ECE 2300 Digital Logic and Computer Organization Fall 2026

Topic 11: Multi-Cycle Processors

School of Electrical and Computer Engineering
Cornell University

revision: 2026-11-17-10-48

1 High-Level Idea for Multi-Cycle Processors **3**

1.1. Transactions and Steps 4

1.2. Technology and System Constraints 5

1.3. First-Order Performance Equation 7

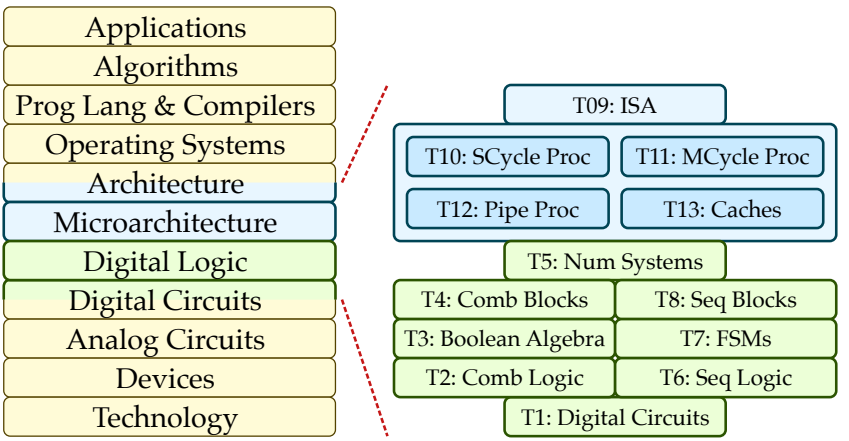
2 15-State Multi-Cycle Processor Microarchitecture **8**

2.1. Analyzing Delay and Area 15

3 62-State Multi-Cycle Processor Microarchitecture **19**

3.1. Analyzing Performance 30

4 TinyRV1 Embedded System **34**



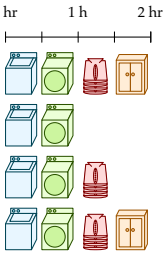
Copyright © 2026 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2300 / ENGRD 2300 Digital Logic and Computer Organization. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

1. High-Level Idea for Multi-Cycle Processors

Transaction Steps

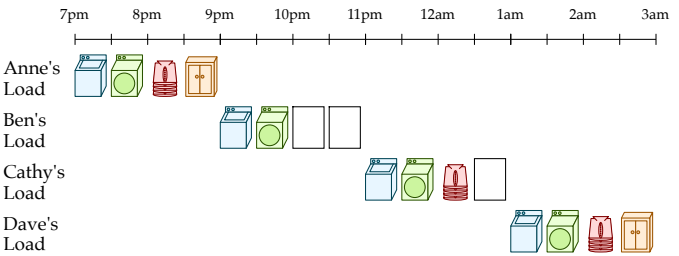


Four Types of Transactions

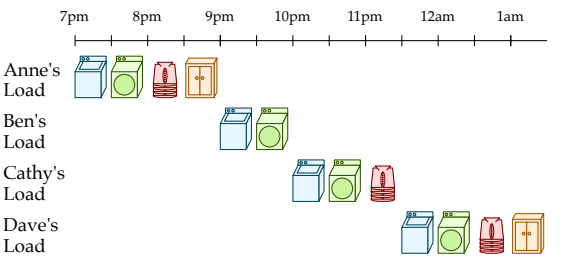


	Transaction Latency	
Anne's Load	2.0 hr	Anne requires all four steps
Ben's Load	1.0 hr	Ben is messy, leaves unfolded clothes in his laundry basket
Cathy's Load	1.5 hr	Cathy does not have a bureau, leaves folded clothes in basket
Dave's Load	2.0 hr	Dave requires all four steps

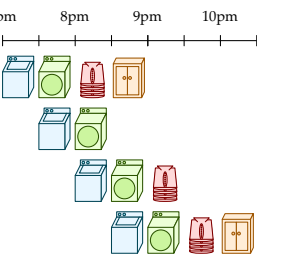
Fixed Time Slot Laundry (Single-Cycle Processors)



Variable Time Slot Laundry (Multi-Cycle Processors)



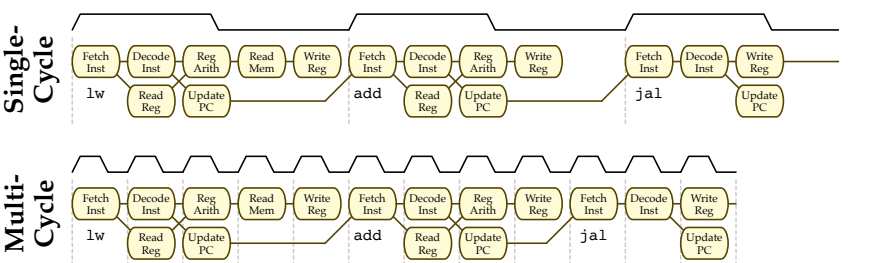
Pipelined Laundry



1.1. Transactions and Steps

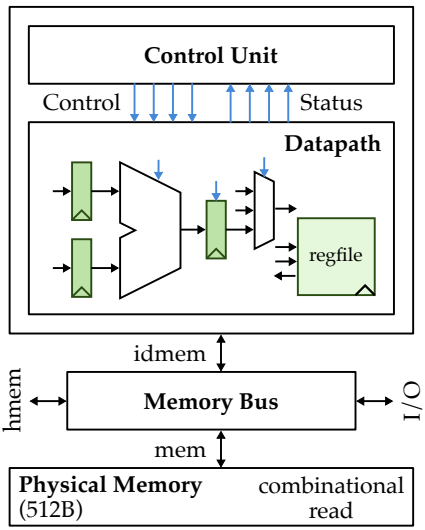
- We can think of each instruction as a transaction
- Executing a transaction involves a sequence of steps

