Optional Lecture 1

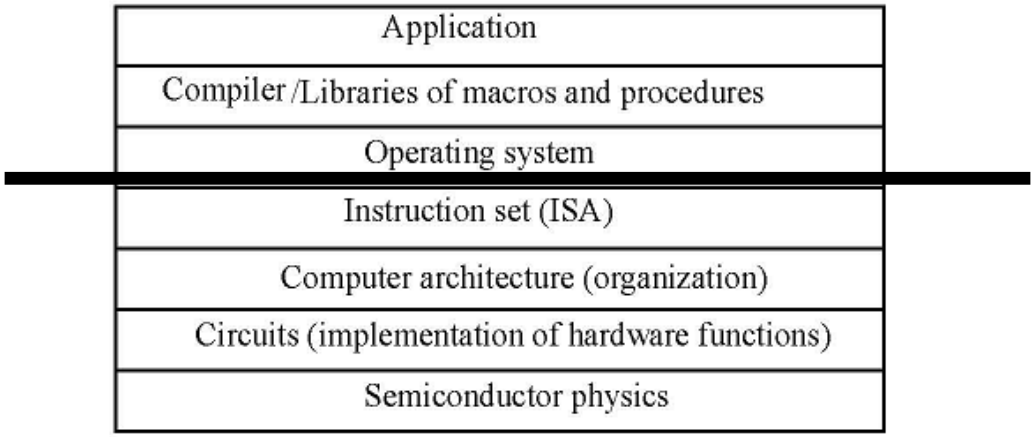
Basic Microarchitecture Concepts

- Instruction set architecture (ISA) (Ch 3.2)
- Statically scheduled pipelines (Ch 3.3.1)

Instruction Set Architecture (ISA) (Ch 3.2)

Instruction Set Architecture

The ISA is the interface between software and hardware



Design objectives

- Functionality and flexibility for OS and compilers
- Implementation efficiency in available technology
- Backward compatibility
- ISAs are typically designed to last through trends of changes in usage and technology
- Tend to grow over time

Instruction Set Architectures (ISA)

Agenda

- Instruction types and opcode
- Instruction operands
- Exceptions
- RISC vs CISC
- Microcoded implementations
- The ISA used in the course

Instruction Types and Operands

Instruction Types and Opcodes 1(2)

The opcode of an instruction specifies the operation to perform

Four classes of instructions are considered:

1. **Integer Arithmetic/Logic instructions**
   - •ADD, SUB, MULT
   - •ADDU, SUBU,MULTU
   - •OR, AND, NOR, NAND
2. **Floating-point instructions**
   - •FADD, FMUL, FDIV
   - •COMPLEX ARITHMETIC
3. **Memory transfer instructions**
   - •LOADS AND STORES
   - •TEST AND SET, AND SWAP

Instruction Types and Opcodes 2(2)

4. **Control instructions**
    - Branches are conditional
        - Condition may be condition bits (ZCVXN)
        - Condition may test the value of a register (set by SLT instruction)
        - Condition may be computed in the branch instruction itself
    - Jumps are unconditional with absolute address or address in register
    - JAL (JUMP AND LINK) needed for procedures

CPU Operands

Include accumulators, evaluation stacks, registers and immediate values

Accumulators

- ADDA <mem_address>
- MOVA <mem_address>

Stack

- PUSH <mem_address>
- ADD
- POP <mem_address>

Registers

- LW R1, <memory-address>
- SW R1, <memory_address>
- ADD R2, <memory_address>
- ADD R1,R2,R4

Load/Store ISAs

- Management by the compiler: register spill/fill

Immediate

- ADDI R1,R2,#5

Example

How is B:=C+D coded in different register models?

Accumulator model:

CLRA

ADDA C

ADDA D

MOVA B

Stack model:

PUSH C

PUSH D

ADD

POP B

Register model (w/ memory operands):

LW R1,C

ADD R1, D

SW R1, B

Register model: (w/ reg operands)

