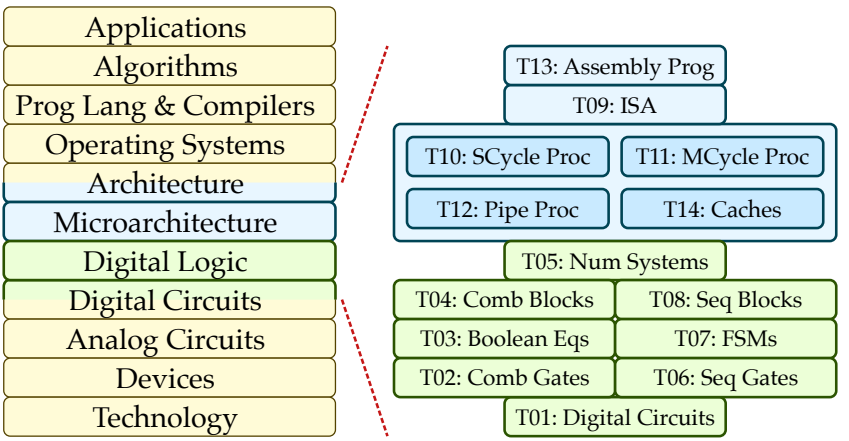
ECE 2300 Digital Logic and Computer Organization Fall 2024

Topic 10: Single-Cycle Processors

School of Electrical and Computer Engineering
Cornell University

revision: 2024-10-31-10-41

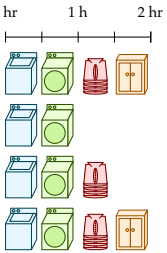


1. High-Level Idea for Single-Cycle Processors

Transaction Steps

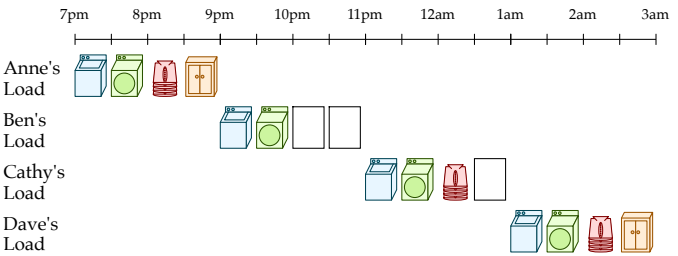
Washing (30 min)
Drying (30 min)
Folding (30 min)
Storing (30 min)

Four Types of Transactions

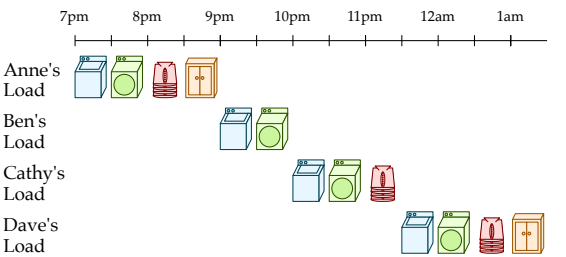


	Transaction Latency	
Anne's Load	2.0 hr	Anne requires all four steps
Ben's Load	1.0 hr	Ben is messy, leaves unfolded clothes in his laundry basket
Cathy's Load	1.5 hr	Cathy does not have a bureau, leaves folded clothes in basket
Dave's Load	2.0 hr	Dave requires all four steps

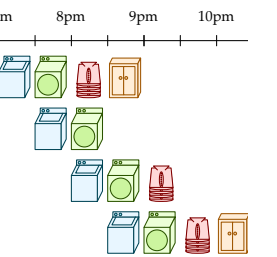
Fixed Time Slot Laundry (Single-Cycle Processors)



Variable Time Slot Laundry (Multi-Cycle Processors)



Pipelined Laundry



1.1. Transactions and Steps

- We can think of each instruction as a transaction
- Executing a transaction involves a sequence of steps

ADD
`add rd, rs1, rs2`
R[rd] ← R[rs1] + R[rs2]
PC ← PC + 4

ADDI
`addi rd, rs1, imm`
R[rd] ← R[rs1] + sext(imm)
PC ← PC + 4

MUL
`mul rd, rs1, rs2`
R[rd] ← R[rs1] × R[rs2]
PC ← PC + 4

LW
`lw rd, imm(rs1)`
R[rd] ← M[R[rs1] + sext(imm)]
PC ← PC + 4

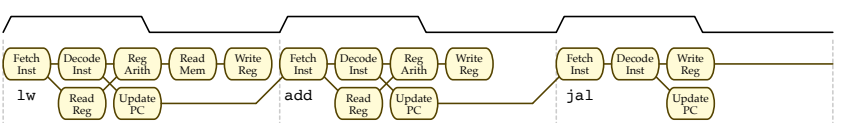
SW
`sw rs2, imm(rs1)`
M[R[rs1] + sext(imm)] ← R[rs2]
PC ← PC + 4