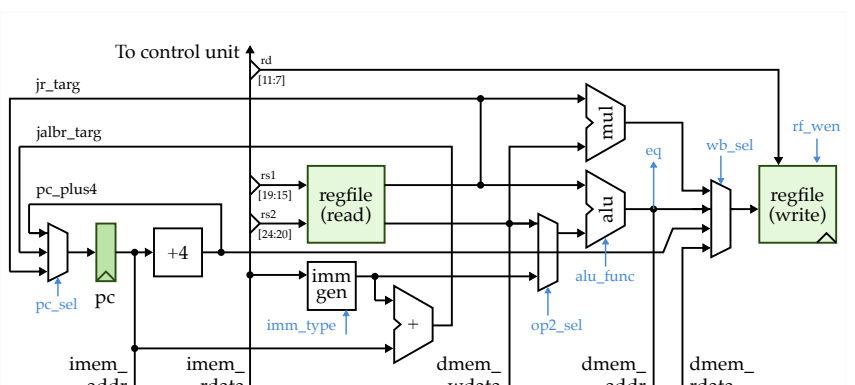
	`add`	`addi`	`mul`	`lw`	`sw`	`jal`	`jr`	`bne`
Fetch Instruction	✓	✓	✓	✓	✓	✓	✓	✓
Decode Instruction	✓	✓	✓	✓	✓	✓	✓	✓
Read Registers	✓	✓	✓	✓	✓		✓	✓
Register Arithmetic	✓	✓	✓	✓	✓			✓
Read Memory				✓				
Write Memory					✓			
Write Registers	✓	✓	✓	✓		✓		
Update PC	✓	✓	✓	✓	✓	✓	✓	✓



1.2. Technology and System Constraints

- Assume technology where logic is expensive, so we want to minimize the amount of registers and combinational logic as much as possible
- Assume multi-ported register files are expensive, so we want to use a single-ported register file if possible
- Assume a multi-ported physical memory is too expensive, so we must use a single-ported physical memory with combinational read, 512B of capacity, and wait signal





Component Delay and Area Models

- Use what we learned in the first part of the course to build area and delay models for processor components
  - Assume carry-look-ahead (CLA) adder with $k = 4$
  - Assume carry-save parallel-array multiplier with final CLA adder
  - Assume register file uses latches and tri-state buffers

Gate	t_{pd}	Area
NOT	1τ	1α
NAND2	2τ	2α
NOR2	3τ	3α
AND2	3τ	3α
OR2	4τ	4α
XOR2	7τ	7α
XNOR2	7τ	7α
NAND	$n\tau$	$n\alpha$
NOR	$(1+n)\tau$	$(1+n)\alpha$
AND	$(1+n)\tau$	$(1+n)\alpha$
OR	$(2+n)\tau$	$(2+n)\alpha$
XOR	$(5+n)\tau$	$(5+n)\alpha$
XNOR	$(5+n)\tau$	$(5+n)\alpha$

Component	t_{pd}	Area
32-bit 2-to-1 Mux	8τ	0.4kα
32-bit 4-to-1 Mux	11τ	0.8kα
32-bit Adder	107τ	1.5kα
32-bit Eq Cmp	20τ	0.3kα
32-bit ALU	115τ	2.1kα
32-bit Multiplier	530τ	17.5kα
32-bit +4 Unit	53τ	0.6kα
ImmGen Unit	11τ	0.8kα
32-bit Reg (t_{cq})	9τ	0.9kα
Register File (t_{cq})	25τ	18.0kα
Register File Read	25τ	
Memory (t_{cq})	650τ	
Memory Read	650τ	

32-bit Reg (t_{setup})	10τ
Register File (t_{setup})	20τ
Memory (t_{setup})	600τ

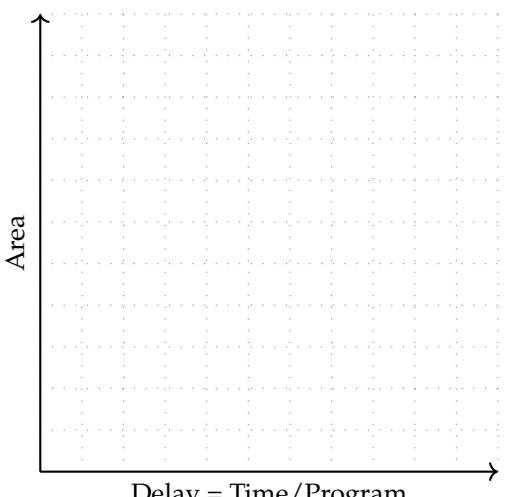
- Single-ported register file
  - Same timing as 2r1w register file
  - Area is 1/3 area of 2r1w register file (6kα)

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	Avg Cycles / Instruction	Time / Cycle	Time / Program	Area
Single-Cycle Processor	1	long	medium	medium
Multi-Cycle Processor	>1	short	high	low
Pipelined Processor	≈1	short	low	high



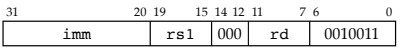
2. 15-State Multi-Cycle Processor Microarchitecture

ADDI

`addi rd, rs1, imm`

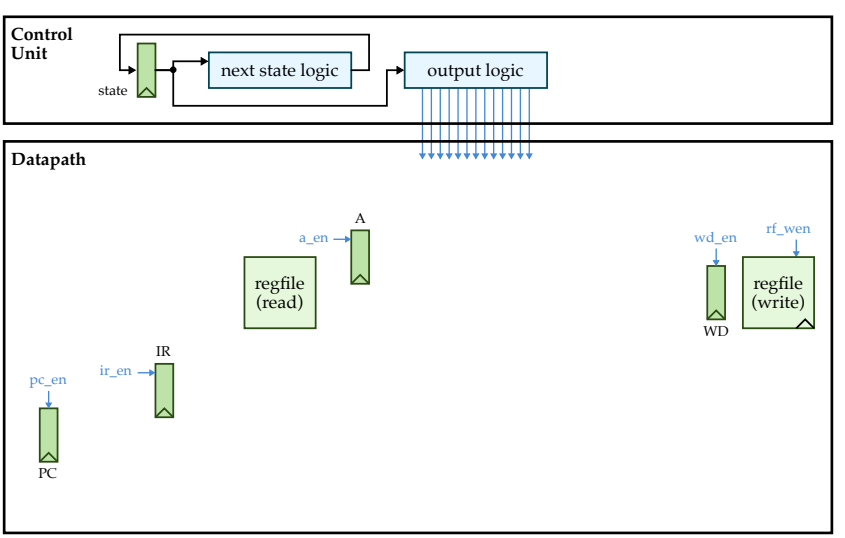
R[`rd`] ← R[`rs1`] + sext(`imm`)