LW R1,C

LW R2,D

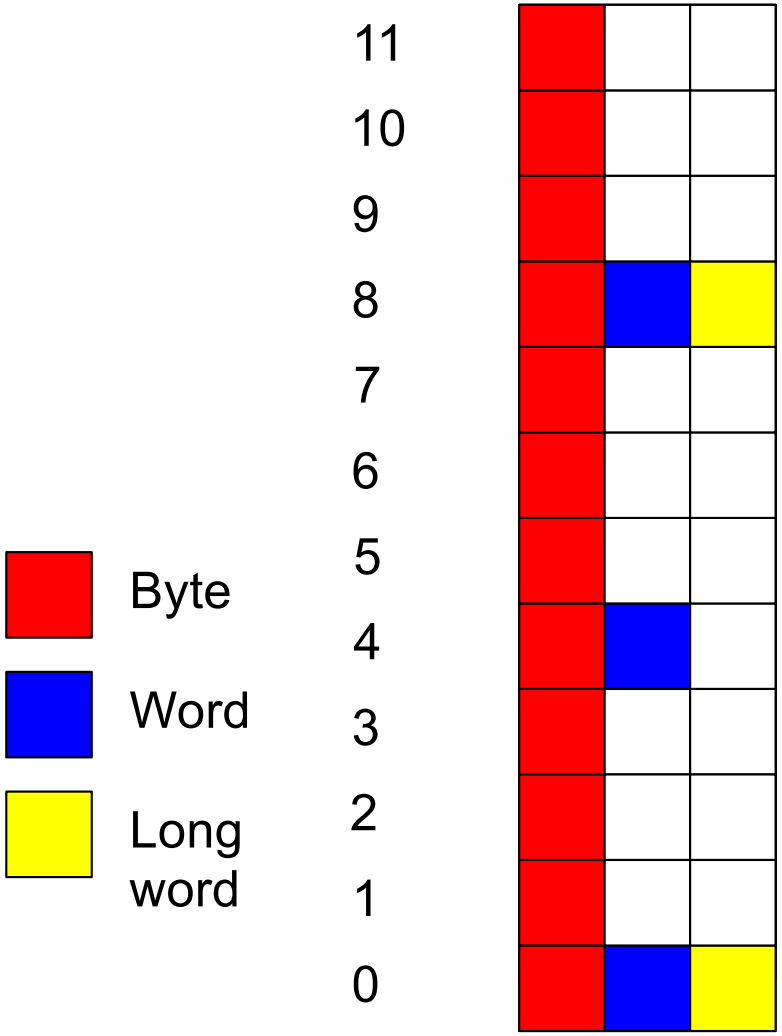
ADD R3,R1,R2

SW R3, B

Memory Operands 1(2)

Operand alignment

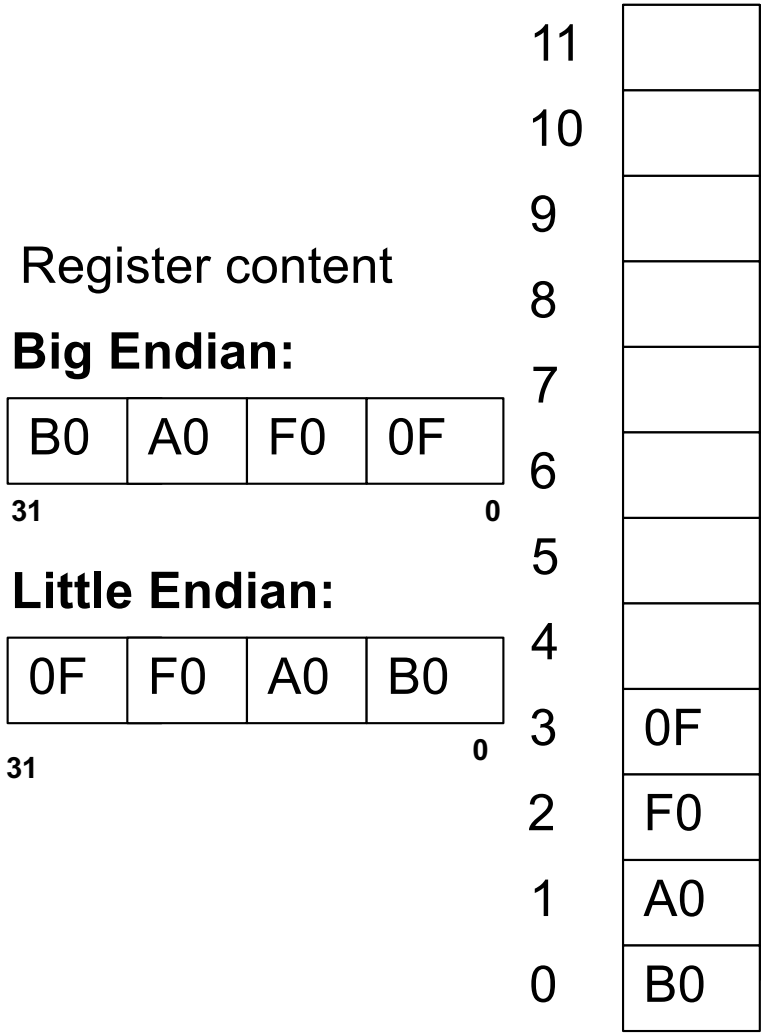
- Byte addressable machines
- Operands of size S must be stored at an address that is a multiple of S
- Bytes are always aligned; words (32 bits) are aligned at 0,4,8…
- Compiler is responsible for aligning operands. Hardware checks for traps if misaligned
- Opcode indicates size (also tags in memory)



