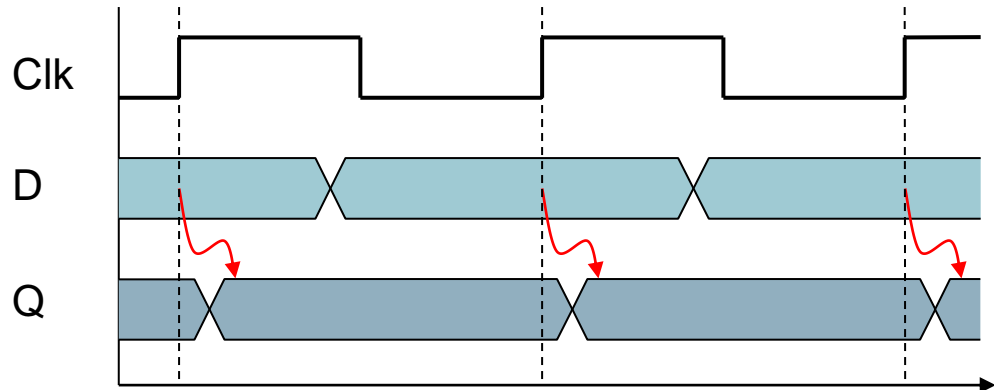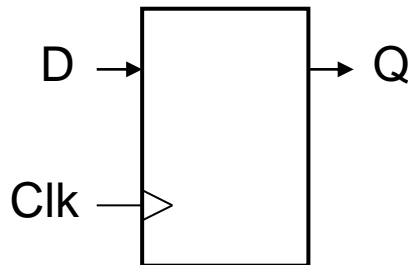
Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later
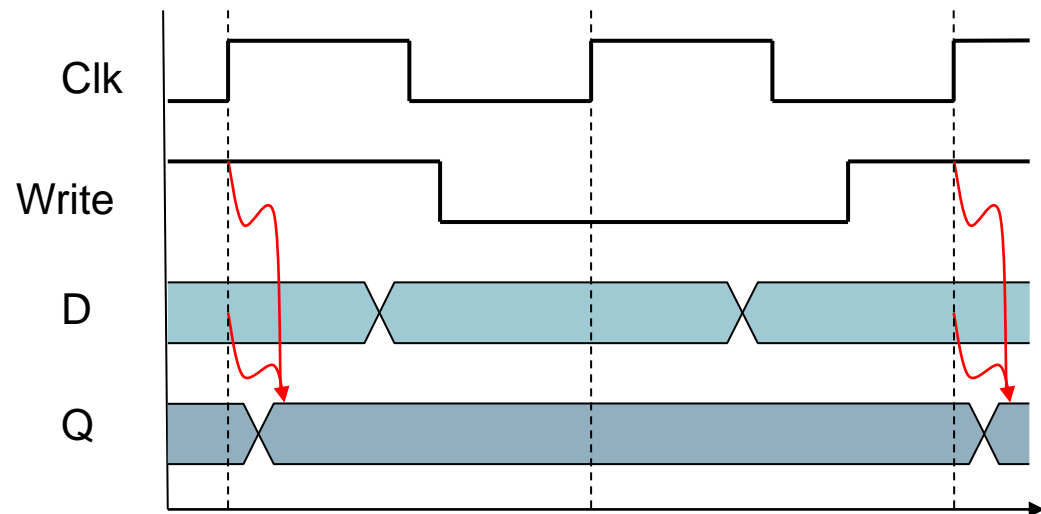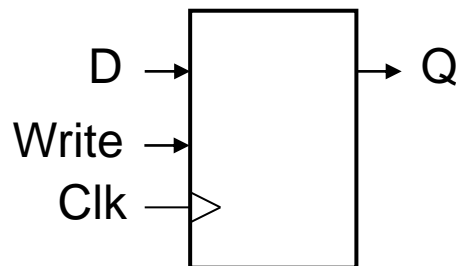
Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period
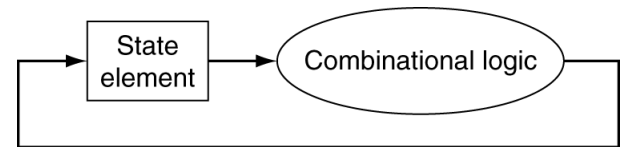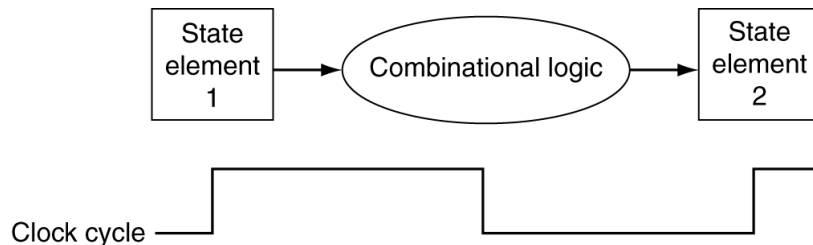
Building a Datapath

- 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

Instruction Fetch

64-bit register

PC

Read address

Instruction

Instruction memory

Add

4

Increment by 4 for next instruction
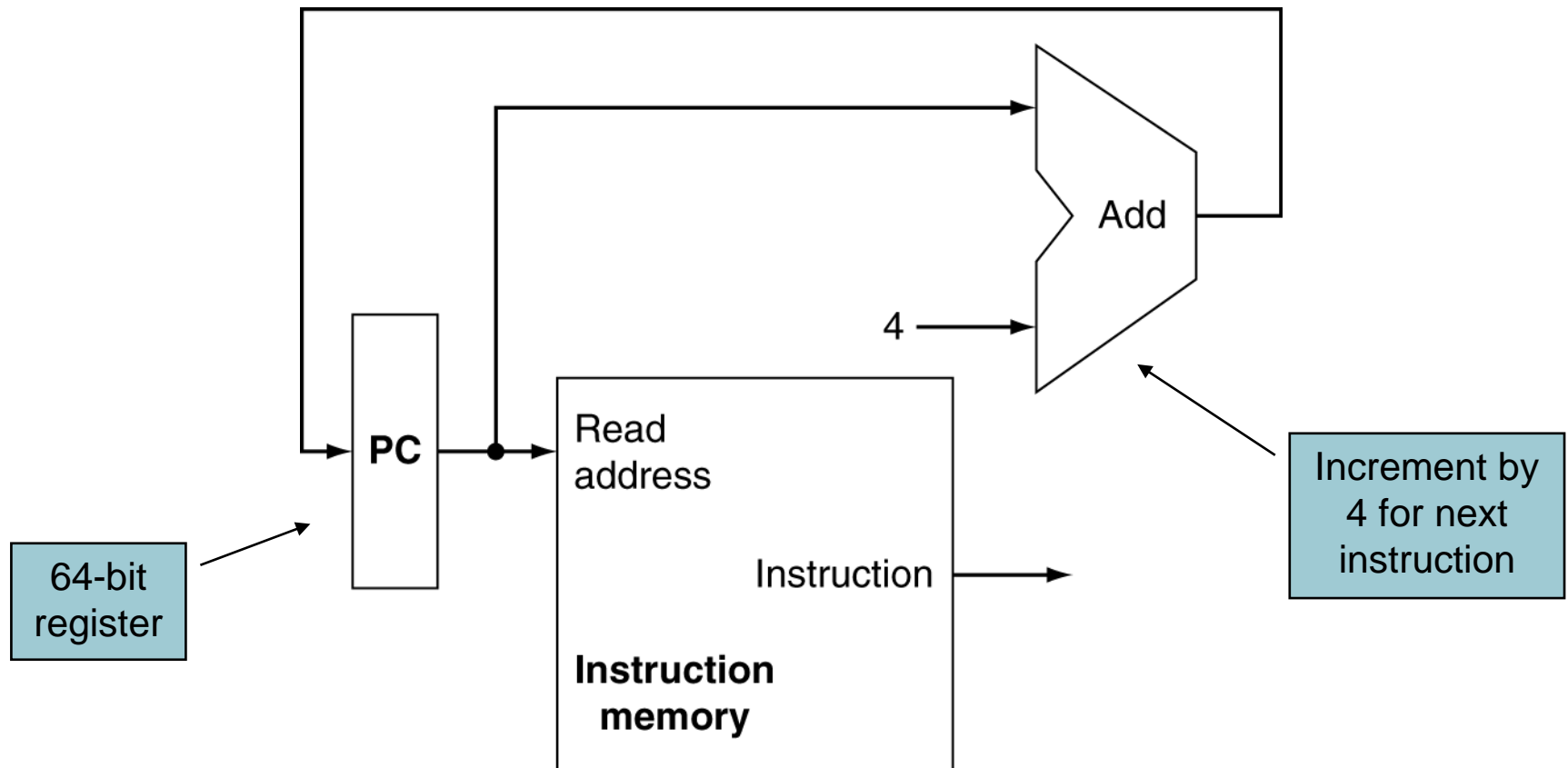
R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result
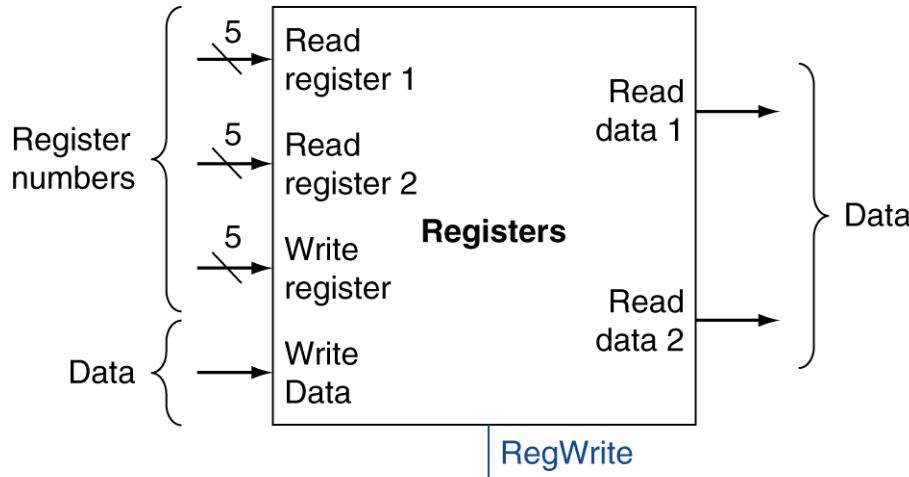
a. Registers
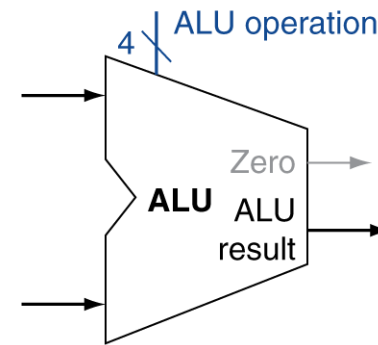
b. ALU

Load/Store Instructions

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

a. Data memory unit

b. Sign extension unit
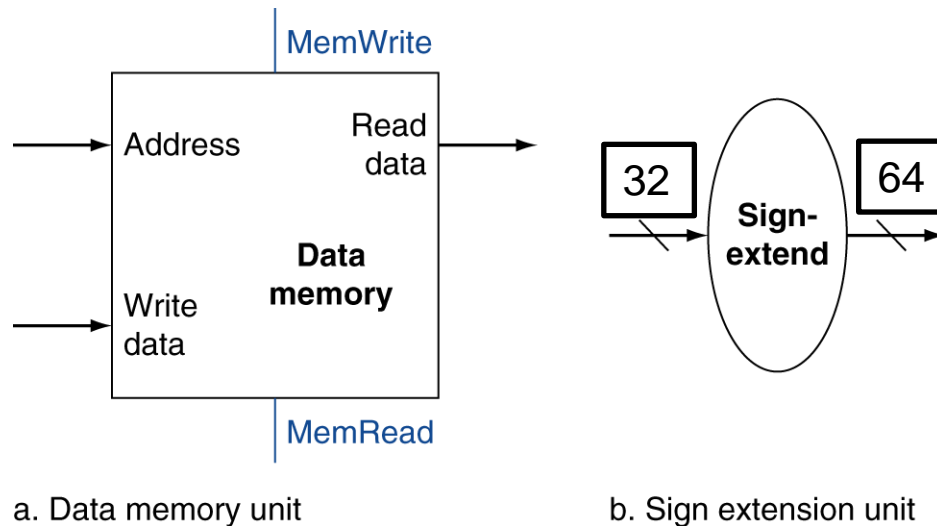
Branch Instructions