PC ← PC + 4



I-type Immediate

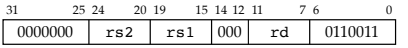
state	pc en	ir en	a en	op1 sel	op2 sel	wd en	rf wen	mem val
F								
D								
AI								
WB								



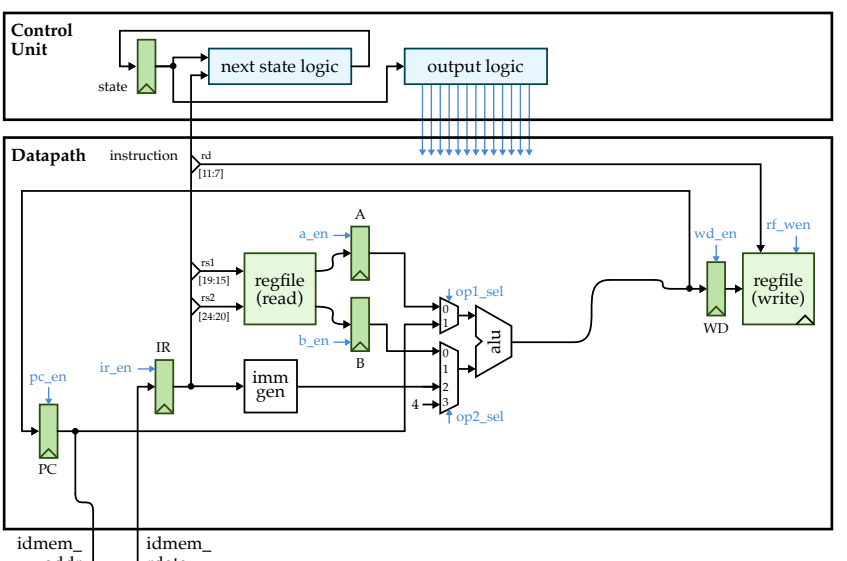
ADD

```
add rd, rs1, rs2
```

R[rd] ← R[rs1] + R[rs2]
PC ← PC + 4



state	pc en	ir en	a en	b en	op1 sel	op2 sel	mem val	wd en	rf wen
F	1	1	0	0	1	3	1	0	0
D	0	0	1	1	x	x	0	0	0
AI	0	0	0	0	0	2	0	1	0
WB	0	0	0	0	x	x	0	0	1
A									

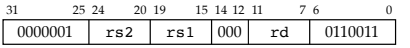


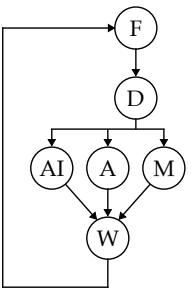
MUL

```
mul rd, rs1, rs2
```

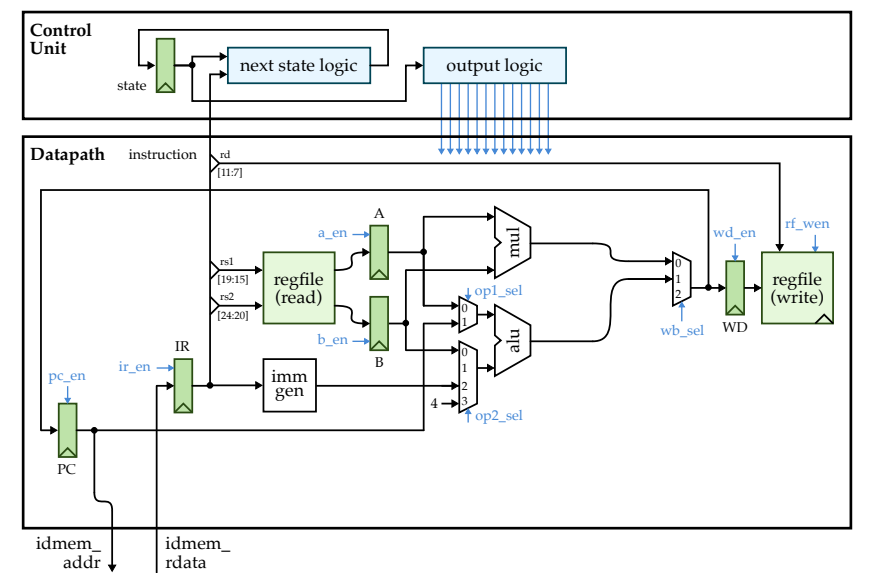
R[rd] ← R[rs1] × R[rs2]

PC ← PC + 4





state	pc en	ir en	a en	b en	op1 sel	op2 sel	wb sel	mem val	wd en	rf wen
F	1	1	0	0	1	3		1	0	0
D	0	0	1	1	x	x		0	0	0
AI	0	0	0	0	0	2		0	1	0
WB	0	0	0	0	x	x		0	0	1
A										
M										

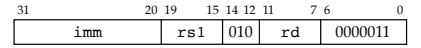


LW

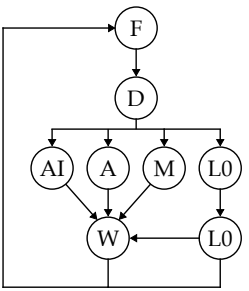
`lw rd, imm(rs1)`

R[`rd`] ← M[R[`rs1`] + sext(`imm`)]