Memory Operands 2(2)

Little vs. big endian

- **Big endian:** MSB is stored at address XXXXXX00
- **Little endian:** LSB is stored at address XXXXXX00
- Portability problems, configurable endianness



Addressing Modes

Addressing Modes

MODE	EXAMPLE	MEANING
REGISTER	ADD R4,R3	reg[R4] <- reg[R4] + reg[R3]
DISPLACEMENT	ADD R4, 100(R1)	reg[R4] <- reg[R4] + Mem[100 + reg[R1]]
REGISTER INDIRECT	ADD R4, (R1)	reg[R4] <- reg[R4] + Mem[reg[R1]]
INDEXED	ADD R3, (R1+R2)	reg[R3] <- reg[R3] + Mem[reg[R1] + reg[R2]]
DIRECT OR ABSOLUTE	ADD R1, (1001)	reg[R1] <- reg[R1] + Mem[1001]
MEMORY INDIRECT	ADD R1, @R3	reg[R1] <- reg[R1] + Mem[Mem[Reg[3]]]
POST INCREMENT	ADD R1, (R2)+	ADD R1, (R2) then R2 <- R2+d
PREDECREMENT	ADD R1, -(R2)	R2 <- R2-d then ADD R1, (R2)
PC-RELATIVE	BEZ R1, 100	if R1==0, PC <- PC+100
PC-RELATIVE	JUMP 200	Concatenate bits of PC and offset

Actual Use of Addressing Modes

Optimize the common case

Displacement and immediate are the most common modes

- **16 bits is usually enough for both types of values**

Several addressing modes are special cases of displacement and immediate

- **Register indirect and memory absolute**

More complex addressing modes can be synthesized

- **Memory indirect: LW R1, @(R2)**

LW R3, 0(R2)

LW R1, 0(R3)

- **Postincrement: LW R1, (R2)++**

LW R1, 0(R2)

ADDI R2, R2, #size

Number of Memory Operands in ALU OPs

Consider HLL statement C := A+B

With 3 memory operands (memory-to-memory instruction)

ADD C,A,B

- **Long instruction**
- **No memory-operand re-use**

With 1 memory operand and 1 register

LW R1,A

ADD R1,B

SW R1,C

With no memory operand (Load/Store architecture)

LW R1,A

LW R2,B

ADD R3,R1,R2

SW R3,C

Exceptions

Exceptions, Traps and Interrupts

Exceptions are rare events triggered by the hardware and forcing the processor to execute a handler

- Includes traps and interrupts

Examples:

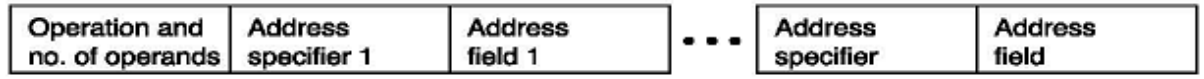
- I/O device interrupts
- Operating system calls
- Instruction tracing and breakpointing
- Integer and floating-point arithmetic exceptions
- Misaligned memory accesses
- Memory protection violations
- Undefined instruction execution
- Hardware failure/alarm
- Power failures
- Page faults

Precise Exceptions

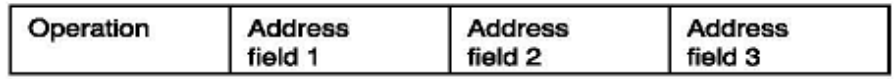
- Synchronized with an instruction
- Must resume execution after handler
- Save the process state at the faulting instruction
- Often difficult in architectures where multiple instructions execute

Encoding the ISA

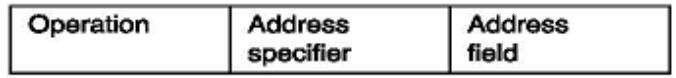
Theoretically any encoding will do. However, watch out for code size and decoding complexity. Decoding is simplified if instruction format is highly predictable

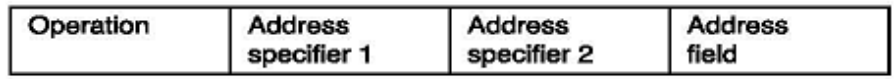


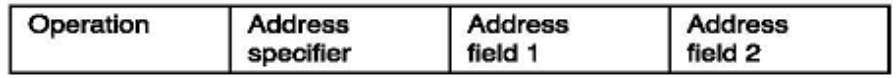
(a) Variable (e.g., VAX, Intel 80x86)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)