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

Branch Instructions

Just re-routes wires

PC+4 from instruction datapath

Add Sum → Branch target

Shift left 2

ALU operation
4
Read data 1
ALU Zero → To branch control logic

Instruction

Registers
Read register 1
Read register 2
Write register
Write data

Read data 2

RegWrite

32 — Sign-extend — 64

Sign-bit wire replicated
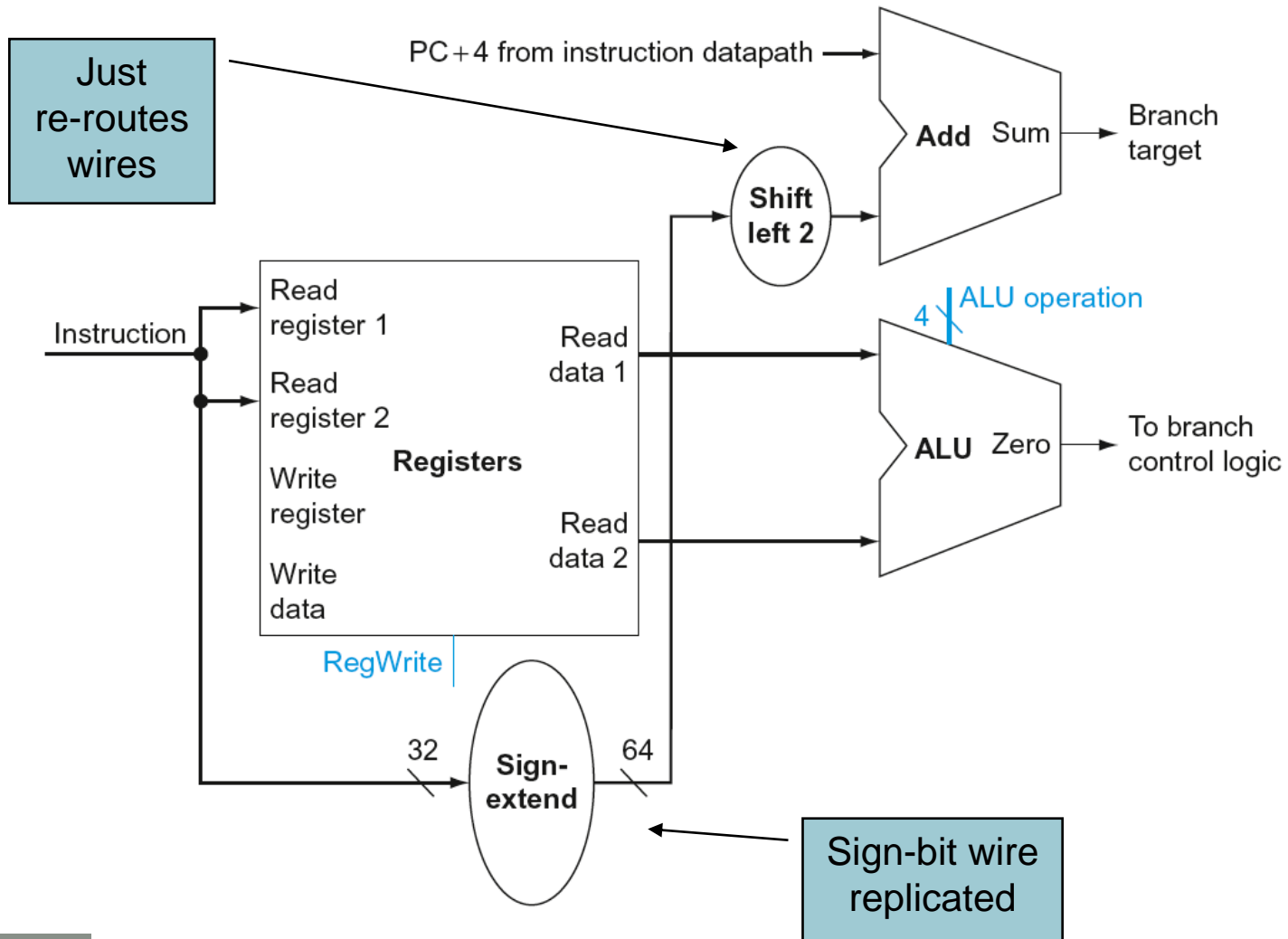
Composing the Elements

- First-cut data path does 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
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath
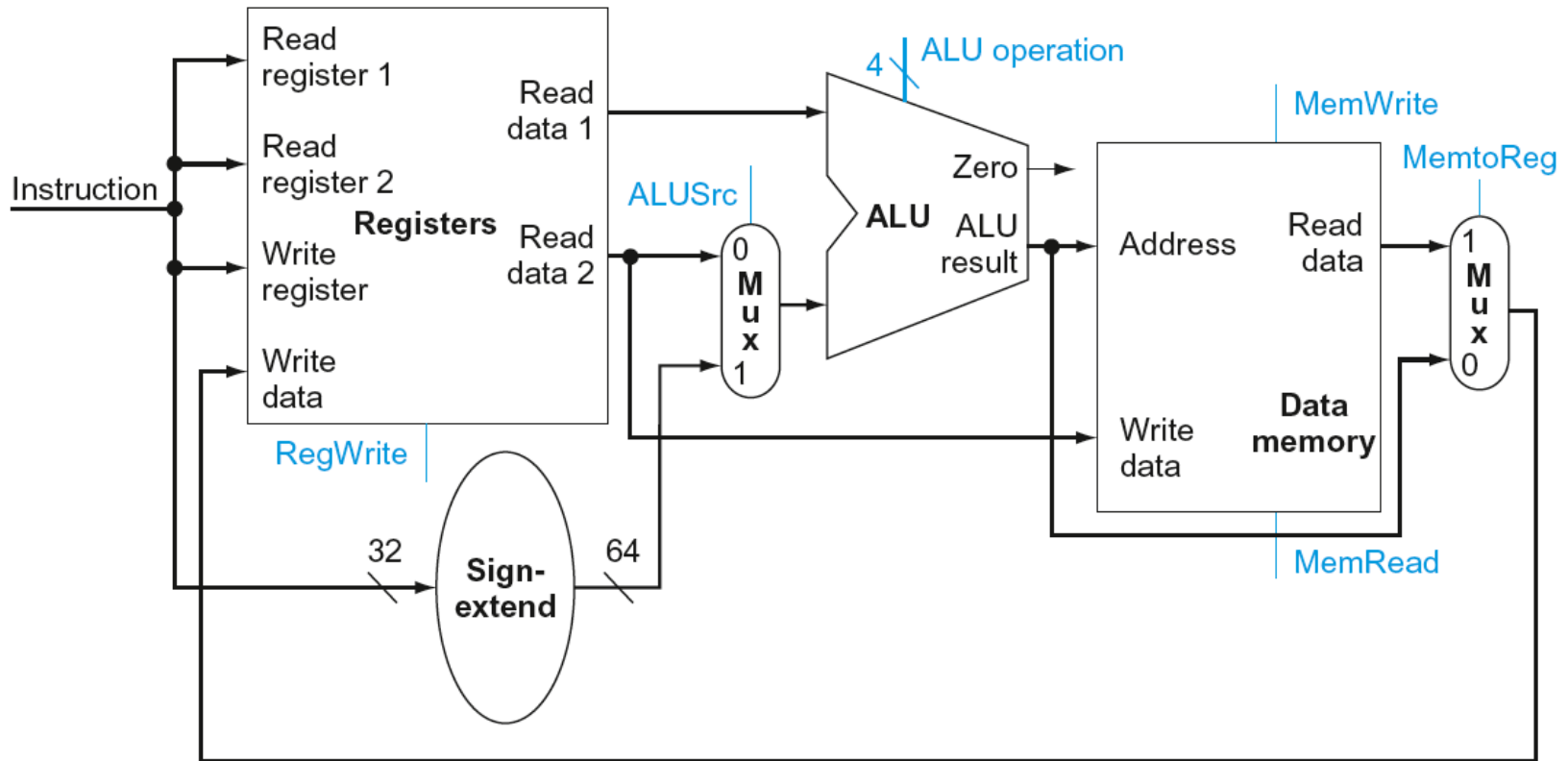
Full Datapath
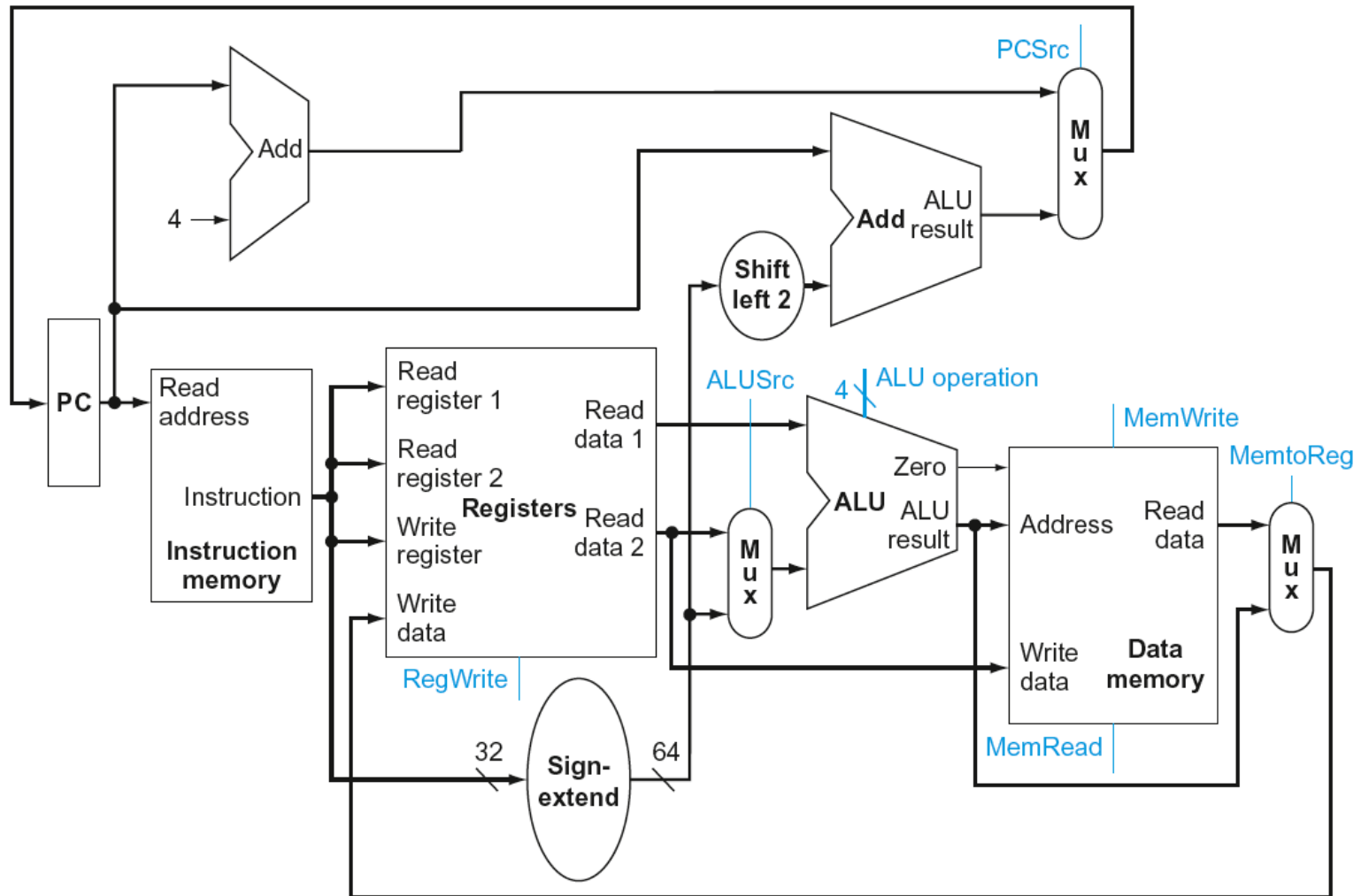
ALU Control

- ## ALU used for

  - ### Load/Store: F = add

  - ### Branch: F = pass input b

  - ### R-type: F 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 derived from opcode
  - Combinational logic derives ALU control

