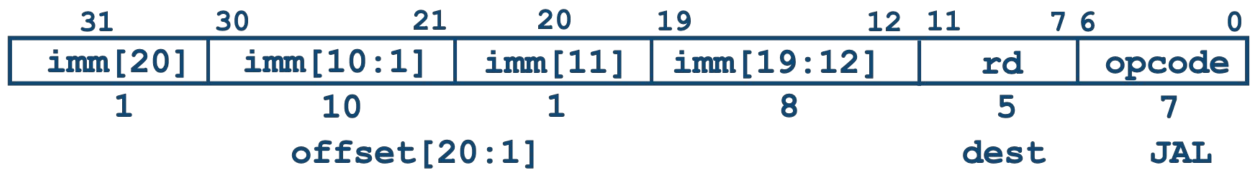
Discussion 6

RISC-V instructions, Pipelining, and Hazards

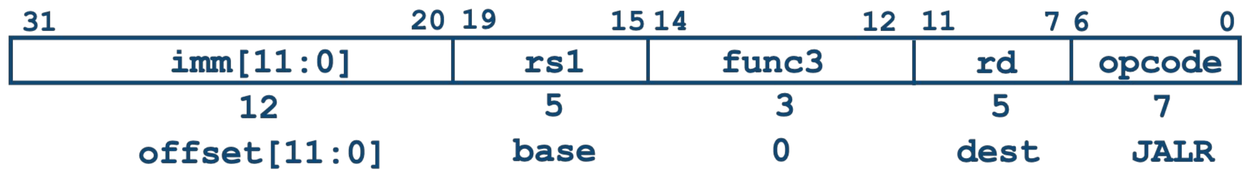
RISC-V ISA

- Branch instructions
    - beq, bne, blt, bge, bltu, bgeu
- How many comparator units are required in the branch unit?
- What’s the maximum branch jumping range?
    - +- 4 kiB
    - Branch immediate implicitly sets LSB to 0 (must be a multiple of 2 bytes). Why? Compressed ISA extension
- How are branches resolved in a pipelined datapath?
    - Branch prediction and recovery/flushing on a mispredict
    - Bubble insertion / stalling

RISC-V ISA



- JAL saves PC+4 in register rd (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart



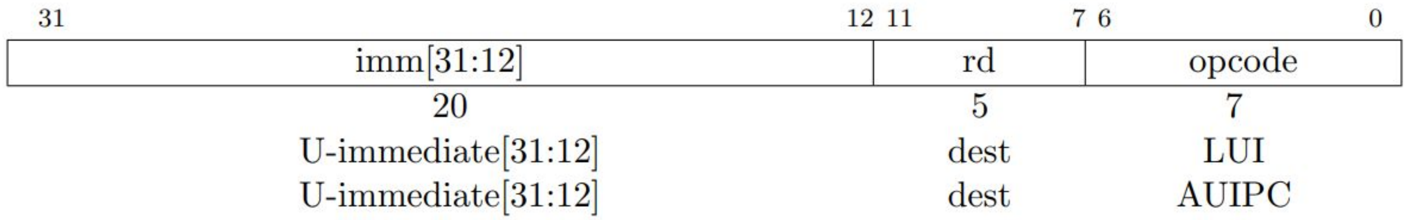
- JALR Writes PC+4 to rd (return address)
- Sets PC = rs1 + immediate (and sets the LSB to 0)
- Uses same immediates as arithmetic and loads

RISC-V ISA

- Jump instructions
- JAL
    - Jump and link
    - Used commonly to jump within a +- 1 MiB range (PC relative)
    - Link = writeback of PC + 4, used to return from a function call
    - Plain unconditional jumps writeback to x0
- JALR
    - Jump and link register
    - Used less frequently to jump to any address (non-PC relative)
    - LUI + JALR is a common combination to jump across large ranges
    - Can also be paired with AUIPC
- What hazards are introduced by jumps in pipelined datapaths?

RISC-V ISA

- Register load instructions, 20-bit U immediate
- AUIPC
    - rd = PC + (imm<<12)
    - Can be used to fetch the current PC
- LUI
    - rd = imm << 12



31 … 12	11 … 7	6 … 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

JAL example

```
0x0: li x1, 0
0x4: jal x2, 8
0x8: li x1, 100
0xc: li x3, 200
```

X1 = ?
X2 = ?
X3 = ?