(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

CISC vs RISC (Complex vs Reduced Instruction Set Computers)

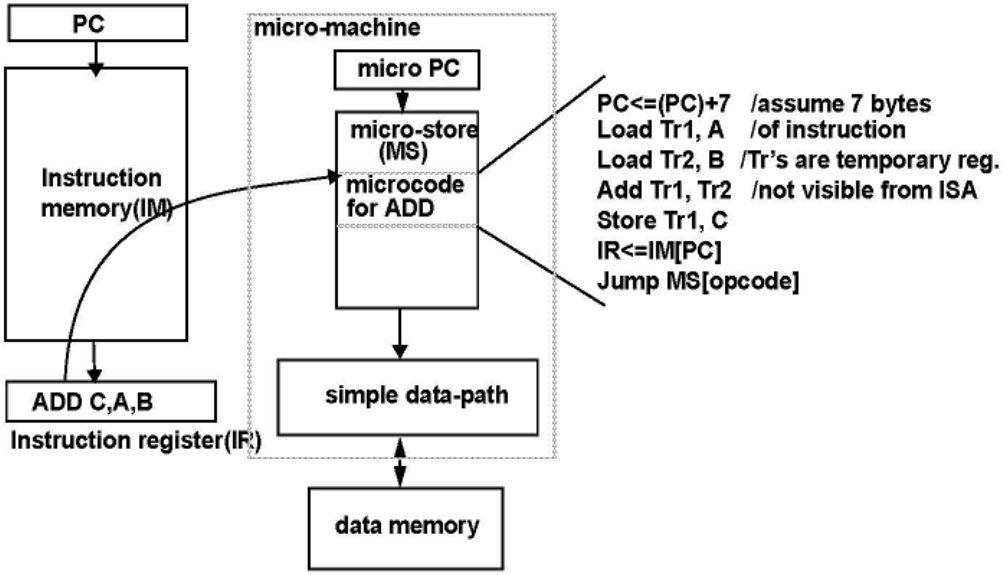
CISC vs RISC

Complex vs reduced (i.e., simple) instruction set computers

Not directly related to implementation

- Complex RISC implementations
- Simple CISC implementations

Simple implementation of CISC machines use *microcode*



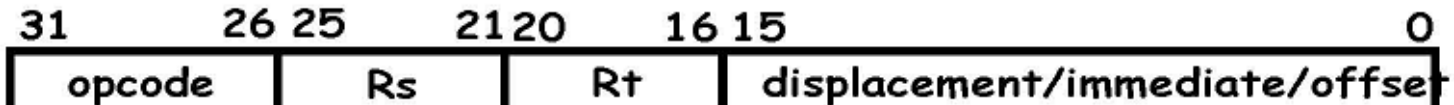
- **Translation overhead from instruction to micro instruction**

ISA used in the Course

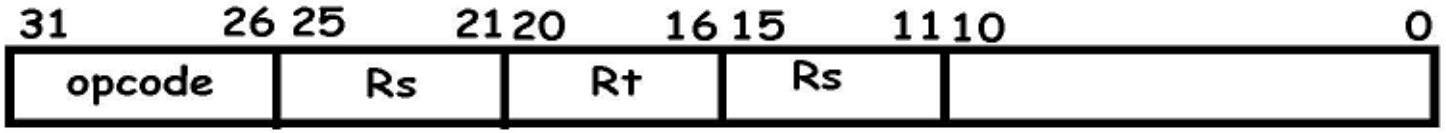
Types	Opcode	Assembly code	Meaning	Comments
Data Transfers	LB, LH, LW, LD	LW R1,#20(R2)	R1<=MEM[(R2)+20]	for bytes, half-words
	SB, SH, SW, SD	SW R1,#20(R2)	MEM[(R2)+20]<=(R1)	words, and double words
	L.S, L.D	L.S F0,#20(R2)	F0<=MEM[(R2)+20]	single/double float load
	S.S, S.D	S.S F0,#20(R2)	MEM[(R2)+20]<=(F0)	single/double float store
ALU operations	ADD, SUB, ADDU, SUBU	ADD R1,R2,R3	R1<=(R2)+(R3)	add/sub signed or unsigned
	ADDI, SUBI, ADDIU, SUBIU	ADDI R1,R2,#3	R1<=(R2)+3	add/sub immediate signed or unsigned
	AND, OR, XOR,	AND R1,R2,R3	R1<=(R2).AND.(R3)	bitwise logical AND, OR, XOR
	ANDI, ORI, XORI,	ANDI R1,R2,#4	R1<=(R2).ANDI.4	bitwise AND, OR, XOR immediate
	SLT, SLTU	SLT R1,R2,R3	R1<=1 if R2<R3 else R1<=0	test on R2,R3 outcome in R1, signed or unsigned comparison
	SLTI, SLTUI	SLTI R1,R2,#4	R1<=1 if R2<4 else R1<=0	test R2 outcome in R1, signed or unsigned comparison
Branches/Jumps	BEQZ, BNEZ	BEQZ R1,label	PC<=label if (R1)=0	conditional branch-equal 0/not equal 0
	BEQ, BNE	BNE R1,R2,label	PC<=label if (R1)=(R2)	conditional branch-equal/not equal
	J	J target	PC<=target	target is an immediate field
	JR	JR R1	PC<=(R1)	target is in register
	JAL	JAL target	R1<=(PC)+4; PC<=target	jump to target after saving the return address in R31
Floating point	ADD.S,SUB.S,MUL.S,DIV.S	ADD.S F1,F2,F3	F1<=(F2)+(F3)	float arithmetic single precision
	ADD.D,SUB.D,MUL.D,DIV.D	ADD.D F0,F2,F4	F0<=(F2)+(F4)	float arithmetic double precision