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

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

Field	180	address	Rt
Bit positions	31:26	23:5	4:0

c. Conditional branch instruction

Datapath With Control
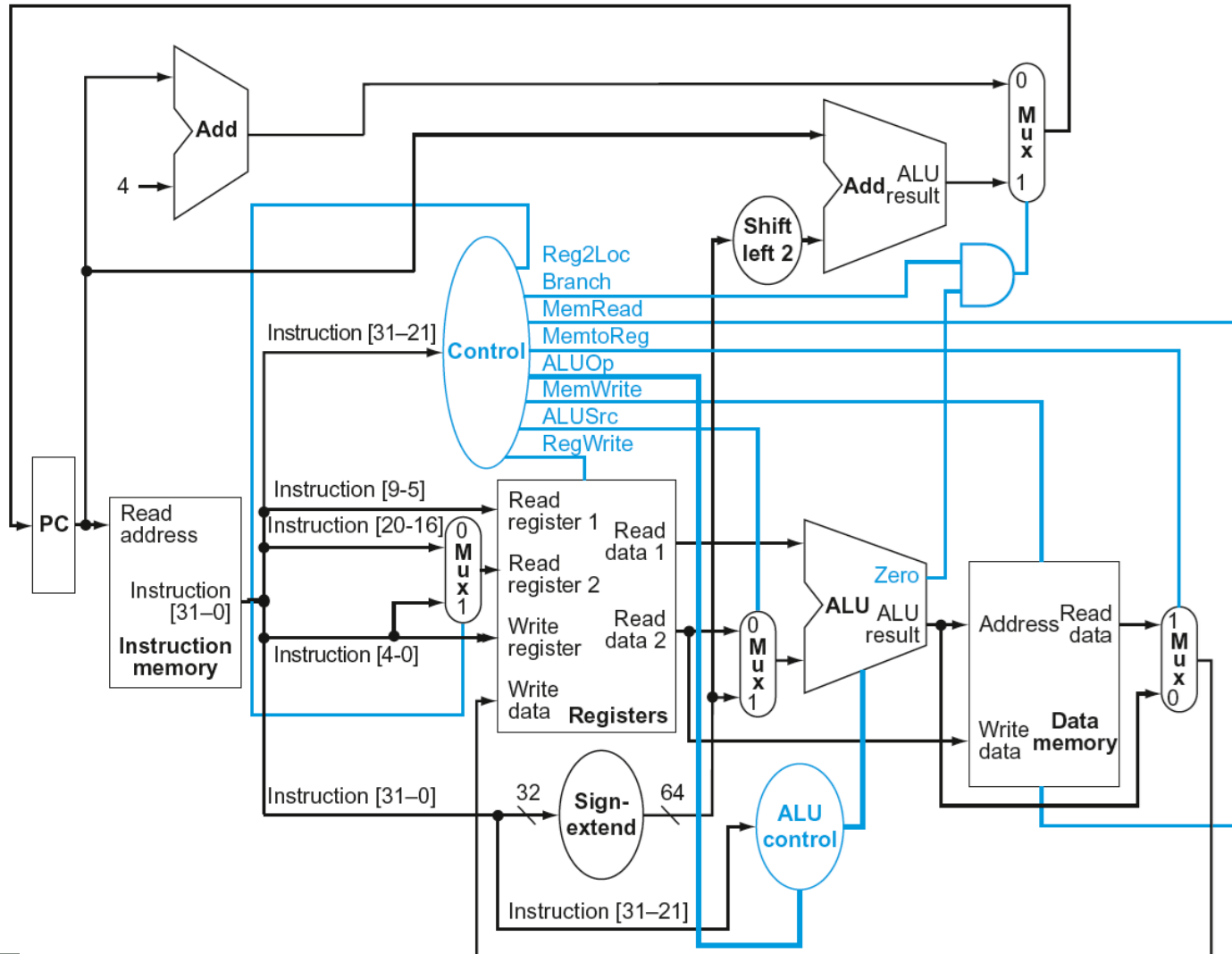
R-Type Instruction
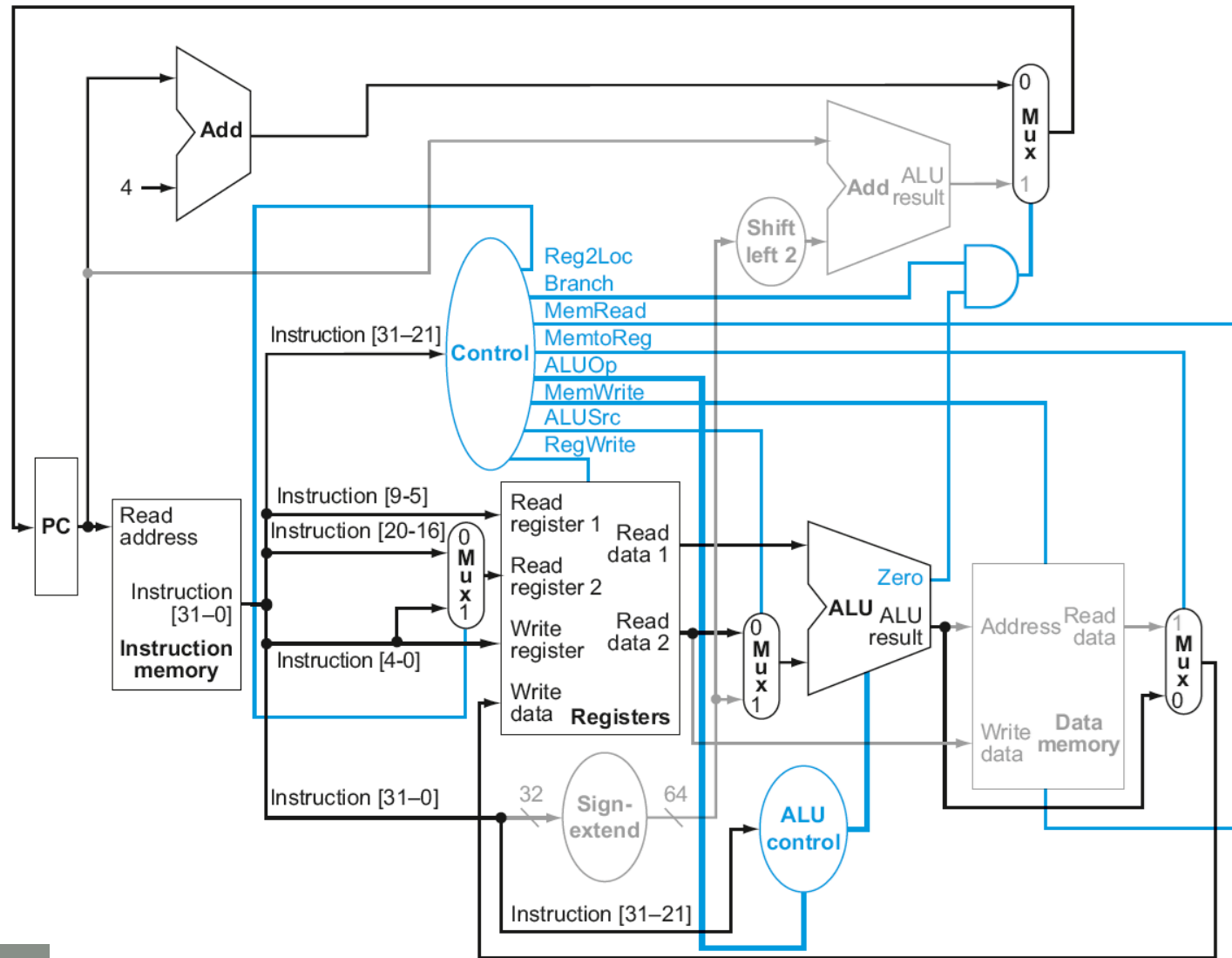
Load Instruction
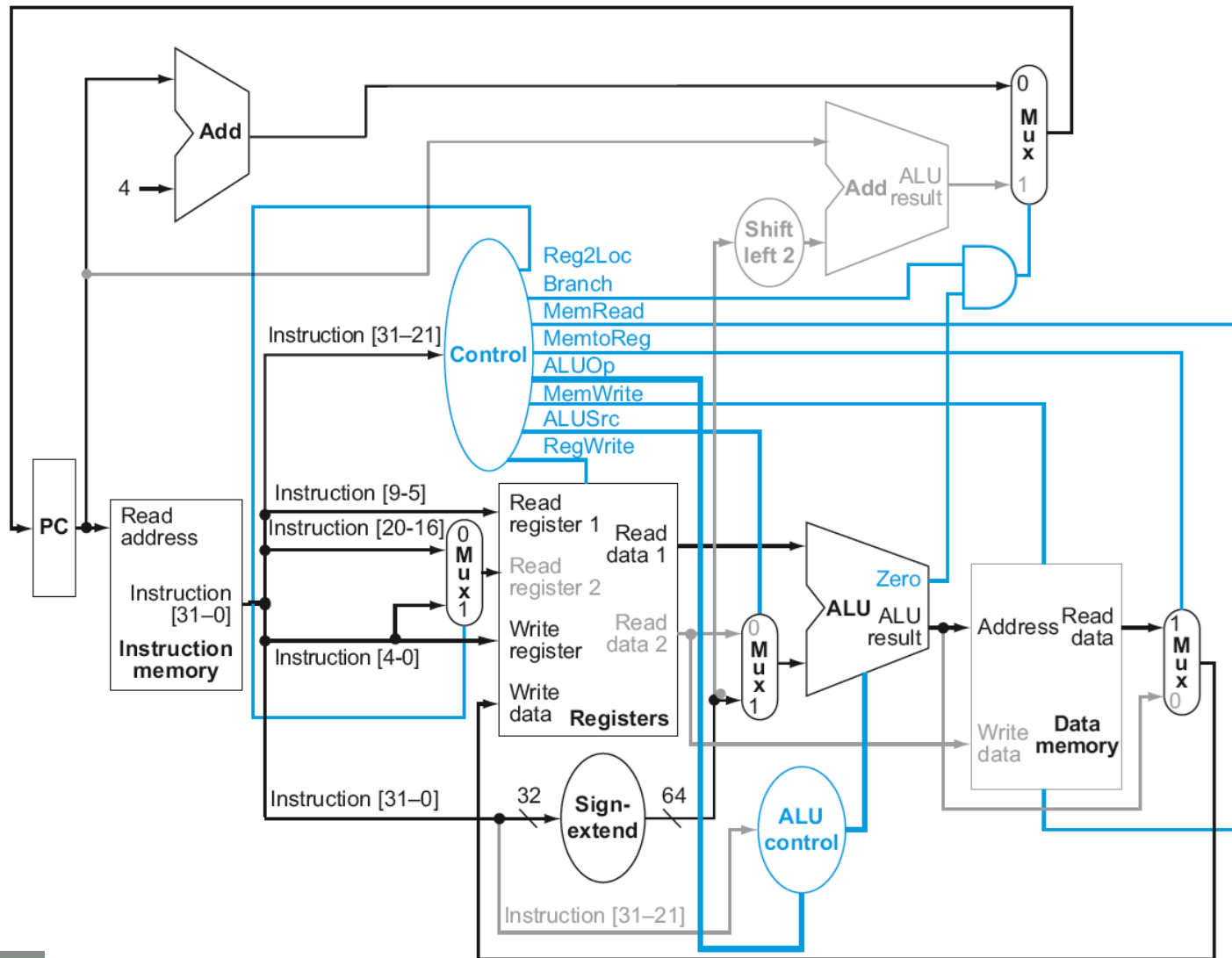
CBZ Instruction
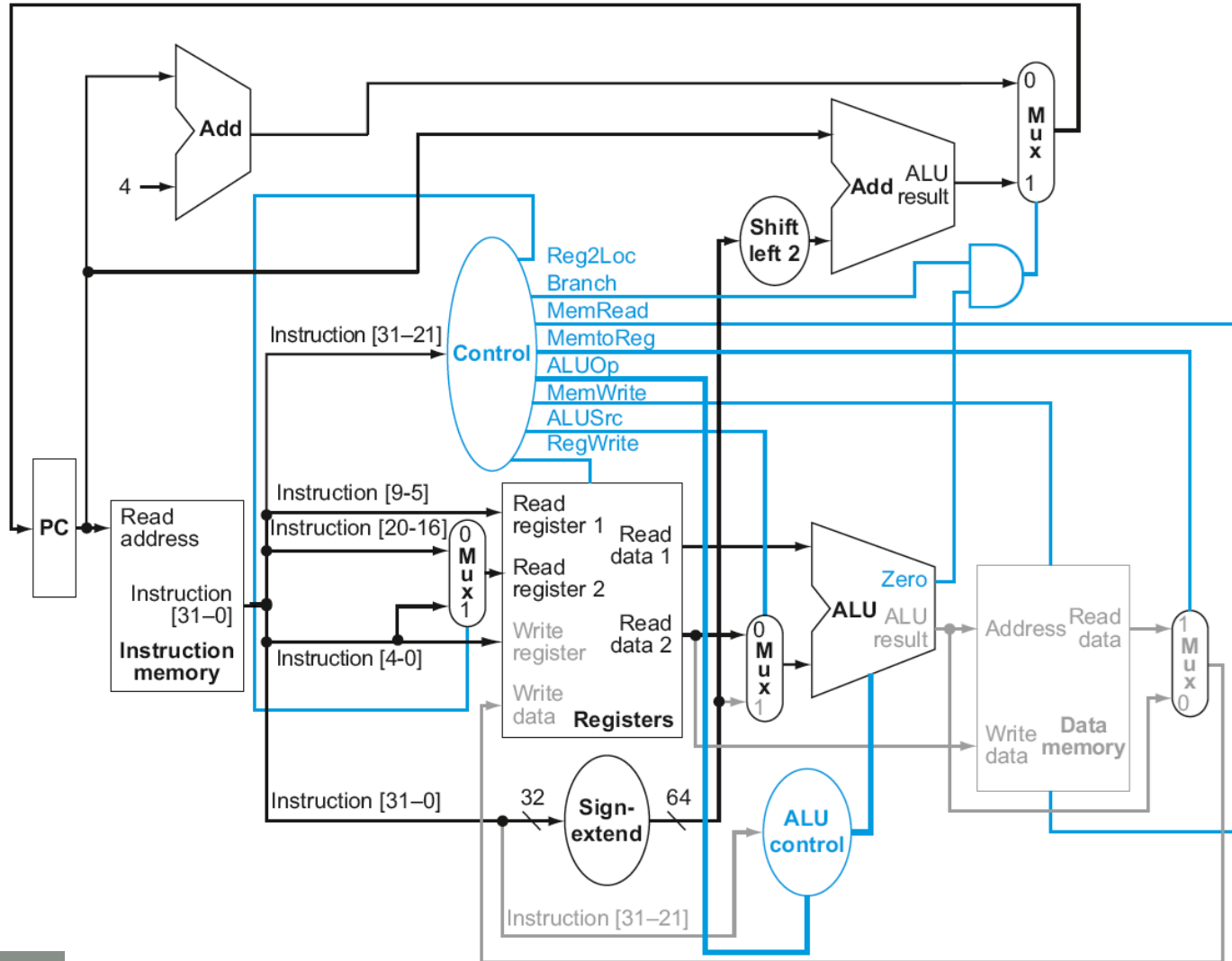
Implementing Uncnd'l Branch

