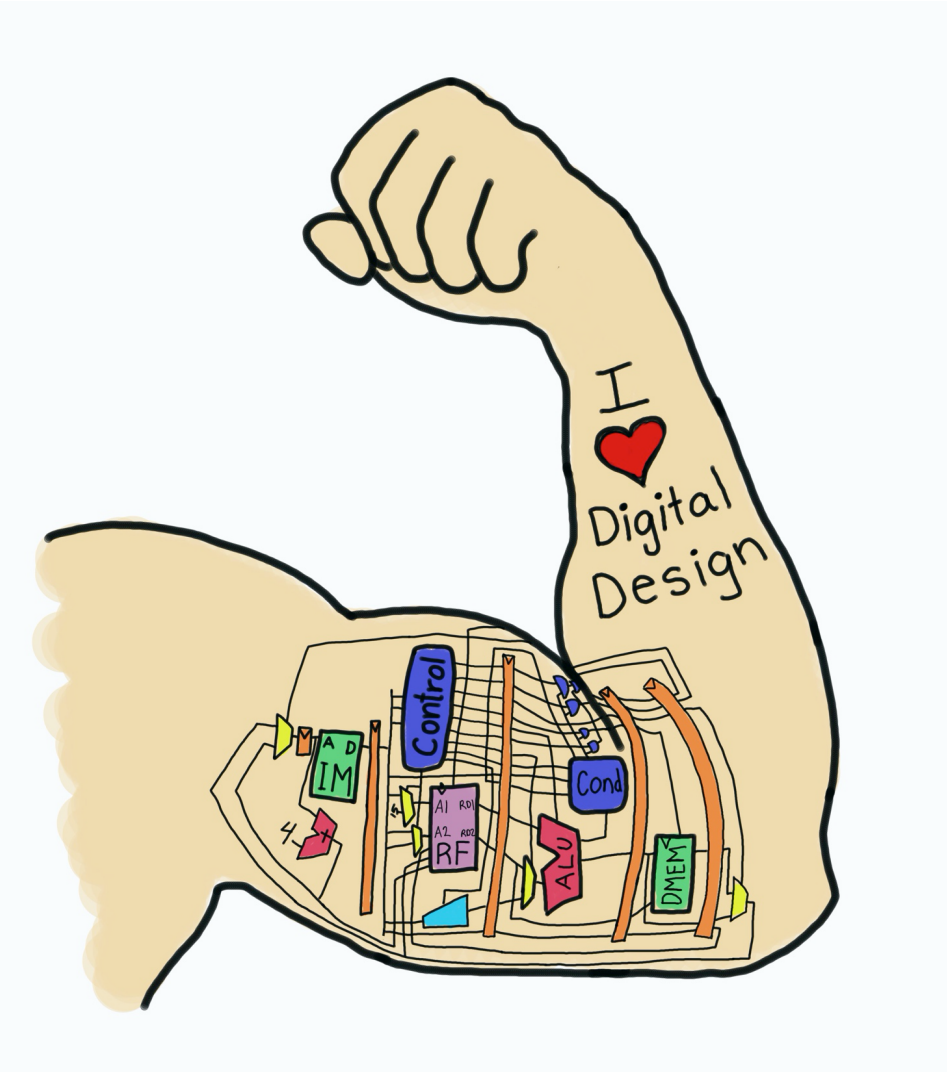
E85 Digital Design & Computer Engineering



Lecture 25:
Final Review

HARVEY MUDD COLLEGE

Course Overview

1st Half Semester

- **Logic Levels**
- **Number Systems**
- **CMOS Transistors**
- **Power Consumption**
- **Combinational Logic Design**
- **Finite State Machines**
- **Timing**
- **Verilog**
- **Arithmetic Circuits**

2nd Half Semester

- **Fixed & Floating Point**
- **Building Blocks**
- **C Programming**
- **Assembly Language**
- **Machine Language**
- **Microarchitecture**

First Half Semester

- See midterm review
- Make sure you can:
  - Design a finite state machine
  - Analyze its timing
  - Express it in Verilog
  - Estimate static and dynamic power

Fixed and Floating Point

Express -5.625 as 4.4 fixed point and float

Fixed:

5.625 =

-5.625 =

Float:

$-5.625 = (-1)(1.01101)(2^2)$

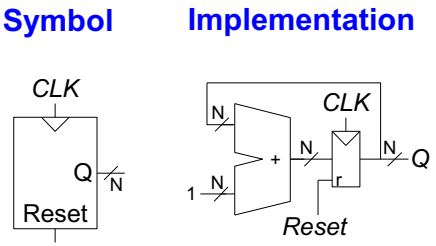
Sign = , biased exp = , fract =

= 0x

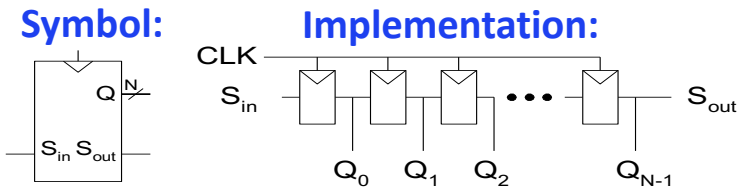
ELSEVIER

Building Blocks

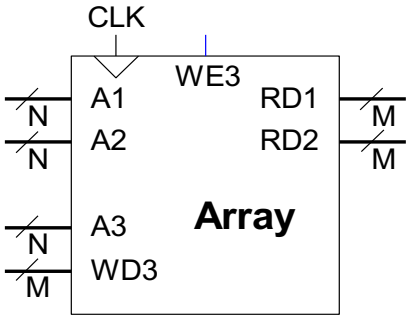
Counters



Shift Registers



Memories



ELSEVIER

FPGA Logic Elements

Each logic element has a 4-input LUT and one flop

Compute any function of up to 4 inputs

Ex:

y = a + b | (c ^ d)

7-input AND

4-bit shift register

3:8 decoder

FSM 3 bits state, 1 in, 2 out

C Programming

- Write basic programs in C
  - Data types & sizes (char, short, int, long, float, double)
  - If/else
  - For, while
  - Structures
  - Arrays
  - Pointers

Structure Example

```
typedef struct materialProps {
        double youngModulus;
        double hardness;
        double density;
        int    color[3];
        char name[36];
} materialProps;  // size = 8 + 8 + 8 + 3*4 + 36*1 = 72

materialProps myMaterials[100]; // base address 2000

double *dp = &myMaterials[2].density; //
```

ELSEVIER

RISC-V Assembly & Machine Language

Architectural State:

- 32 registers (x0 is 0)
- 32-bit program counter (PC)
- Up to 2^{32} bytes of memory

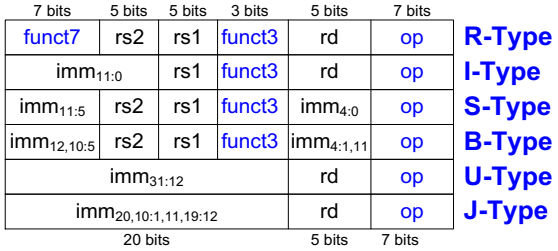
32-bit instructions: 6 formats

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
funct7	rs2	rs1	funct3	rd	op	**R-Type**
$imm_{11:0}$		rs1	funct3	rd	op	**I-Type**
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
$imm_{31:12}$				rd	op	**U-Type**
$imm_{20,10:1,11,19:12}$				rd	op	**J-Type**
20 bits				5 bits	7 bits	

ELSEVIER

RISC-V Instructions

op	f3	f7	Type	Instruction	Description	Operation
3	0	-	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
3	1	-	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
3	2	-	I	lw rd, imm(rs1)	load word	rd = [Address]
3	4	-	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
3	5	-	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
19	0	-	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
19	1	0	R	slli rd, rs1, shamt	shift left logical immediate	rd = rs1 << shamt
19	2	-	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
19	3	-	I	sltiu rd, rs1, imm	set less than immediate unsigned	rd = (rs1 < SignExt(imm))
19	4	-	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
19	5	0	R	srli rd, rs1, shamt	shift right logical immediate	rd = rs1 >> shamt
19	5	32	R	srai rd, rs1, shamt	shift right arithmetic immediate	rd = rs1 >>> shamt
19	6	-	I	ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
19	7	-	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
23	-	-	U	auipc rd, imm	add upper immediate to PC	rd = {imm$_{31:12}$, 12'b0} + PC
35	0	-	S	sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
35	1	-	S	sh rs2, imm(rs1)	store half	[Address]$_{15:0}$ = rs2$_{15:0}$
35	2	-	S	sw rs2, imm(rs1)	store word	[Address] = rs2
51	0	0	R	add rd, rs1, rs2	add	rd = rs1 + rs2
51	0	32	R	sub rd, rs1, rs2	sub	rd = rs1 - rs2
51	1	0	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2$_{4:0}$
51	2	0	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
51	3	0	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
51	4	0	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
51	5	0	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2
51	5	32	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2
51	6	0	R	or rd, rs1, rs2	or	rd = rs1 | rs2
51	7	0	R	and rd, rs1, rs2	and	rd = rs1 & rs2
55	-	-	U	lui rd, imm	load upper immediate	rd = {imm$_{31:12}$, 12'b0}
99	0	-	B	beq rs1, rs2, label	branch if =	if (rs1==rs2) PC = BTA
99	1	-	B	bne rs1, rs2, label	branch if !=	if (rs1!=rs2) PC = BTA
99	4	-	B	blt rs1, rs2, label	branch if <	if (rs1< rs2) PC = BTA
99	5	-	B	bge rs1, rs2, label	branch if ≥	if (rs1>=rs2) PC = BTA
99	6	-	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1< rs2) PC = BTA
99	7	-	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1>=rs2) PC = BTA
103	0	-	I	jalr rd, rs1, imm	jump and link register	rd = PC + 4, PC = rs1 + SignExt(imm)
111	-	-	J	jal rd, label	jump and link	rd = PC + 4, PC = JTA