Pipelining
(Ch 3.1.1)

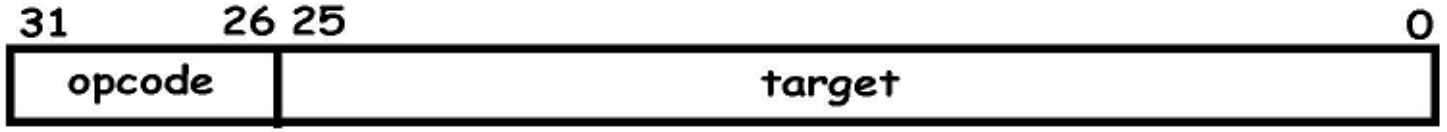
Instruction Formats



LW Rt, displacement(Rs)
SW Rt, displacement(Rs)
ADDI Rt, Rs, immediate
BEQ Rt, Rs, offset



ADD Rd, Rt, Rs



J target
JAL target

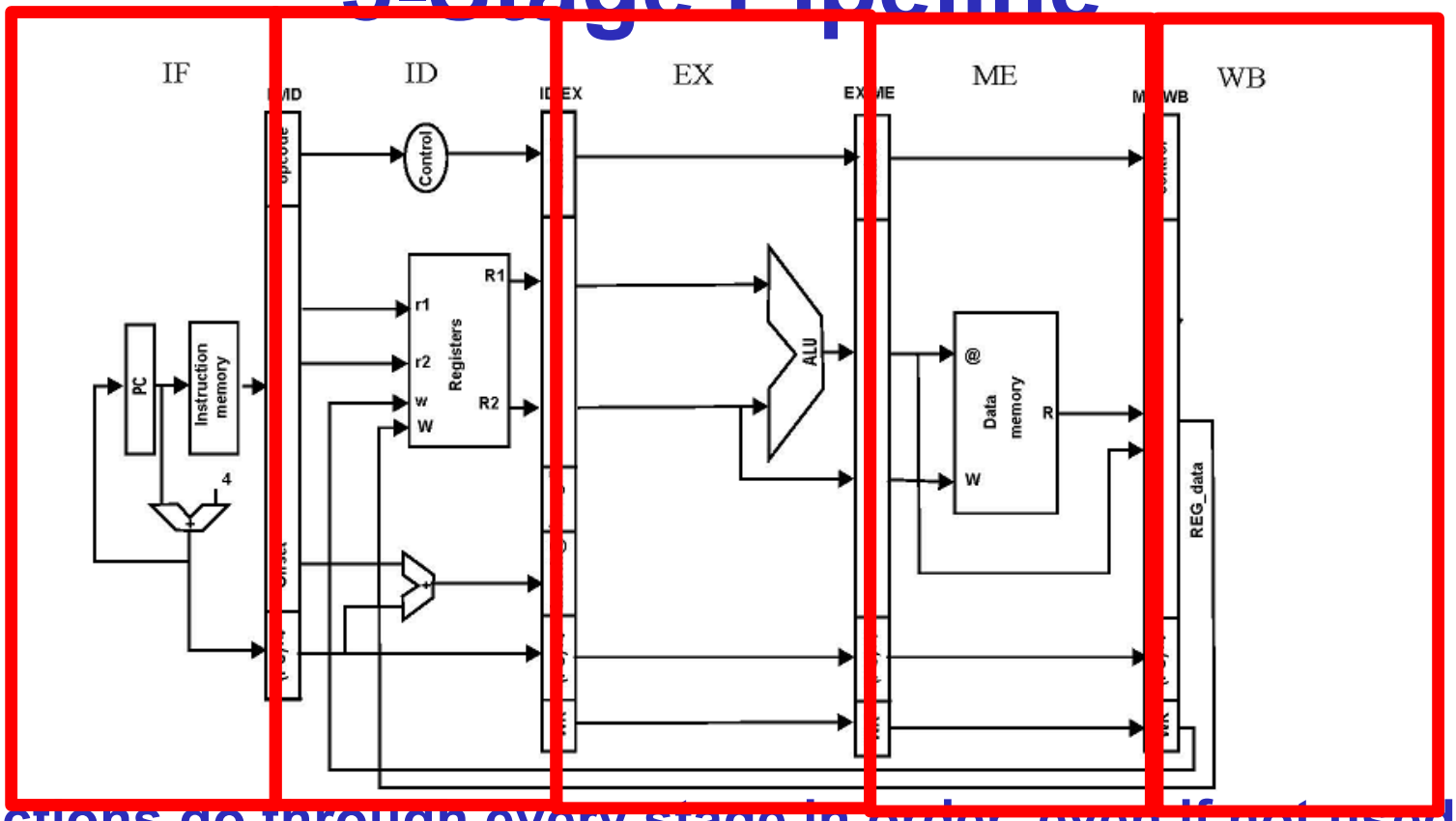
Execution Steps in Instructions

Instruction	I-Fetch(IF)	I-Decode(ID)	Execute(EX)	Memory(ME)	Write-Back(WB)
LW R1,#20(R2)	Fetch; PC+=4	Decode; Fetch R2	Compute address = (R2)+20	Read	Write in R1
SW R1,#20(R2)	Fetch; PC+=4	Decode; Fetch R1 and R2	Compute address = (R2)+20	Write	--
ADD R1,R2,R3	Fetch; PC+=4	Decode; Fetch R2 and R3	Compute (R2)+(R3)	--	Write in R1
ADDI R1,R2,imm.	Fetch; PC+=4	Decode; Fetch R2	Compute (R2)+imm	--	Write in R1
BEQ R1,R2,offset	Fetch; PC+=4	Decode; Fetch R1 and R2 Compute target-- address (PC)+offset	Subtract (R1) and (R2) Take branch if zero	--	--
J target	Fetch; PC+=4	Decode; Take jump	--	--	--

THESE STEPS CAN BE PIPELINED

WE'LL DEAL WITH FLOATING-POINT INSTRUCTIONS LATER

5-Stage Pipeline



Instructions go through every stage in order, even if not used

Note: Control implementation

- Instruction carries control
- This is a general approach: ”Each instruction carries its baggage”

Pipeline Hazards

Pipeline Hazards

Structural hazards

- Caused by resource contention (e.g. register file, memory fetch)