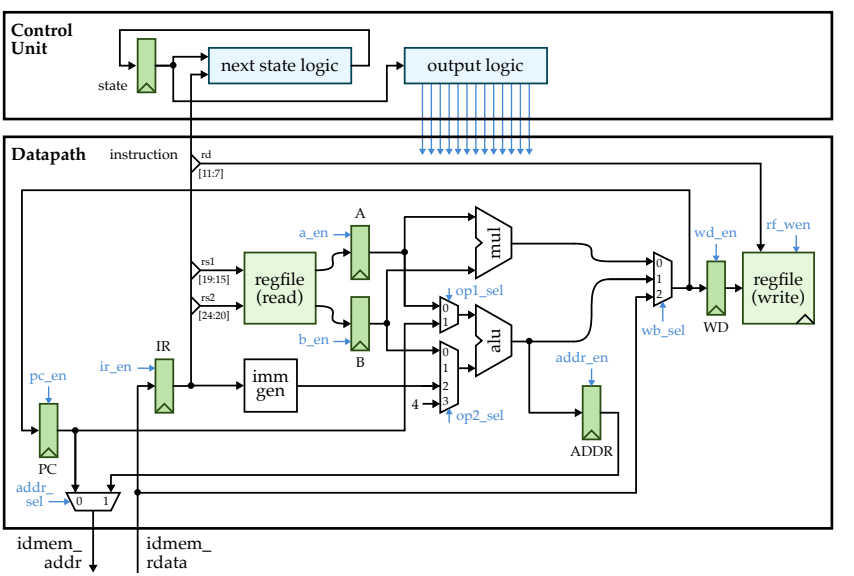
PC ← PC + 4

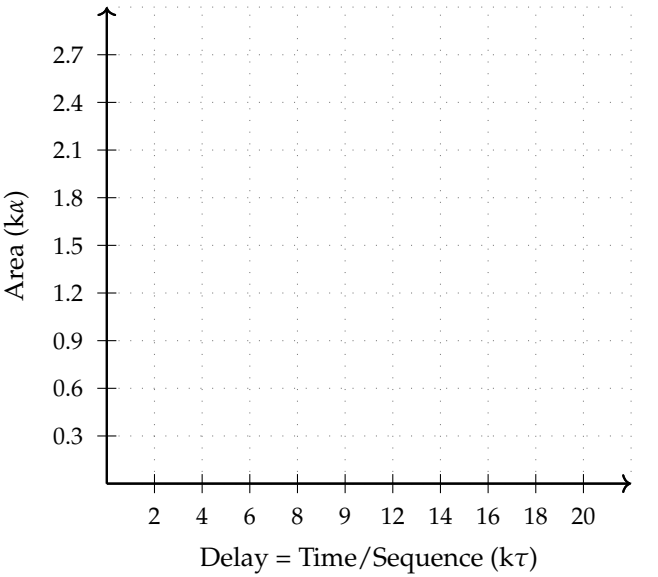


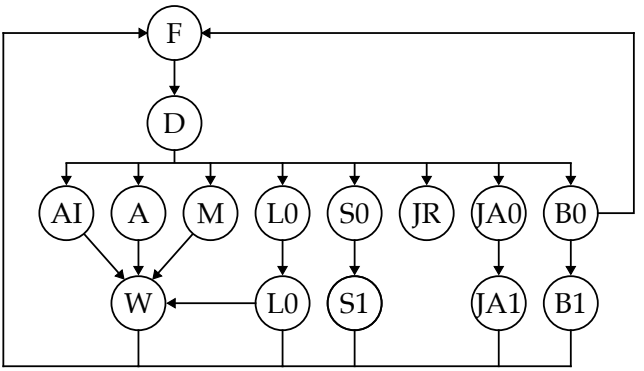
I-type Immediate

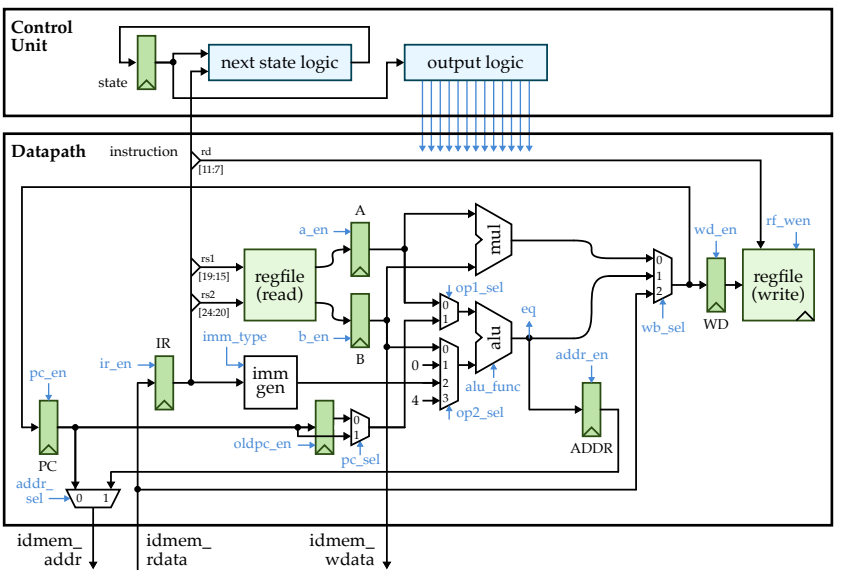


state	pc en	ir en	a en	b en	op1 sel	op2 sel	wb sel	addr sel	addr en	mem val	wd en	rf wen
F	1	1	0	0	1	3				1	0	0
D	0	0	1	1	x	x				0	0	0
AI	0	0	0	0	0	2				0	1	0
WB	0	0	0	0	x	x				0	0	1
A												
M												
L0												
L1												

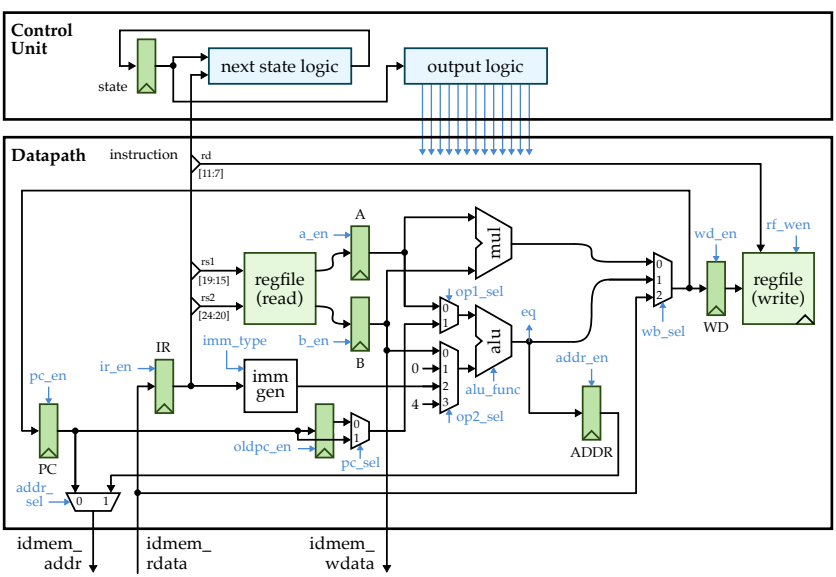



2.7
2.4
2.1
1.8
1.5
1.2
0.9
0.6
0.3
Area (kα)
2
4
6
8
9
12
14
16
18
20
Delay = Time/Sequence (kτ)