ELSEVIER

Assembly Language Programming

```
void dotproduct (int v1[], int v2[], int len) {
      int i, sum = 0;
      for (i=0; i<len; i++)
            sum += v1[i] * v2[i];
      return sum;
}

void main(void) {
      int a[3] = {1, 2, 3};
      int b[3] = {4, 6, 6};
      int d;
      d = dotproduct(a, b, 3);
}
```

ELSEVIER

Assembly Language Programming

```
// v1 in a0, v2 in a1, len in a2
// i in s0, sum in s1
dotproduct
        addi sp, sp, -8          // room to save registers on stack
        sw s0, 0(sp)             // save s0
        sw s1, 4(sp)             // save s1
        addi s1, zero, 0         // sum = 0
        addi s0, zero, 0         // i = 0;
for
        bge s0, a2, done         // i < len?
        slli t0, s0, 2           // i * 4
        add t1, s0, t0           // address of v1[i]
        lw t1, 0(t1)             // v1[i]
        add t2, s0, t0           // address of v2[i]
        lw t2, 0(t2)             // v2[i]
        mul t3, t1, t2           // v1[i] * v2[i]
        add s1, s1, t3           // sum += v1[i] * v2[i]
        addi s0, s0, 1           // i++
        j for                    // repeat for loop
done
        mv a0, s1                // return sum
        lw s0, 0(sp)             // restore s0
        lw s1, 4(sp)             // restore s1
        addi sp, sp, 8           // restore stack pointer
        jr ra                    // and go back to caller
```

ELSEVIER

Assembly to Machine Language

```
xor x3, x8, x9

funct7 =
rs2    =
rs1    =
funct3 =
rd     =
op     =
```

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
funct7	rs2	rs1	funct3	rd	op	**R-Type**
$imm_{11:0}$		rs1	funct3	rd	op	**I-Type**
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
$imm_{31:12}$				rd	op	**U-Type**
$imm_{20,10:1,11,19:12}$				rd	op	**J-Type**
20 bits				5 bits	7 bits	

ELSEVIER

Machine to Assembly Language

0x00A5A823

Based on the bottom 7 bits, op = 35 (S-type)

Interpret the other bits according to S-type

0000000 01010 01011 010 10000 0100011

$Imm_{11:5}$ =

rs2 =

rs1 =

funct3 =

$Imm_{4:0}$ =

op =

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
funct7	rs2	rs1	funct3	rd	op	**R-Type**
$imm_{11:0}$		rs1	funct3	rd	op	**I-Type**
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
$imm_{31:12}$				rd	op	**U-Type**
$imm_{20,10:1,11,19:12}$				rd	op	**J-Type**
20 bits				5 bits	7 bits	