- Can be avoided by adding resources (unless too costly)
- No such hazard in the 5-stage pipeline

Example: single memory in the 5-stage pipeline

Data hazards

- Due to program dependencies (RAW, WAW, WAR)
- Both for memory and register operand accesses
- Difference between data dependencies and data hazards
- Software vs. hardware implementation
- In 5-stage pipeline: Only RAW dependencies on registers cause hazards because all instructions go through pipeline stage in process order.

Control hazards

- Branch, jump, exceptions

Data Hazards

RAW Hazards on Register Values

RAW hazard with a preceding ALU instruction

	CLOCK ==>	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	ADD R1,R2,R3	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	EX	ME	WB			
I3	LW R5,0(R1)			IF	ID	EX	ME	WB		
I4	ORI R6,R1,#20				IF	ID	EX	ME	WB	
I5	SUBI R1,R1,R7					IF	ID	EX	ME	WB

I1 and I5 are fine

Between I1 and I4:

- Register forwarding: value stored = value read

Between I1 and I2

- Value is available and can be forwarded from ME input to EX input

Between I1 and I3

- Value is available and can be forwarded from WB input to EX input

Forwarding: Provides direct paths between ME/WB into EX. Forwarding is implemented by detecting in a forwarding unit (FU) that the instruction in ME/WB writes into one input register of the instruction in EX.