Jump	2	address
	31:26	25:0
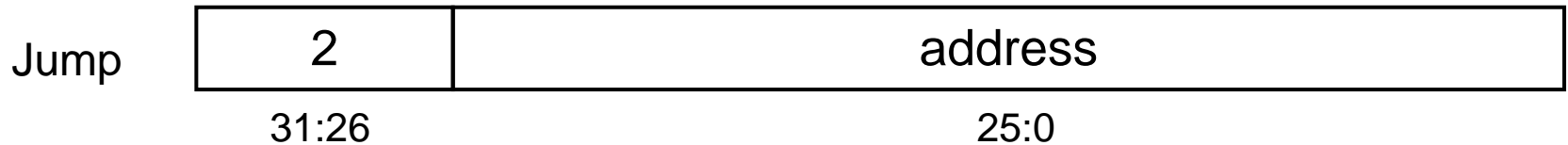
- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

Datapath With B Added
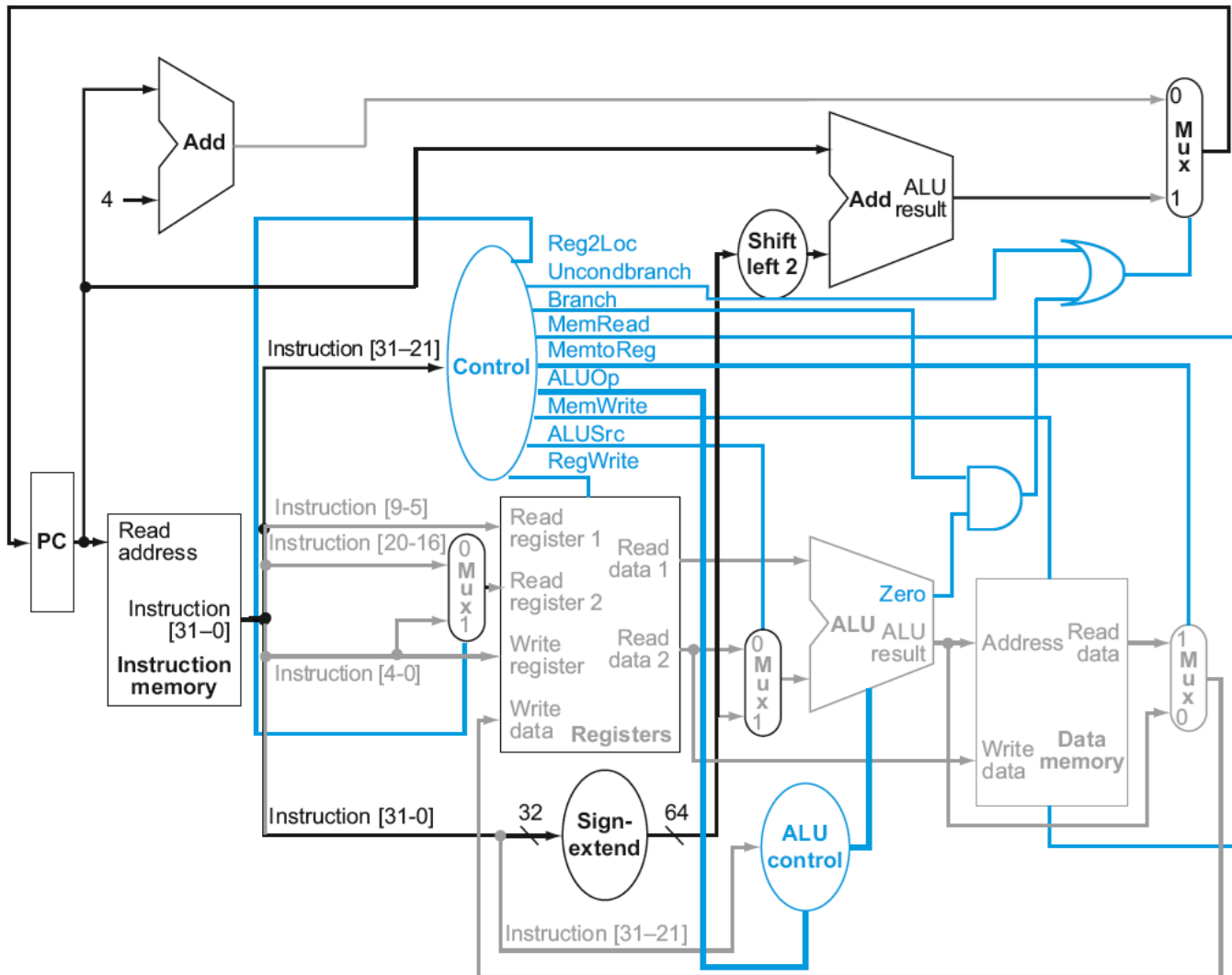
Performance Issues

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

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance
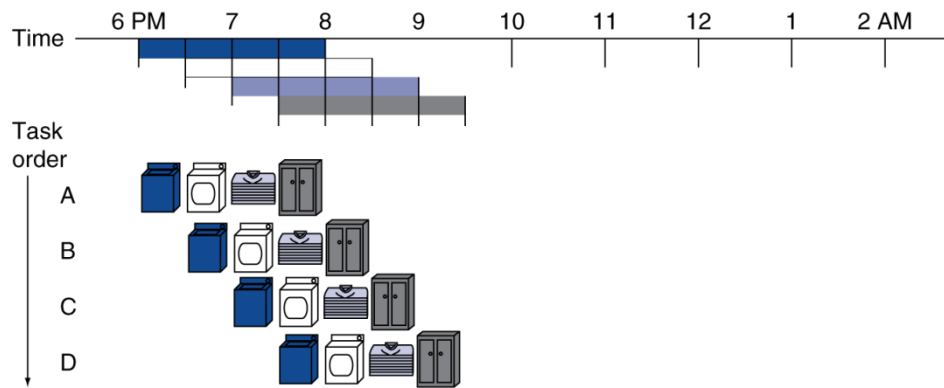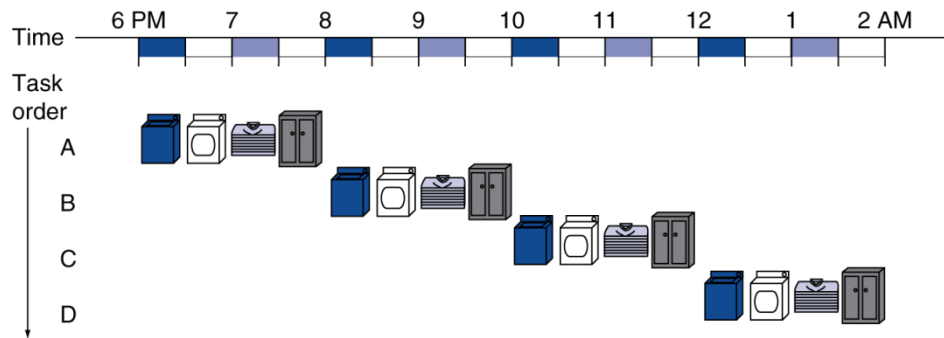
- Four loads:
  - Speedup = 8/3.5 = 2.3

- Non-stop:
  - Speedup = 2n/0.5n + 1.5 ≈ 4 = number of stages

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

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

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
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

Single-cycle (T_c= 800ps)
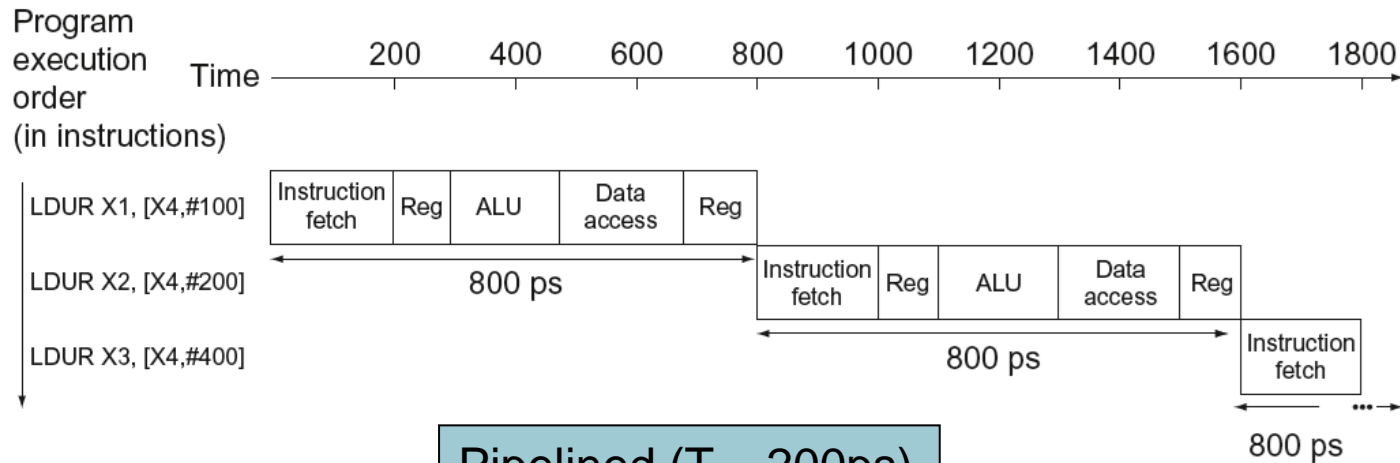
Pipelined (T_c= 200ps)
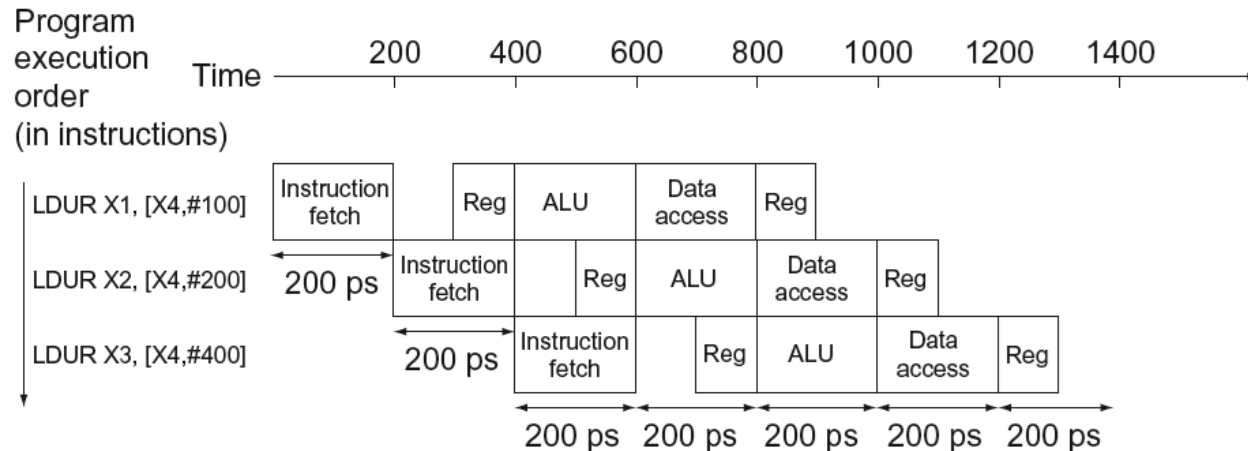
Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$
    $$= \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$$

- If not balanced, speedup is less

- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining and ISA Design

- LEGv8 ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Hazards

- Situations that prevent starting the next instruction in the next cycle

- Structure hazards
  - A required resource is busy

- 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/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
    - ADD   X19, X0, X1
      SUB   X2, X19, X3
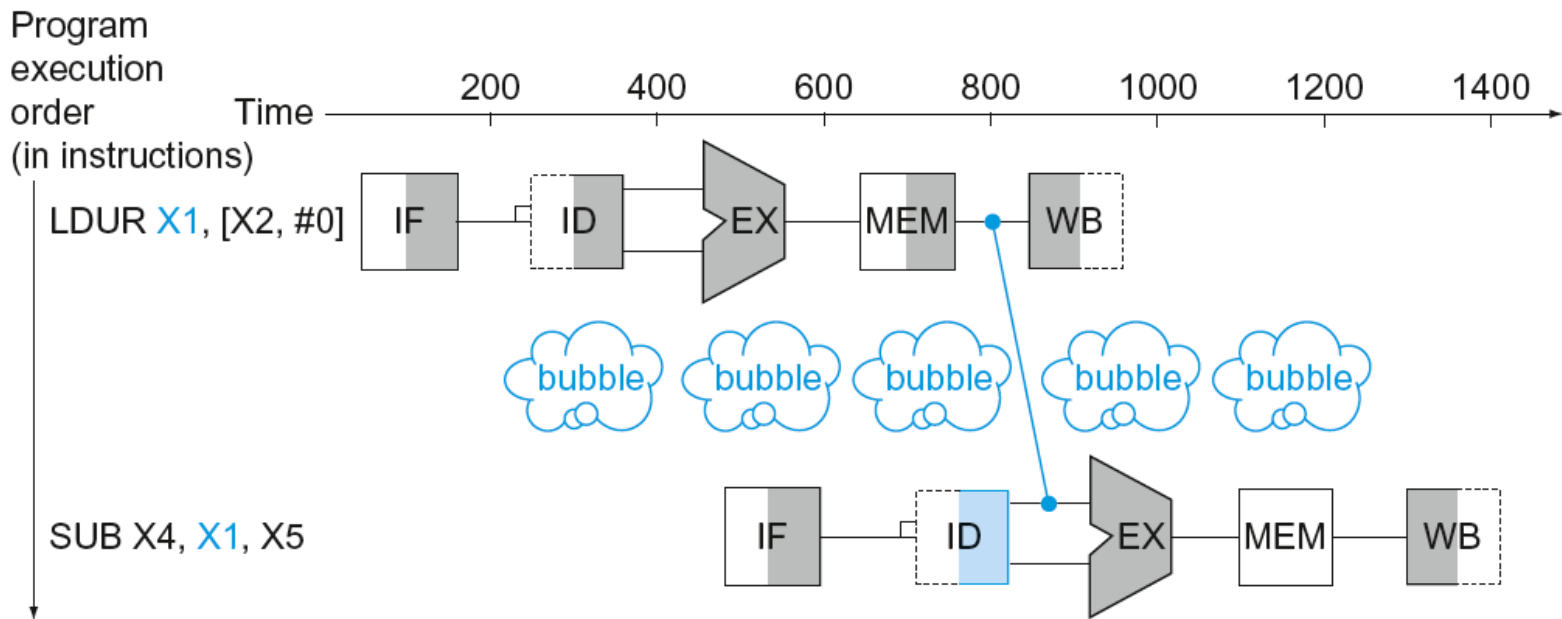
Forwarding (aka Bypassing)