F
D
AI
A
M
L0
S0
JR
JA0
B0
W
L0
S1
JA1
B1


Control Unit
state
next state logic
output logic
Datapath
instruction
rd [11:7]
rs1 [19:15]
rs2 [24:20]
regfile (read)
A
a_en
B
b_en
imm_type
imm gen
IR
ir_en
pc_en
PC
addr_sel
oldpc_en
pc_sel
op1_sel
op2_sel
alu_func
alu
mul
eq
addr_en
ADDR
wd_en
rf_wen
wb_sel
WD
regfile (write)
idmem_ addr
idmem_ rdata
idmem_ wdata

	pc en	oldpc en	ir en	a en	b en	addr sel	pc sel	op1 sel	op2 sel	imm type	alu func	addr sel	addr en	mem val	mem type	wb sel	wd en	rf wen
F	1	1	1	0	0	0	1	1	3	x	add	0	0	1	0	1	0	0
D	0	0	0	1	1	x	x	x	x	x	x	x	0	0	x	x	0	0
AI	0	0	0	0	0	x	x	0	2	I	add	x	0	0	x	1	1	0
WB	0	0	0	0	0	x	x	x	x	x	x	x	0	0	x	0	0	1
A	0	0	0	0	0	x	x	0	0	x	add	x	0	0	x	1	1	0
M	0	0	0	0	0	x	x	x	x	x	x	x	0	0	x	0	1	0
L0	0	0	0	0	0	x	x	0	2	I	add	x	1	0	x	x	0	0
L1	0	0	0	0	0	1	x	x	x	x	x	1	0	1	0	2	1	0
S0	0	0	0	0	0	x	x	0	2	S	add	x	1	0	x	x	0	0
S1	0	0	0	0	0	1	x	x	x	x	x	1	0	1	1	x	0	0
JR	1	0	0	0	0	x	x	0	1	x	add	x	0	0	0	1	0	0
JA0	0	0	0	0	0	x	0	1	3	x	add	x	0	0	0	1	1	0
JA1	1	0	0	0	0	x	0	1	2	J	add	x	0	0	0	1	0	1
B0	0	0	0	0	0	x	x	0	0	x	cmp	x	0	0	0	x	0	0
B1	1	0	0	0	0	x	0	1	2	B	add	x	0	0	0	1	0	0

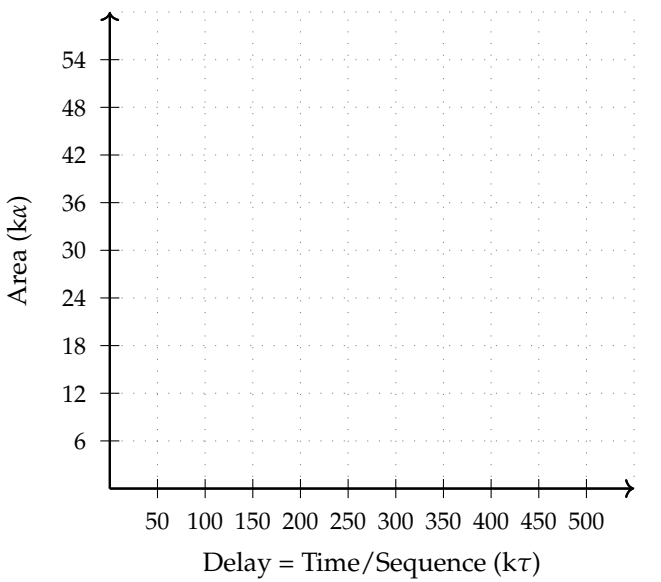


2.1. Analyzing Delay and Area

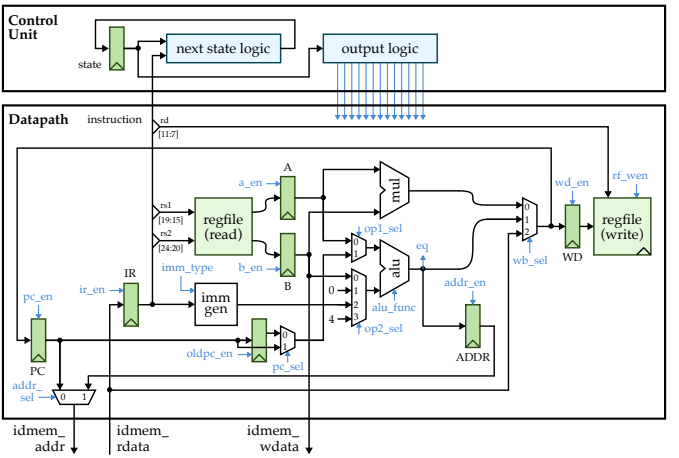
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycles}}$$

Results for vector-vector-add micro-benchmark

µArch	Inst/Prog	Cycle/Inst	Time/Cycle	Time/Prog	Area
SC	189	1.0	1.48 kτ	280 kτ	43.4 kα
MC-15					



Estimating minimum T_C and area



Component	t_{pd}	Area
32-bit 2-to-1 Mux	8τ	0.4kα
32-bit 4-to-1 Mux	11τ	0.8kα
32-bit Adder	107τ	1.5kα
32-bit Eq Cmp	20τ	0.3kα
32-bit ALU	115τ	2.1kα
32-bit Multiplier	530τ	17.5kα
32-bit +4 Unit	53τ	0.6kα
ImmGen Unit	11τ	0.8kα
32-bit Reg (t_{cq})	9τ	0.9kα
Register File (t_{cq})	25τ	18.0kα
Register File Read	25τ	
Memory (t_{cq})	650τ	
Memory Read	650τ	

32-bit Reg (t_{setup})	10τ
Register File (t_{setup})	20τ
Memory (t_{setup})	600τ

Estimating execution time

How long in units of τ will it take to execute the vector-vector add program assuming `n` is 21?