JAL example

```
0x0: li x1, 0
0x4: jal x2, 8
0x8: li x1, 100
0xc: li x3, 200
```

X1 = 0
X2 = PC + 4 = 0x4 + 4 = 0x8
X3 = 200

JALR example

```
0x0: li x1, 4
0x4: jalr x2, x1, 8
0x8: li x3, 100
0xc: li x3, 200
```

X1 = ?
X2 = ?
X3 = ?

JALR example

```
0x0: li x1, 4
0x4: jalr x2, x1, 8
0x8: li x3, 100
0xc: li x3, 200
```

X1 = 4

X2 = PC + 4 = 0x4 + 4 = 0x8

X3 = 200

JALR example

```
0x0: nop
0x4: nop
0x8: nop
0xc: auipc x1, 0x8
```

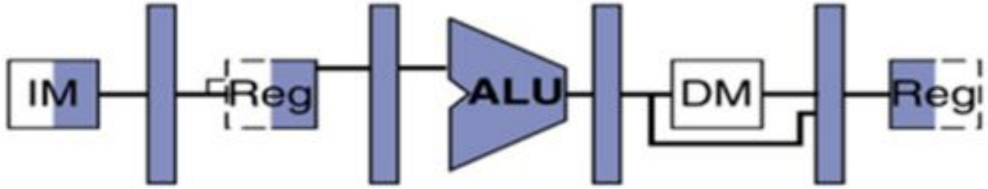
X1 = PC + (0x8 << 12) = ?

JALR example

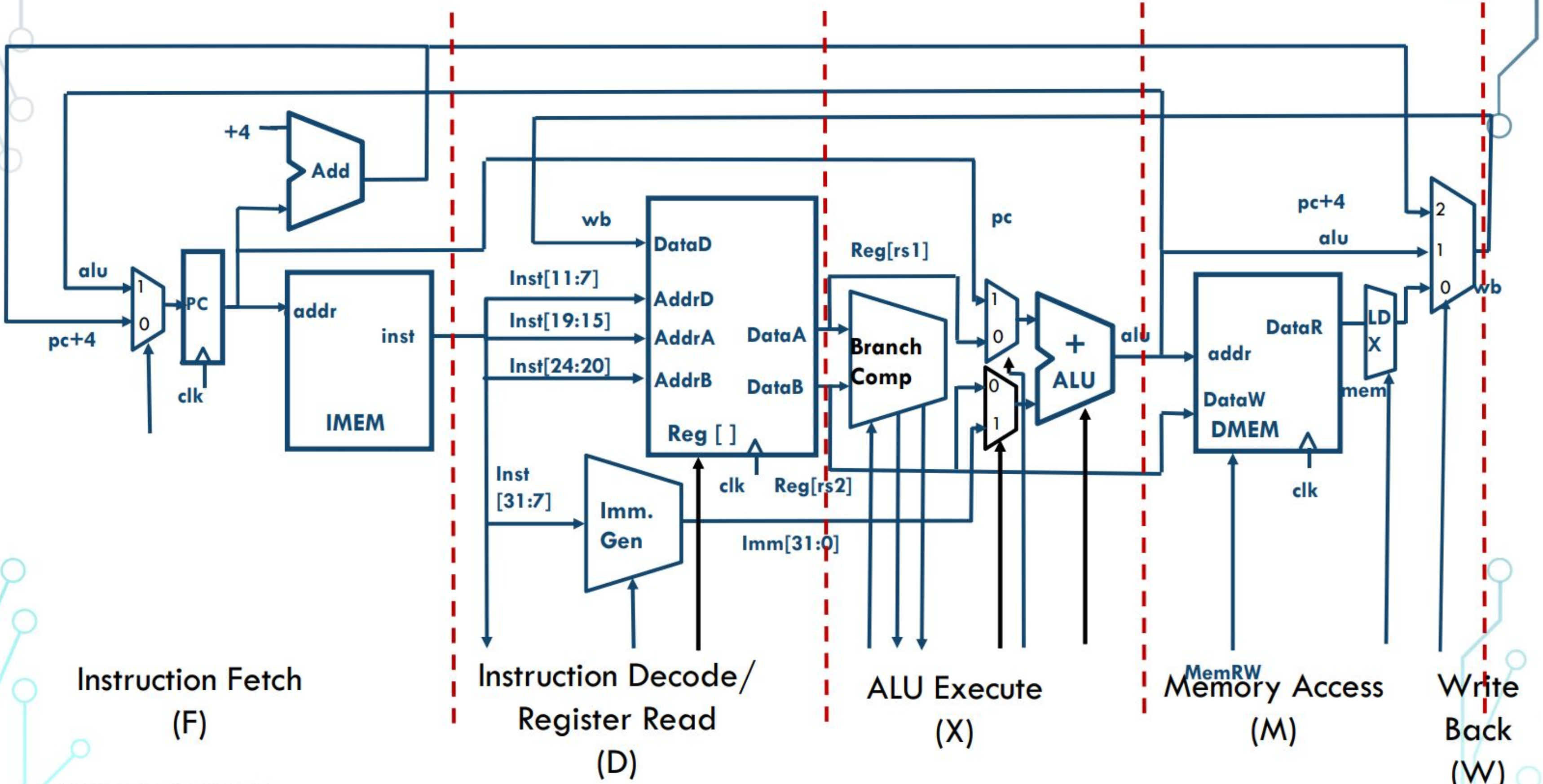
```
0x0: nop
0x4: nop
0x8: nop
0xc: auipc x1, 0x8
```