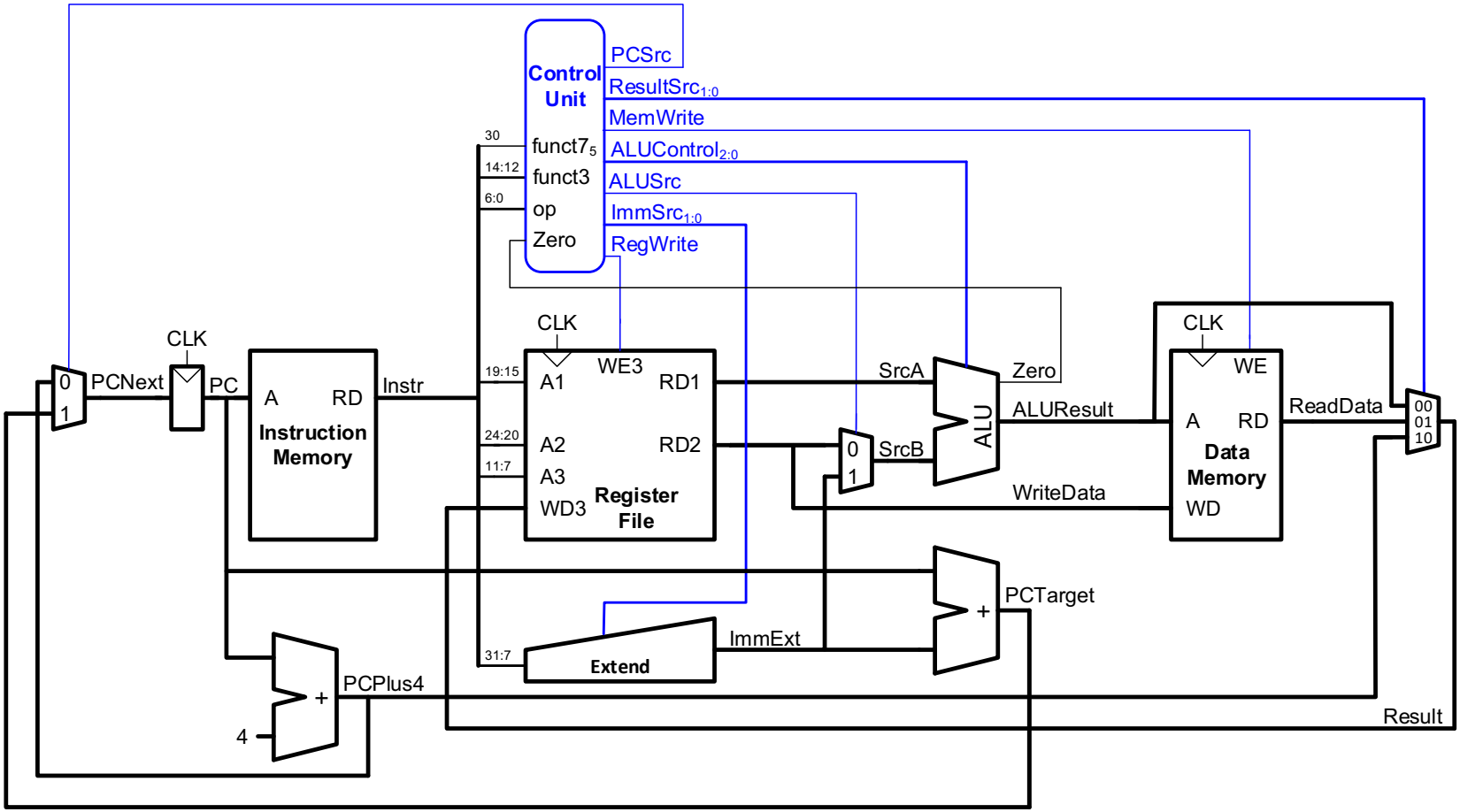
ELSEVIER

Microarchitecture

- Three Microarchitectures
  - Single-cycle
  - Multicycle
  - Pipelined
- Be able to add new features, such as new instructions
  - Examples in the book/lecture and homework
- Determine cycle time and performance
- Don’t need to memorize diagrams or tables