RAW Hazards on Register Values

Dependencies with prior Loads pose special complications

	CLOCK ==>	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	LW R1,0(R3)	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	EX	ME	WB			
I3	LW R5,0(R1)			IF	ID	EX	ME	WB		
I4	ORI R6,R1,#20				IF	ID	EX	ME	WB	
I5	SUBI R1,R1,R7					IF	ID	EX	ME	WB

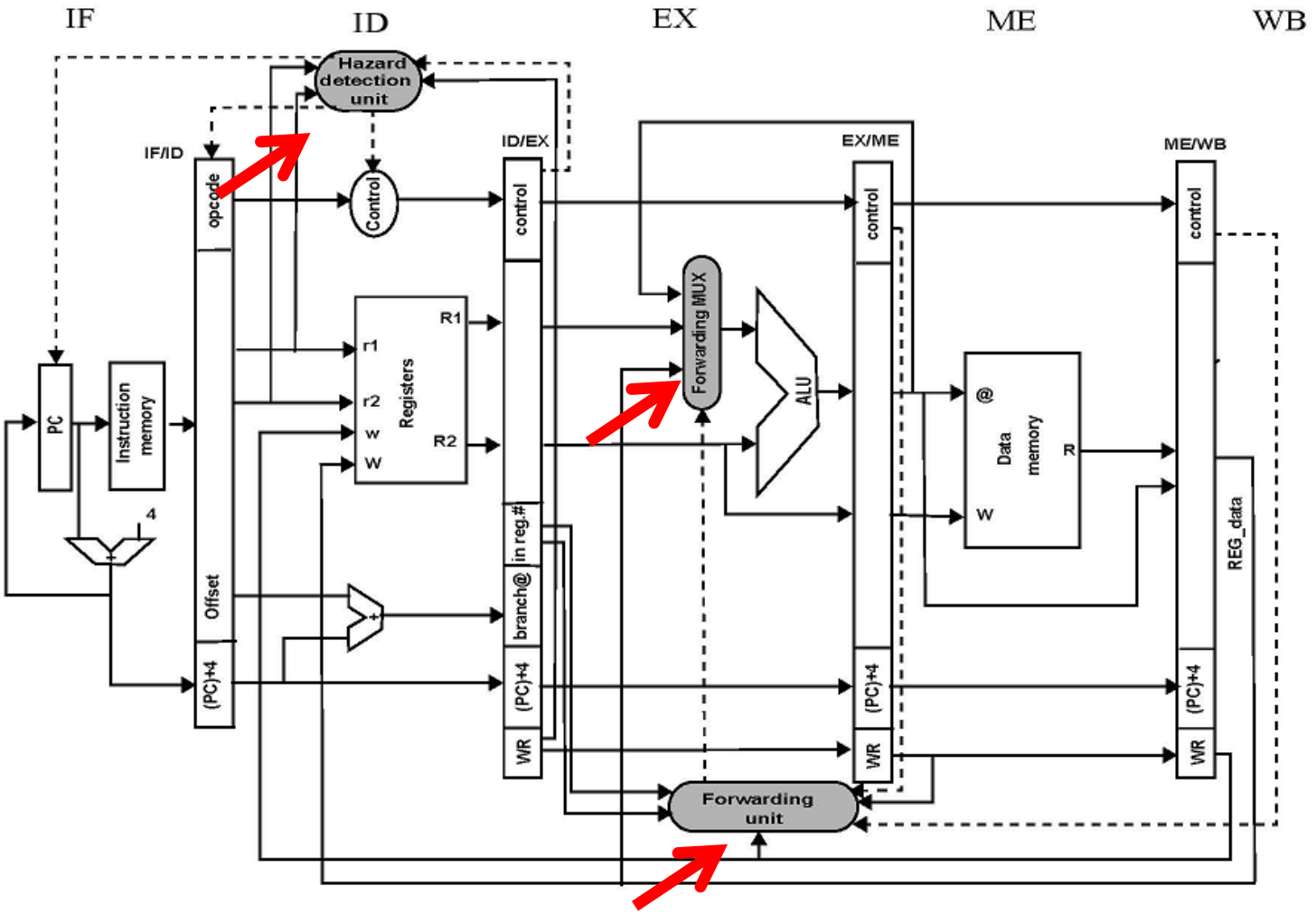
In this case it is not possible to forward the value from I1 to I2 because the value is available only in cycle C5

- However, the value can be forwarded from I1 to I3
- I2 must be stalled to wait for the value of I1

	CLOCK ==>	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	LW R1,0(R3)	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	ID	EX	ME	WB		
I3	LW R5,0(R1)			IF	IF	ID	EX	ME	WB	
I4	ORI R6,R1,#20					IF	ID	EX	ME	WB
I5	SUBI R1,R1,R7						IF	ID	EX	ME

I2 and I3 must be stalled in IF and ID for one cycle (C4) by a hazard detection unit (HDU) in cycle C4. A NOOP has been clocked in EX