X1 = PC + (0x8 << 12) = 0xc + 0x8000 = 0x800c

Pipelining



Complete RV32I Datapath with Control

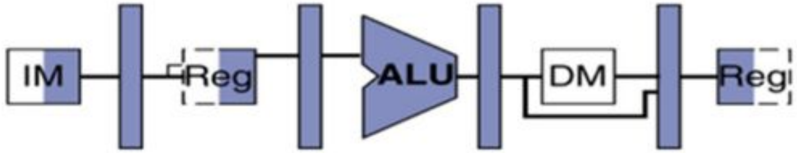


Pipeline flow diagram (cycle 1)

Assume x1, x2, x3 are initialized

1	2	3	4	5
F(i1)	D	Ex	Mem	Wb

i1. Add x6, x1, x2
i2. Add x7, x3, x3
i3. Sub x8, x1, x2
i4. And x8, x1, x3
i5. Lw x9, 40(x1)
i6. Sw x10, 20(x6)

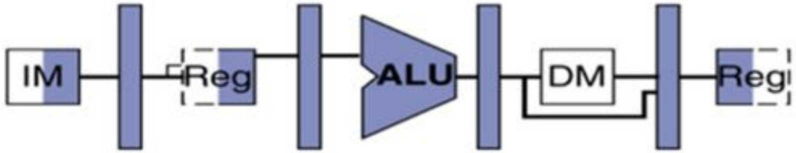


Pipeline flow diagram (cycle 2)

Assume x1, x2, x3 are initialized

i1. Add x6, x1, x2
i2. Add x7, x3, x3
i3. Sub x8, x1, x2
i4. And x8, x1, x3
i5. Lw x9, 40(x1)
i6. Sw x10, 20(x6)

1	2	3	4	5	
F(i1)	D(i1)	Ex	Mem	Wb	
	F(i2)	D	Ex	Mem	Wb

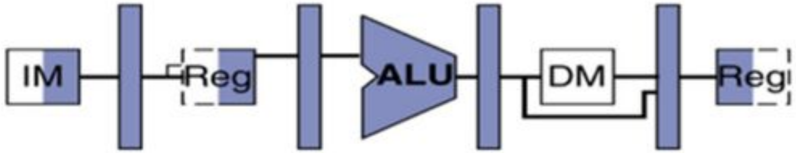


Pipeline flow diagram (cycle 3)

Assume x1, x2, x3 are initialized

i1. Add x6, x1, x2
i2. Add x7, x3, x3
i3. Sub x8, x1, x2
i4. And x8, x1, x3
i5. Lw x9, 40(x1)
i6. Sw x10, 20(x6)

1	2	3	4	5		
F(i1)	D(i1)	Ex(i1)	M	Wb		
	F(i2)	D(i2)	Ex	M	Wb	
		F(i3)	D	Ex	M	Wb

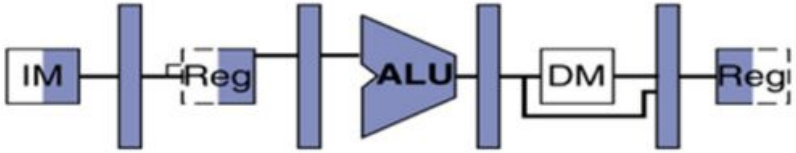


Pipeline flow diagram (cycle 4)