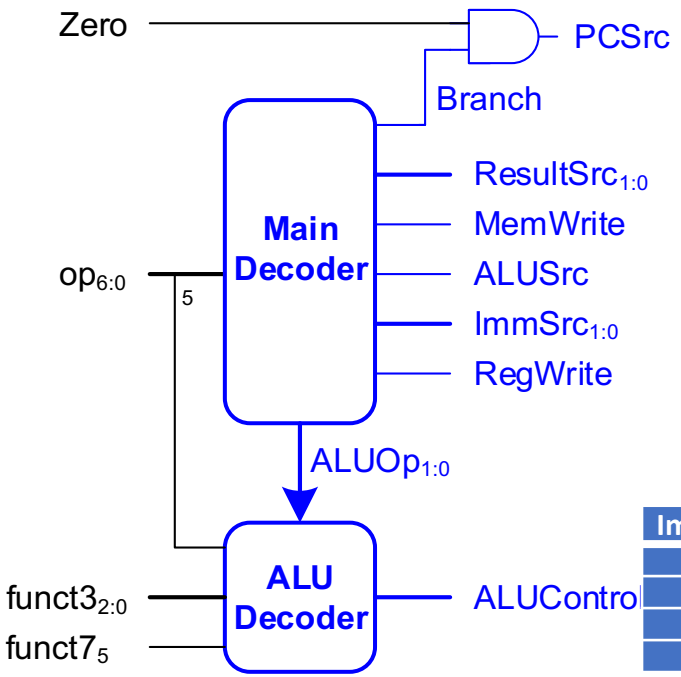
ELSEVIER

Single Cycle Processor



ELSEVIER

Single Cycle Controller



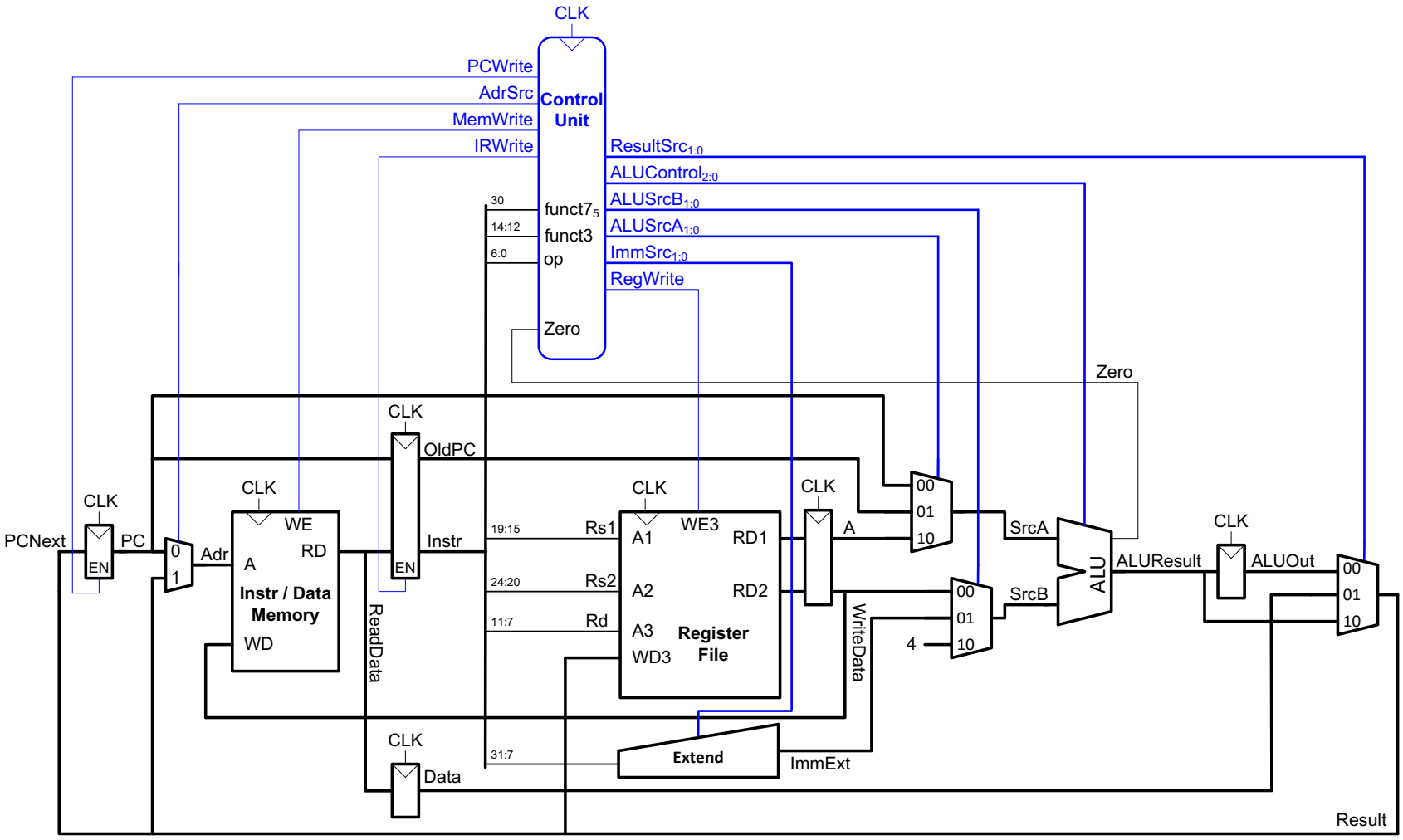
ALUOp	funct3	op_5, $funct7_5$	ALUControl	Instruction
00	x	xx	010 (add)	lw, sw
01	x	xx	110 (subtract)	beq
10	000	00, 01, 10	010 (add)	add
10	000	11	110 (subtract)	sub
10	010	xx	111 (set less than)	slt
10	110	xx	001 (or)	or
10	111	xx	000 (and)	and

ImmSrc	ImmExt	Type	Description
00	{{20{Instr[31]}}, Instr[31:20]}	I	12-bit signed immediate
01	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	S	12-bit signed immediate
10	{{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}	B	13-bit signed immediate
11	{{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0}	J	21-bit signed immediate

