EECS 151/251A Discussion 5

Daniel Grubb 9/28, 9/29, 10/4

Administrivia

- Homework 4 due Oct 4th
- No new homework this week
- Lab 4 due Friday Oct 1st
- Lab 5 this week - due after midterm
- Midterm
  - Oct 7th, 7-8:30PM
  - Scope: all lectures and material through 10/4
  - Logistics details TBA

Review session on 10/4 or 10/5

RISC-V

- “risk-five”
- Developed right here at Berkeley!
- Why RISC-V?
  - Free, flexible, extensible
  - Great for education in this course, and more and more prolific in industry
- Look through the spec here!
  - Includes RV32I for this class plus 64b, extensions, etc.
- Basis of the course project!

RISC-V Glossary

- **ISA** - Instruction Set Architecture
- **ALU** - arithmetic logic unit
- **PC** - program counter
- **Immediate** - a constant value used in instructions
- **Byte** - 8 bit value
- **Word** - 32 bit value (4 bytes; may be 64, etc. depending on implementation)
- **Half-word** - 16 bit value (or half of word length)

RV32I

Open Reference Card

Base Integer Instructions: RV32I

Category	Name	Fmt	RV32I Base	Category	Name	Fmt	RV32I Base
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	Loads	Load Byte	I	LB rd,rs1,imm
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt		Load Halfword	I	LH rd,rs1,imm
	Shift Right Logical	R	SRL rd,rs1,rs2		Load Byte Unsigned	I	LBU rd,rs1,imm
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt		Load Half Unsigned	I	LHU rd,rs1,imm
	Shift Right Arithmetic	R	SRA rd,rs1,rs2		Load Word	I	LW rd,rs1,imm
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	Stores	Store Byte	S	SB rs1,rs2,imm
Arithmetic	ADD	R	ADD rd,rs1,rs2		Store Halfword	S	SH rs1,rs2,imm
	ADD Immediate	I	ADDI rd,rs1,imm		Store Word	S	SW rs1,rs2,imm
	SUBtract	R	SUB rd,rs1,rs2	Branches	Branch =	B	BEQ rs1,rs2,imm
	Load Upper Imm	U	LUI rd,imm		Branch ≠	B	BNE rs1,rs2,imm
	Add Upper Imm to PC	U	AUIPC rd,imm		Branch <	B	BLT rs1,rs2,imm
Logical	XOR	R	XOR rd,rs1,rs2		Branch ≥	B	BGE rs1,rs2,imm
	XOR Immediate	I	XORI rd,rs1,imm		Branch < Unsigned	B	BLTU rs1,rs2,imm
	OR	R	OR rd,rs1,rs2		Branch ≥ Unsigned	B	BGEU rs1,rs2,imm
	OR Immediate	I	ORI rd,rs1,imm	Jump & Link	J&L	J	JAL rd,imm
	AND	R	AND rd,rs1,rs2		Jump & Link Register	I	JALR rd,rs1,imm
	AND Immediate	I	ANDI rd,rs1,imm				
Compare	Set <	R	SLT rd,rs1,rs2	Synch	Synch thread	I	FENCE
	Set < Immediate	I	SLTI rd,rs1,imm				
	Set < Unsigned	R	SLTU rd,rs1,rs2	Environment	CALL	I	ECALL
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm		BREAK	I	EBREAK

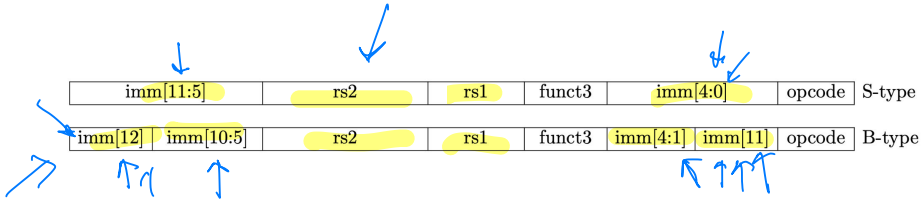
x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31

- Additional state?

RISC-V Instruction Formats

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

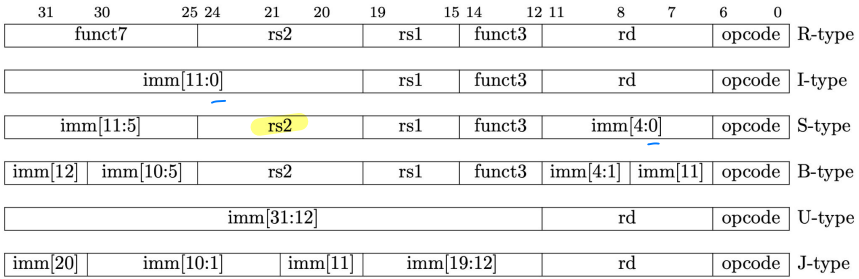
Design Patterns



- Load/store architecture: operate on registers, not directly on memory
- Decoding: designed for hardware efficiency
  - Fixed length (32b) instruction
  - Locations of register fields common among instructions (why?) fewer muxes
  - Immediate bit rotation for J- and B-type (why?)
    - Why skip lower bits of immediate?
- RV32I is enough to run any C program
  - Register loading, arithmetic, logic, memory load/store, branches, jumps
  - Additional pseudo-instructions (spec table 25.2, 25.3)
  - Reserved opcodes for extensions
- Additional resources beyond spec can be found in lecture and CS61C

Instruction Details

- Arithmetic (R-, I-)
  - ALU modes
  - Different versions of same instruction:
  - **slt**, **sltu, slti**
  - **srl**, **sra, srli** (a vs. l?)
- Load/store
  - Load: I-type, Store: S-type
  - Byte-addressing, little endian
  - Load/store granularity:
  - **sw**, **sh**, **sb**
  - **lw**, **lh**, **lhu**, **lb**, **lbu**
    - Sign extension