Assume x1, x2, x3 are initialized

i1. Add x6, x1, x2
i2. Add x7, x3, x3
i3. Sub x8, x1, x2
i4. And x8, x1, x3
i5. Lw x9, 40(x1)
i6. Sw x10, 20(x6)

1	2	3	4	5				
F(i1)	D(i1)	Ex(i1)	M(i1)	Wb				
	F(i2)	D(i2)	Ex(i2)	M	Wb			
		F(i3)	D(i3)	Ex	M	Wb		
			F(i4)	D	Ex	M	Wb	

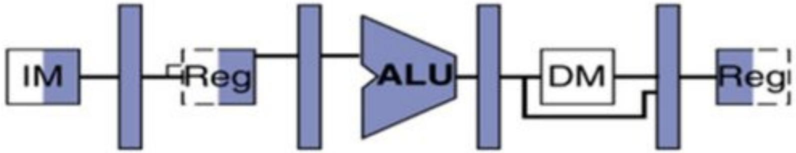


Pipeline flow diagram (cycle 5)

Assume x1, x2, x3 are initialized

i1. Add x6, x1, x2
i2. Add x7, x3, x3
i3. Sub x8, x1, x2
i4. And x8, x1, x3
i5. Lw x9, 40(x1)
i6. Sw x10, 20(x6)

1	2	3	4	5				
F(i1)	D(i1)	Ex(i1)	M(i1)	W(i1)				
	F(i2)	D(i2)	Ex(i2)	M(i2)	W			
		F(i3)	D(i3)	Ex(i3)	M	W		
			F(i4)	D(i4)	Ex	M	W	
				F(i5)	D	Ex	M	W



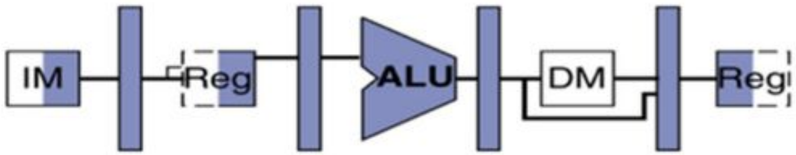
Pipeline flow diagram (cycle 6)

Assume x1, x2, x3 are initialized

DONE!

i1. Add x6, x1, x2
i2. Add x7, x3, x3
i3. Sub x8, x1, x2
i4. And x8, x1, x3
i5. Lw x9, 40(x1)
i6. Sw x10, 20(x6)

1	2	3	4	5	6				
F(i1)	D(i1)	Ex(i1)	M(i1)	W(i1)					
	F(i2)	D(i2)	Ex(i2)	M(i2)	W(i2)				
		F(i3)	D(i3)	Ex(i3)	M(i3)	W			
			F(i4)	D(i4)	Ex(i4)	M	W		
				F(i5)	D(i5)	Ex	M	W	
					F(i6)	D	Ex	M	W



Pipeline flow diagram (cycle 6+)

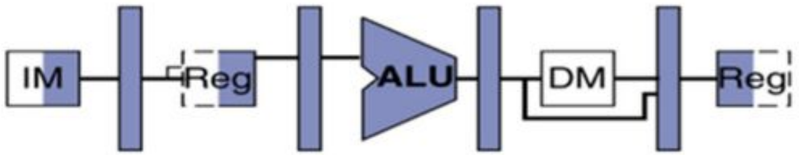
Assume x1, x2, x3 are initialized

i1. Add x6, x1, x2
i2. Add x7, x3, x3
i3. Sub x8, x1, x2
i4. And x8, x1, x3
i5. Lw x9, 40(x1)
i6. Sw x10, 20(x6)

DONE!

ALSO DONE!

1	2	3	4	5	6	7				
F(i1)	D(i1)	Ex(i1)	M(i1)	W(i1)						
	F(i2)	D(i2)	Ex(i2)	M(i2)	W(i2)					
		F(i3)	D(i3)	Ex(i3)	M(i3)	W(i3)				
			F(i4)	D(i4)	Ex(i4)	M(i4)	W			
				F(i5)	D(i5)	Ex(i5)	M	W		
					F(i6)	D(i6)	Ex	M	W	
						F(xx)	D	Ex	M	W



… .... etc... etc...