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
lw	0000011	1	00	1	0	01	0	00	0
sw	0100011	0	01	1	1	xx	0	00	0
R-type	0110011	1	xx	0	0	00	0	10	0
beq	1100011	0	10	0	0	xx	1	01	0
addi	0010011	1	00	0	0	00	0	10	0
jal	1101111	1	11	x	0	10	0	xx	1

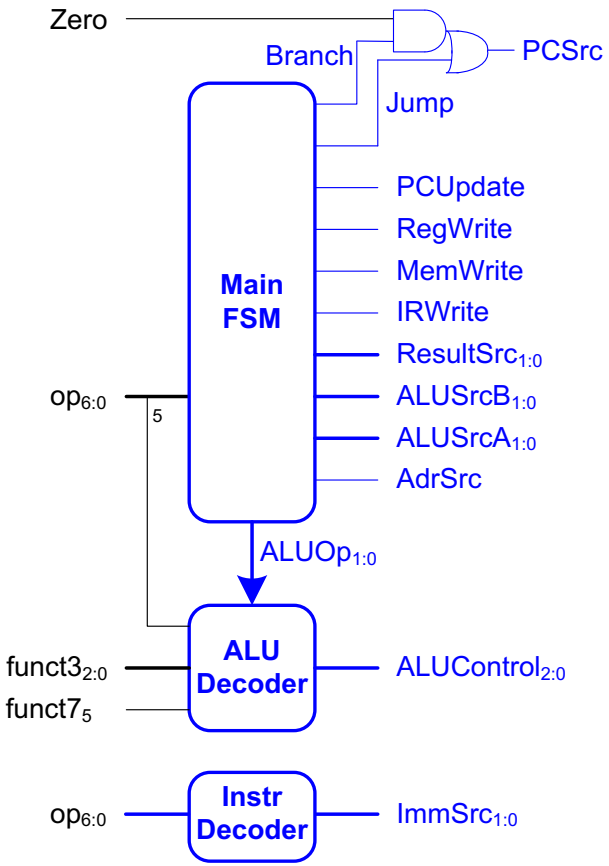
ELSEVIER