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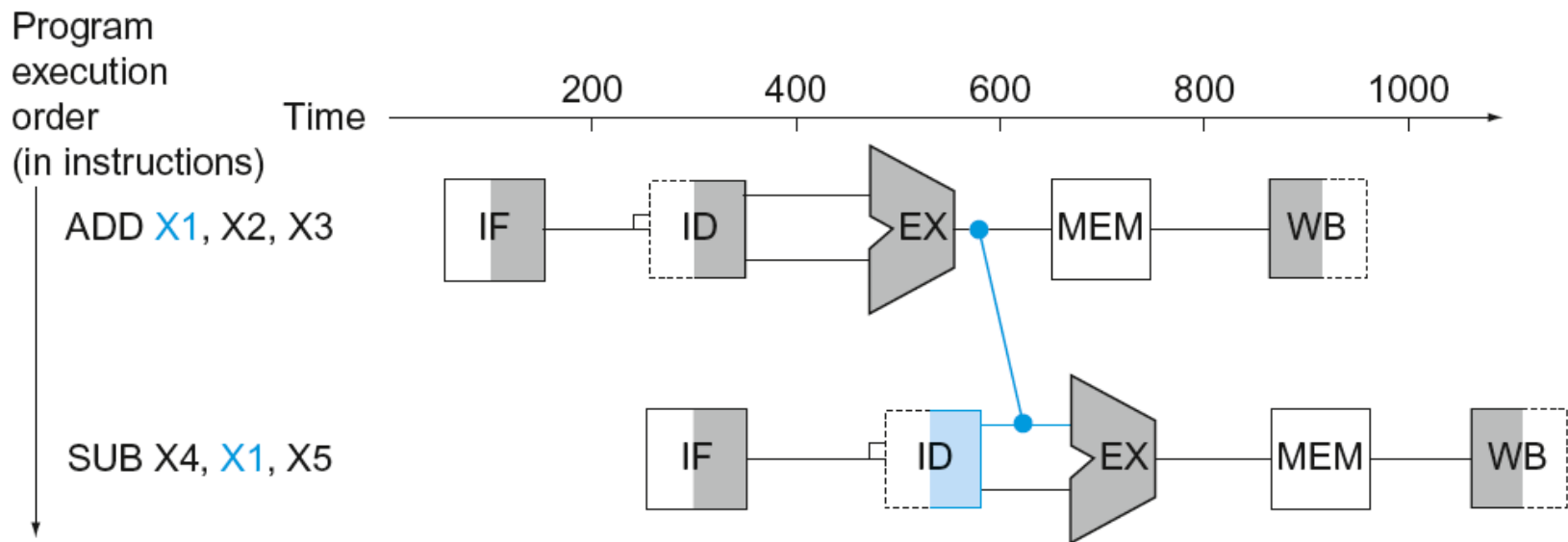
Load-Use Data Hazard

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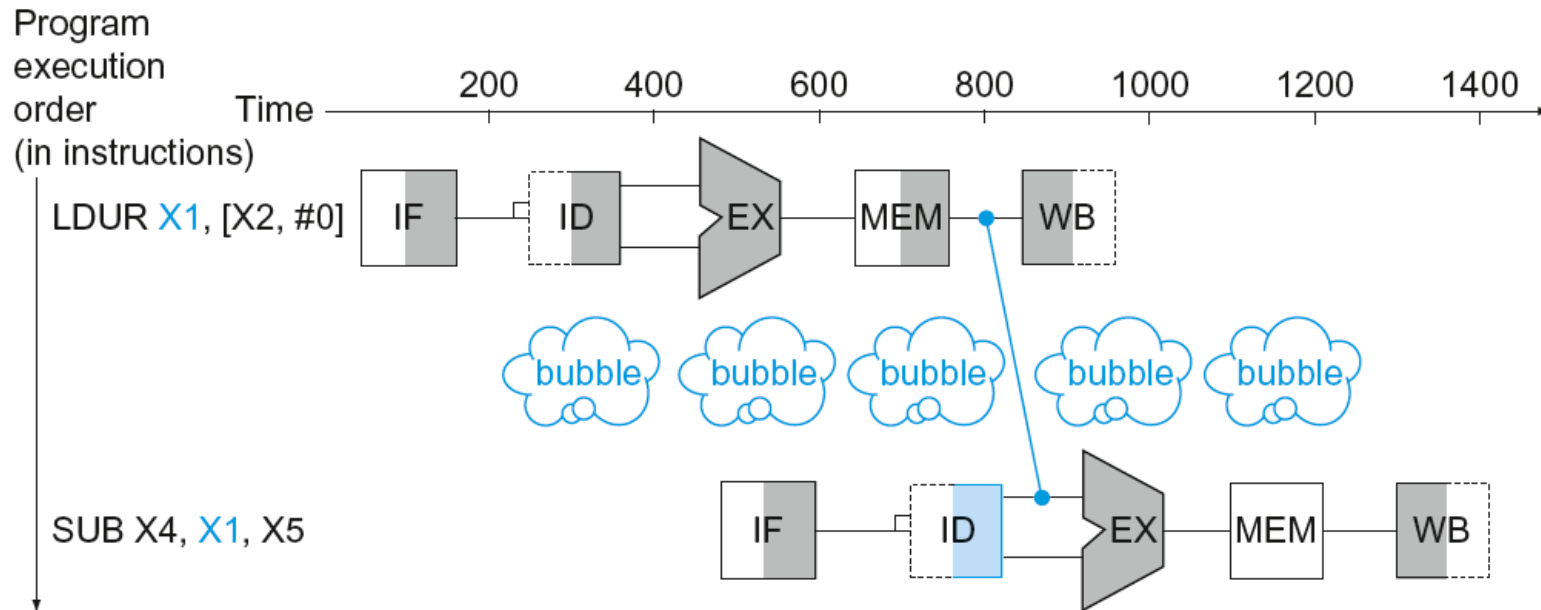
Code Scheduling to Avoid Stalls

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

	LDUR X1, [X0,#0]
	LDUR X2, [X0,#8]
	ADD X3, X1, X2
	STUR X3, [X0,#24]
	LDUR X4, [X0,#16]
	ADD X5, X1, X4
	STUR X5, [X0,#32]
		LDUR X1, [X0,#0]
		LDUR X2, [X0,#8]
		LDUR X4, [X0,#16]
		ADD X3, X1, X2
		STUR X3, [X0,#24]
		ADD X5, X1, X4
		STUR X5, [X0,#32]

stall

stall

13 cycles

11 cycles
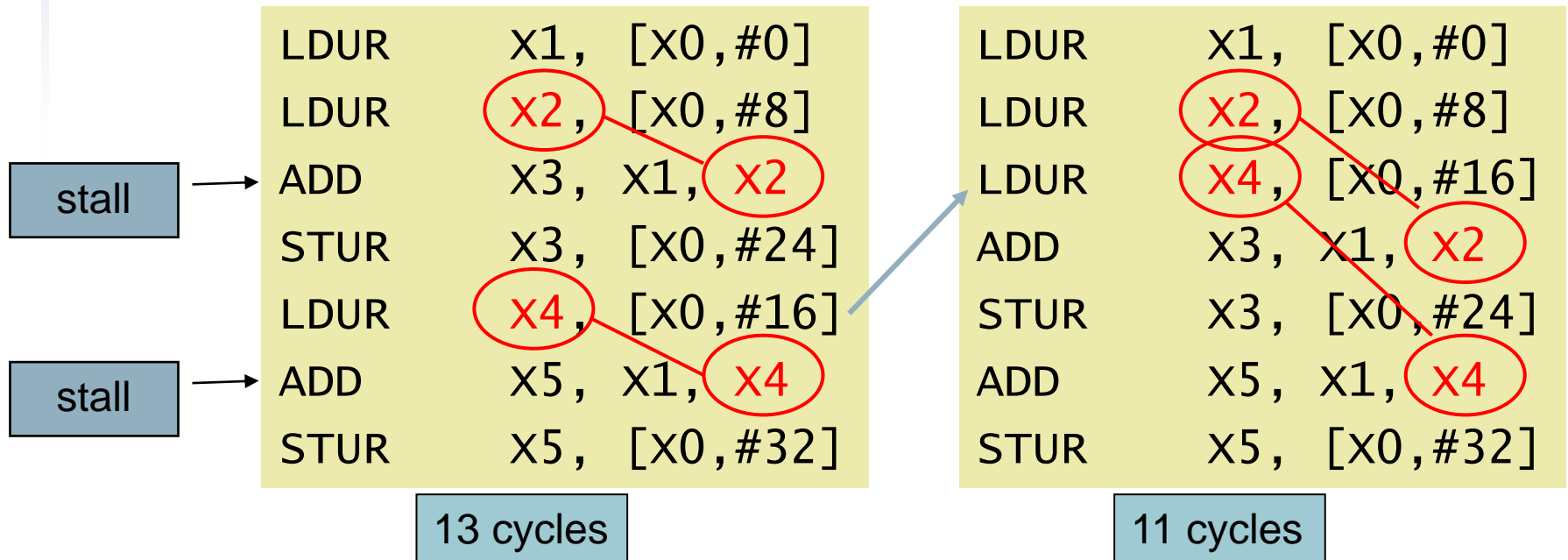
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch

- In LEGv8 pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction
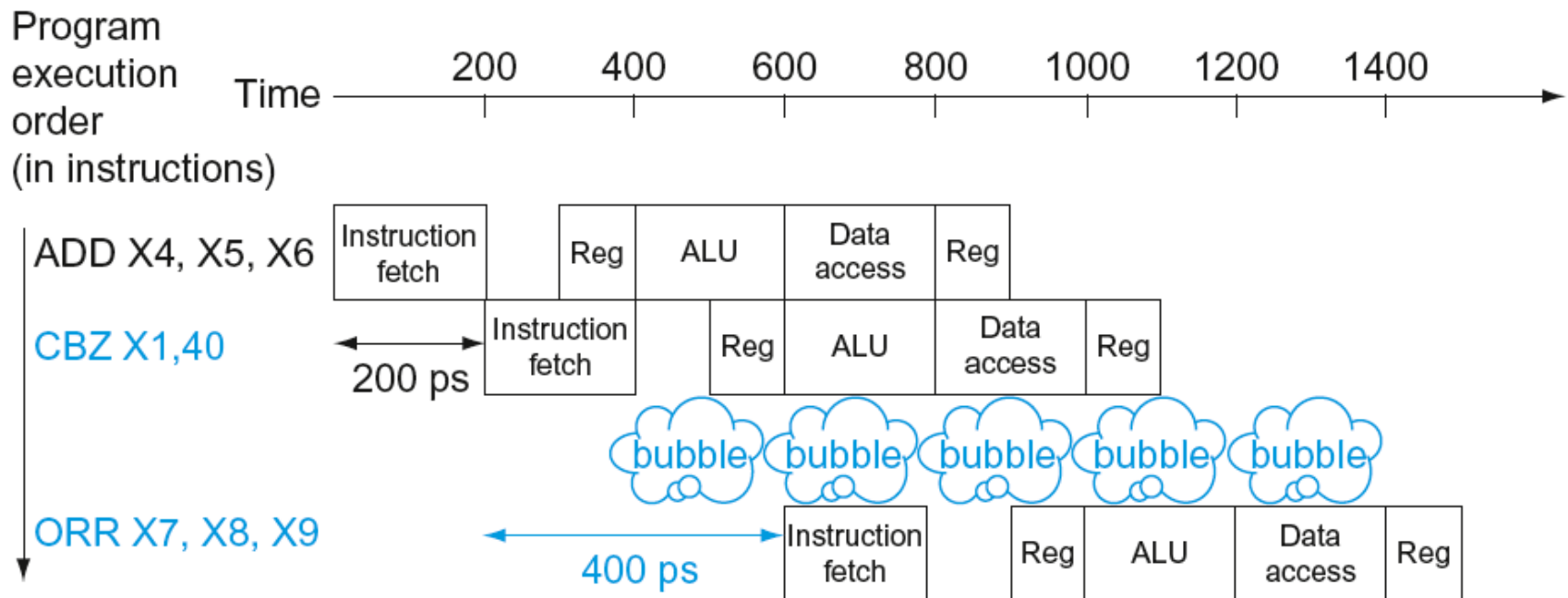
Branch Prediction