Pipeline flow diagram (round 2)

Assume x1, x2, x3 are initialized

1	2	3	4	5
F(i1)	D	Ex	Mem	Wb

i1. Add x6, x1, x2
i2. Add x7, x3, x3
i3. Sub x8, x6, x2

Pipeline flow diagram (round 2)

Assume x1, x2, x3 are initialized

i1. Add x6, x1, x2
i2. Add x7, x3, x3
i3. Sub x8, x6, x2

1	2	3	4	5		
F(i1)	D	Ex	Mem	Wb		
	F(i2)	D(i2)	Ex(i2)	M(i2)	W(i2)	
		F(i3)	D(i3)	Ex(i3)	M(i3)	W(i3)

What happens here?

Pipeline flow diagram (round 2)

Assume x1, x2, x3 are initialized

i1. Add x6, x1, x2
i2. Add x7, x3, x3
i3. Sub x8, x6, x2

1	2	3	4	5		
F(i1)	D	Ex	Mem	Wb		
	F(i2)	D(i2)	Ex(i2)	M(i2)	W(i2)	
		F(i3)	D(i3)	$\boxed{\text{Ex(i3)}}$	M(i3)	W(i3)

- Pipelining leads to hazards
  - Data hazards (read after write)
    - ALU -> ALU dependency (add then add)
    - Memory -> ALU dependency (lw then add)
    - ALU -> Memory dependency (add then sw)
    - Memory -> Memory dependency (lw then sw)
  - Control hazards (jal, jalr, branch)

Hazards

Pipelining Hazards

- A hazard is a situation that prevents starting the next instruction in the next clock cycle

i1. Add **x6**, x1, x2
i2. Add x7, x3, x3
i3. Sub x8, **x6**, x2

- 3 Types:
  - Structural hazards
    - A required resource is busy
  - Data hazards
    - Data dependency between nearby instructions
  - Control hazards
    - Flow of execution depends on previous instruction (branches)

Solving Hazards

- Depends on type of hazard you want to solve
  - Add more hardware (duplicate ALUs, etc…)
  - Add more ports to memories (can read/write multiple values at a time!)
  - Add branch prediction to make educated guess about control flow!
  - Forward values as soon as they're calculated (sometimes referred to as bypassing)

- Stalling (a.k.a. adding bubbles a.k.a. interlocking)
  - Will always work but will cost performance

Fa15 MT2 #5

5. [12pts/12pts] MIPS microarchitecture.

Consider the design of the single-cycle MIPS processor, as we we discussed in class. Suppose you want to pipeline this design using *2 pipeline stages*. Functionality is divided into 2 pipeline stage as shown below:

stage 1	stage 2
instruction fetch/decode	ALU operation
regfile access	Dmem access

In the space below, explain what *hazards* will result from this pipelining (prior to any measures to remove the hazards). For each hazard, i) describe why it occurs, ii) write a short MIPS assembly language instruction sequence to demonstrate a case when the hazard would occur, and iii) describe what you can do to deal with the hazard. Write your answers in the space provided. (You might not need all answer spaces.)

Fa15 MT2 #5

(a) Hazard 1

i. **Control hazard occurs on every branch instruction because the branch comparison is done in stage 2 and the branch target address is needed in stage 1.**

ii. *Any insruction sequence with a branch.*

iii. **1) Stall the pipeline to allow the branch instruction to complete, 2) implement branch prediction.**

(b) Hazard 2

i. **Data hazard occurs on r-type instruction dependent on the immediately preceeding instruction, because ALU output is available sometime in stage 2 but needed at the beginning of stage 2.**

ii.

```
add $1, $2, $3
add $4, $5, $1
```

iii. **1) Stall the pipeline to allow r-type to store results back to register file, 2) Selectively feed output of ALU to ALU input register.**

Fa15 MT2 #5

(c) Hazard 3

i. **Data hazard on load from memory when an instruction is dependent on load instruction immediately preceeding. Result from memory load is available at end of stage 2 but needed at beginning of stage 2.**

ii.

```
lw $1, 0($1)
add $4, $5, $1
```

iii. **1) Selectively direct output of memory to input register of ALU, 2) stall the load instruction.**