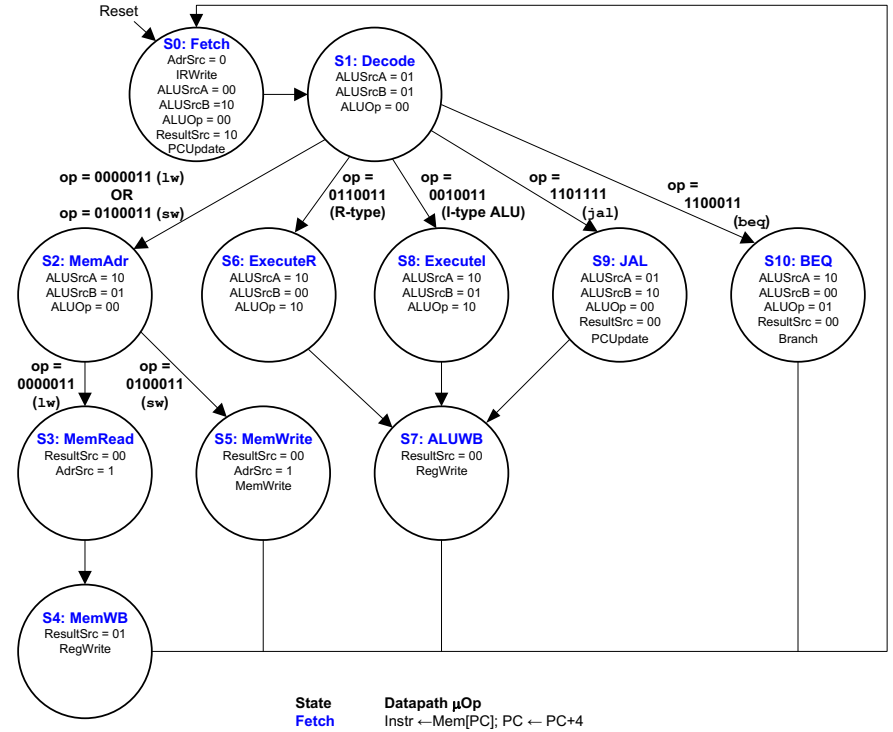
Multicycle Processor



ELSEVIER

Multicycle Controller

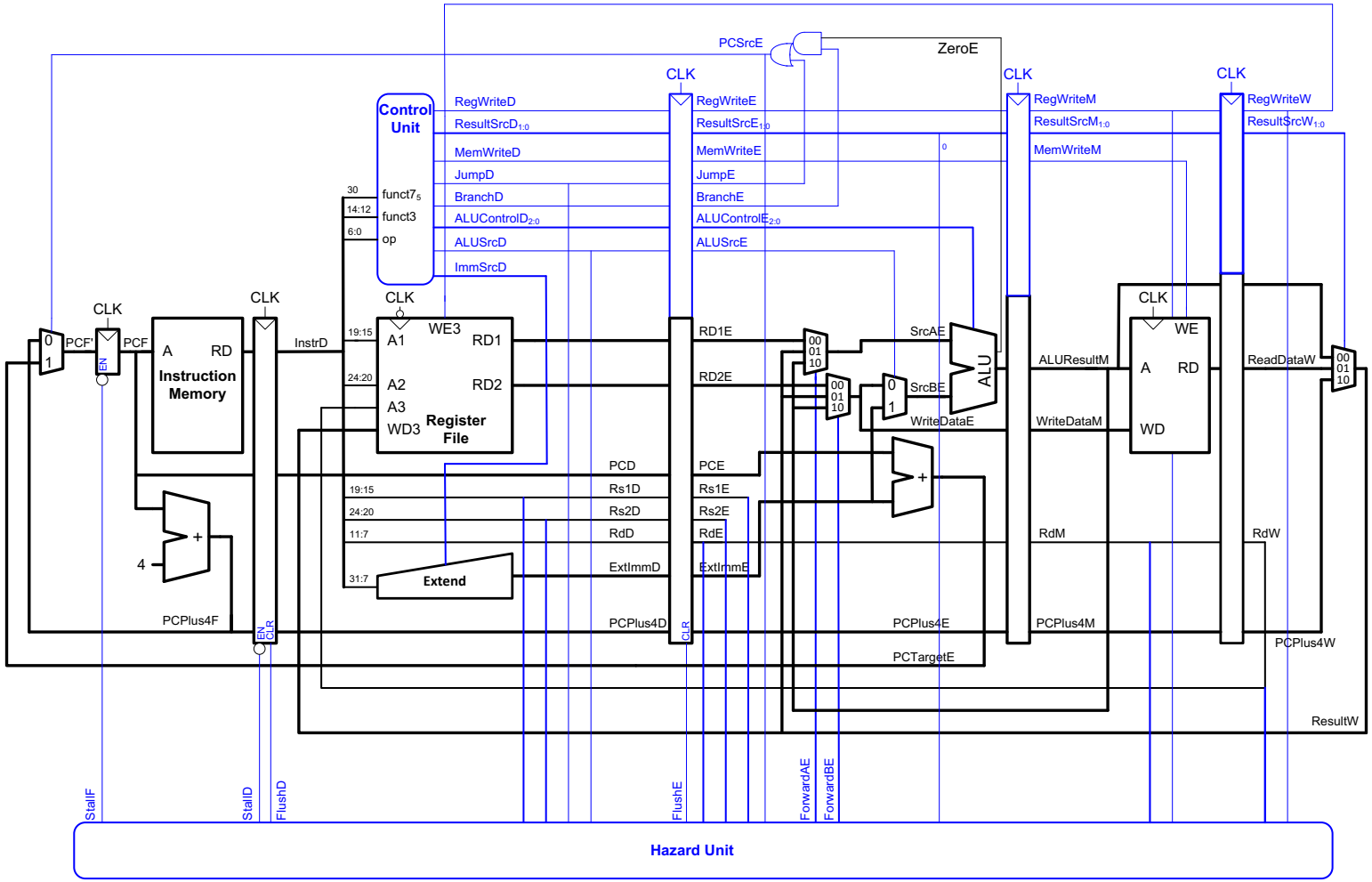




State	Datapath µOp
Fetch	Instr ←Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1 op rs2
ExecuteI	ALUOut ← rs1 op imm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4