- Longer pipelines can't readily determine branch outcome early
    - Stall penalty becomes unacceptable
- Predict outcome of branch
    - Only stall if prediction is wrong
- In LEGv8 pipeline
    - Can predict branches not taken
    - Fetch instruction after branch, with no delay

More-Realistic Branch Prediction

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

Pipeline Summary

The BIG Picture

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

LEGv8 Pipelined Datapath

MEM
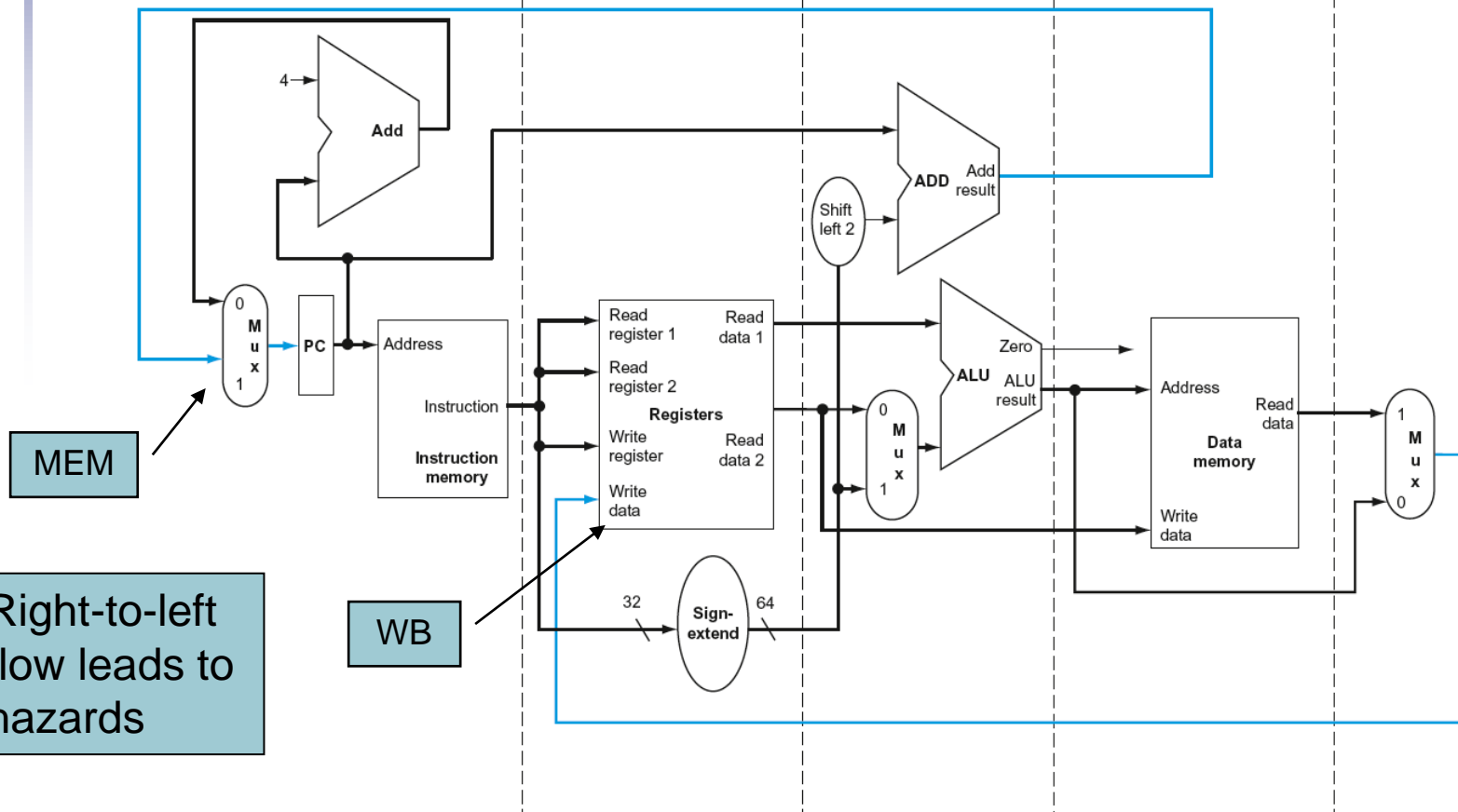
Right-to-left flow leads to hazards

WB

Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle
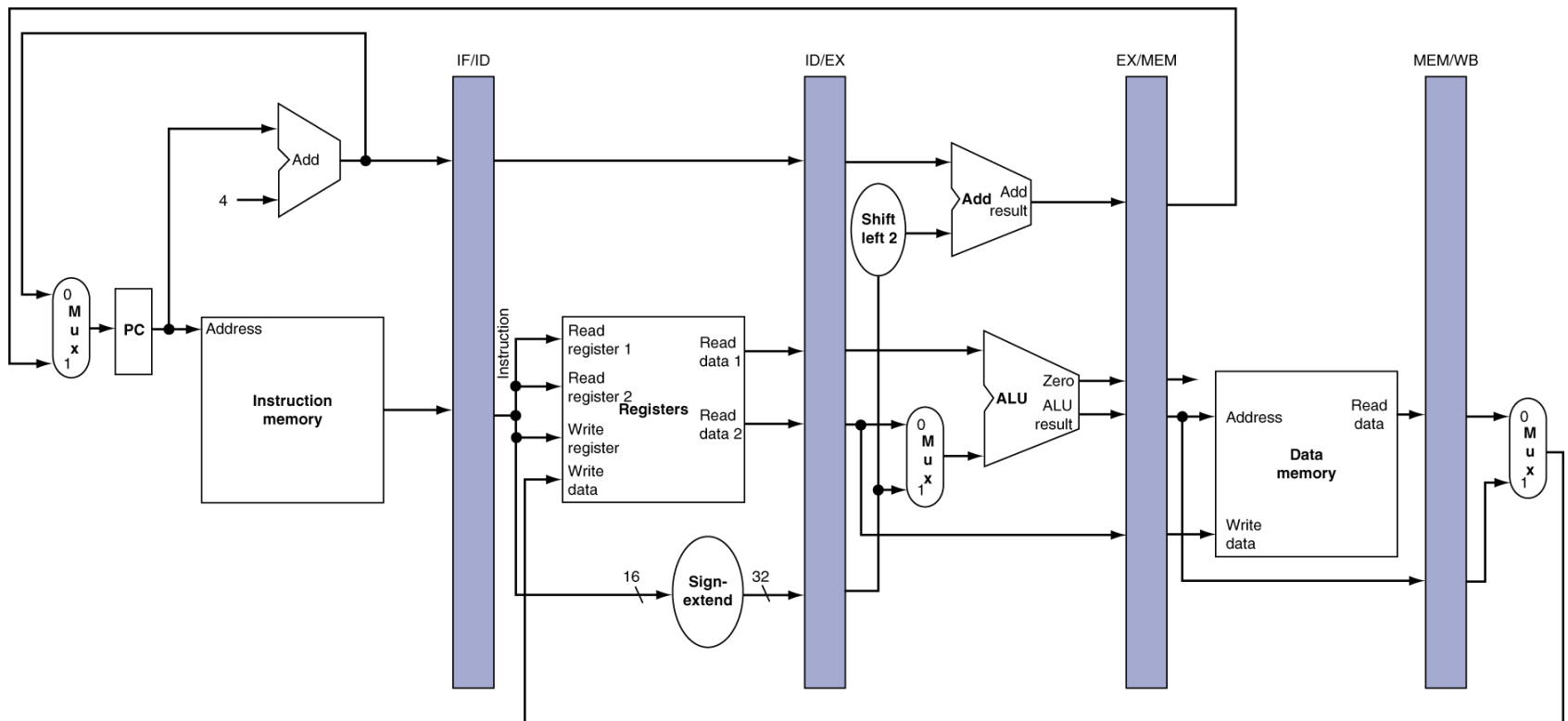
Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel

- To increase ILP

  - Deeper pipeline

    - Less work per stage $\Rightarrow$ shorter clock cycle

  - Multiple issue

    - Replicate pipeline stages $\Rightarrow$ multiple pipelines

    - Start multiple instructions per clock cycle

    - CPI < 1, so use Instructions Per Cycle (IPC)

    - E.g., 4GHz 4-way multiple-issue

      - 16 BIPS, peak CPI = 0.25, peak IPC = 4

    - But dependencies reduce this in practice

Multiple Issue

- ## Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into "issue slots"
  - Compiler detects and avoids hazards

- ## Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Speculation

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include "fix-up" instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
    - e.g., speculative load before null-pointer check
- Static speculation
    - Can add ISA support for deferring exceptions
- Dynamic speculation
    - Can buffer exceptions until instruction completion (which may not occur)

Static Multiple Issue

- Compiler groups instructions into "issue packets"
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - $\Rightarrow$ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
    - Reorder instructions into issue packets
    - No dependencies with a packet
    - Possibly some dependencies between packets
        - Varies between ISAs; compiler must know!
    - Pad with nop if necessary

LEGv8 with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

LEGv8 with Static Dual Issue
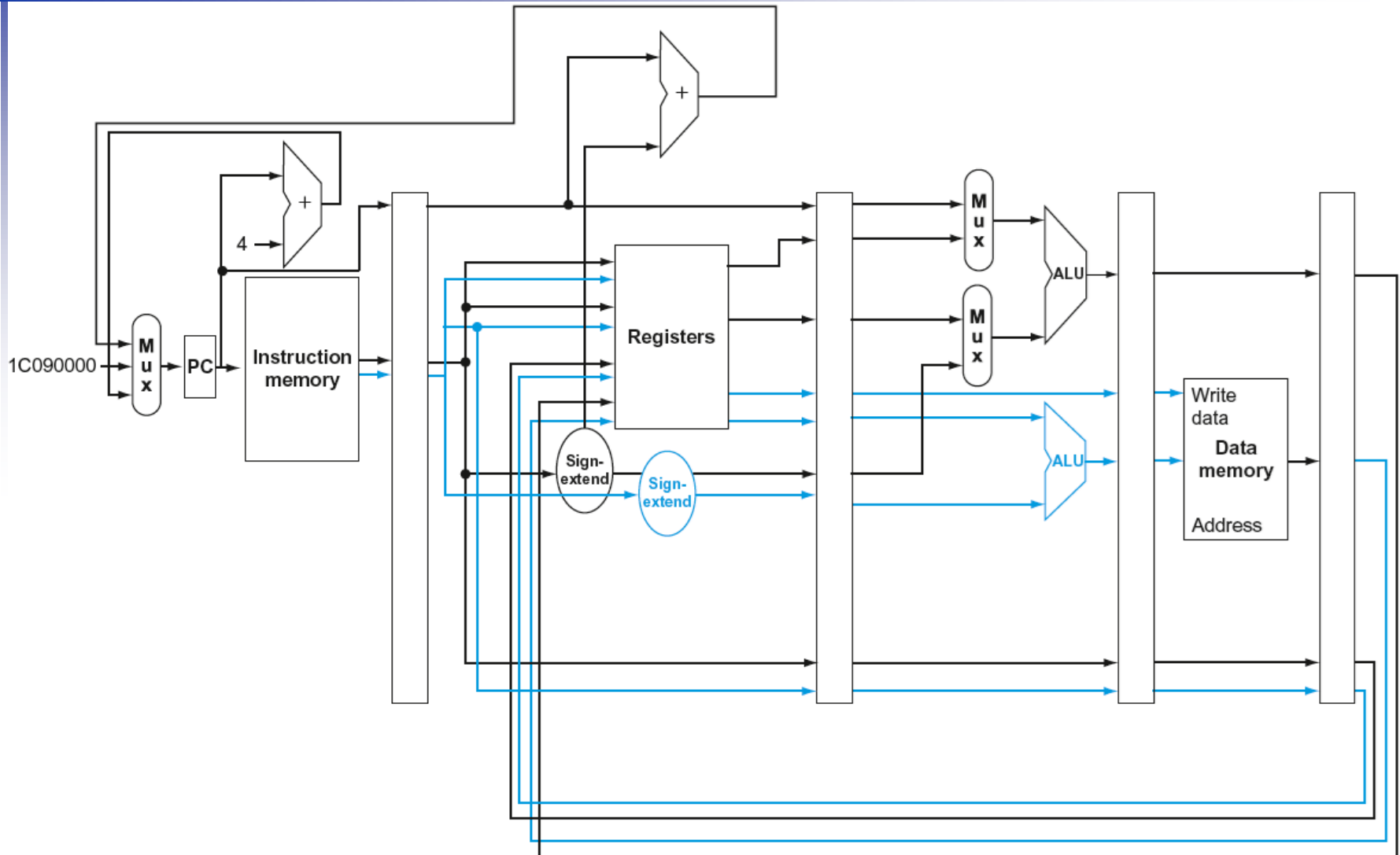
Hazards in the Dual-Issue LEGv8

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - ADD X0, X0, X1
      LDUR X2, [X0,#0]
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Scheduling Example