JAL
`jal rd, imm`
R[rd] ← PC + 4
PC ← PC + sext(imm)

JR
`jr rs1`
PC ← R[rs1]

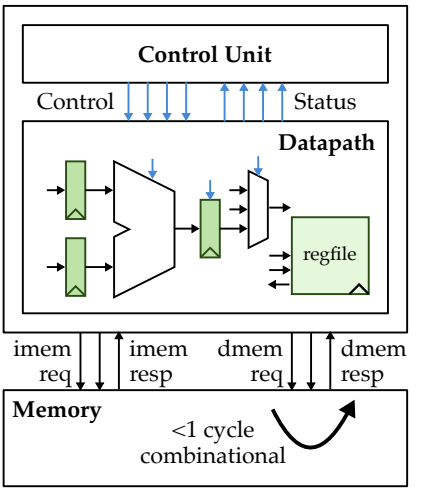
BNE
`bne rs1, rs2, imm`
if (R[rs1] != R[rs2]) PC ← PC + sext(imm)
else PC ← PC + 4

	add	addi	mul	lw	sw	jal	jr	bne
Fetch Instruction								
Decode Instruction								
Read Registers								
Register Arithmetic								
Read Memory								
Write Memory								
Write Registers								
Update PC								



1.2. Technology Constraints

- Assume technology where logic is not too expensive, so we do not need to overly minimize the number of registers and combinational logic
- Assume multi-ported register file with a reasonable number of ports is feasible
- Assume a dual-ported combinational memory



1.3. First-Order Performance Equation

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- Instructions / program depends on source code, compiler, ISA
- Avg cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

Microarchitecture	CPI	Cycle Time
Single-Cycle Processor	1	long
Multi-Cycle Processor	>1	short
Pipelined Processor	≈1	short

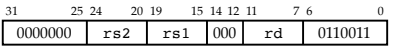
2. Single-Cycle Processor Datapath

ADD

`add rd, rs1, rs2`

R[rd] ← R[rs1] + R[rs2]

PC ← PC + 4

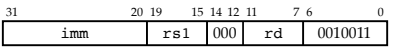


ADDI

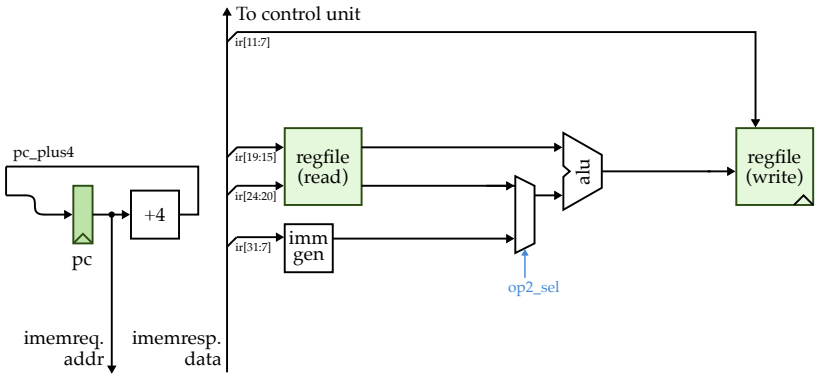
`addi rd, rs1, imm`

R[rd] ← R[rs1] + sext(imm)

PC ← PC + 4



Implementing ADD and ADDI Instructions

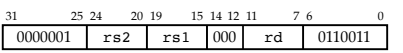


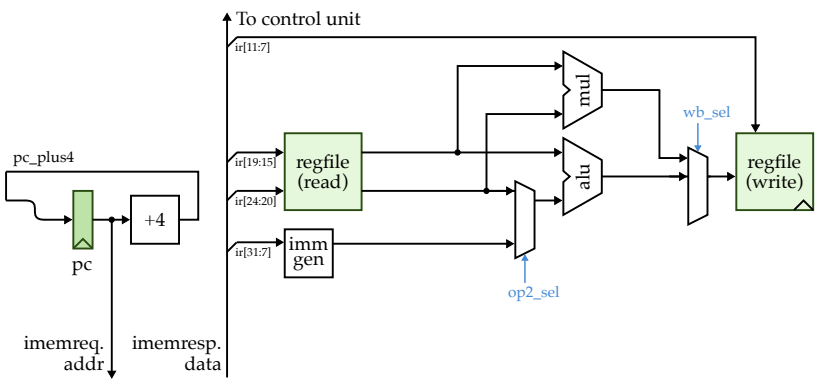
MUL

`mul rd, rs1, rs2`

R[`rd`] ← R[`rs1`] × R[`rs2`]