5-Stage Pipeline with FU and HDU

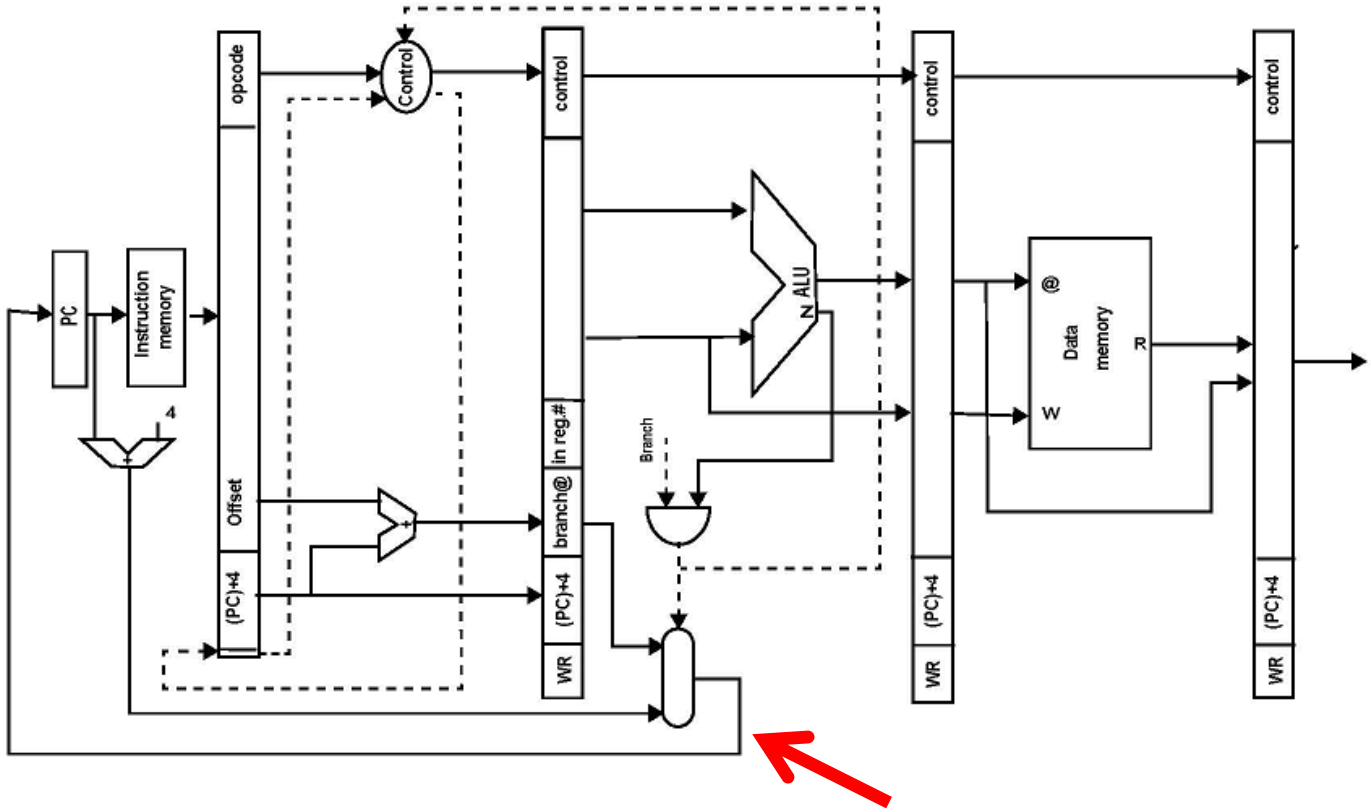


Control Hazards

Control Hazards

Need target address (computed in ID) and condition (computed in EX)

- Predict Branch not Taken and take it in EX if taken



If the branch must be taken then IF and ID must be flushed and the target address must be clocked into the PC

Exceptions 1(2)

Pipeline benefits come from overlapped instruction executions

- Things go wrong when the flow of instructions is suddenly interrupted (EXCEPTIONS)

Many exceptions must be precise. On a precise exception:

1. All instructions preceding the faulting instruction must complete
2. The faulting instruction and all following instructions must be squashed (flushed)
3. The handler must start its execution.

Exceptions 2(2)

It is not practical to take an exception in the cycle when it happens

- Multiple exceptions in the same cycle
- It is complex to take exceptions in various pipeline stages
- The major reason is that exceptions must be taken in processor order and not in temporal order

Instead, flag the exception and record its cause when it happens.

Keep it “silent” and wait until the instruction reach the WB stage to take it.

You should know up to now

- Major components of an instruction-set architecture (ISA)
  - Instruction types
  - Register and memory operands
  - Addressing modes
  - Impact of usage on the design of the ISA
  - Exceptions
  - Instruction format
- Structure of a basic 5-stage pipeline
- The different hazard types and how to deal with them
- Precise exceptions and how to deal with them