Pseudo-Code

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

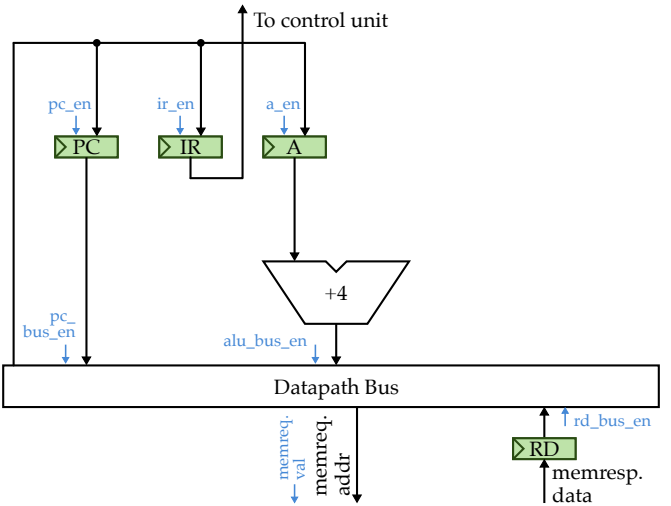
Assembly Code

```
 # assume x1 holds address of dest[i]
 # assume x2 holds address of src0[i]
 # assume x3 holds address of src1[i]
 # assume x4 holds n (counting down)
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	14	15	16	17
`lw x5, 0(x1)`																		
`lw x6, 0(x2)`																		
`add x7, x5, x6`																		
`sw x7, 0(x3)`																		

	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
`addi x1, x1, 4`																				
`addi x2, x2, 4`																				
`addi x3, x3, 4`																				
`addi x4, x4, -1`																				
`bne x4, x0, loop`																				

3. 62-State Multi-Cycle Processor Microarchitecture

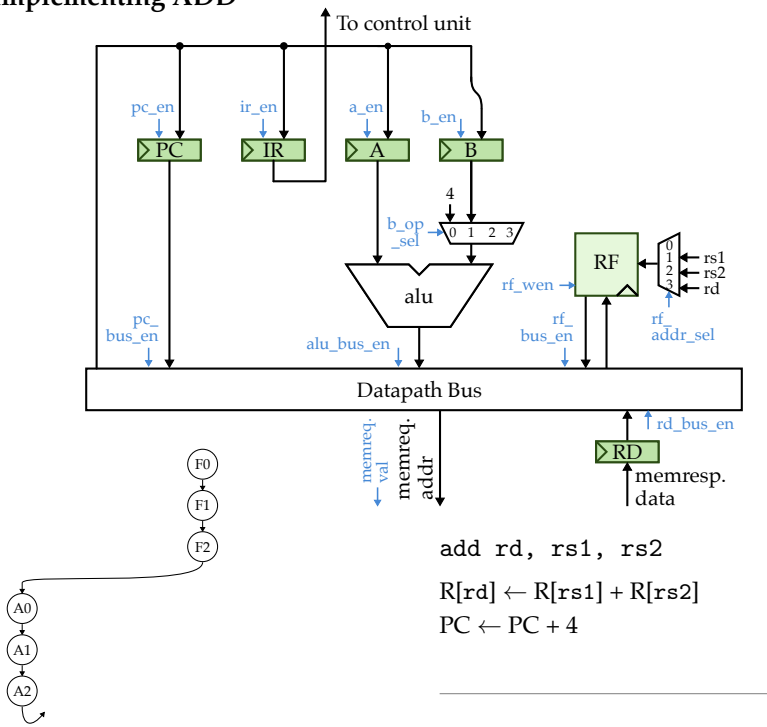




(pseudo-control-signal syntax)

	Bus Enables			Reg Enables			MReq
state	pc	alu	rd	pc	ir	a	val
F0	1	0	0	0	0	1	1
F1	0	0	1	0	1	0	0
F2	0	1	0	1	0	0	0

Implementing ADD

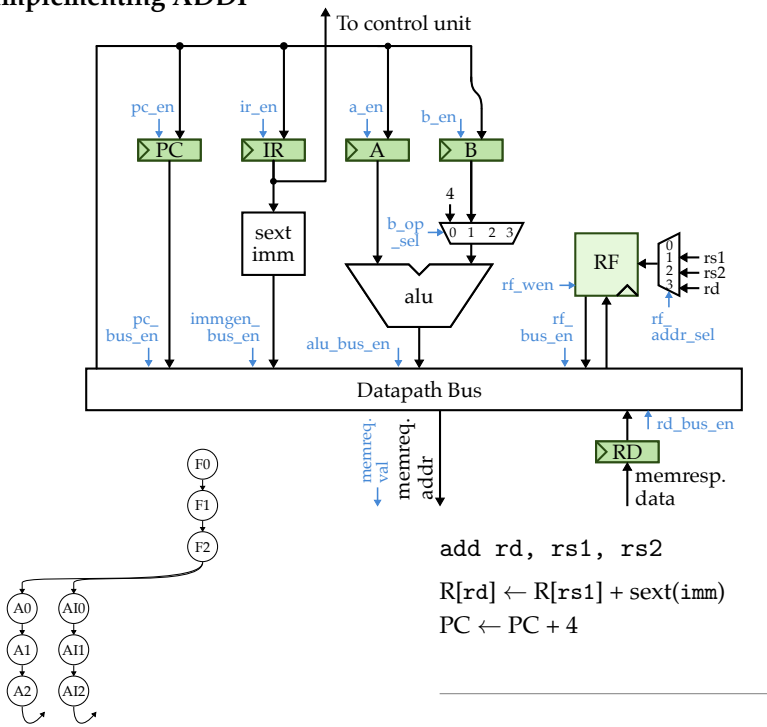