PC ← PC + 4



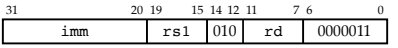


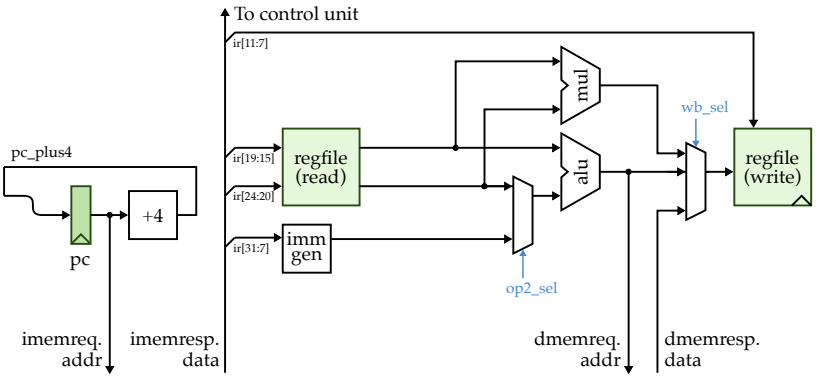
LW

`lw rd, imm(rs1)`

R[`rd`] ← M[R[`rs1`] + sext(`imm`)]

PC ← PC + 4



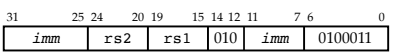


SW

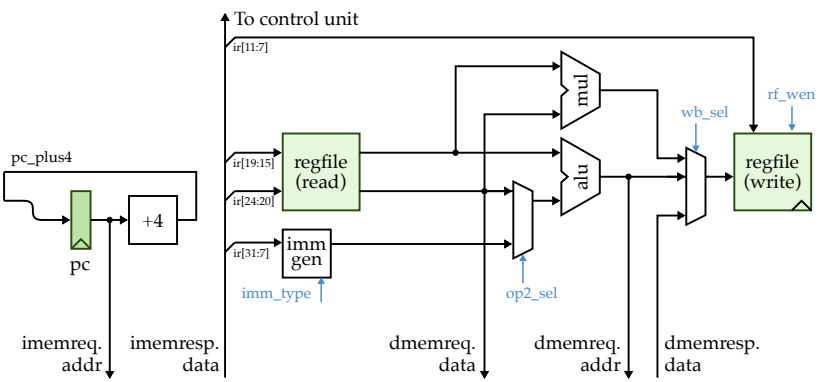
`sw rs2, imm(rs1)`

M[R[`rs1`] + sext(`imm`)] ← R[`rs2`]

PC ← PC + 4



`imm = { inst[31:25], inst[11:7] }`

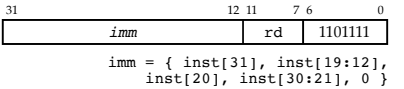


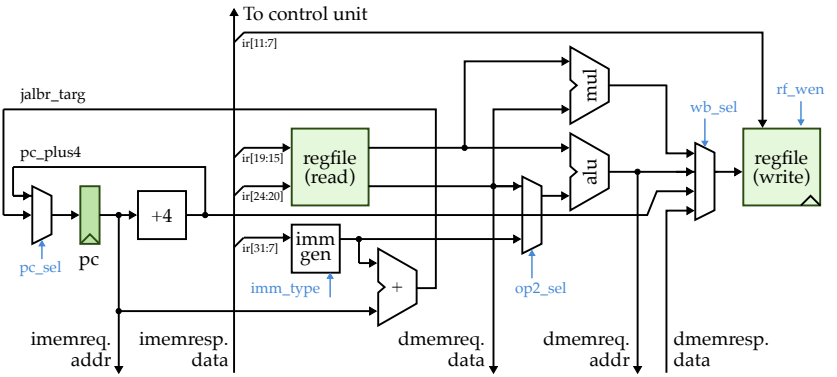
JAL

`jal rd, imm`

R[`rd`] ← PC + 4

PC ← PC + sext(`imm`)

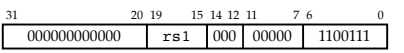


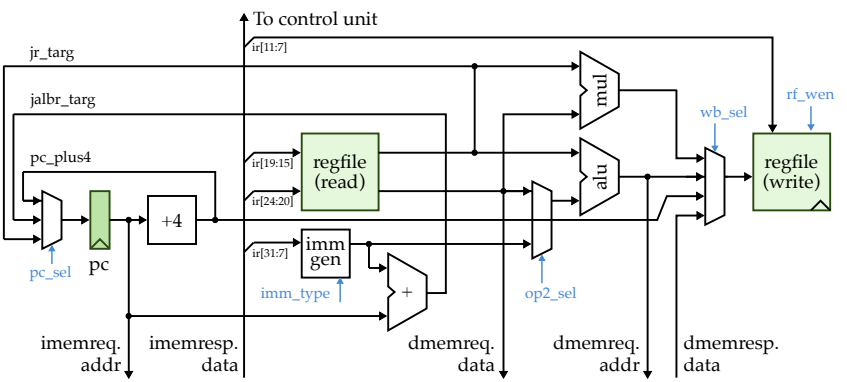


JR

`jr rs1`

PC ← R[`rs1`]

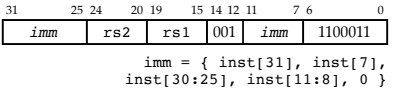


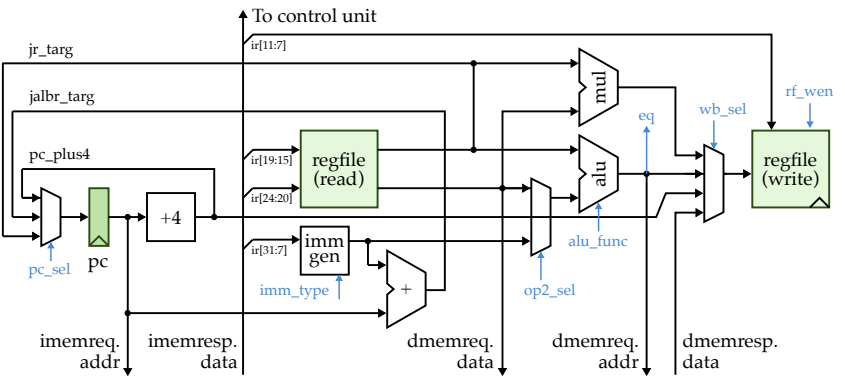


BNE

`bne rs1, rs2, imm`

if (R[`rs1`] != R[`rs2`]) PC ← PC + sext(`imm`)
else PC ← PC + 4





- List of control signals (from control unit to datapath)
  - pc_sel
  - imm_type
  - op2_sel
  - alu_func
  - wb_sel
  - rf_wen
  - imm_req_val
  - dmm_req_val
- List of status signals (from datapath to control unit)
  - eq

Adding a New Auto-Incrementing Load Instruction

Draw on the datapath diagram what paths we need to use as well as any new paths we will need to add in order to implement the following auto-incrementing load instruction.

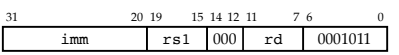
LW.AI

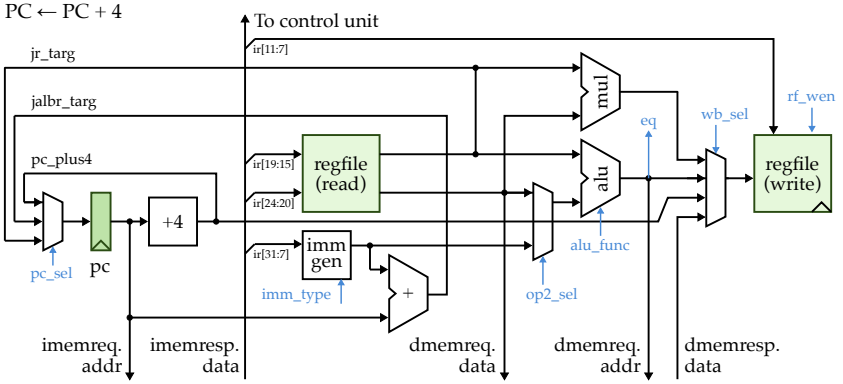
`lw.ai rd, imm(rs1)`

R[`rd`] ← M[R[`rs1`] + sext(`imm`)]

R[`rs1`] ← R[`rs1`] + 4

PC ← PC + 4





3. Single-Cycle Processor Control Unit

inst	pc sel	imm type	op2 sel	alu func	wb sel	rf wen	imem req val	dmem req val
`add`	pc+4	x	rf	+	alu	1	1	0
`addi`								
`mul`	pc+4	x	x	x	mul	1	1	0
`lw`	pc+4	i	imm	+	mem	1	1	1
`sw`								
`jal`								
`jr`	jr	x	x	x	x	0	1	0
`bne`								

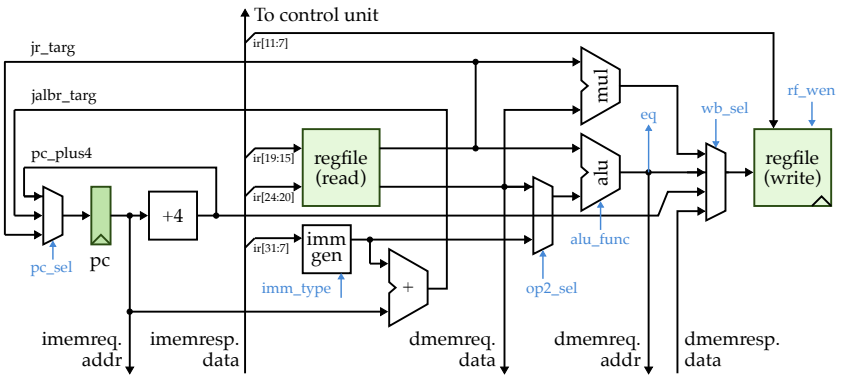
Need to factor `eq` status signal into `pc_sel` signal for BNE!

4. Analyzing Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycles}}$$

- Instructions / program depends on source code, compiler, ISA
- Cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

Estimating minimum clock period (cycle time)



	t_{pd}
32-bit 2-to-1 Mux	4τ
32-bit 4-to-1 Mux	8τ
32-bit Adder	60τ
32-bit ALU	64τ
32-bit Multiplier	100τ
32-bit +4 Unit	30τ
ImmGen Unit	12τ
32-bit Reg Clk-to-Q	9τ
32-bit Reg Setup	10τ
Register File Read	25τ
Register File Setup	20τ
Memory Read	120τ
Memory Setup	120τ

Estimating execution time

How long in units of τ will it take to execute the vector-vector add program assuming `n` is 64?

Pseudo-Code

```
for i in range(n):
  dest[i] = src0[i] + src1[i]
```

Assembly Code

```
 # addr(dest[i]):x1, addr(src0[i]):x2
 # addr(src1[i]):x3, n:x4
loop:
 lw    x5, 0(x1)
 lw    x6, 0(x2)
 add   x7, x5, x6
 sw    x7, 0(x3)
 addi  x1, x1, 4
 addi  x2, x2, 4
 addi  x3, x3, 4
 addi  x4, x4, -1
 bne   x4, x0, loop
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
`lw x5, 0(x1)`														
`lw x6, 0(x2)`														
`add x7, x5, x6`														
`sw x7, 0(x3)`														
`addi x1, x1, 4`														
`addi x2, x2, 4`														
`addi x3, x3, 4`														
`addi x4, x4, -1`														
`bne x4, x0, loop`														
`lw x5, 0(x1)`														
`lw x6, 0(x2)`														
`add x7, x5, x6`														
`sw x7, 0(x3)`														

How long in units of τ will it take to execute the find program assuming `n` is 64 and only the first element matches the given value.

Pseudo-Code

```
found = 0
for i in range(n):
  if ( src0[i] == value )
    found = 1
```

Assembly Code

```
 # addr(src0[i]):x1, n:x2, value:x3
 addi  x5, x0, 0

loop:
 lw    x4, 0(x1)
 bne   x4, x3, neq
 addi  x5, x0, 1

neq:
 addi  x1, x1, 4
 addi  x2, x2, -1
 bne   x2, x0, loop
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
`addi x5, x0, 0`														
`lw x4, 0(x1)`														
`bne x4, x3, neq`														
`addi x5, x0, 1`														
`addi x1, x1, 4`														
`addi x2, x2, -1`														
`bne x2, x0, loop`														
`lw x4, 0(x1)`														
`bne x4, x3, neq`														
`addi x1, x1, 4`														
`addi x2, x2, -1`														
`bne x2, x0, loop`														