ELSEVIER

Pipelined Processor



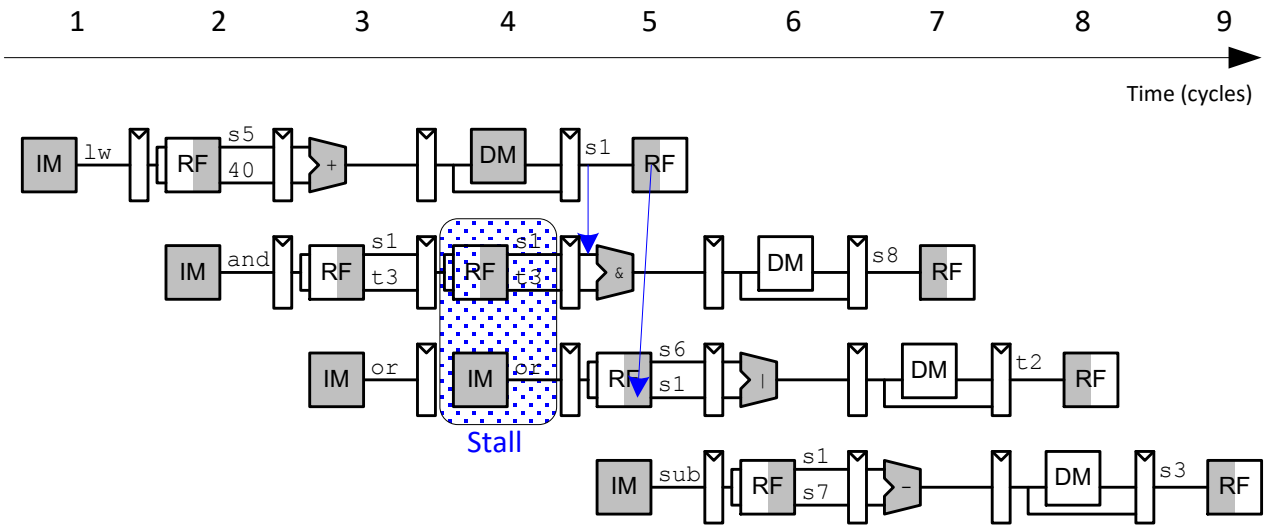
ELSEVIER

Pipeline Hazards

- Data hazards
  - Instruction depends on result of previous instruction
  - Handled by forwarding
  - Most data hazards have zero latency
  - 1-cycle stall between load and use
- Control hazards
  - What instruction to fetch after branch
  - Predict branch not taken, keep fetching from PC+4
  - Flush two instructions if branch is actually taken

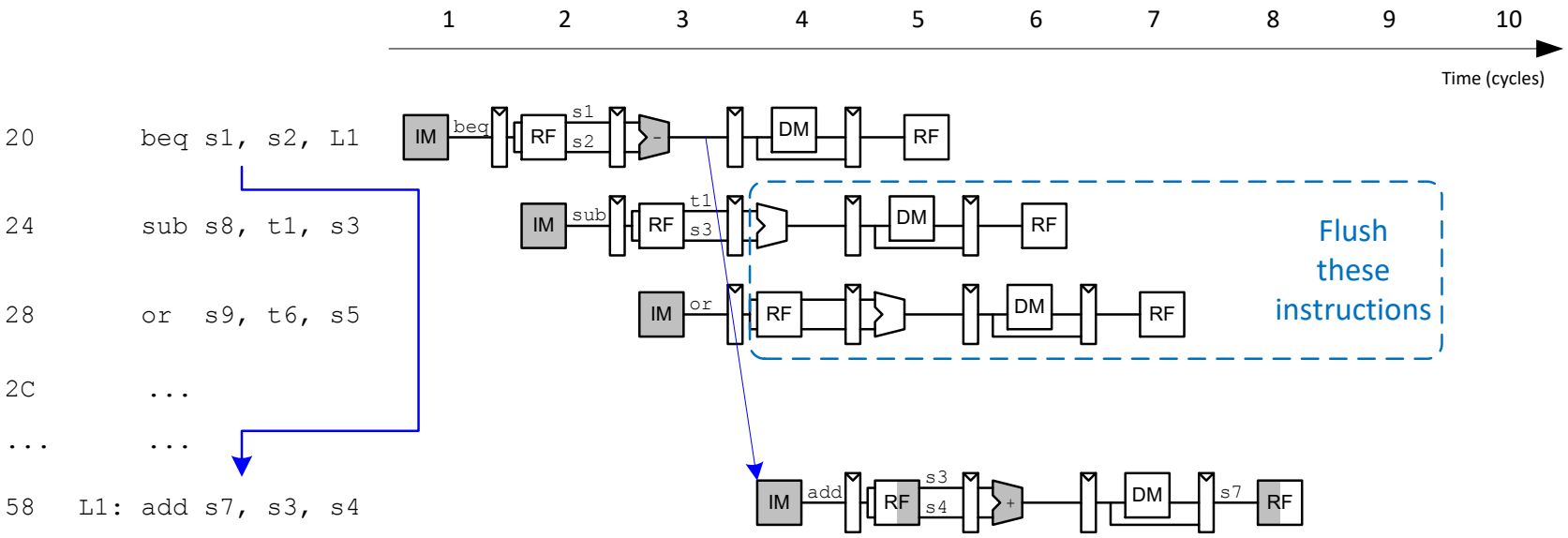
Data Hazards

```
lw   s1, 40(s5)
and  s8, s1, t3
or   t2, s6, s1
sub  s3, s1, s7
```



ELSEVIER

Control Hazards



ELSEVIER