```
add rd, rs1, rs2
```

R[rd] ← R[rs1] + R[rs2]
PC ← PC + 4

	Bus Enables				Register Enables				Mux	RF		MReq
state	pc	alu	rf	rd	pc	ir	a	b	bop	sel	wen	val
A0												
A1												
A2												

Implementing ADDI



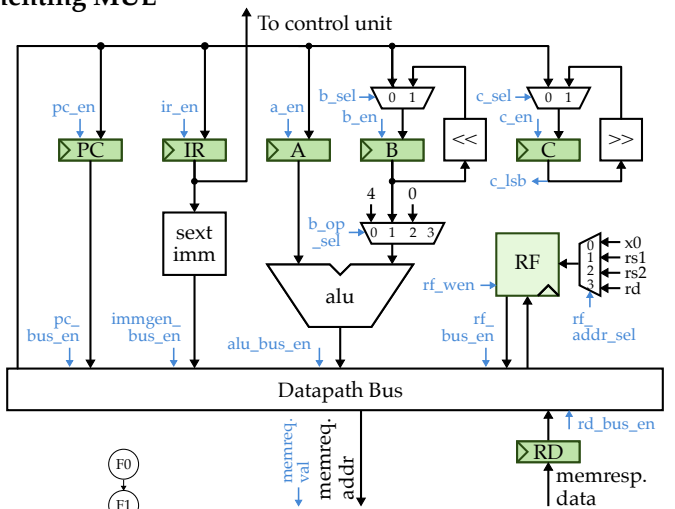
`add rd, rs1, rs2`

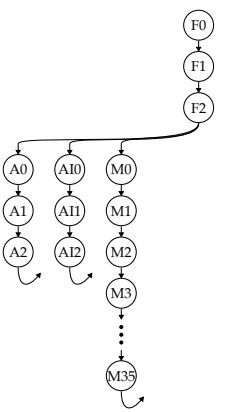
R[`rd`] ← R[`rs1`] + sext(`imm`)

PC ← PC + 4

	Bus Enables					Register Enables				Mux	RF		MReq
state	pc	ig	alu	rf	rd	pc	ir	a	b	bop	sel	wen	val
AI0													
AI1													
AI2													

Implementing MUL



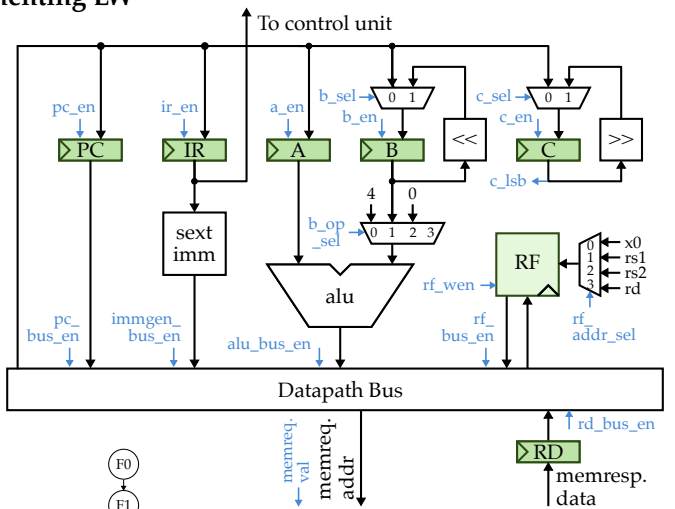


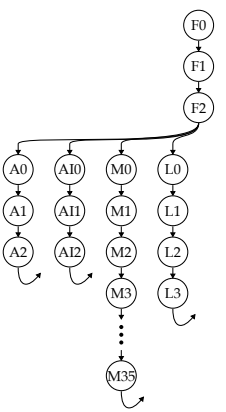
`mul rd, rs1, rs2`

R[`rd`] ← R[`rs1`] × R[`rs2`]

PC ← PC + 4

Implementing LW



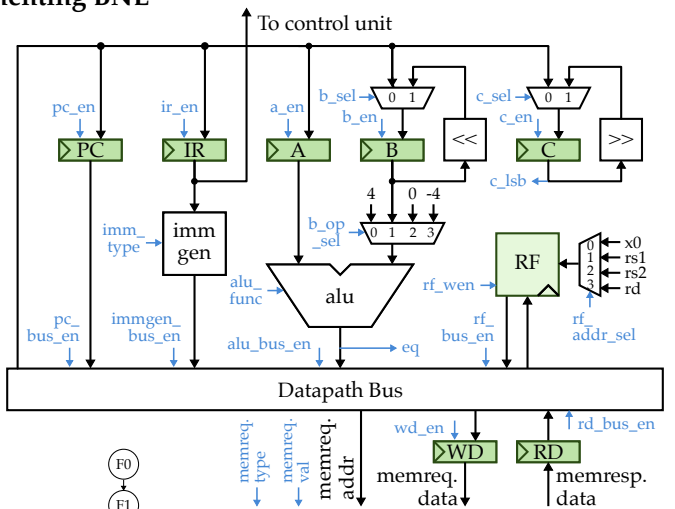


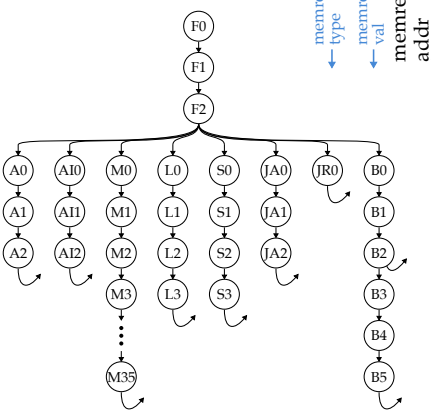
`lw rd, imm(rs1)`

R[`rd`] ← M[R[`rs1`] + sext(imm)]

PC ← PC + 4

Implementing BNE





`bne rs1, rs2, imm`

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

Fetch States

F0: memreq.addr ← PC; A ← PC;
RD ← M[memreq.addr]
F1: IR ← RD
F2: PC ← A + 4; goto inst

`add rd, rs1, rs2`

A0: A ← R[`rs1`]
A1: B ← R[`rs2`]
A2: R[`rd`] ← A + B

`addi rd, rs1, imm`

AI0: A ← R[`rs1`]
AI1: B ← sext(`imm_i`)
AI2: R[`rd`] ← A + B

`mul rd, rs1, rs2`

M0: A ← R[x0]
M1: B ← R[`rs1`]
M2: C ← R[`rs2`]
M3: A ← C[0] ? A + B : A;
B ← B << 1; C ← C >> 1
M4: A ← C[0] ? A + B : A;
B ← B << 1; C ← C >> 1
...
M34: R[`rd`] ← C[0] ? A + B : A;
goto F0

`lw rd, imm(rs1)`

L0: A ← R[`rs1`]
L1: B ← sext(`imm_i`)
L2: memreq.addr ← A + B;
RD ← M[memreq.addr]
L3: R[`rd`] ← RD; goto F0