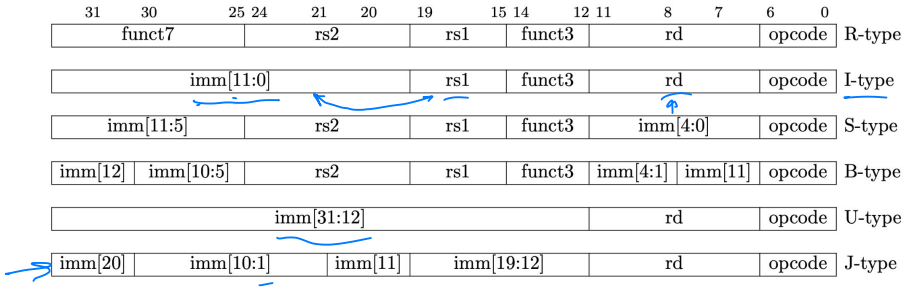


R-type assembly:
```
<inst> rd, rs1, rs2
```

I-type assembly:
```
<inst> rd, rs1, imm
<inst> rd, imm(rs1)
```

S-type assembly:
```
<inst> rs1, imm(rs2)
```

Instruction Details



- Conditional branches
  - B-type instructions are similar to S-type (what's different?)
  - Branch comparison types - **bltu**, **blt**, **beq**
- Jumps
  - **jal** (J-type), **jalr** (I-type) "j"?
    - 21b vs 12b LUI + JALR → 32b
  - Both write PC+4 to rd unless rd == x0
- Upper Immediate
  - U-type used to get upper 20 bits of an immediate into rd
  - **auipc** also adds immediate to current PC