- ## Schedule this for dual-issue LEGv8

```
Loop: LDUR X0, [X20,#0]      // X0=array element
      ADD  X0, X0,X21        // add scalar in X21
      STUR X0, [X20,#0]      // store result
      SUBI X20, X20,#4       // decrement pointer
      CMP  X20, X22          // branch $s1!=0
      BGT Loop
```

	ALU/branch	Load/store	cycle
Loop:	nop	LDUR X0, [X20,#0]	1
	SUBI X20, X20,#4	nop	2
	ADD X0, X0,X21	nop	3
	CMP X20, X22		4
	BGT Loop	STUR X0, [X20,#0]	5

- IPC = 6/5 = 1.2 (c.f. peak IPC = 2)

Loop Unrolling

- ## Replicate loop body to expose more parallelism

  - ### Reduces loop-control overhead

- ## Use different registers per replication

  - ### Called "register renaming"

  - ### Avoid loop-carried "anti-dependencies"

    - Store followed by a load of the same register

    - Aka "name dependence"

      - Reuse of a register name

Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	SUBI X20, X20,#32	LDUR X0, [X20,#0]	1
	nop	LDUR X1, [X20,#24]	2
	ADD X0, X0, X21	LDUR X2, [X20,#16]	3
	ADD X1, X1, X21	LDUR X3, [X20,#8]	4
	ADD X2, X2, X21	STUR X0, [X20,#32]	5
	ADD X3, X3, X21	STUR X1, [X20,#24]	6
	CMP X20,X22	STUR X2, [X20,#16]	7
	BGT Loop	STUR X3, [X20,#8]	8

- ## IPC = 15/8 = 1.875
  - Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
    - But commit result to registers in order
- Example

    ```
    LDUR X0, [X21,#20]
    ADD  X1, X0, X2
    SUB  X23,X23,X3
    ANDI X5, X23,#20
    ```

    - Can start SUB while ADD is waiting for LDUR

Dynamically Scheduled CPU
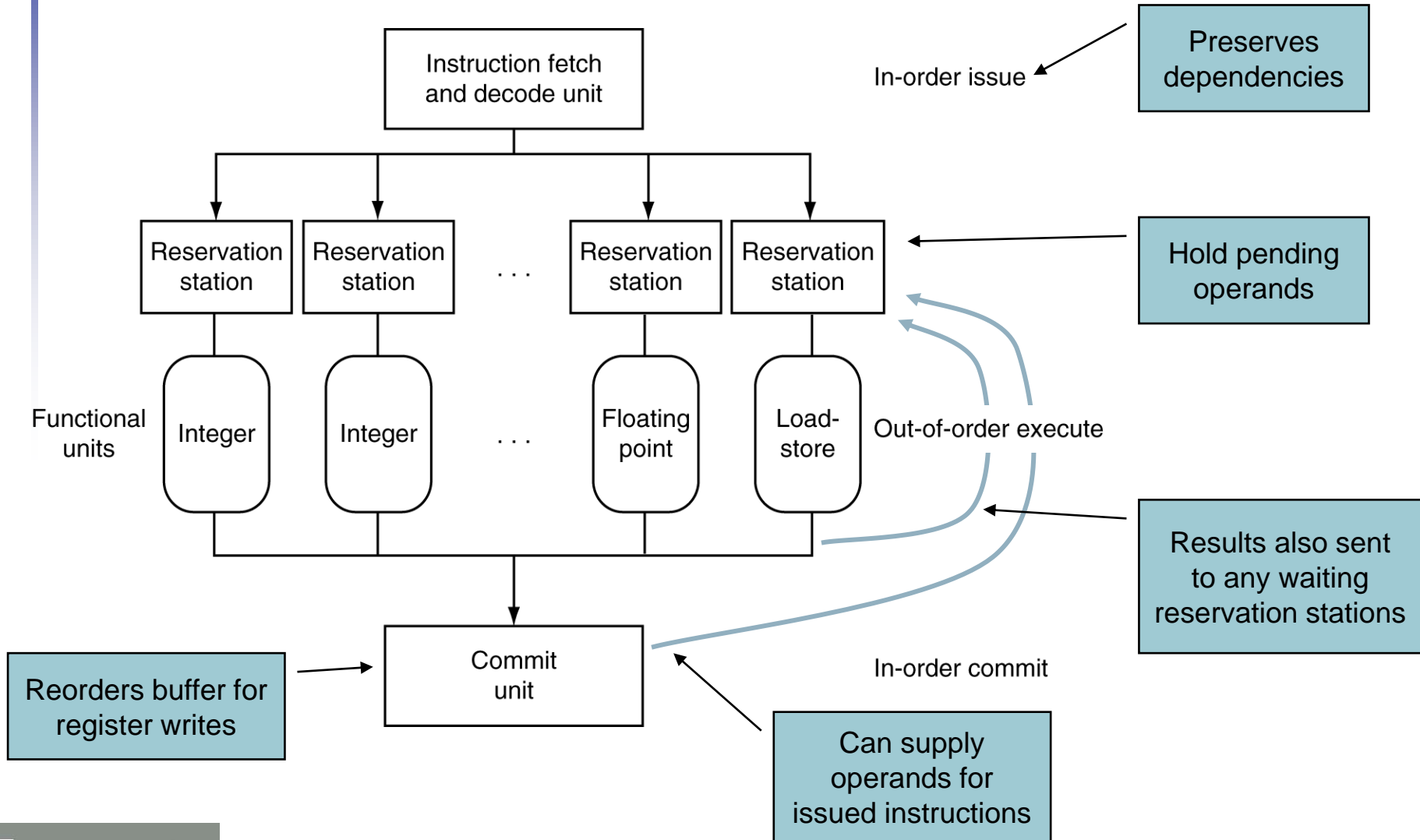
Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming

- On instruction issue to reservation station

  - If operand is available in register file or reorder buffer

    - Copied to reservation station

    - No longer required in the register; can be overwritten

  - If operand is not yet available

    - It will be provided to the reservation station by a function unit

    - Register update may not be required

Speculation

- Predict branch and continue issuing
  - Don't commit until branch outcome determined
- Load speculation
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - Don't commit load until speculation cleared

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?

- Not all stalls are predicable
  - e.g., cache misses

- Can't always schedule around branches
  - Branch outcome is dynamically determined

- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like

- Programs have real dependencies that limit ILP

- Some dependencies are hard to eliminate
  - e.g., pointer aliasing

- Some parallelism is hard to expose
  - Limited window size during instruction issue

- Memory delays and limited bandwidth
  - Hard to keep pipelines full

- Speculation can help if done well

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power

- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Intel Core	2006	2930MHz	14	4	Yes	2	75W
Intel Core i5 N	2010	3300MHz	14	4	Yes	2-4	87W
Intel Core i5 Ivy	2012	3400MHz	14	4	Yes	8	77W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