`sw rs2, imm(rs1)`

S0: WD ← R[`rs2`]
S1: A ← R[`rs1`]
S2: B ← sext(`imm_s`)
S3: memreq.addr ← A + B
M[memreq.addr] ← WD;
goto F0

`jal rd, imm`

JA0: R[`rd`] ← PC
JA1: B ← sext(`imm_j`)
JA2: PC ← A + B; goto F0

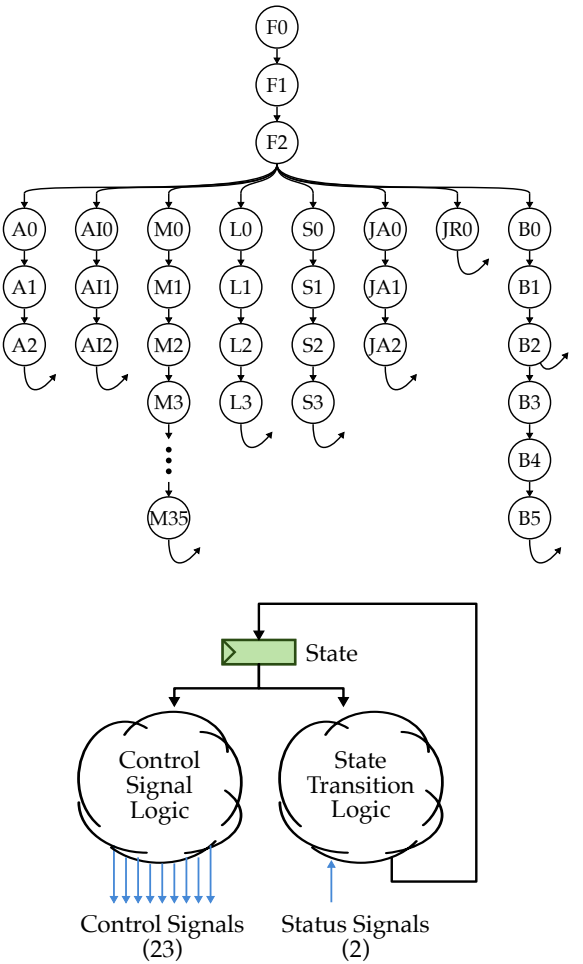
`jr rs1`

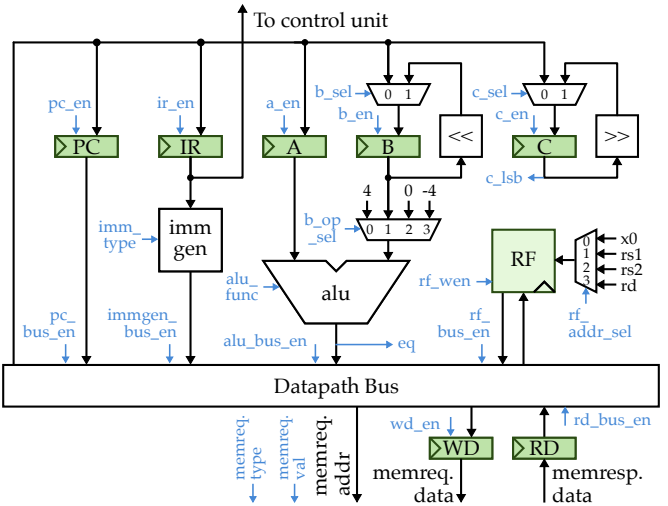
JR0: PC ← R[`rs1`]; goto F0

`bne rs1, rs2, imm`

B0: A ← R[`rs1`]
B1: B ← R[`rs2`]
B2: B ← sext(`imm_b`);
if A == B goto F0
B3: A ← PC
B4: A ← A − 4
B5: PC ← A + B; goto F0

Multi-Cycle Processor Final Control Unit





F0: memreq.addr ← PC; A ← PC
F1: IR ← RD
F2: PC ← A + 4; goto inst

A0: A ← R[`rs1`]
A1: B ← R[`rs2`]
A2: R[`rd`] ← A + B; goto F0

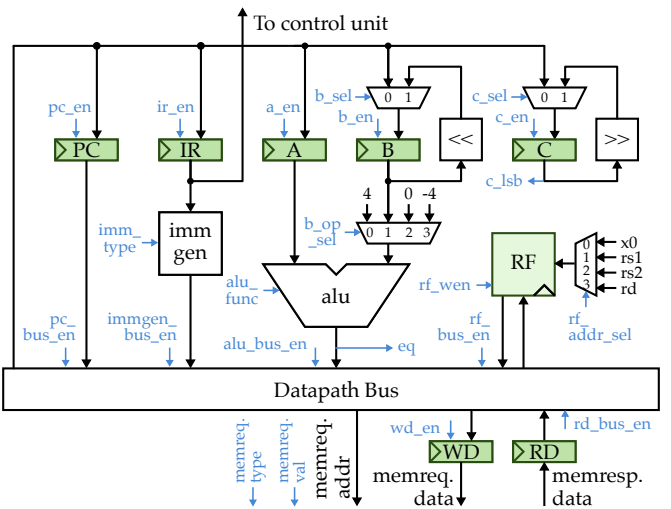
	Bus Enables					Register Enables						Mux			Func		RF		MReq	
state	pc	ig	alu	rf	rd	pc	ir	a	b	c	wd	b	c	bop	ig	alu	sel	wen	val	type
F0	1	0	0	0	0	0	0	1	0	0	0	x	x	x	x	x	x	0	1	0
F1	0	0	0	0	1	0	1	0	0	0	0	x	x	x	x	x	x	0	0	x
F2	0	0	1	0	0	1	0	0	0	0	0	x	x	0	x	0	x	0	0	x
A0	0	0	0	1	0	0	0	1	0	0	0	x	x	x	x	x	1	0	0	x
A1	0	0	0	1	0	0	0	0	1	0	0	0	x	x	x	x	2	0	0	x
A2	0	0	1	1	0	0	0	0	0	0	0	x	x	1	x	0	3	1	0	x

Adding a Complex Instruction

Adding new complex instructions usually does not require datapath modifications, only additional states.

```
add.mm rd, rs1, rs2
```

M[R[rd]] ← M[R[rs1]] + M[R[rs2]]



Adding a New Auto-Incrementing Load Instruction

Implement the following auto-incrementing load instruction using pseudo-control-signal syntax. Modify the datapath if necessary.

`lw.ai rd, imm(rs1)`

R[`rd`] ← M[R[`rs1`] + sext(`imm_i`)]; R[`rs1`] ← R[`rs1`] + 4