LUI 20b → register

imm. → lower 12b

j LABEL

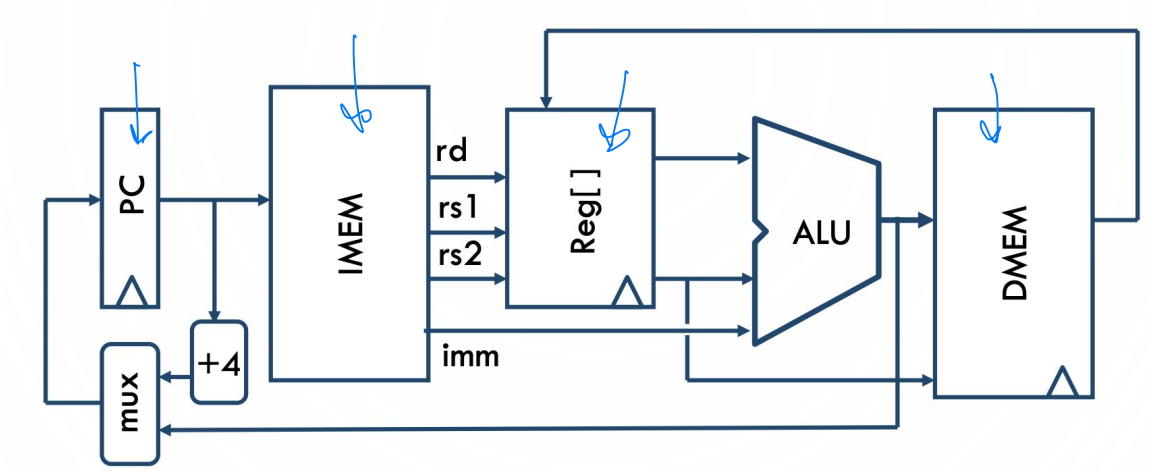
B-type assembly:
`<inst> rs1, imm(rs2)`

J-type assembly:
`jal rd, label`

U-type assembly:
`<inst> rd, imm`

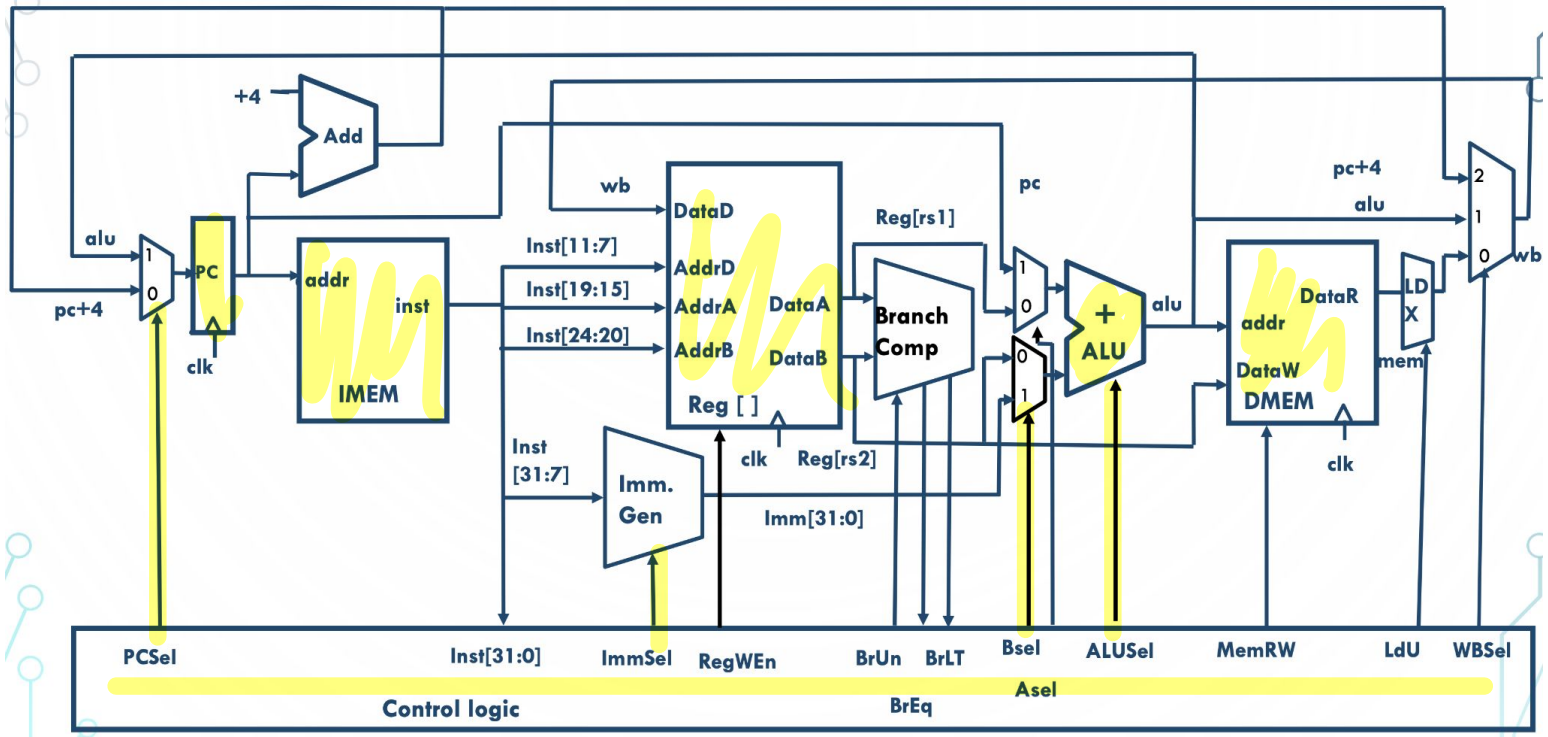
RISC-V CPU Implementation

- Components
  - Datapath
  - Control
  - External memory
- Stages
  - IF
  - ID
  - Exec
  - Mem
  - WB



State?
PC+4?

RISC-V Datapath + Control



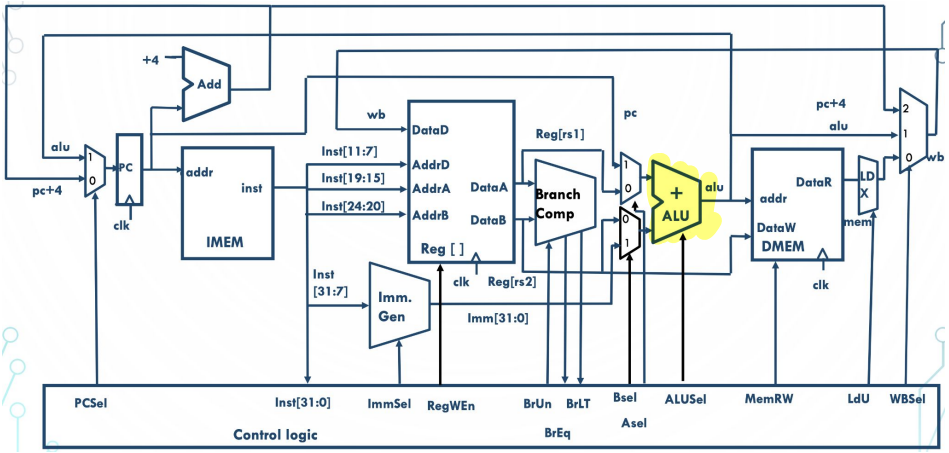
Control Signals

- Inputs
  - Instruction (read from instruction memory)
  - **BrEq**: inputs of branch comparison equal
  - **BrLT**: input 1 less than input 2 of branch comparison
- Outputs
  - **PCSel**: ALU output vs PC+4
  - **ImmSel**: select 1 of 5 types (based on instruction type)
  - **RegWEn**: enable writeback to register file
  - **BrUn**: branch comparison unsigned mode (eg. bltu)
  - **ASel/BSel**: select ALU inputs between (PC or Reg[rs1]) and (imm or Reg[rs2])
  - **ALUSel**: select 1 of 10 operations
  - **MemRW**: read from/write to data memory
  - **WBSel**: rd data from data memory, ALU, or PC+4

ALU Example

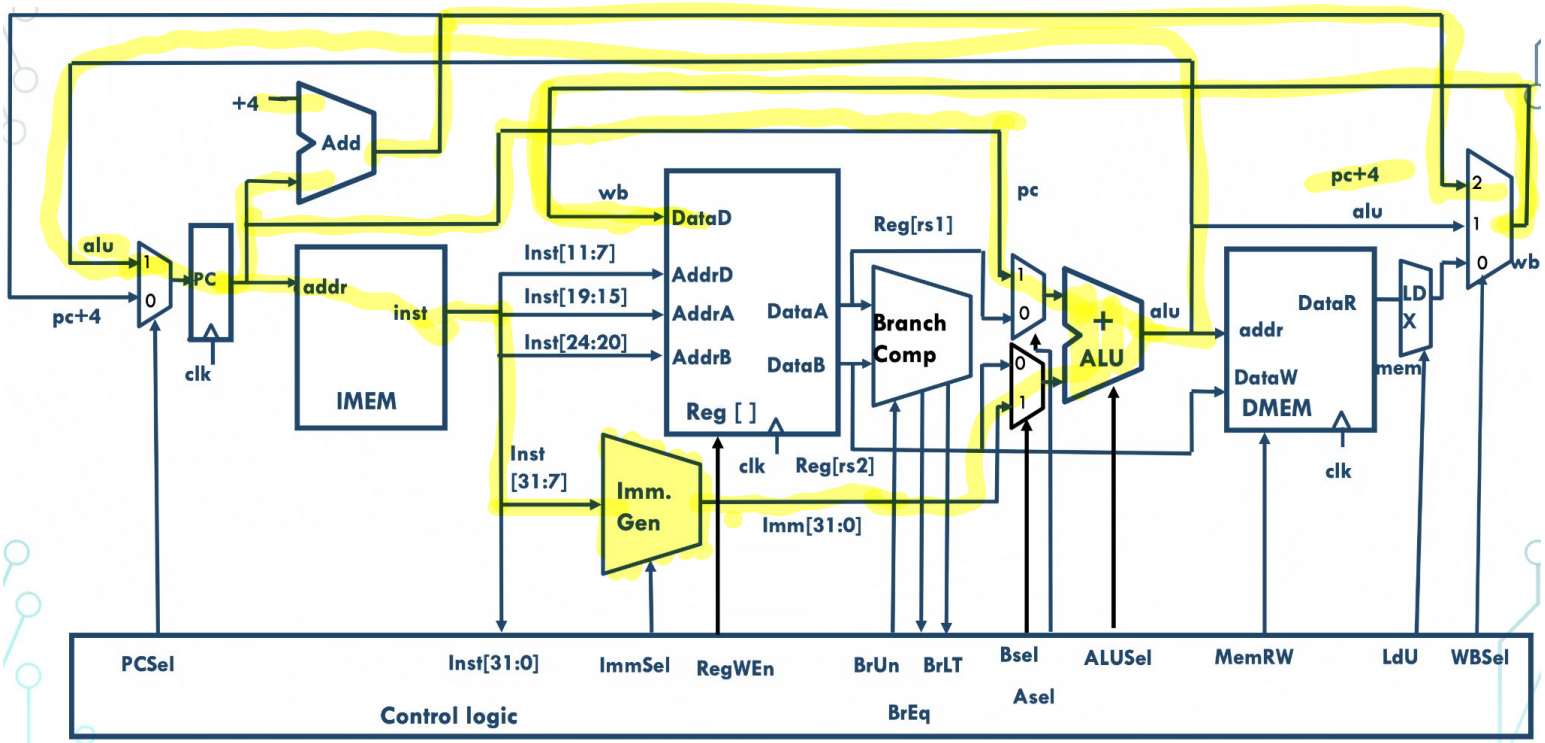
- 10 modes
- Incomplete example: (from past midterm)

```
wire signed [31:0] in1s, in2s;
assign in1s = in1;
assign in2s = in2;
always @(*) begin
    case (ALUSel)
        ADD:          alu = in1 + in2;
        SUB:          alu = in1 - in2;
        SHIFT_LEFT:   alu = in1 << in2[4:0];
        LESS_THAN_S:  alu = (in1s < in2s) ? 32'b1 : 32'b0;
        SHIFT_RIGHT:  alu = in1 >> in2[4:0];
        OR:           alu = in1 | in2;
        AND:          alu = in1 & in2;
        PASS:         alu = in2;
    endcase
end
```



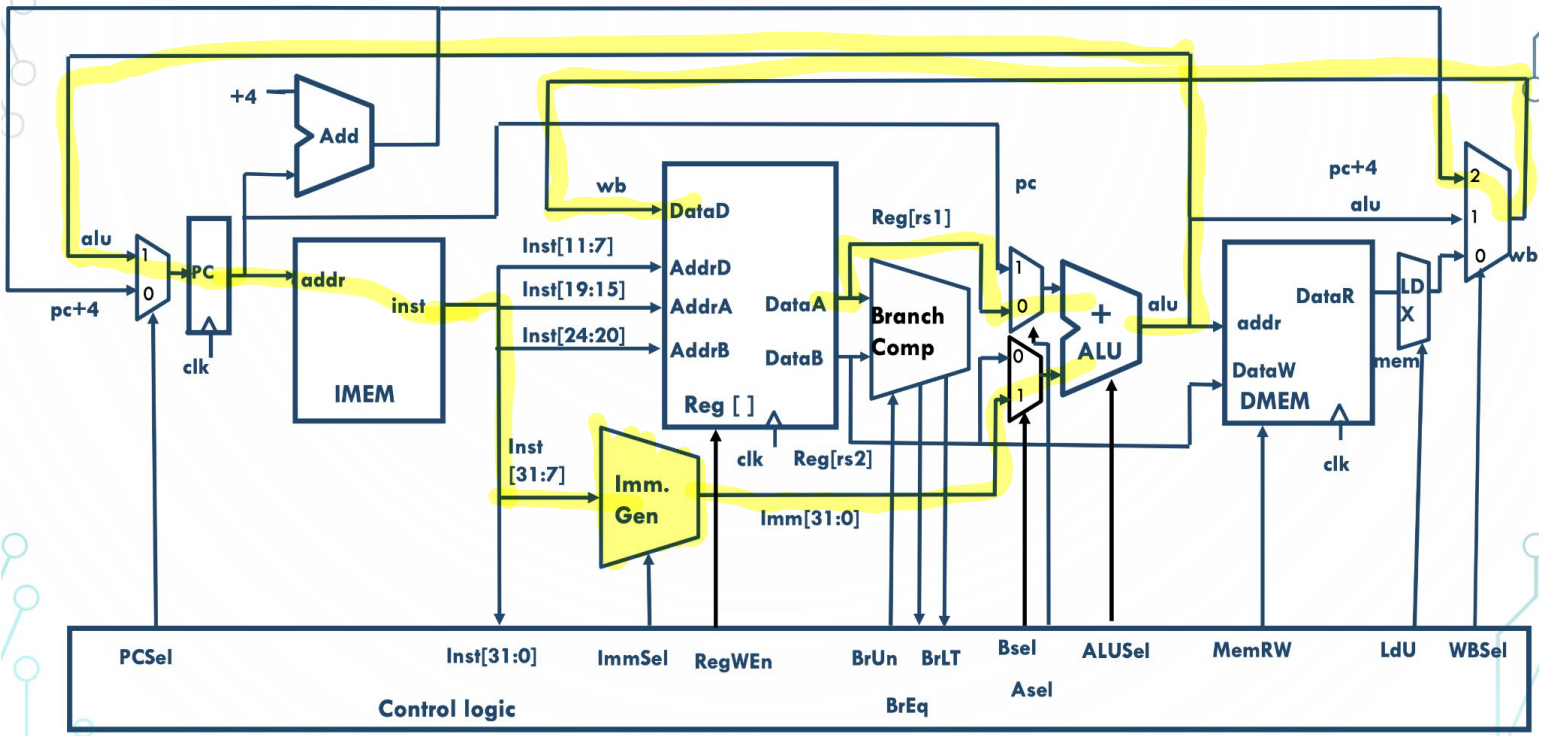
Pipeline Exercise

JAL



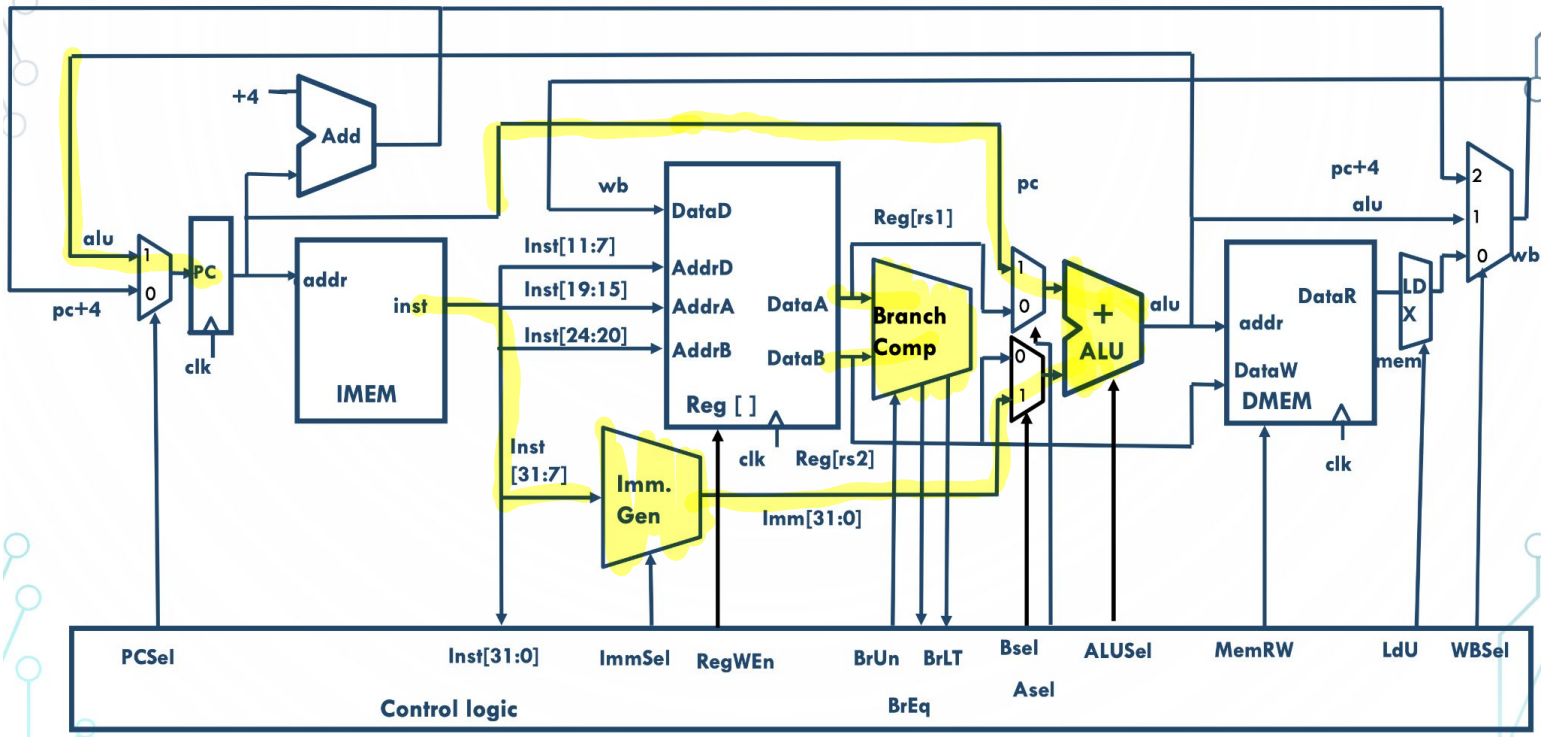
Pipeline Exercise

JALR I-type



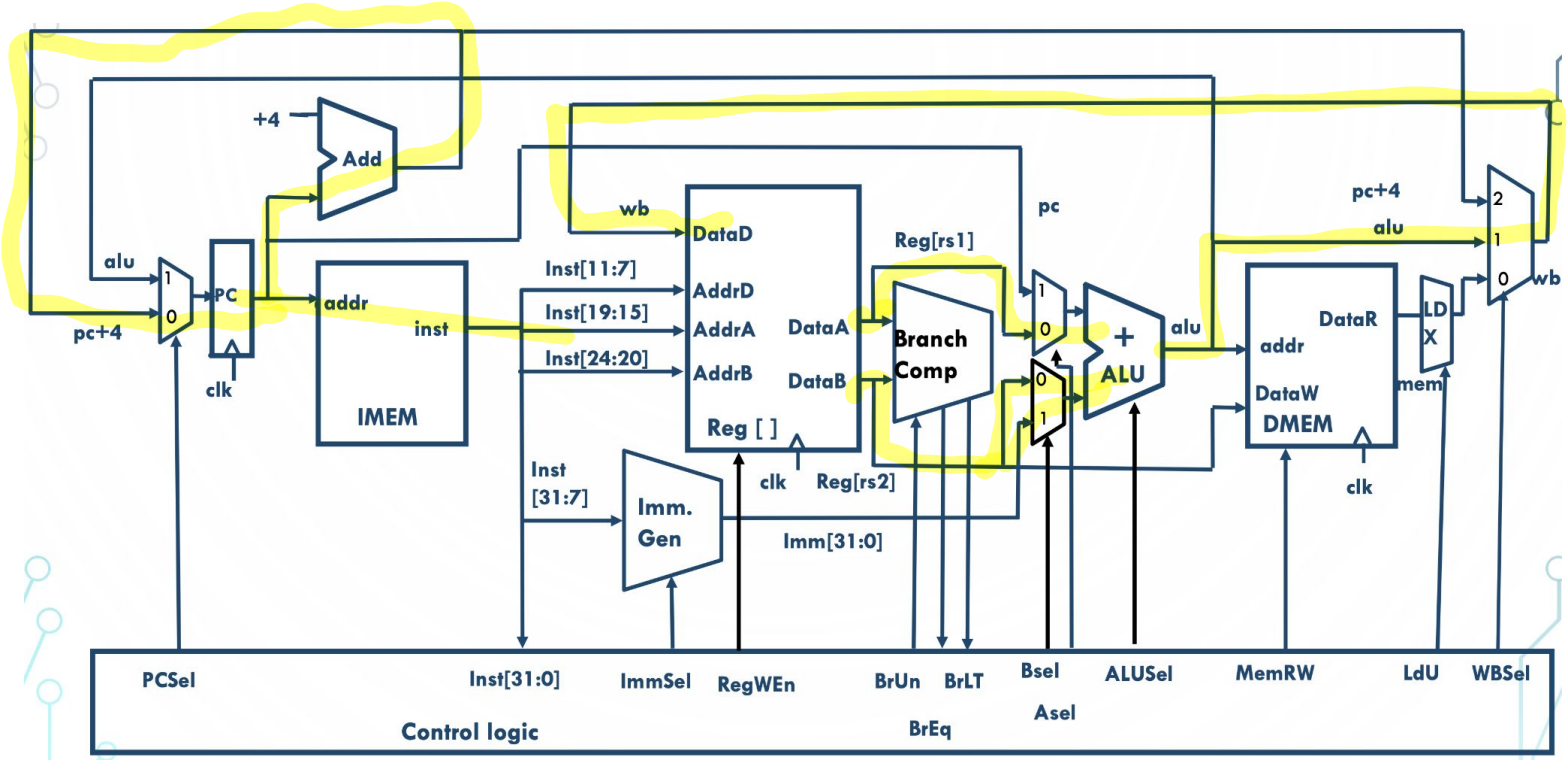
Pipeline Exercise

Beq



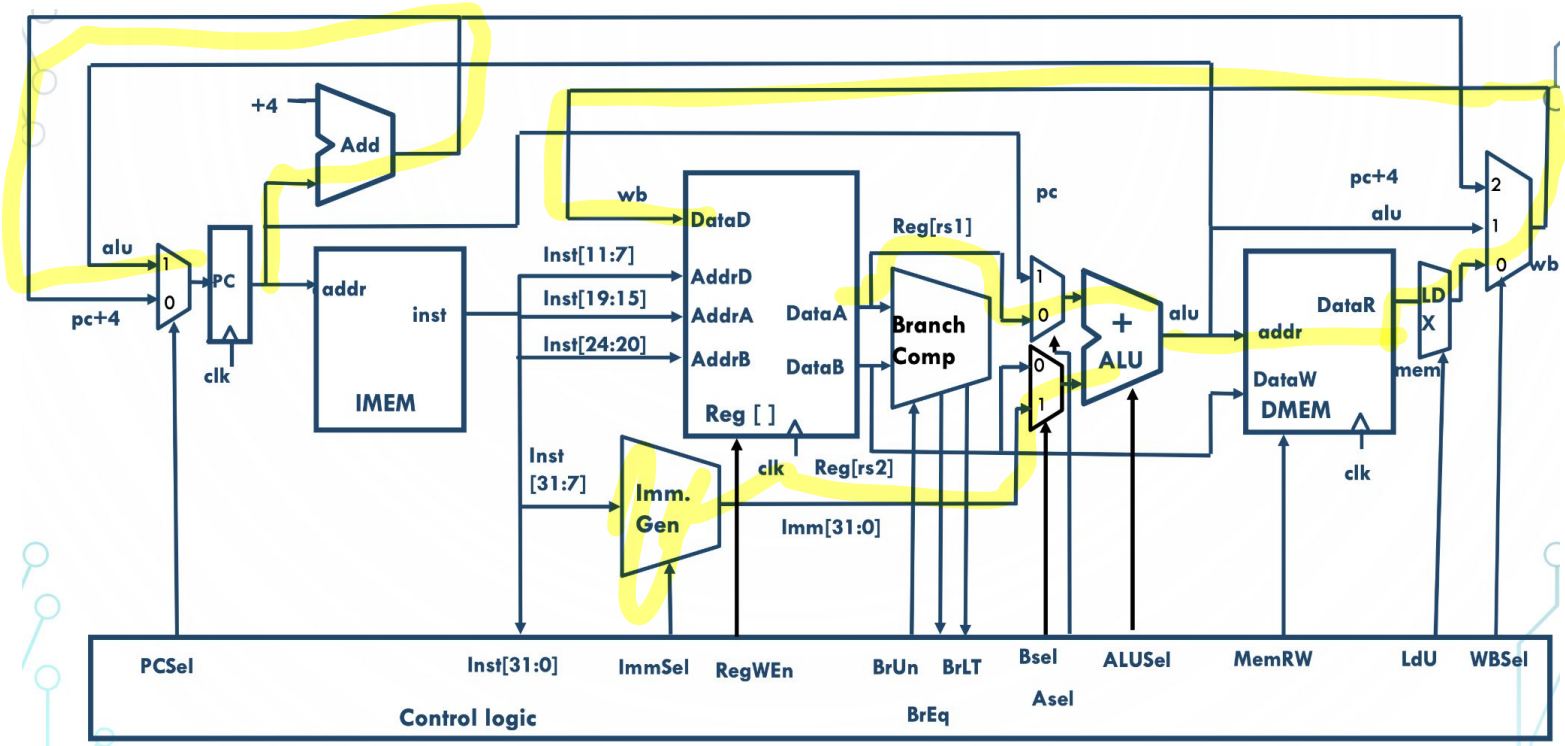
Pipeline Exercise

ADD



Pipeline Exercise

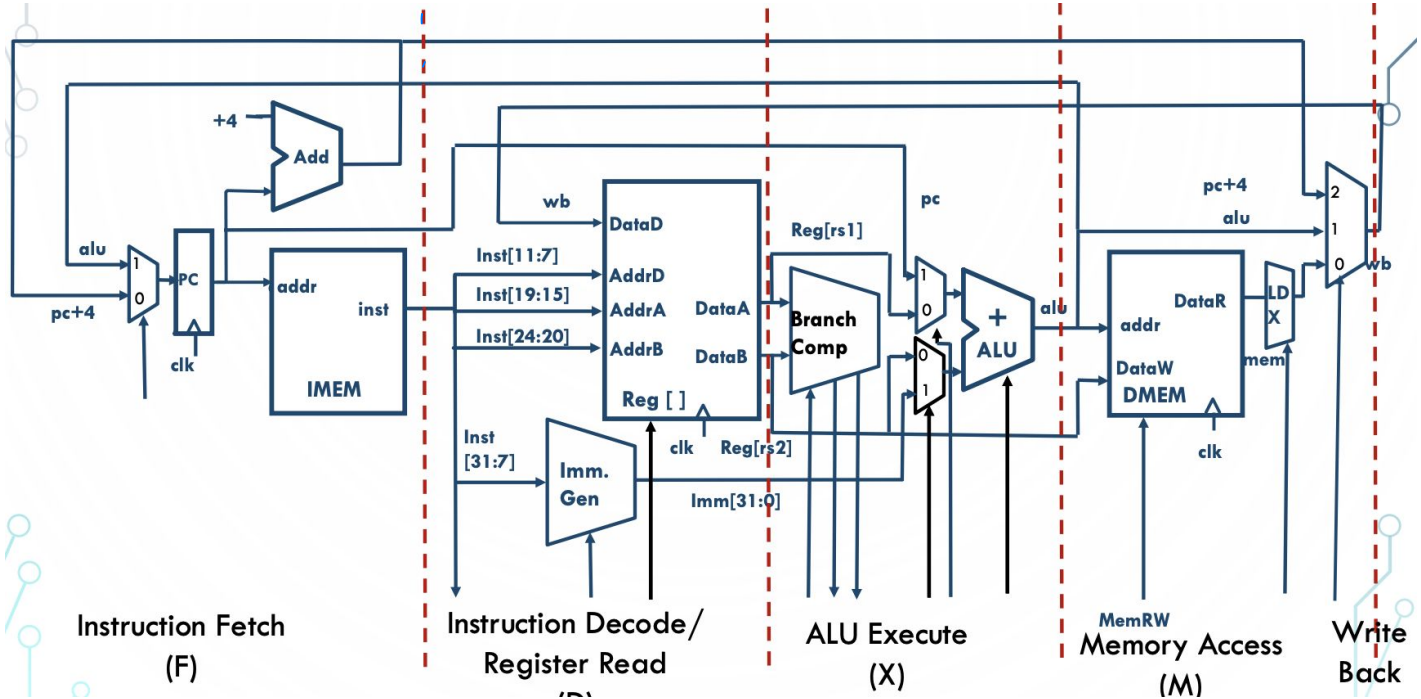
LW i-type



RISC-V Pipelining

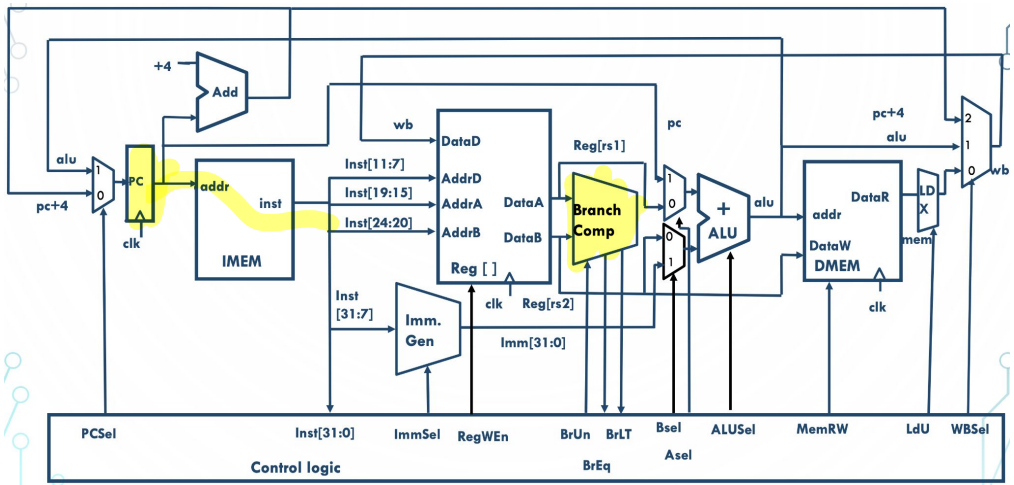
control signals?

- Why do we pipeline? Why not just single-cycle?



Hazards

- Pipelining is great! But what if…
  - in-flight instructions need the same functional unit?
  - in-flight instructions need the same data?
  - we need to check if we jump somewhere else?
- A hazard prevents instructions from being executed properly
- 3 main types
  - Structural
  - Data
  - Control



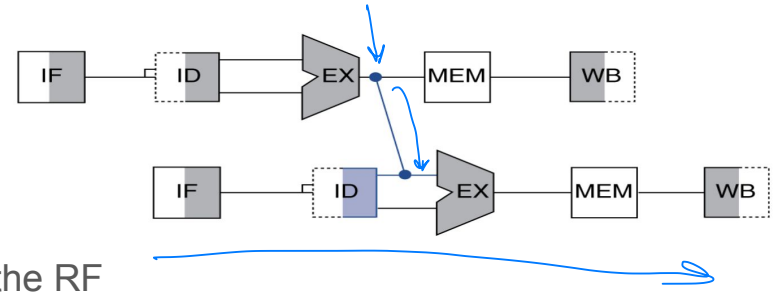
Hazard Types

- Structural
  - The resource that an instruction needs is busy
  - Eg. register file access, instruction & data memory
  - Generally avoided in RISC-V
- Data
  - Data dependency between instructions
  - Later instruction needs the output of a following instruction
    - But the pipeline stages!
  - Example?
- Control
  - Conditional branches - how do we know what instruction to execute?
  - Need execute stage to resolve?

ADD x1, x2, x3
SUB x2, x1, x3

Dealing with Hazards

- What's the easiest thing to do? Just wait!
  - stall/nop/bubble
  - But hurts CPI (cycles per instruction)
- Structural
  - Add more hardware
- Data
  - Writing and reading in the same cycle - speed of the RF
  - Forwarding
    - Adds additional logic to check dependencies; uses result directly
- Control
  - Guess one way, then fix if you're wrong!
    - Flush/kill instructions with control logic
  - Use branch prediction to reduce number of wrong guesses



Hazard Example

- Logic for fwd1 and fwd2?

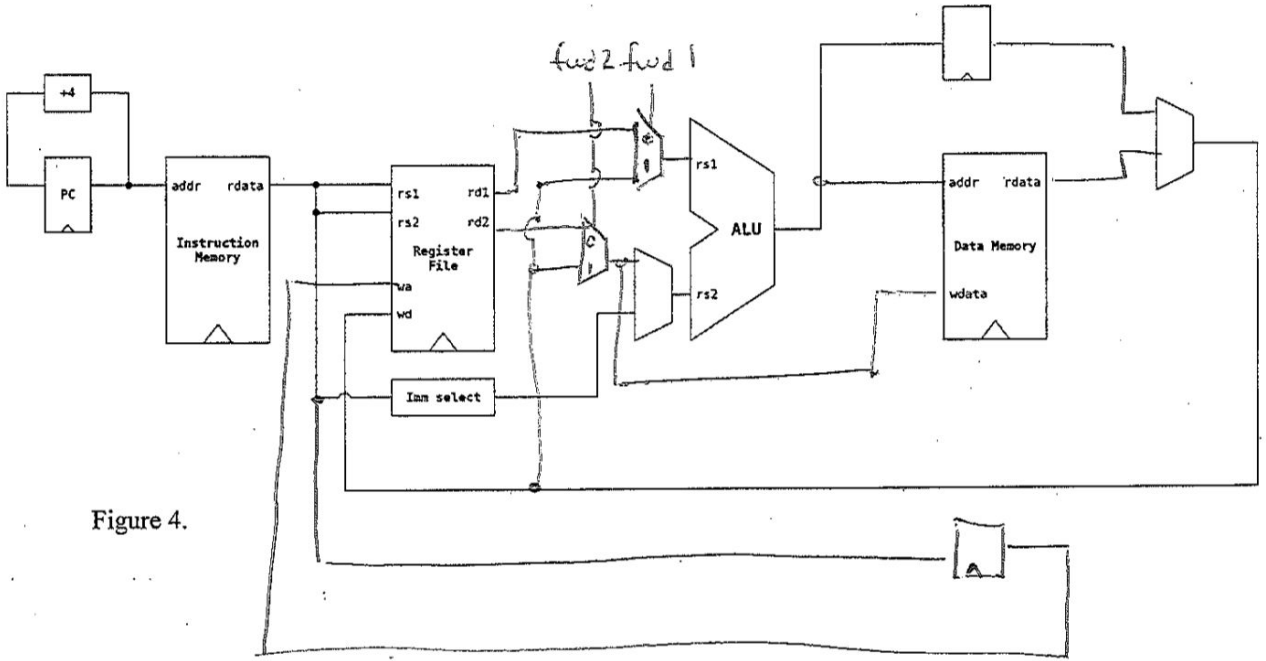


Figure 4.