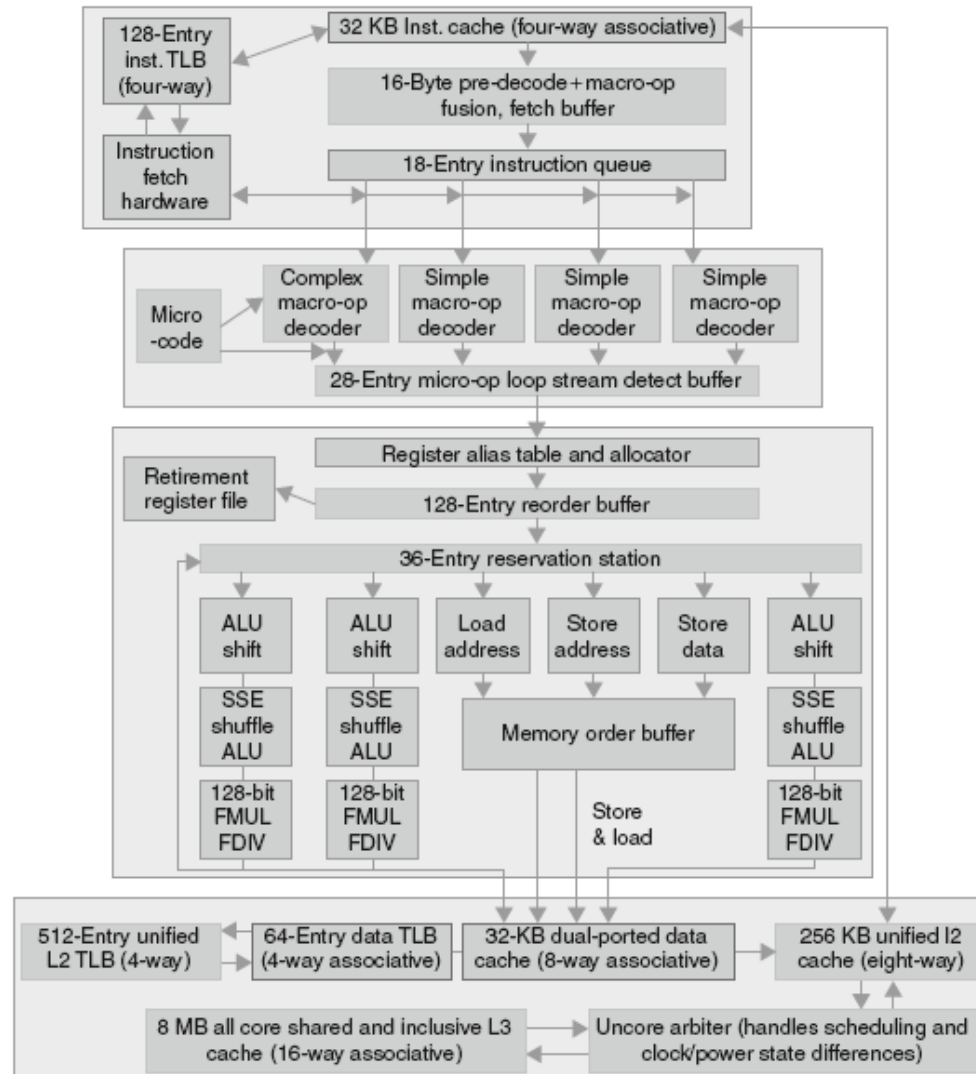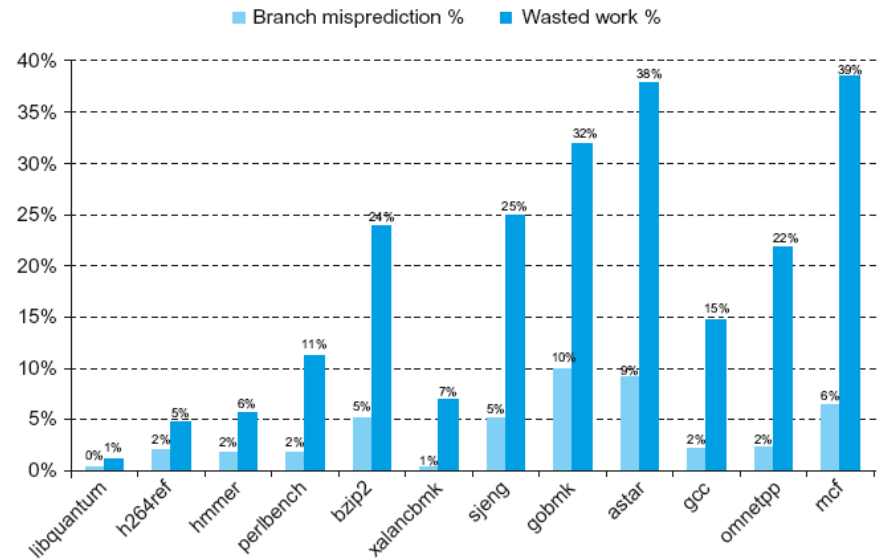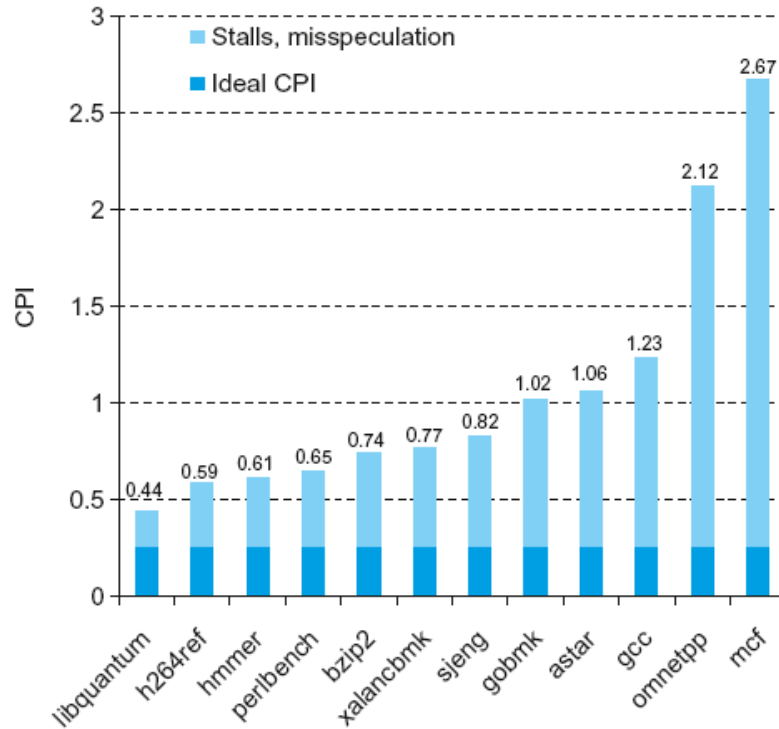# ARM Cortex-A53 Pipeline
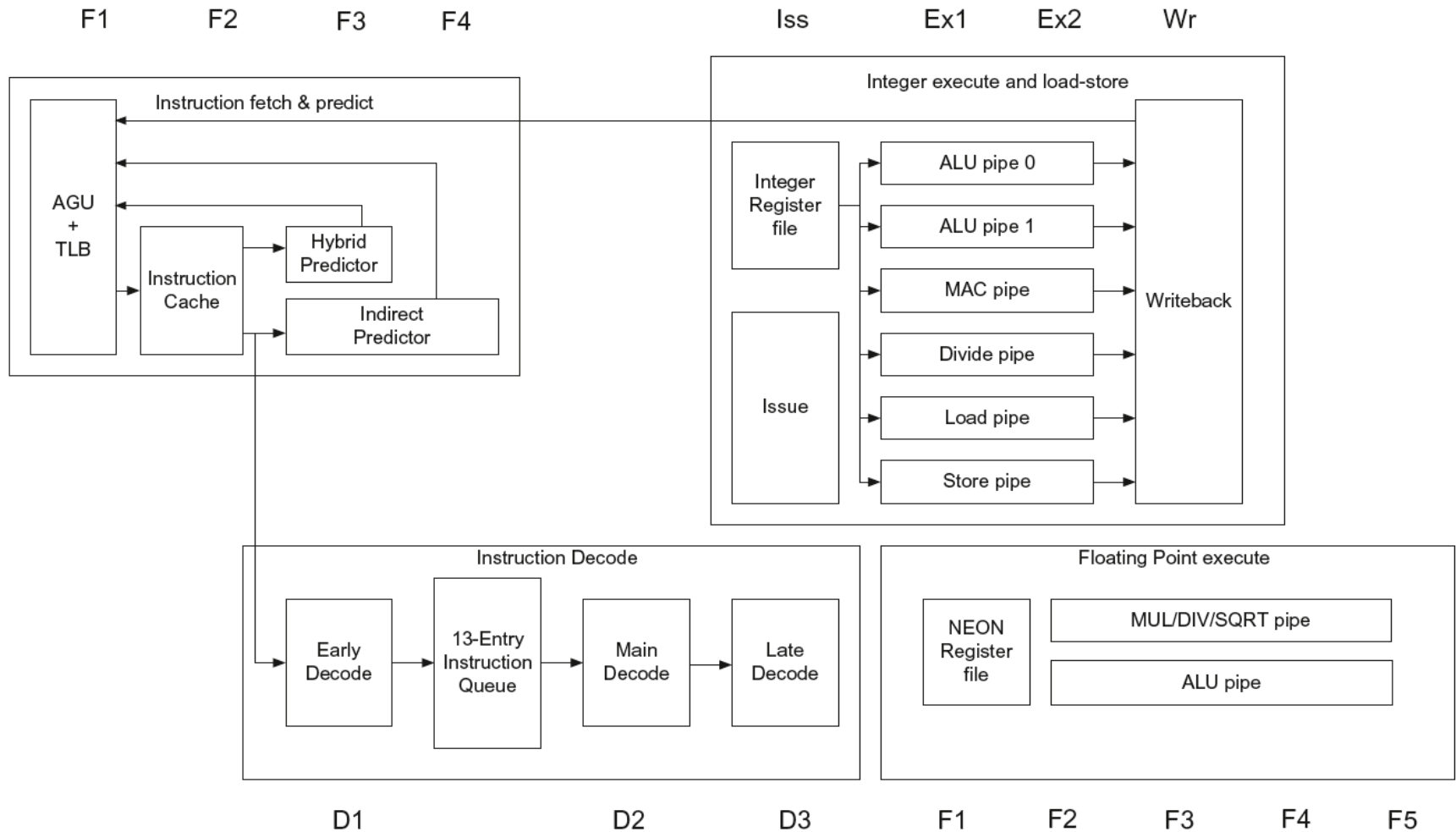
ARM Cortex-A53 Performance
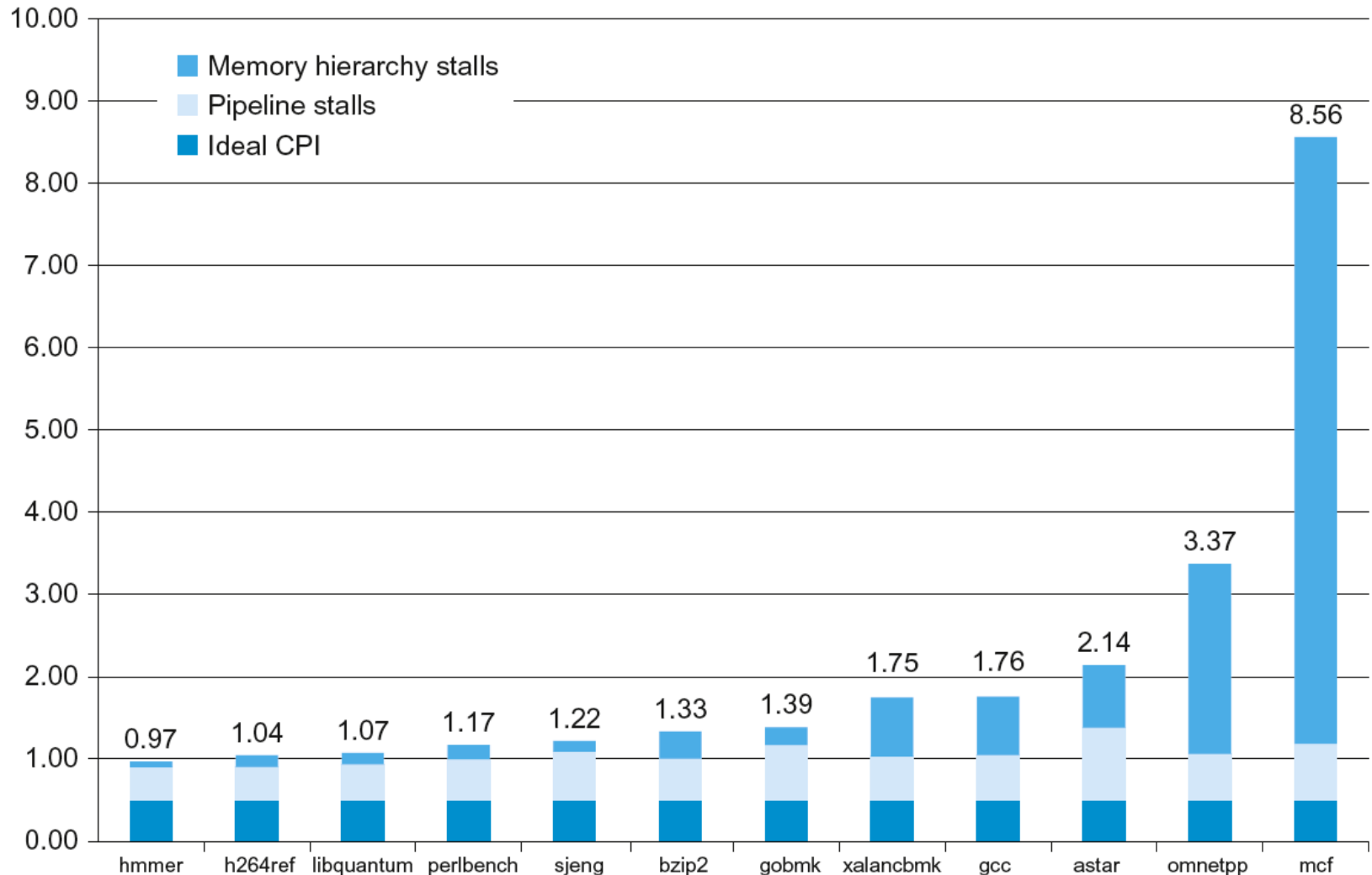
Core i7 Pipeline

Core i7 Performance

Matrix Multiply

- ## Unrolled C code

```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6  for ( int i = 0; i < n; i+=UNROLL*4 )
7   for ( int j = 0; j < n; j++ ) {
8    __m256d c[4];
9    for ( int x = 0; x < UNROLL; x++ )
10    c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12   for( int k = 0; k < n; k++ )
13   {
14    __m256d b = _mm256_broadcast_sd(B+k+j*n);
15    for (int x = 0; x < UNROLL; x++)
16    c[x] = _mm256_add_pd(c[x],
17                        _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18   }
19
20   for ( int x = 0; x < UNROLL; x++ )
21    _mm256_store_pd(C+i+x*4+j*n, c[x]);
22  }
23 }
```

Matrix Multiply

- ## Assembly code:

```
1 vmovapd (%r11),%ymm4                     # Load 4 elements of C into %ymm4
2 mov %rbx,%rax                            # register %rax = %rbx
3 xor %ecx,%ecx                            # register %ecx = 0
4 vmovapd 0x20(%r11),%ymm3                 # Load 4 elements of C into %ymm3
5 vmovapd 0x40(%r11),%ymm2                 # Load 4 elements of C into %ymm2
6 vmovapd 0x60(%r11),%ymm1                 # Load 4 elements of C into %ymm1
7 vbroadcastsd (%rcx,%r9,1),%ymm0          # Make 4 copies of B element
8 add $0x8,%rcx # register %rcx = %rcx + 8
9 vmulpd (%rax),%ymm0,%ymm5                # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4                # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5           # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3                # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5           # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0           # Parallel mul %ymm1,4 A elements
15 add %r8,%rax                            # register %rax = %rax + %r8
16 cmp %r10,%rcx                           # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2                # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1                # Parallel add %ymm0, %ymm1
19 jne 68 <dgemm+0x68>                     # jump if not %r8 != %rax
20 add $0x1,%esi                           # register % esi = % esi + 1
21 vmovapd %ymm4,(%r11)                    # Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)                # Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)                # Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)                # Store %ymm1 into 4 C elements
```

Performance Impact
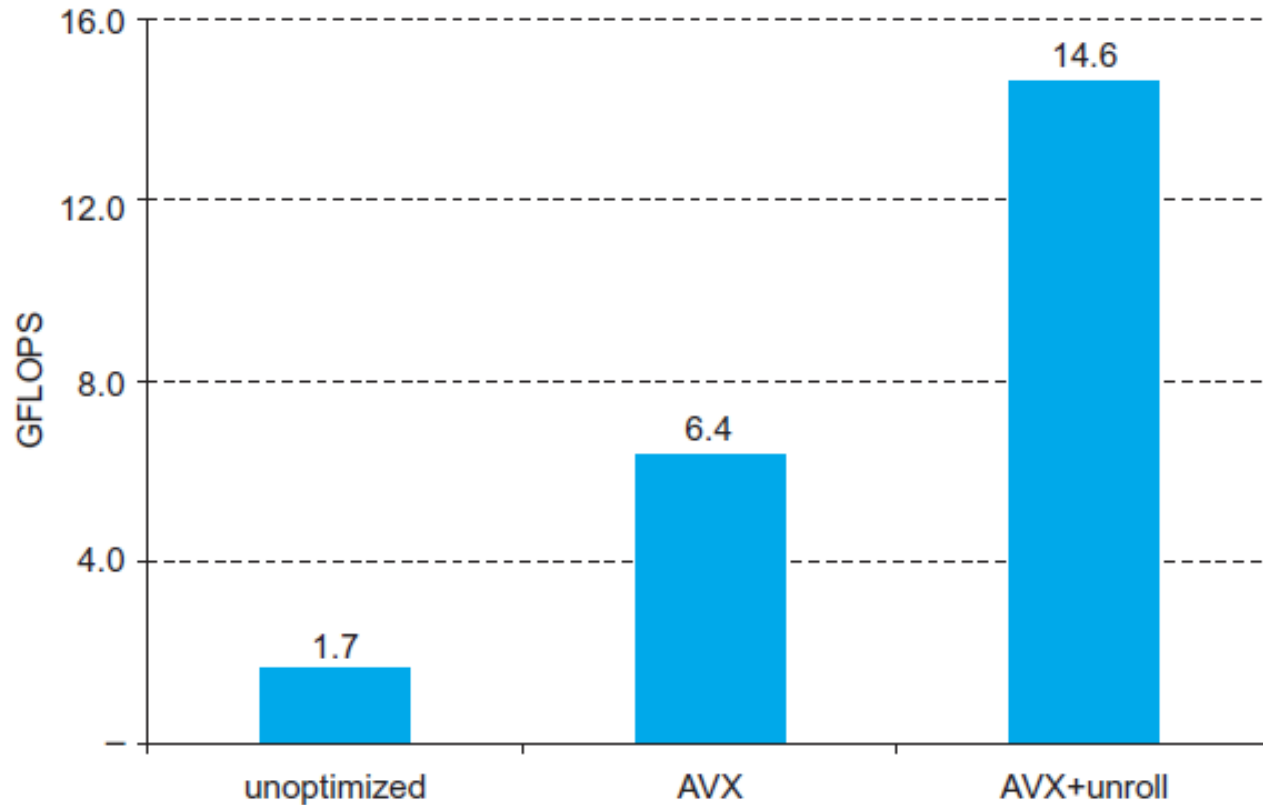
Fallacies

- 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
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall