Class Review

CMPT 295 Week 12

Caches and Virtual Memory

Cache Code Analysis Problem

- Cache-A Direct-mapped, 4KB, 64 sets
- Cache-B Set-associate, 4KB, 2 ways, 32 sets

```
int size = 4096; // int is 4 bytes
int a[size];
long long int a_long[size]; // long long int is 8 bytes
/* loop 1 */
for (int i = 0; i < size; i++) {
    a[i] = i;
}
/* loop 2 */
for (int i = 0; i < size; i++) {
    a_long[i] = i;
}
/* loop 3 */
for (int i = 0; i < size/2; i += 1) {
    a[(size/2+i] = a[i];
}
```

Question: Hit rates for loop1, loop 2 (assume loop 1 has run) and loop 3 (assume loops 1 and 2 have run) for both caches?

Sources of Cache Misses: The 3Cs

- Compulsory: (Many names: cold start, process migration (switching processes), 1st reference)
  - First access to block impossible to avoid; Effect is small for long running programs
- Capacity:
  - Cache cannot contain all blocks accessed by the program, so full associativity won't hold all blocks
- Conflict: (collision)
  - Multiple memory locations mapped to the same cache location, so there is a lack of associativity

Cache & VM Problem

- 14-bit virtual addresses, 12-bit physical address
- Page size = 64 bytes
- Cache: Direct-mapped with $K = 4$ B, $C/K = 16$

TLB:

Set	Tag	PPN	V	Tag	PPN	V	Tag	PPN	V	Tag	PPN	V
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page table (partial):

VPN	PPN	V
0	28	1
1	–	0
2	33	1
3	02	1
4	–	0
5	16	1
6	–	0
7	–	0

VPN	PPN	V
8	13	1
9	17	1
A	09	1
B	–	0
C	–	0
D	2D	1
E	–	0
F	0D	1

Cache:

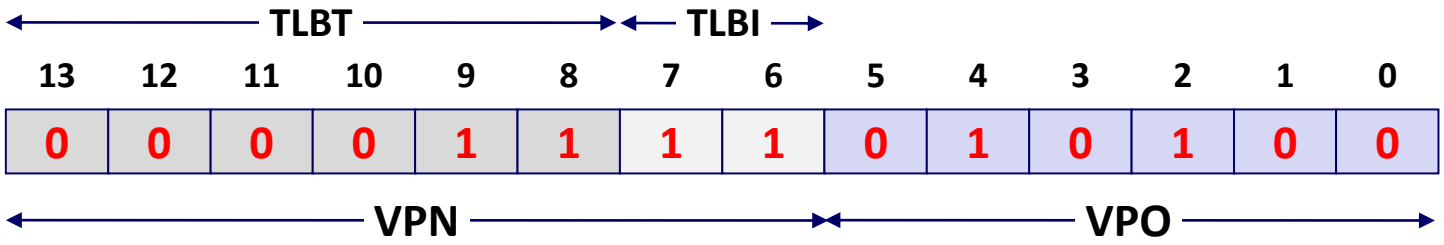
Index	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Index	Tag	V	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Memory Request Example #1

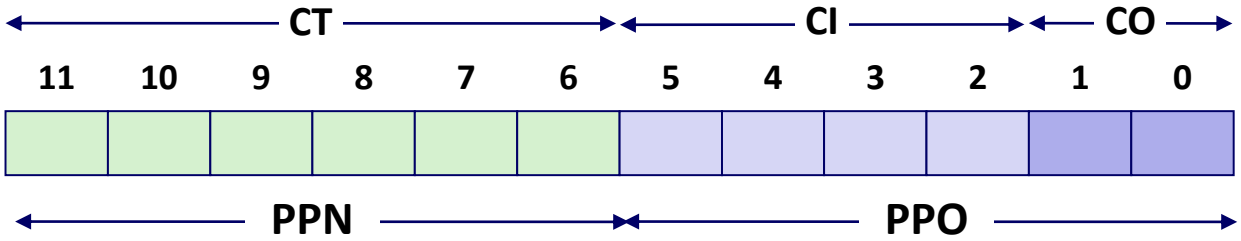
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x03D4`



VPN _____ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:

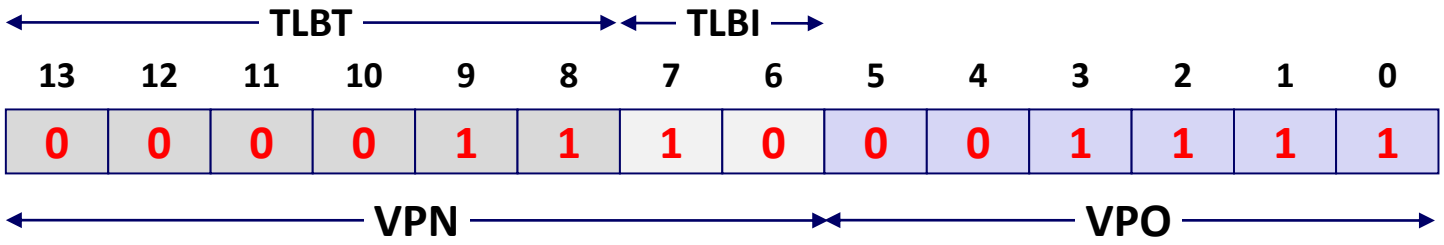


CT _____ CI _____ CO _____ Cache Hit? ___ Data (byte) _____

Memory Request Example #2

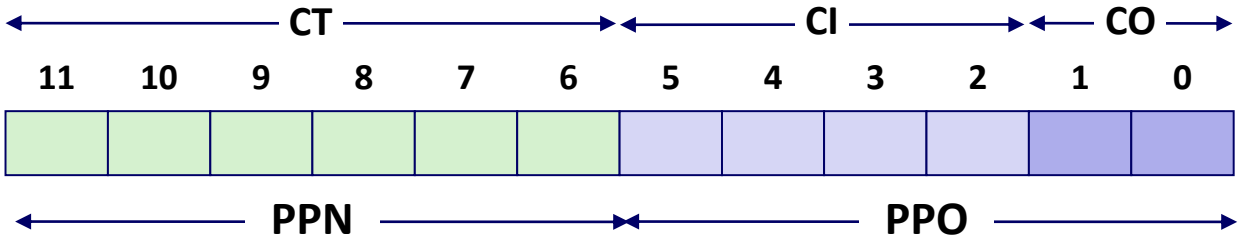
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x038F`



VPN _____ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:

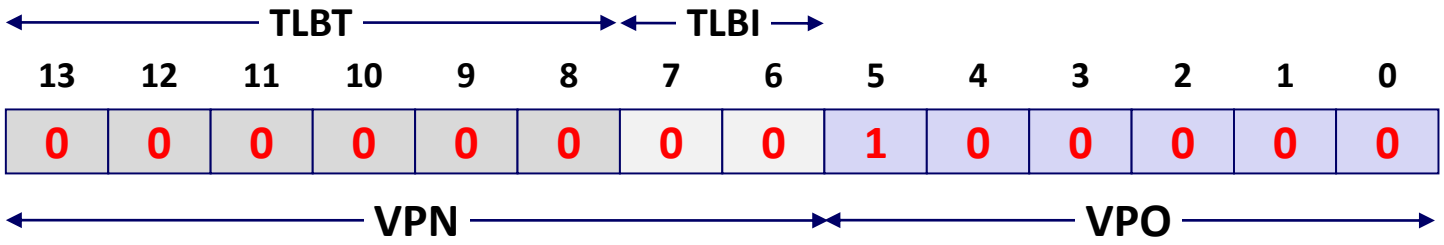


CT _____ CI _____ CO _____ Cache Hit? ___ Data (byte) _____

Memory Request Example #3

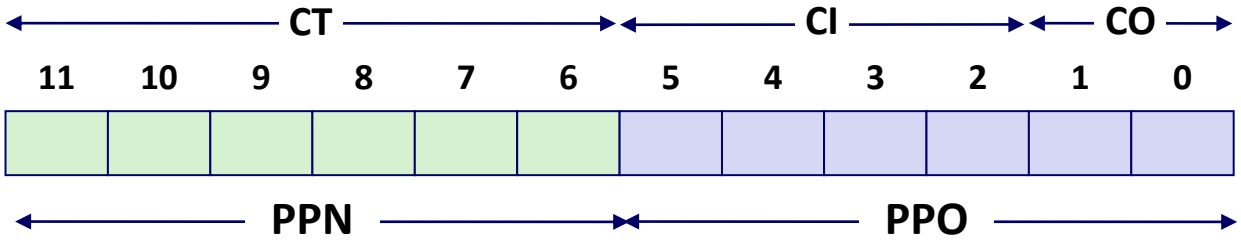
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x0020`



VPN _____ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:

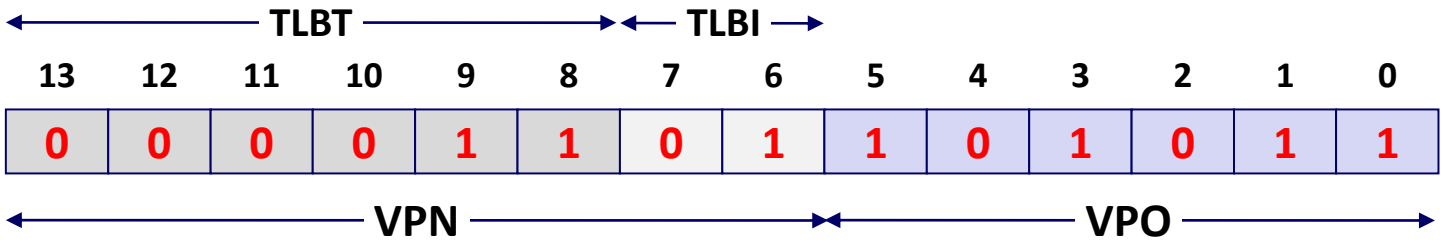


CT _____ CI _____ CO _____ Cache Hit? ___ Data (byte) _____

Memory Request Example #4

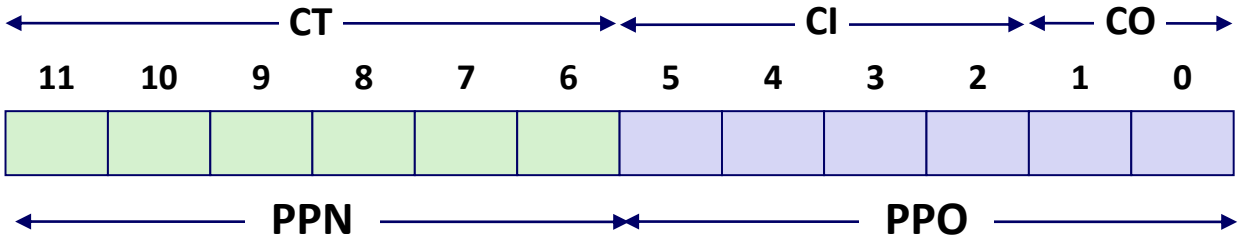
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x036B`



VPN _____ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:



CT _____ CI _____ CO _____ Cache Hit? ___ Data (byte) _____

Digital Logic

Laws of Boolean Algebra

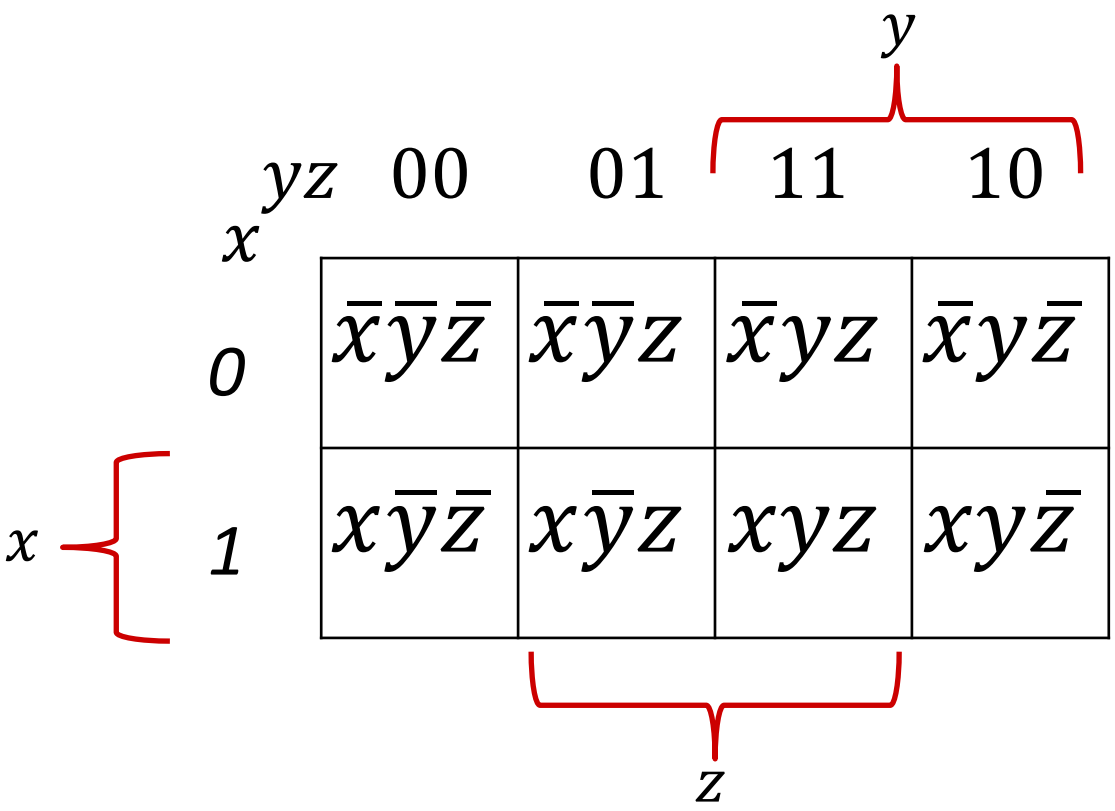
These laws allow us to simplify Boolean expressions:

$x \cdot \overline{x} = 0$	$x + \overline{x} = 1$	complementarity
$x \cdot 0 = 0$	$x + 1 = 1$	laws of 0's and 1's
$x \cdot 1 = x$	$x + 0 = x$	identities
$x \cdot x = x$	$x + x = x$	idempotent law
$x \cdot y = y \cdot x$	$x + y = y + x$	commutativity
$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$	associativity
$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$	distribution
$xy + x = x$	$(x + y)x = x$	uniting theorem
$\overline{x}y + x = x + y$	$(\overline{x} + y)x = xy$	uniting theorem v.2
$\overline{x \cdot y} = \overline{x} + \overline{y}$	$\overline{x + y} = \overline{x} \cdot \overline{y}$	DeMorgan's Law

Simplifying Boolean Expressions: Example

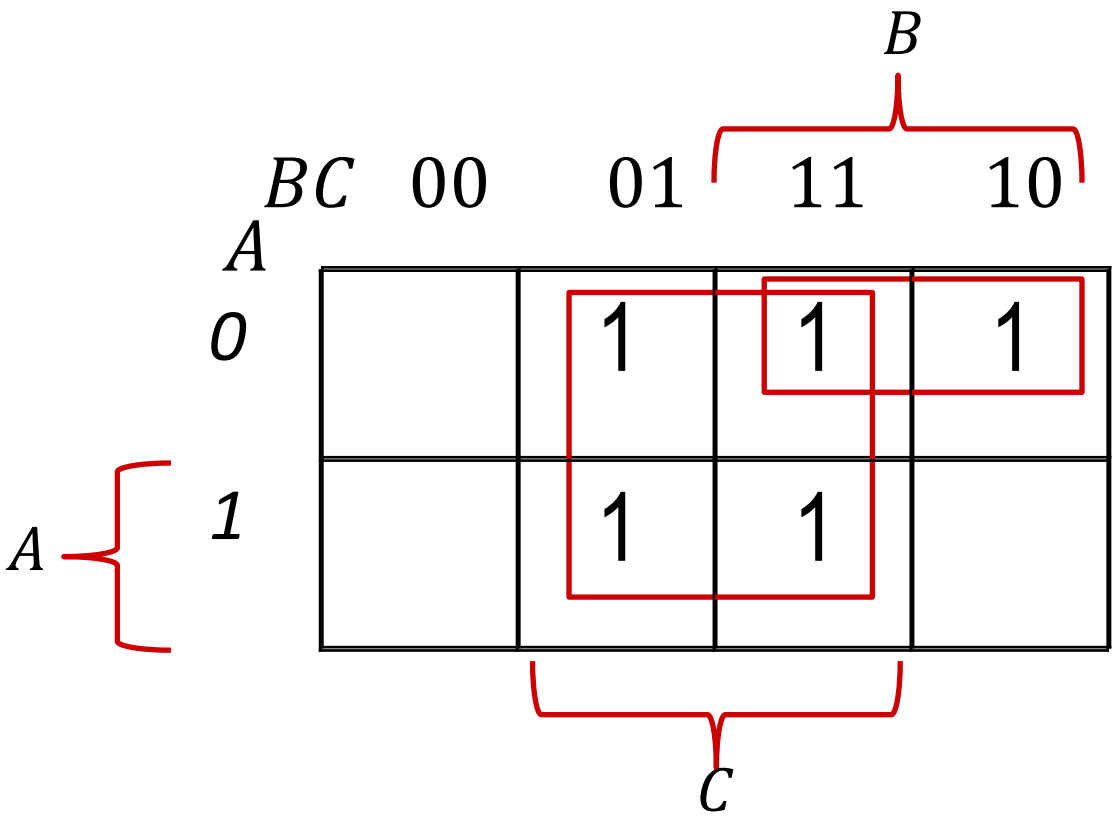
$$y = ab + a + c$$

Three Variable Karnaugh Maps



Example: Simplify 3-Variable Expression

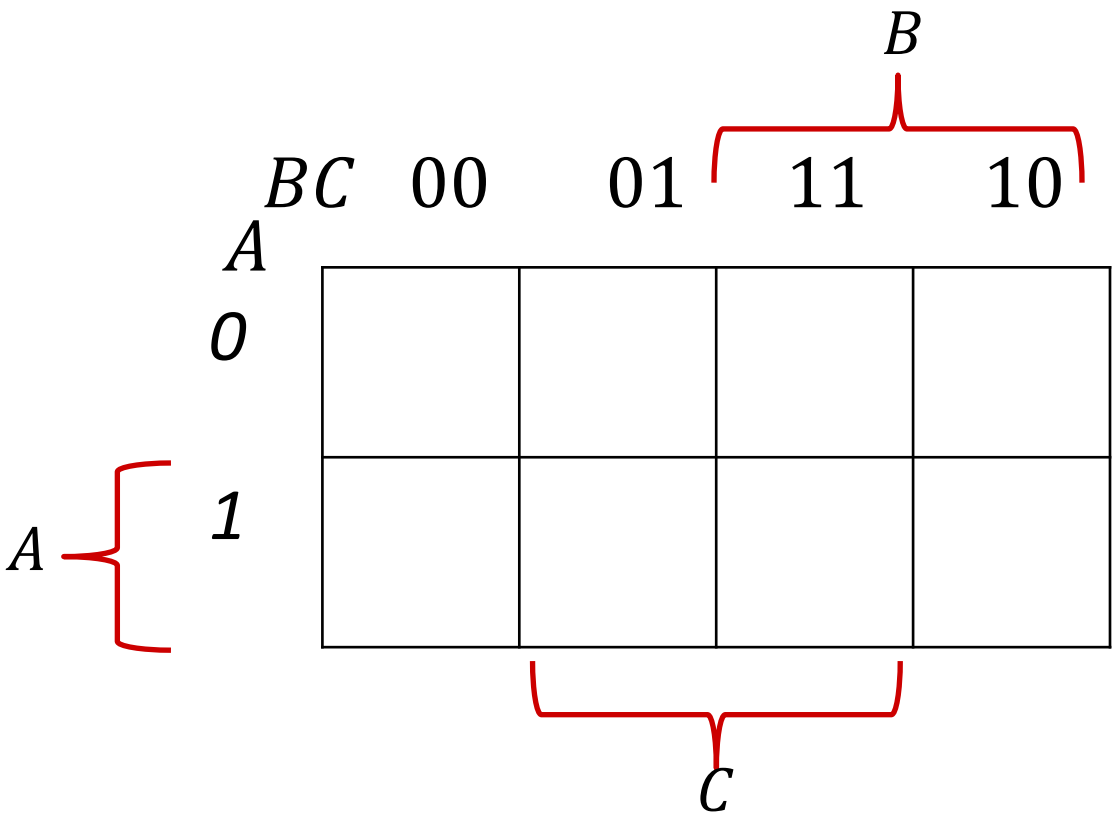
Question: Simplify $\bar{A}C + \bar{A}B + A\bar{B}C + BC$



Answer: $C + \bar{A}B$

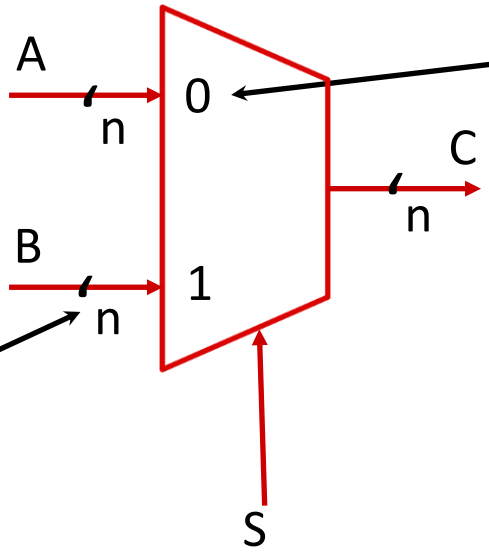
Example: Simplify 3-Variable Expression

Question: Simplify $AB + A + A\bar{B}C + C$



Data Multiplexor (MUX)

- Multiplexor ("MUX") is a *selector*
  - Place one of multiple inputs onto output (N-to-1)
- Shown below is an n-bit 2-to-1 MUX
  - Input S selects between two inputs of n bits each

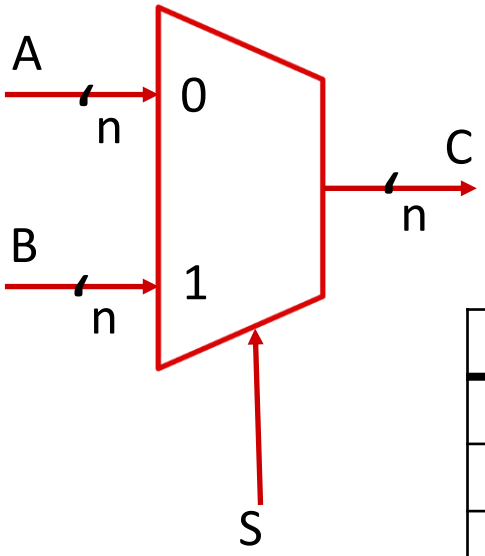


This input is passed to output if selector bits match shown value

Represents that this input has n bits

Implementing a 1-bit 2-to-1 MUX

- **Schematic:**



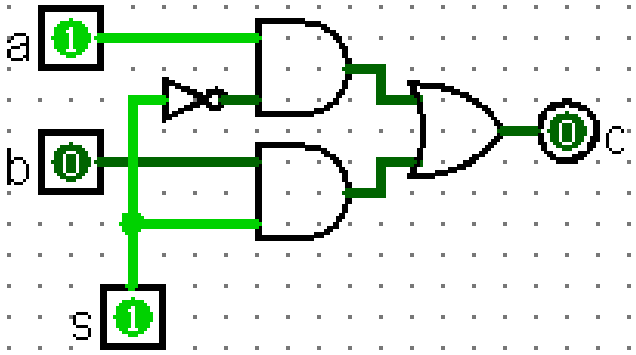
- **Truth Table:**

s	a	b	c
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- **Boolean Algebra:**

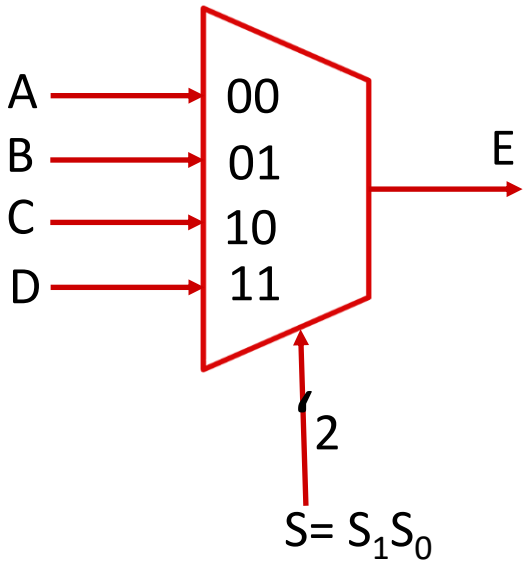
$$\begin{aligned} c &= \bar{s}a\bar{b} + \bar{s}ab + s\bar{a}b + sab \\ &= \bar{s}(a\bar{b} + ab) + s(\bar{a}b + ab) \\ &= \bar{s}(a(\bar{b} + b)) + s((\bar{a} + a)b) \\ &= \bar{s}(a(1) + s((1)b) \\ &= \bar{s}a + sb \end{aligned}$$

- **Circuit Diagram:**



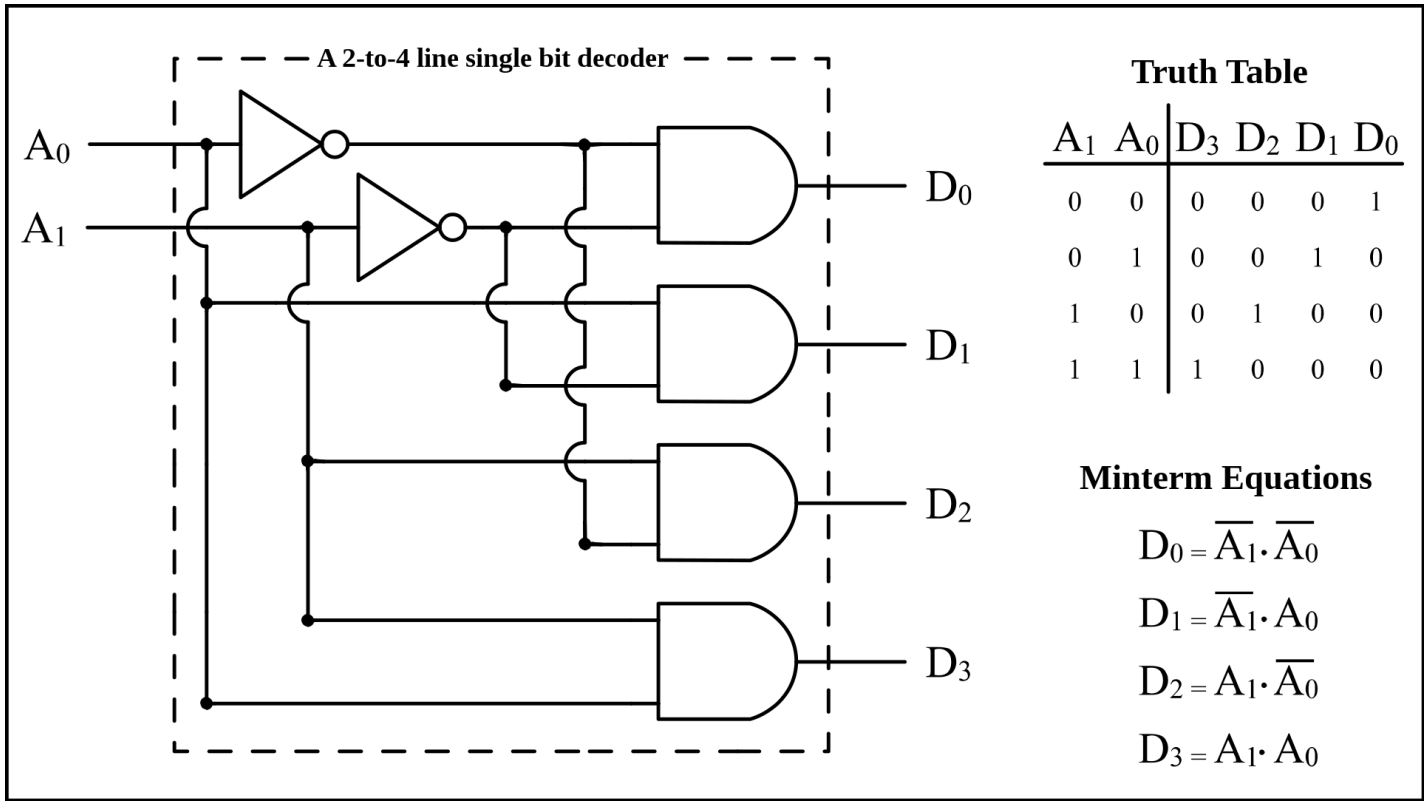
1-bit 4-to-1 MUX

- **Schematic:**



Decoder

- Enable one of 2^N outputs based on N input
- Example: 2-to-4 decoder

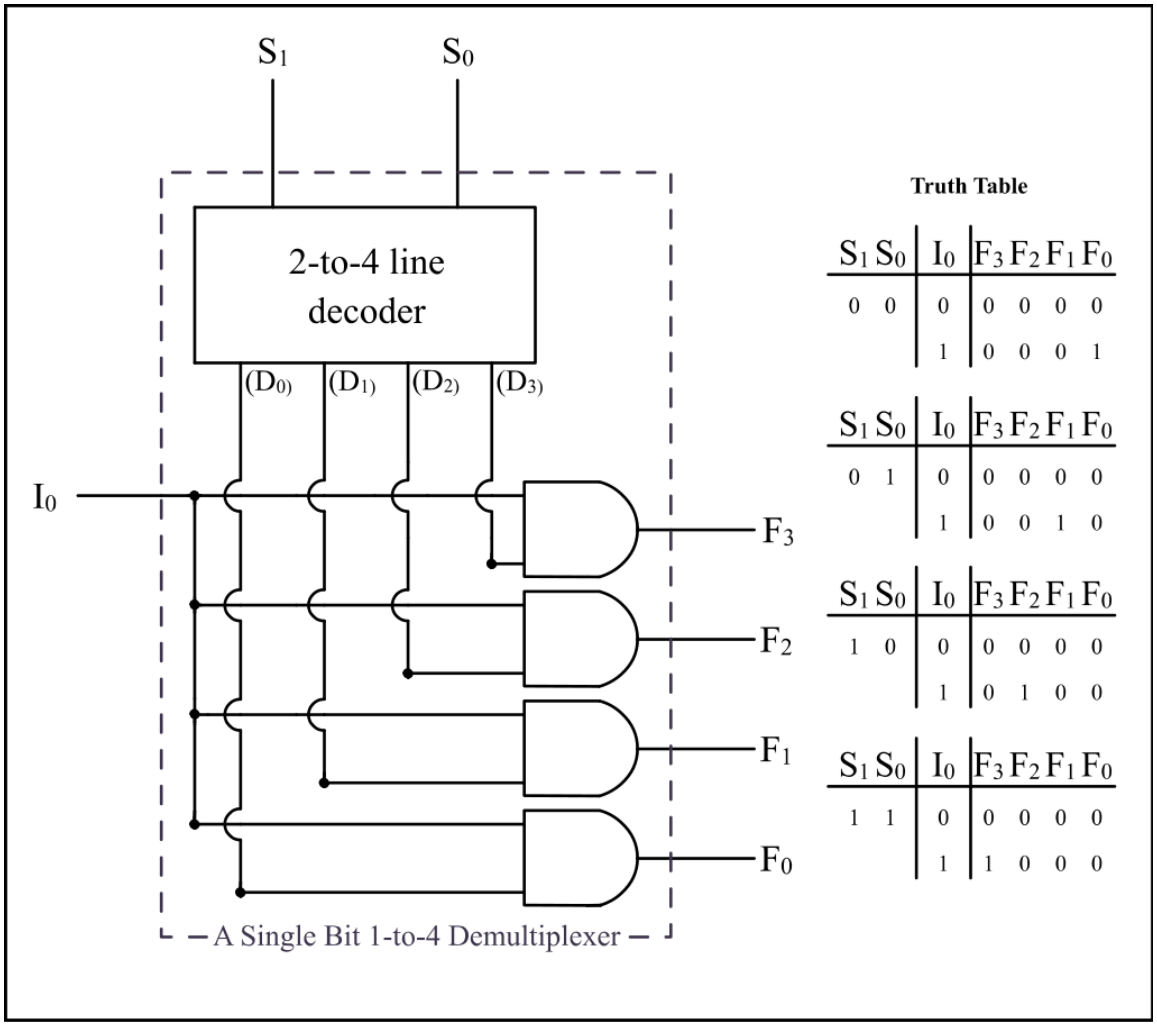


By BlueJester0101, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3668293

- Use case: Choose ALU operation based on instruction op-code

Demultiplexer (Demux)

- Similar to decoder with an enable signal



Truth Table

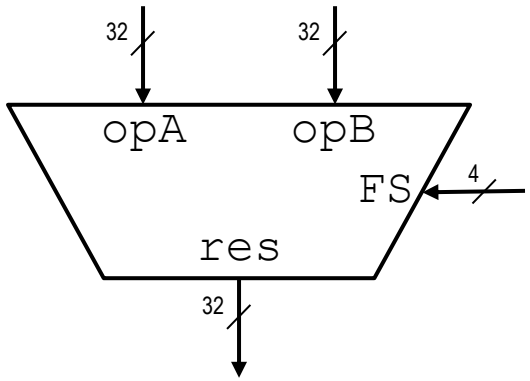
S_1	S_0	I_0	F_3	F_2	F_1	F_0
0	0	0	0	0	0	0
		1	0	0	0	1
0	1	0	0	0	0	0
		1	0	0	1	0
1	0	0	0	0	0	0
		1	0	1	0	0
1	1	0	0	0	0	0
		1	1	0	0	0

Functional Unit

Hardware circuits are fixed

- Can't adjust wires / gates while running
- Build control wires to parametrize its function

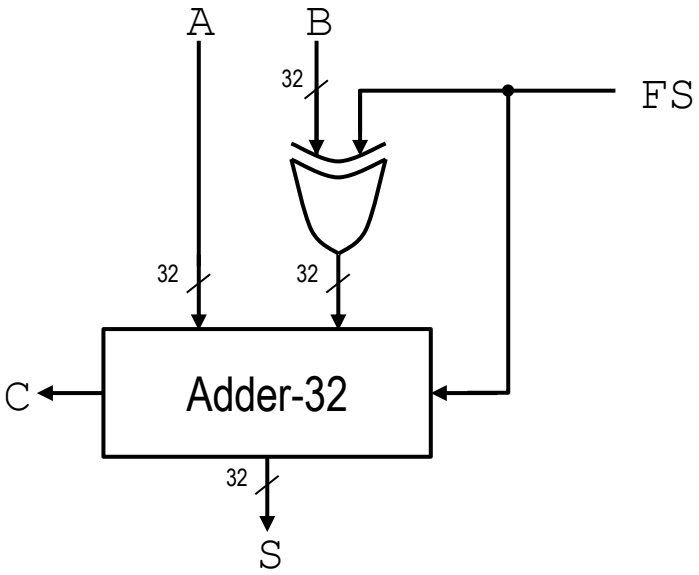
Function Unit:



Function Select:

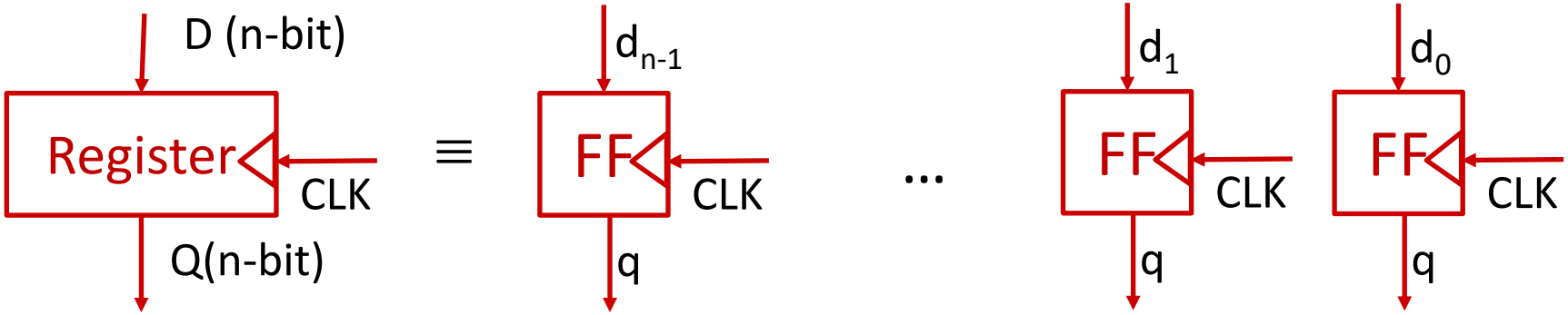
FS	func
0001	A + B
0010	A – B
1000	A * B
0100	A ^ B
0101	A + 1
1101	B

Functional Unit: Adder-Subtractor

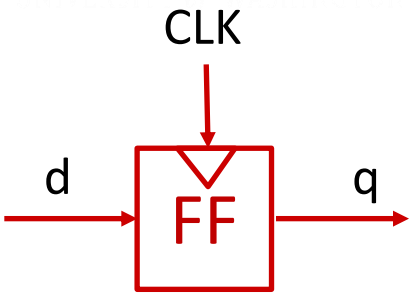


- if `FS == 0` then
  `S = A + B`
- if `FS == 1` then
  `S = A + B̄ + 1`
  `= A − B`

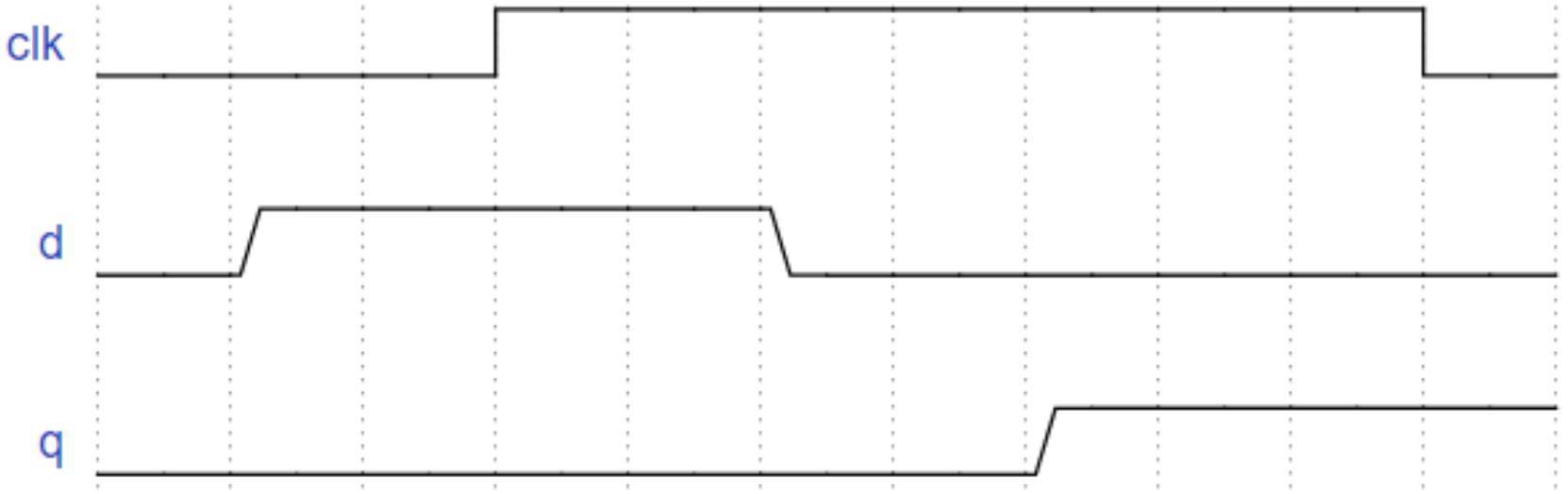
Sequential Circuits: Registers

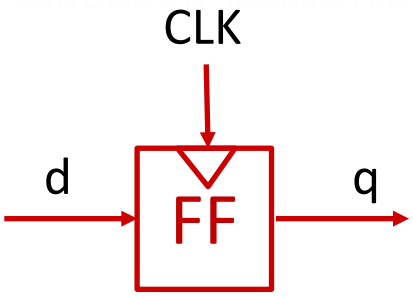


- N-bit register ≡ n instances of a *"Flip-Flop"*
  - Output flips and flops between 0 and 1
- Specifically this is a "D-type Flip-Flop"
  - D is "data input", Q is "data output"
  - A group of wires when interpreted as a bit field is called a *bus*

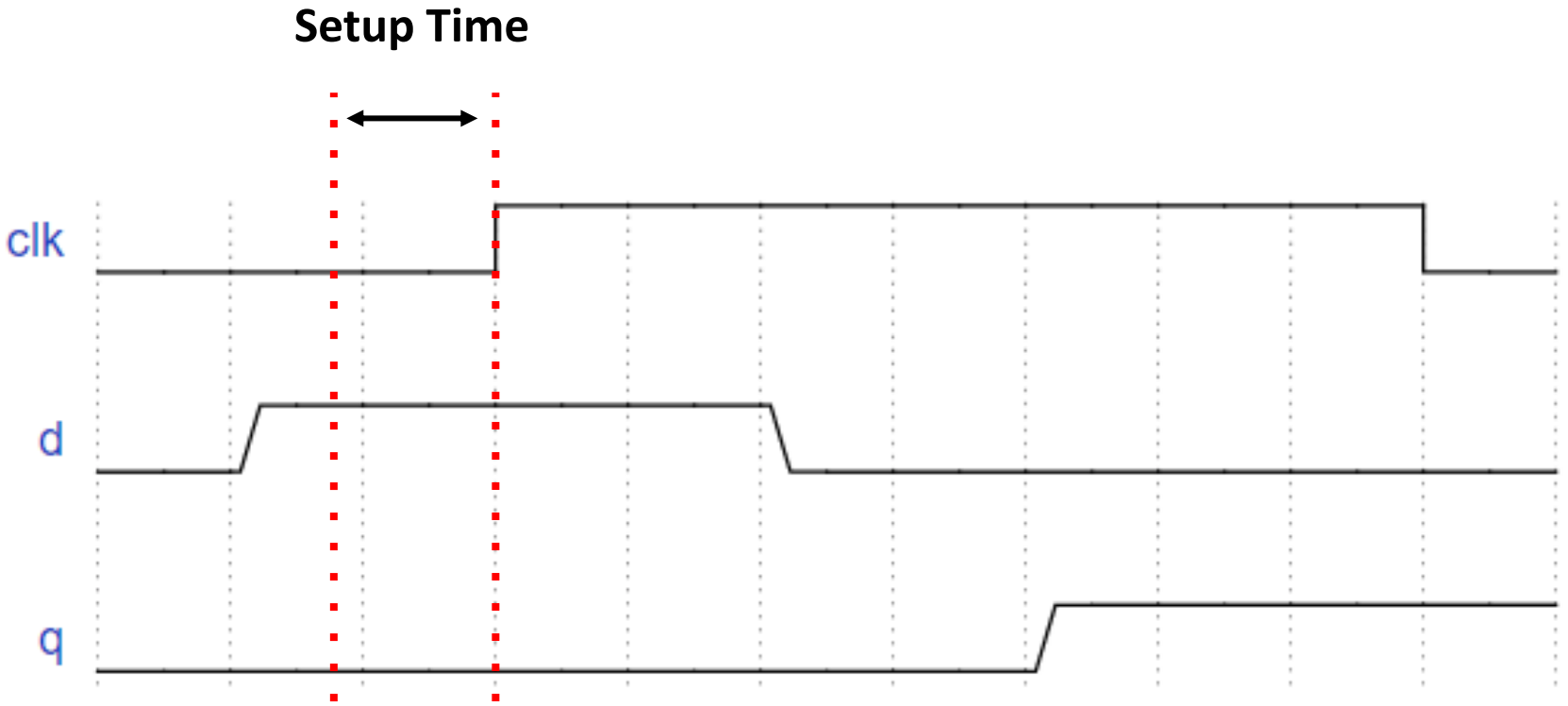


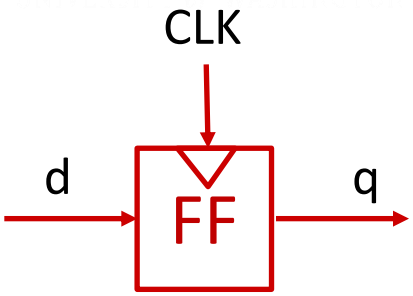
Flip-Flop Timing Behavior



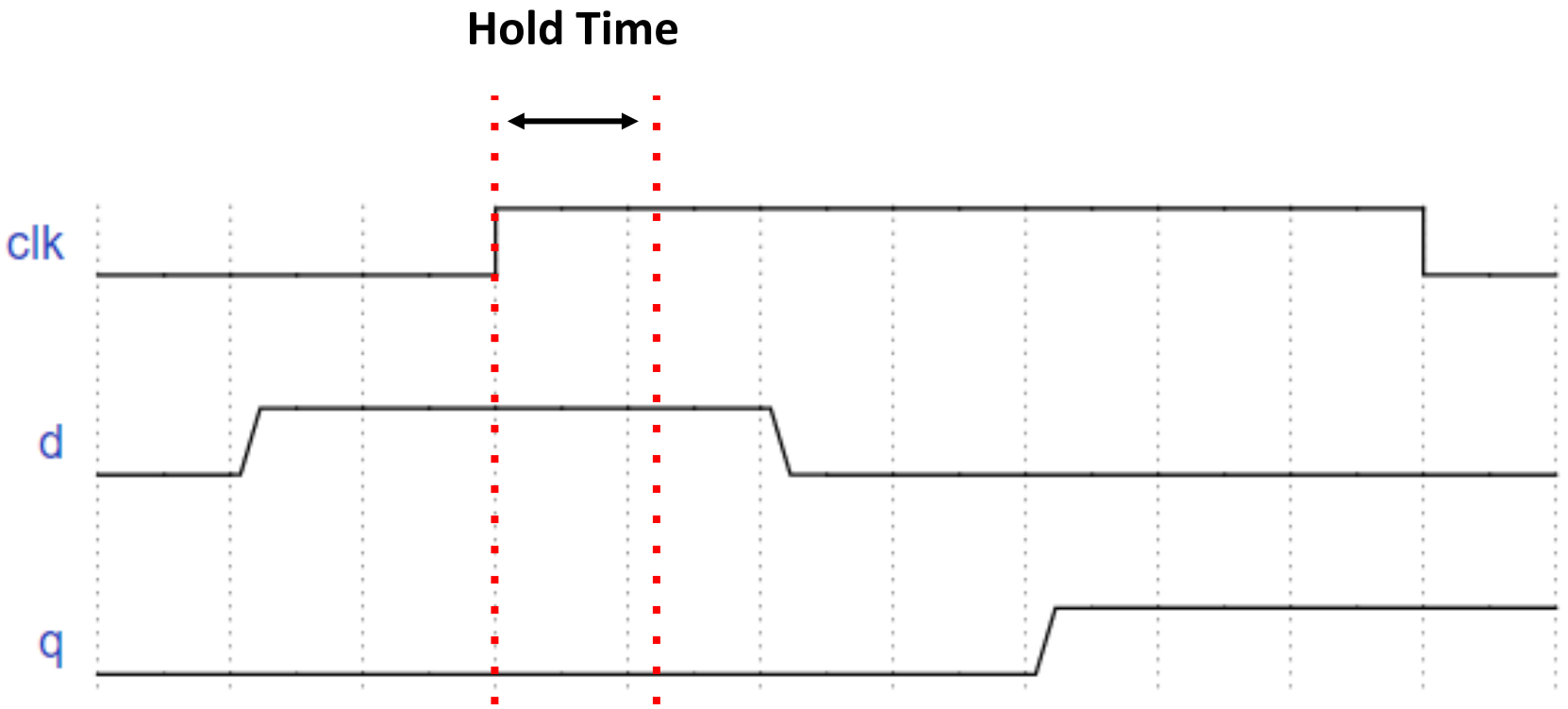


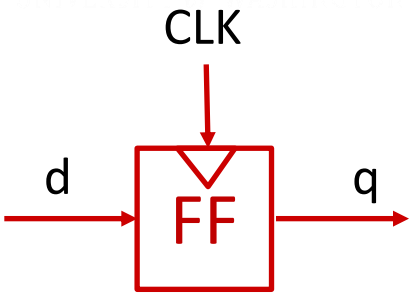
Flip-Flop Timing Behavior



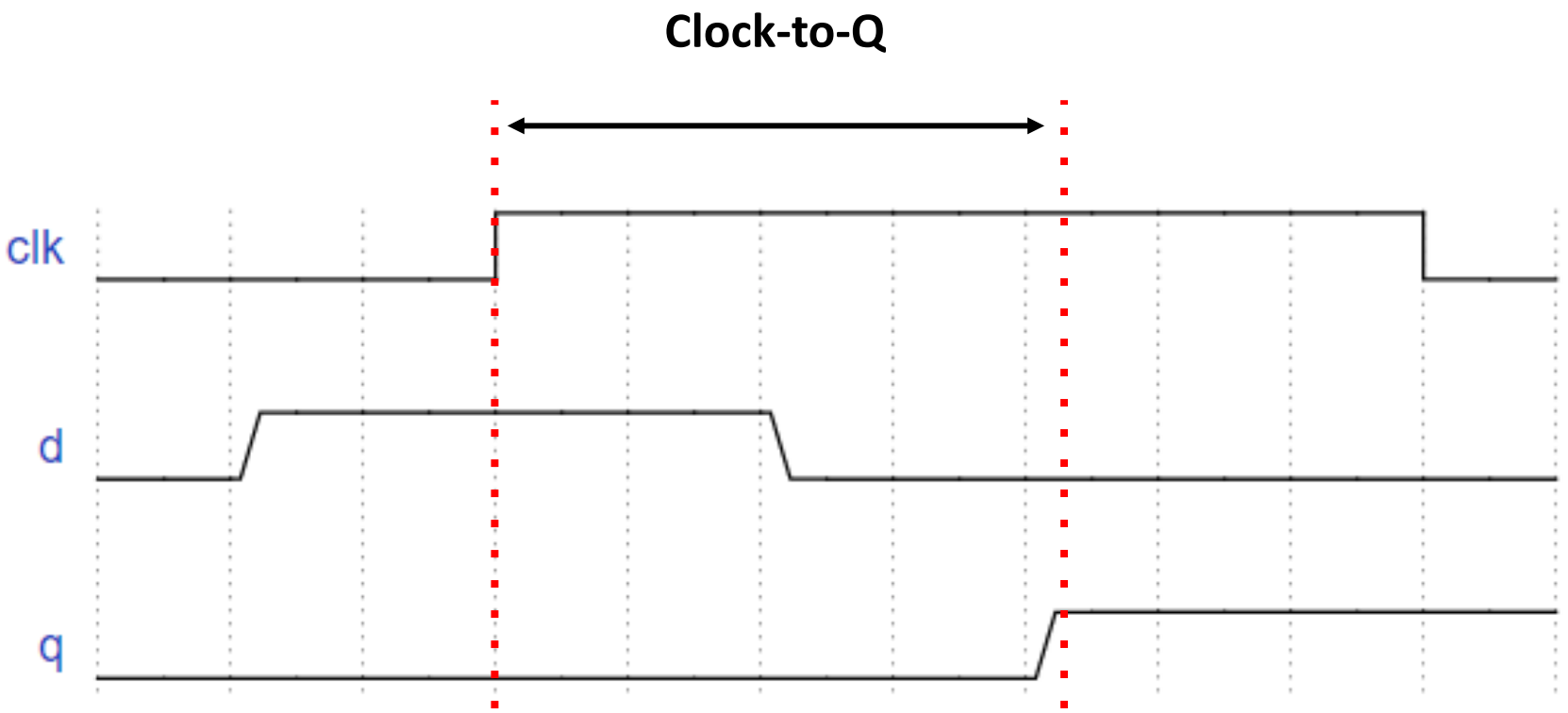


Flip-Flop Timing Behavior





Flip-Flop Timing Behavior

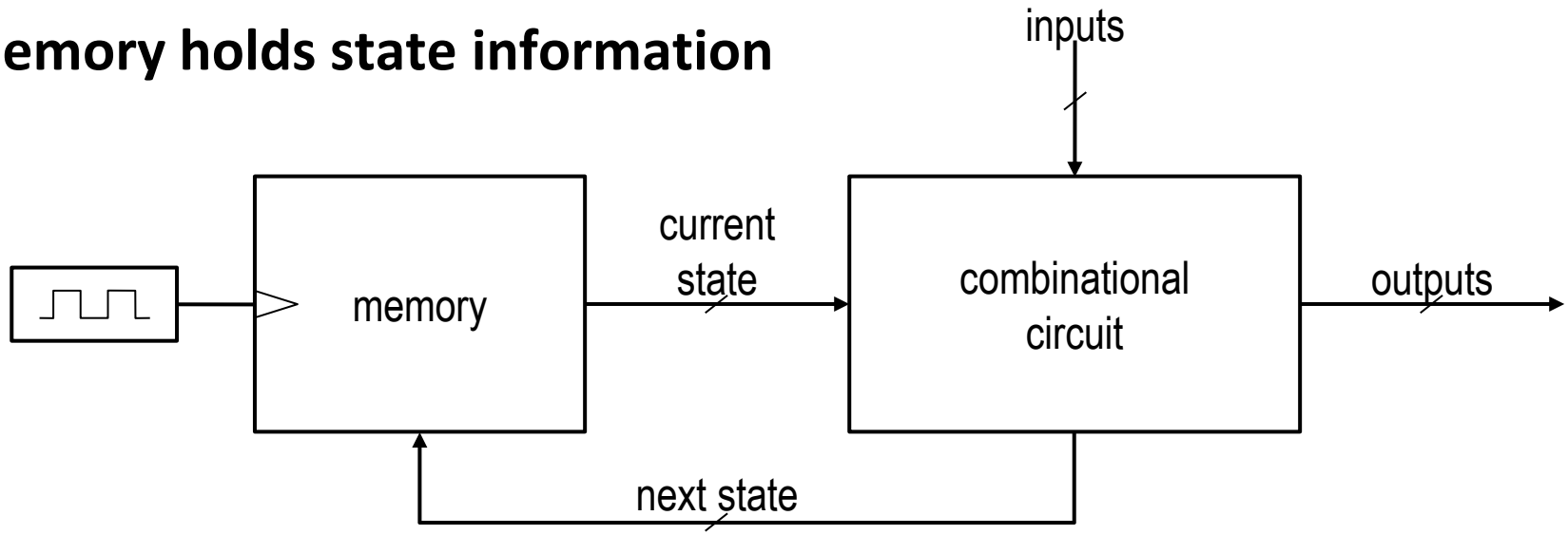


Timing Terms

- Clock: Steady square wave that synchronizes system
- Register: Several bits of state that samples on rising edge of Clock (positive edge-triggered); also has RESET
- Setup Time: When input must be stable *before* Clock trigger
- Hold Time: When input must be stable *after* Clock trigger
- Clock-to-Q Delay: How long it takes output to change from Clock trigger

Digital State Machines

Memory holds state information

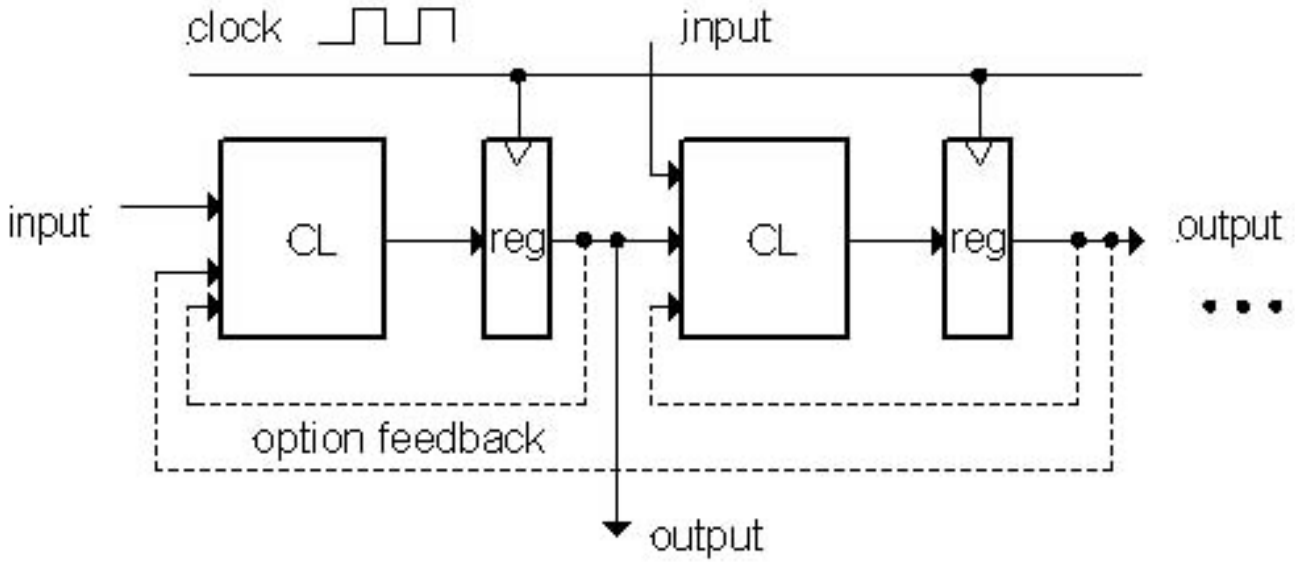


- **compute next state based on (current state, inputs)**
- **compute outputs based on (current state, inputs)**

Q. What does this imply about the clock period?

- **clock period must exceed (t_{pd} of combinational circuit + t_{pd} of registers) where t_{pd} is propagational delay**

Model for Synchronous Systems

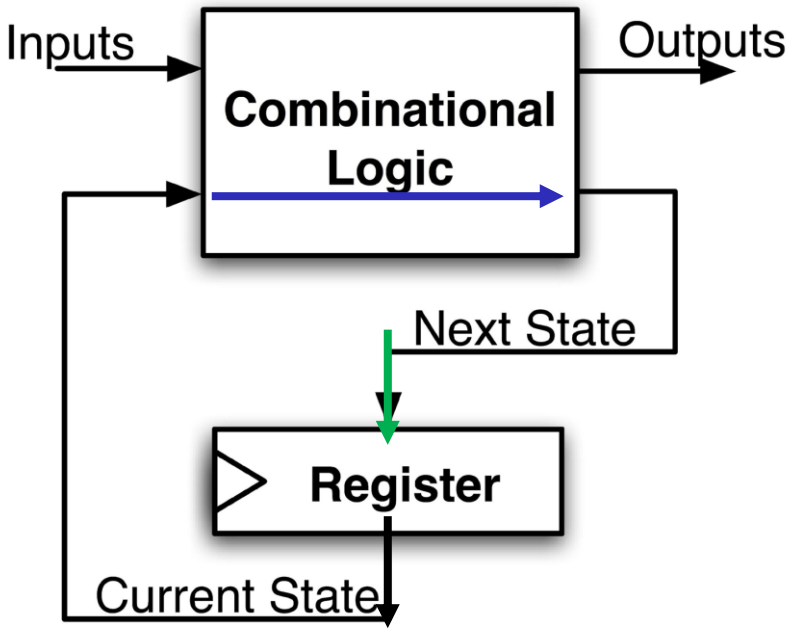


- Combinational logic blocks separated by registers
  - Clock signal connects only to sequential logic elements
  - Feedback is optional depending on application
- How do we ensure proper behavior?
  - How fast can we run our clock?

Maximum Clock Frequency

- What is the max frequency of this circuit?
  - Limited by how much time needed to get correct Next State to Register (t_{setup} constraint)

Assumes Max Delay > Hold Time



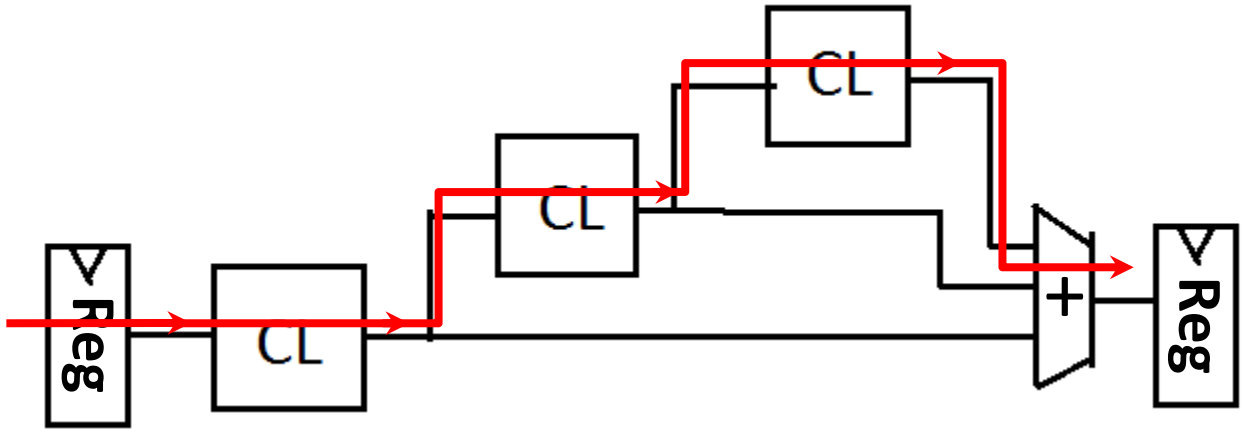
Max Delay = CLK-to-Q Delay
+ CL Delay
+ Setup Time

Min Period = Max Delay

Max Freq = 1/Min Period

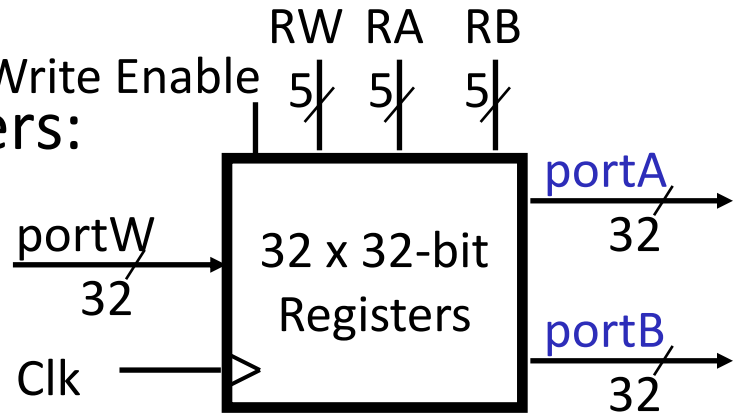
The Critical Path

- The *critical path* is the longest delay between *any* two registers in a circuit
- The clock period must be *longer* than this critical path, or the signal will not propagate properly to that next register



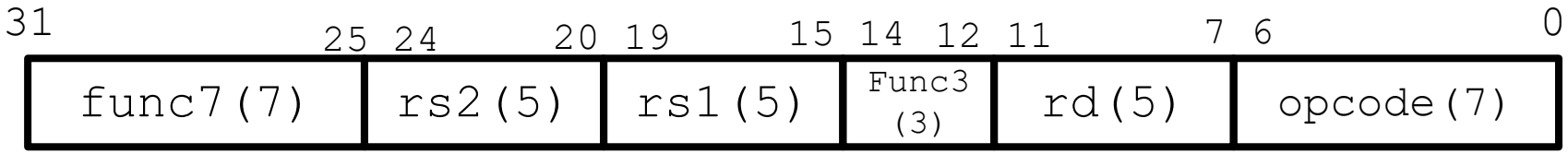
RISC-V *CPU Datapath, Control Intro*

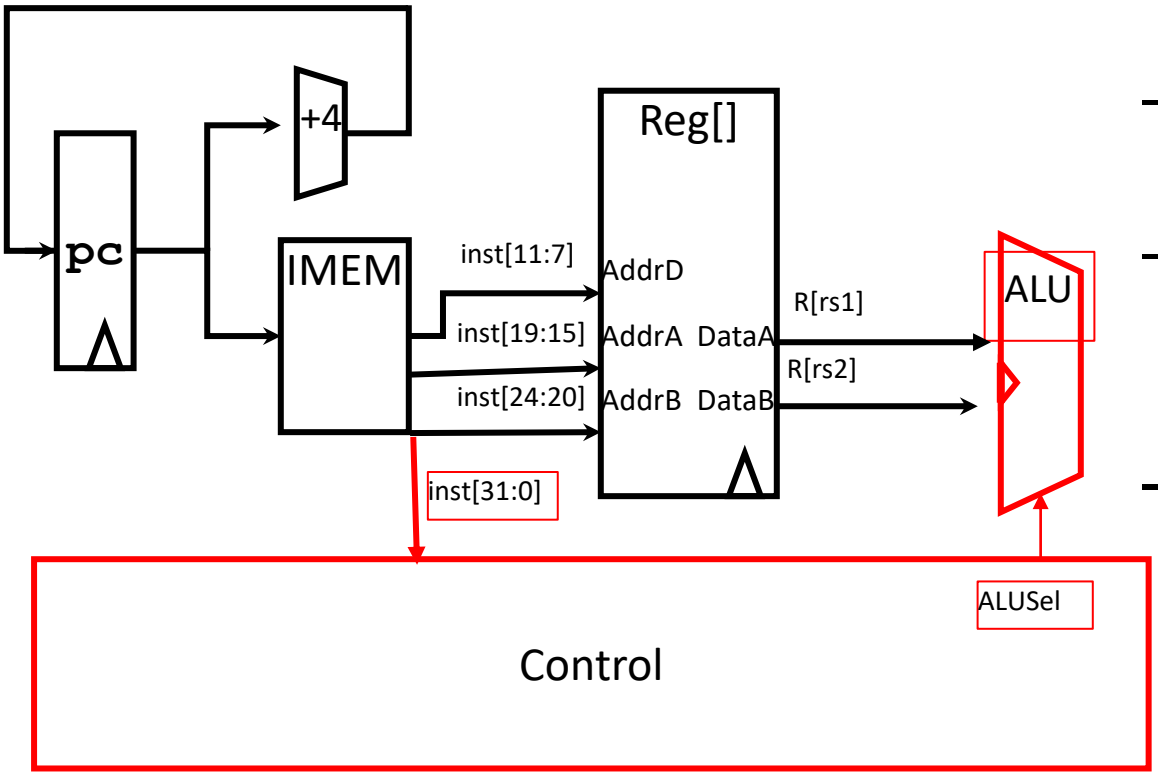
Storage Element: Register File



- *Register File* consists of 32 registers:
  - Output ports portA and portB
  - Input port portW
- Register selection
  - Place data of register RA (number) onto portA
  - Place data of register RB (number) onto portB
  - Store data on portW into register RW (number) when Write Enable is 1
- Clock input (CLK)
  - CLK is passed to all internal registers so they can be written to if they match RW and Write Enable is 1

Implementing R-Types

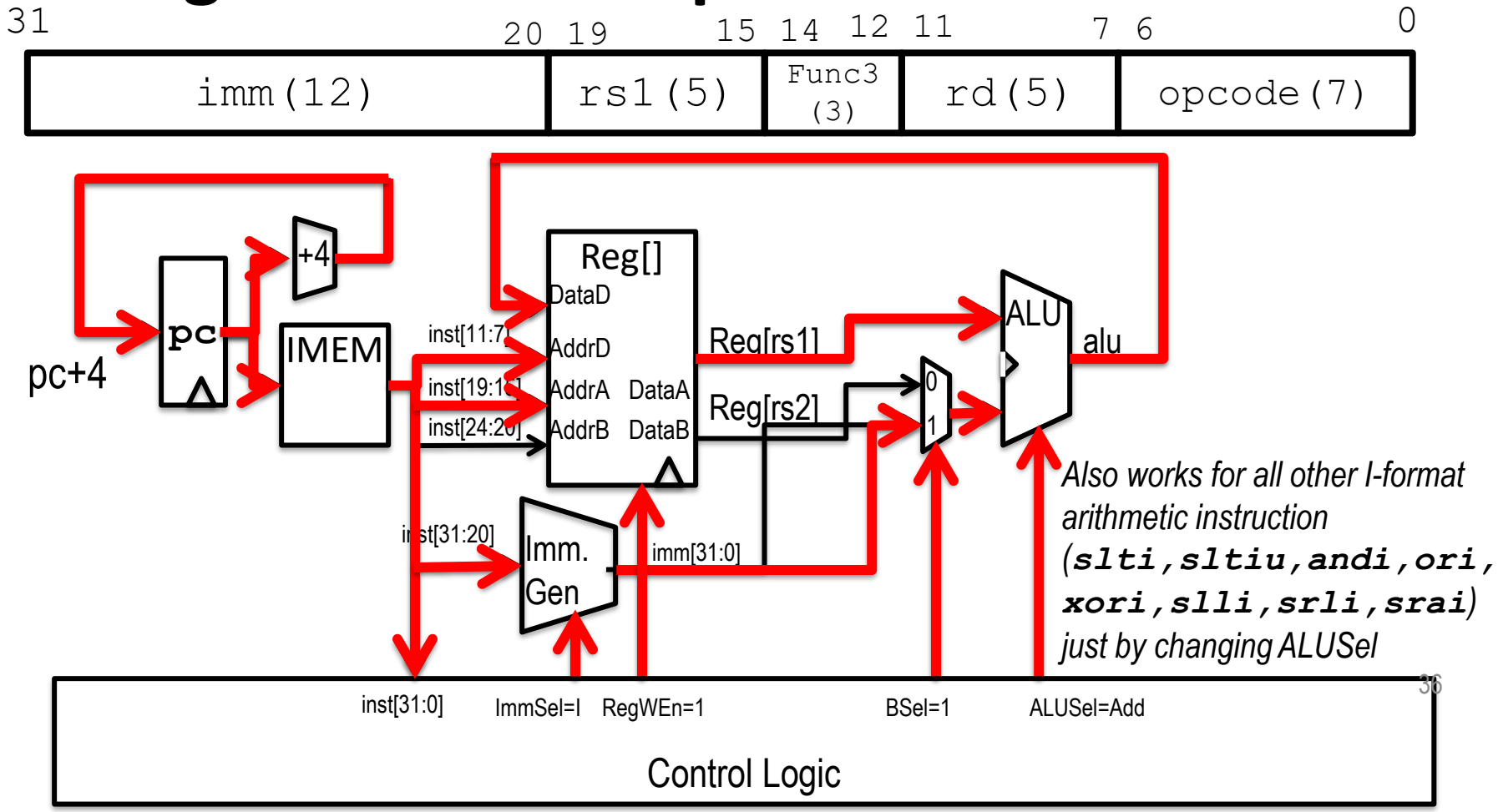




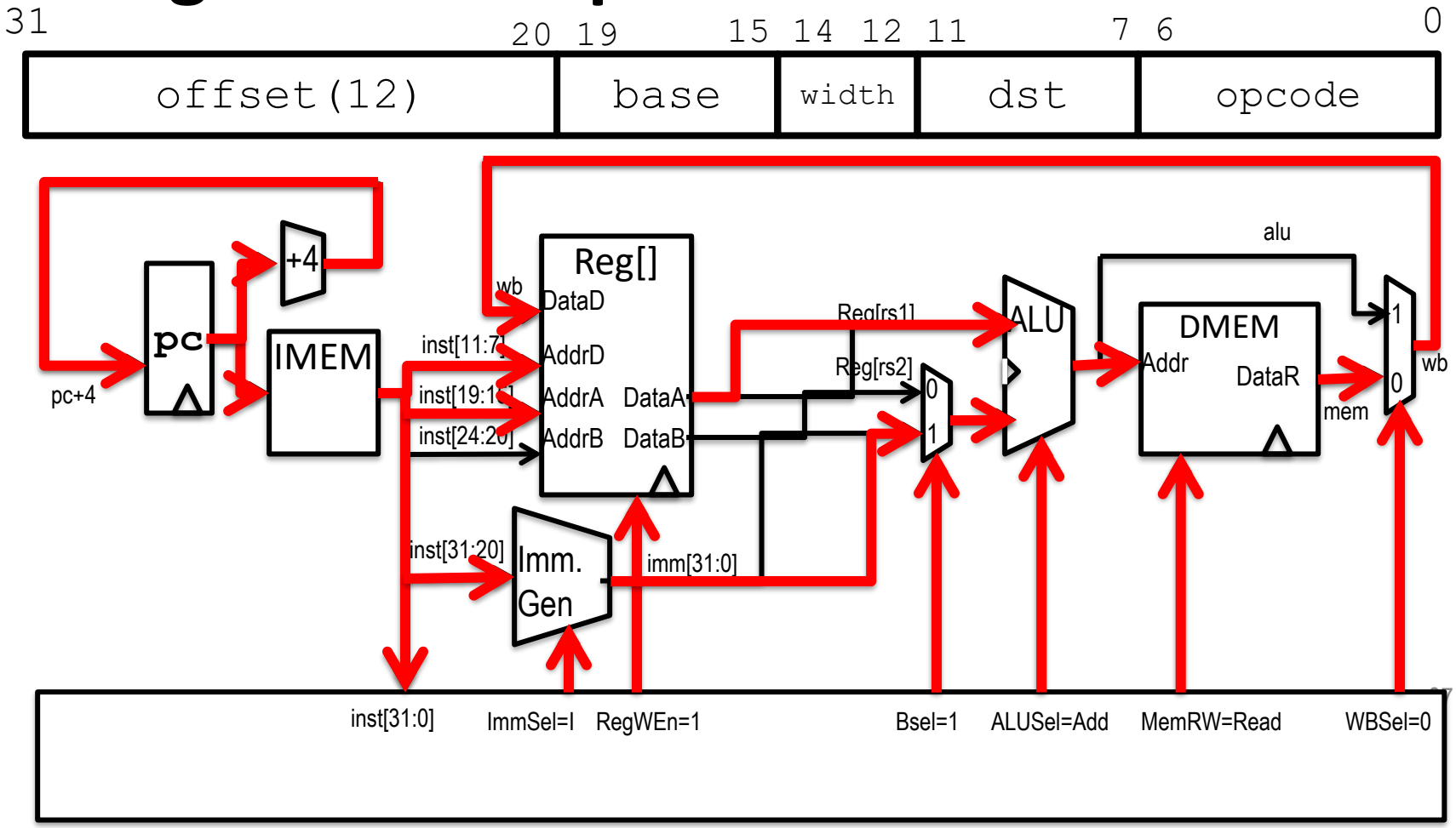
Perform operation

- New hardware: ALU (Arithmetic Logic Unit)
- Abstraction for adders, multipliers, dividers, etc.
- How do we know what operation to execute?
  - Our first control bit! ALUSel(ect)

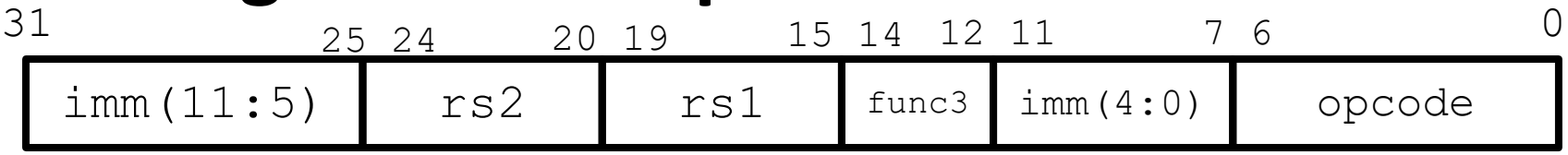
Adding addi to datapath

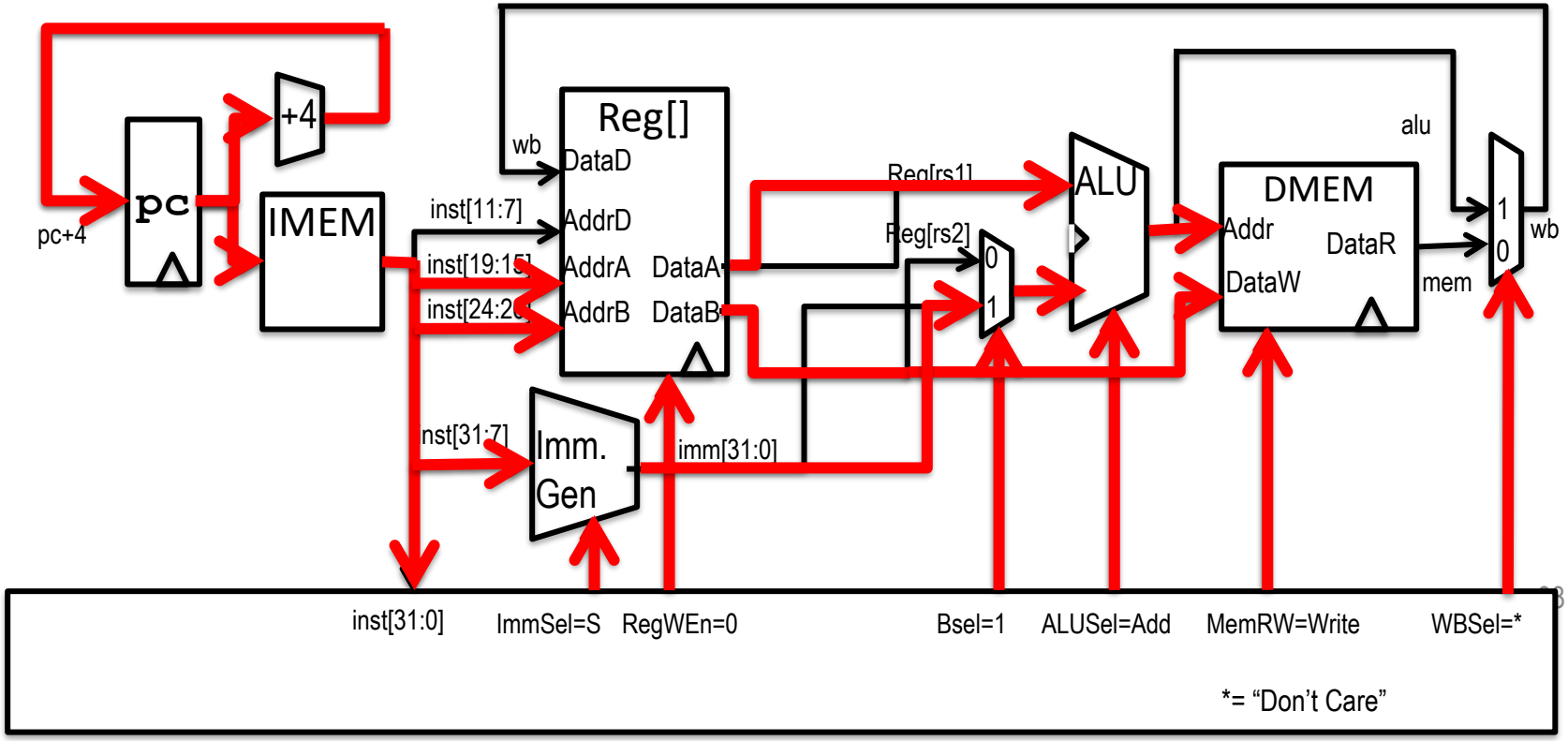


Adding `lw` to datapath

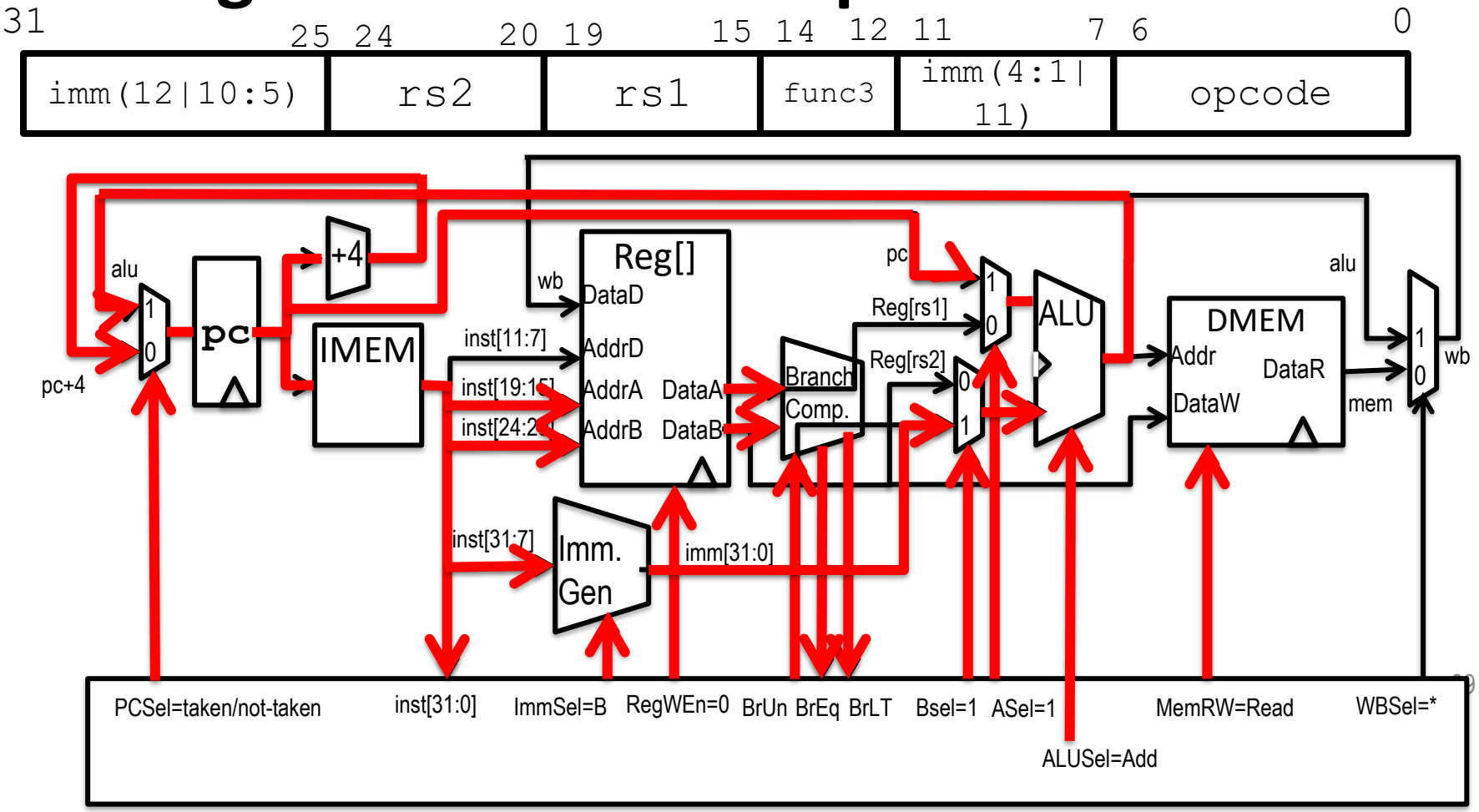


Adding sw to datapath

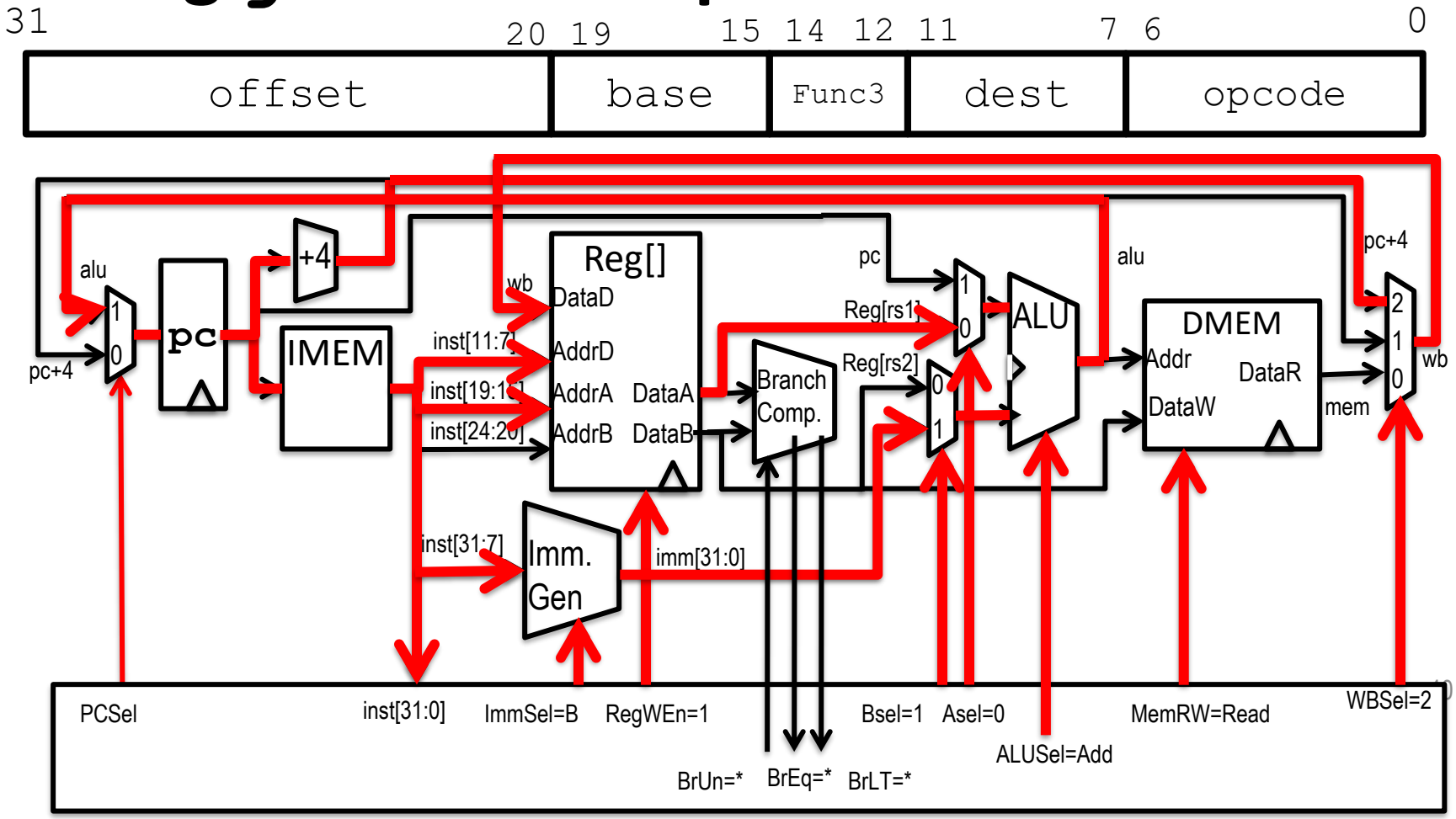




Adding branches to datapath

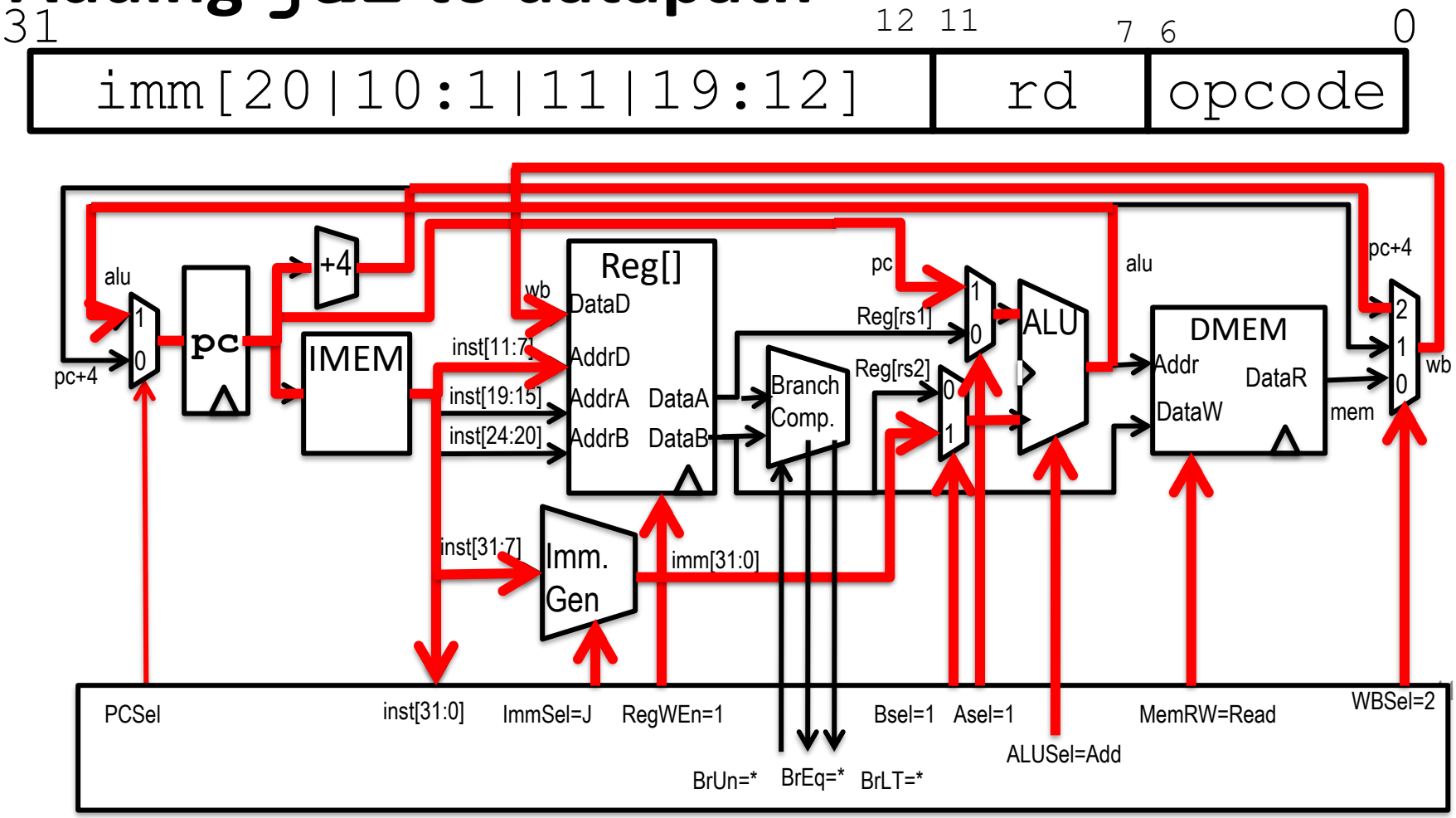


Adding `jalr` to datapath



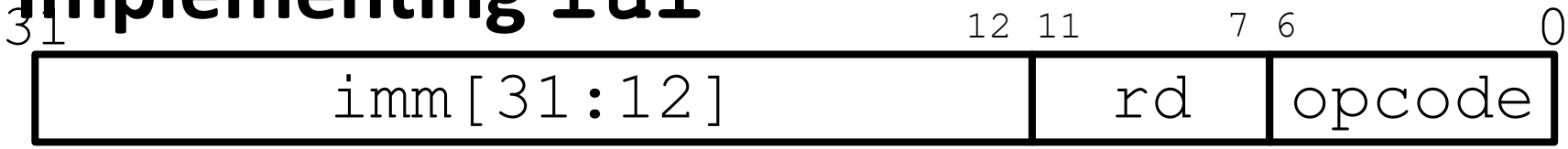
- Writes PC+4 to `dest` (return address)
- Sets PC = `base` + `offset`

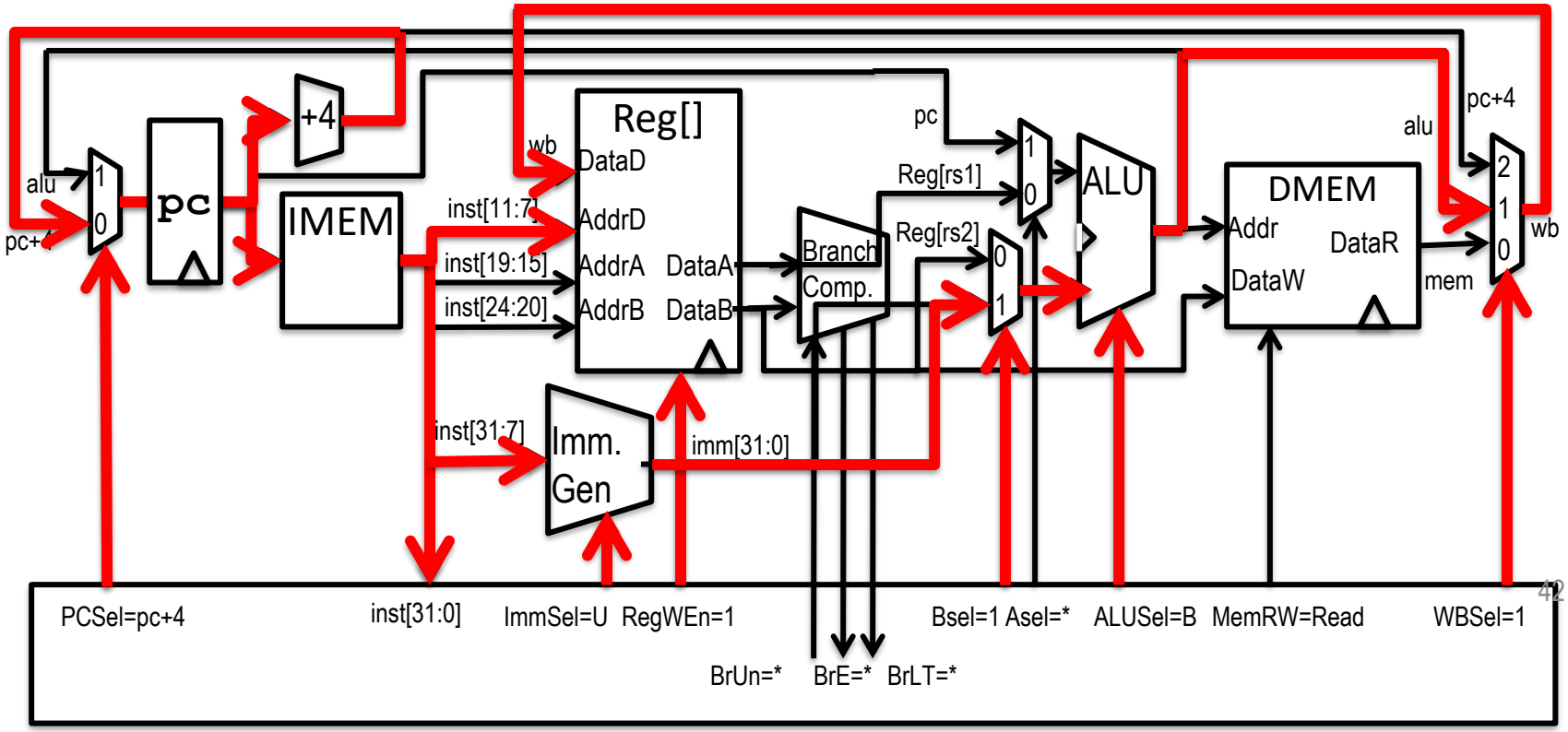
Adding `jal` to datapath



- `jal` saves PC+4 in register `rd` (the return address)
- Set PC = PC + offset (PC-relative jump)

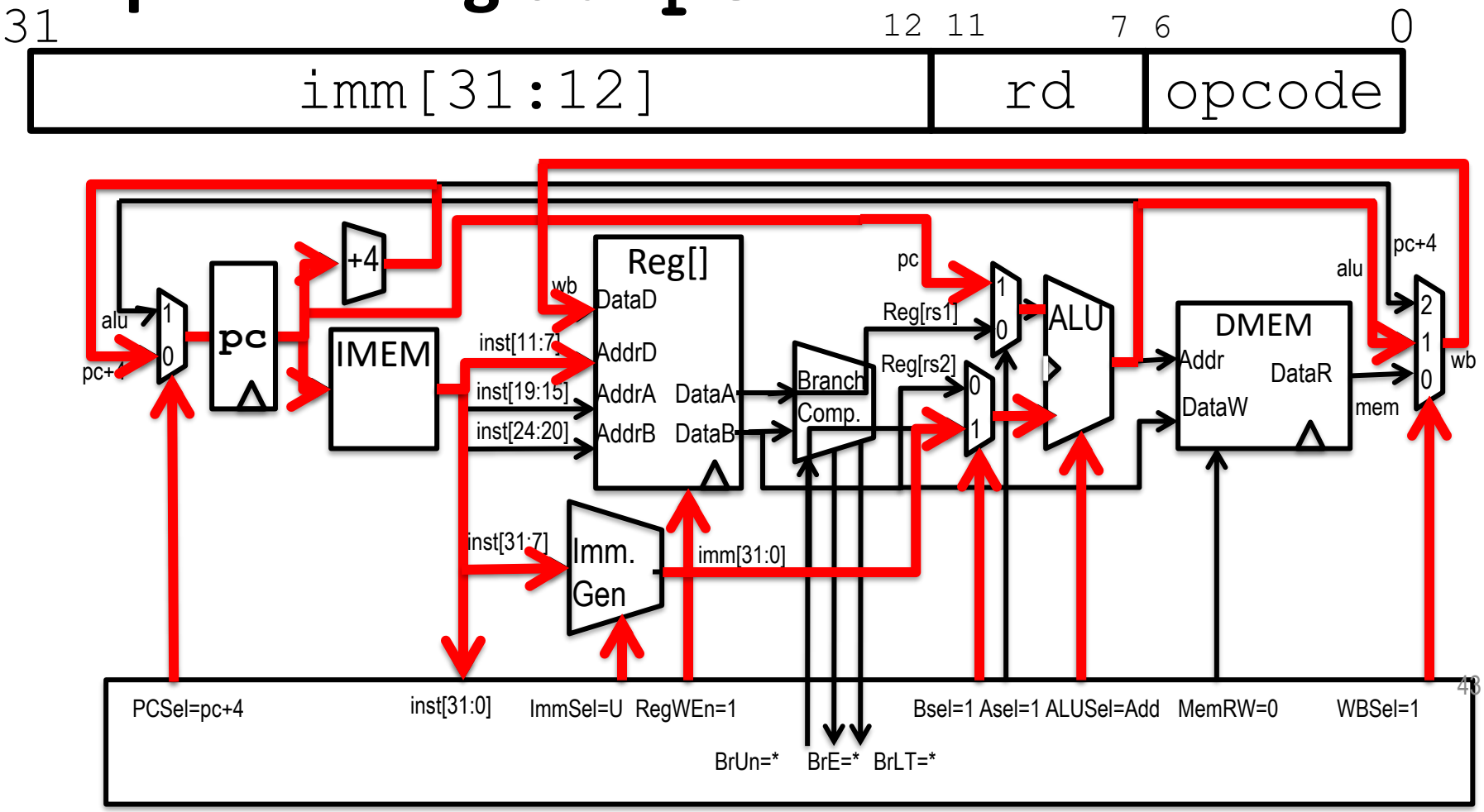
Implementing `lui`





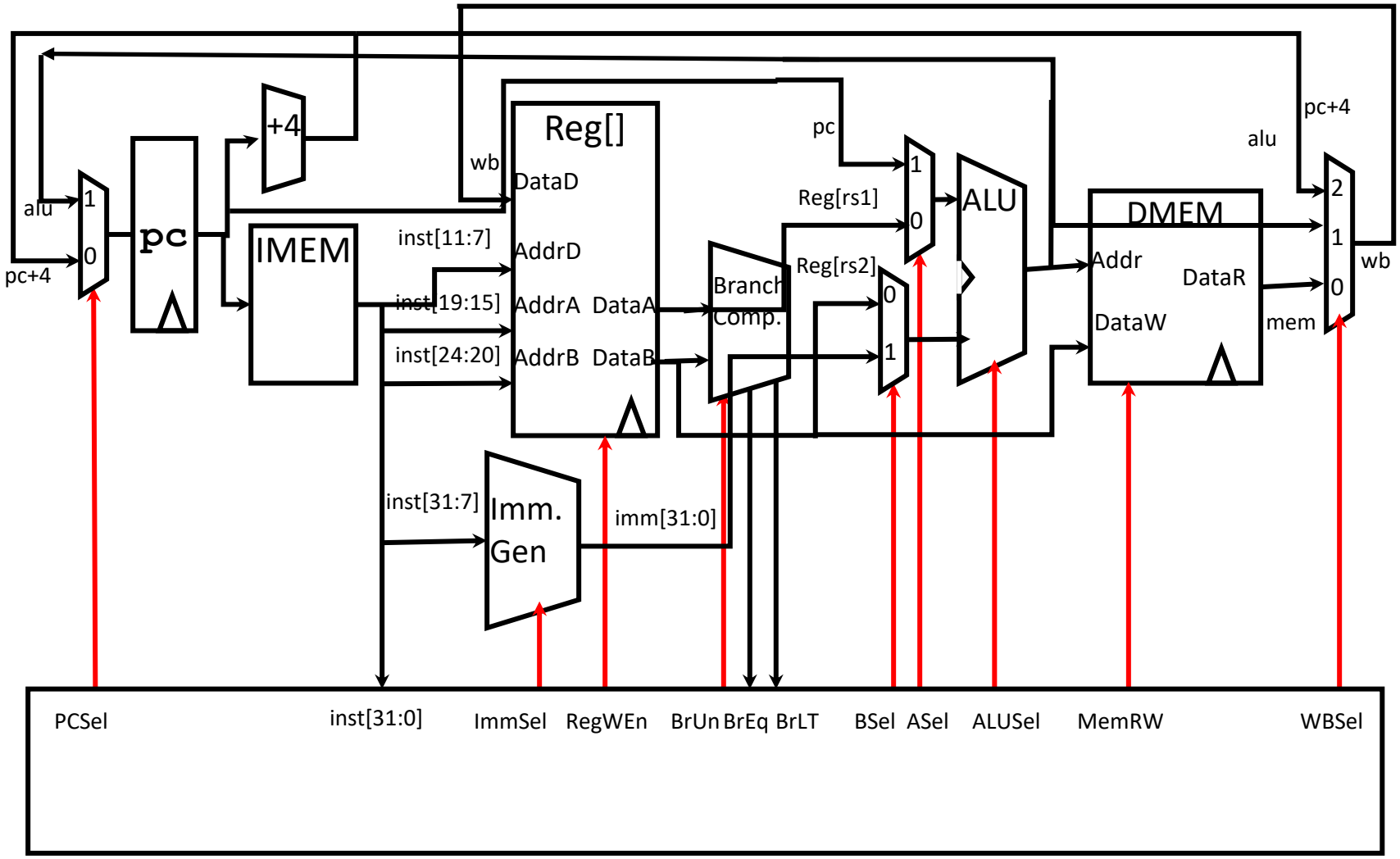
`lui` writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits

Implementing auipc



Adds upper immediate value to PC and places result in destination register

Control Signals: ADD



ADD: Control Signals

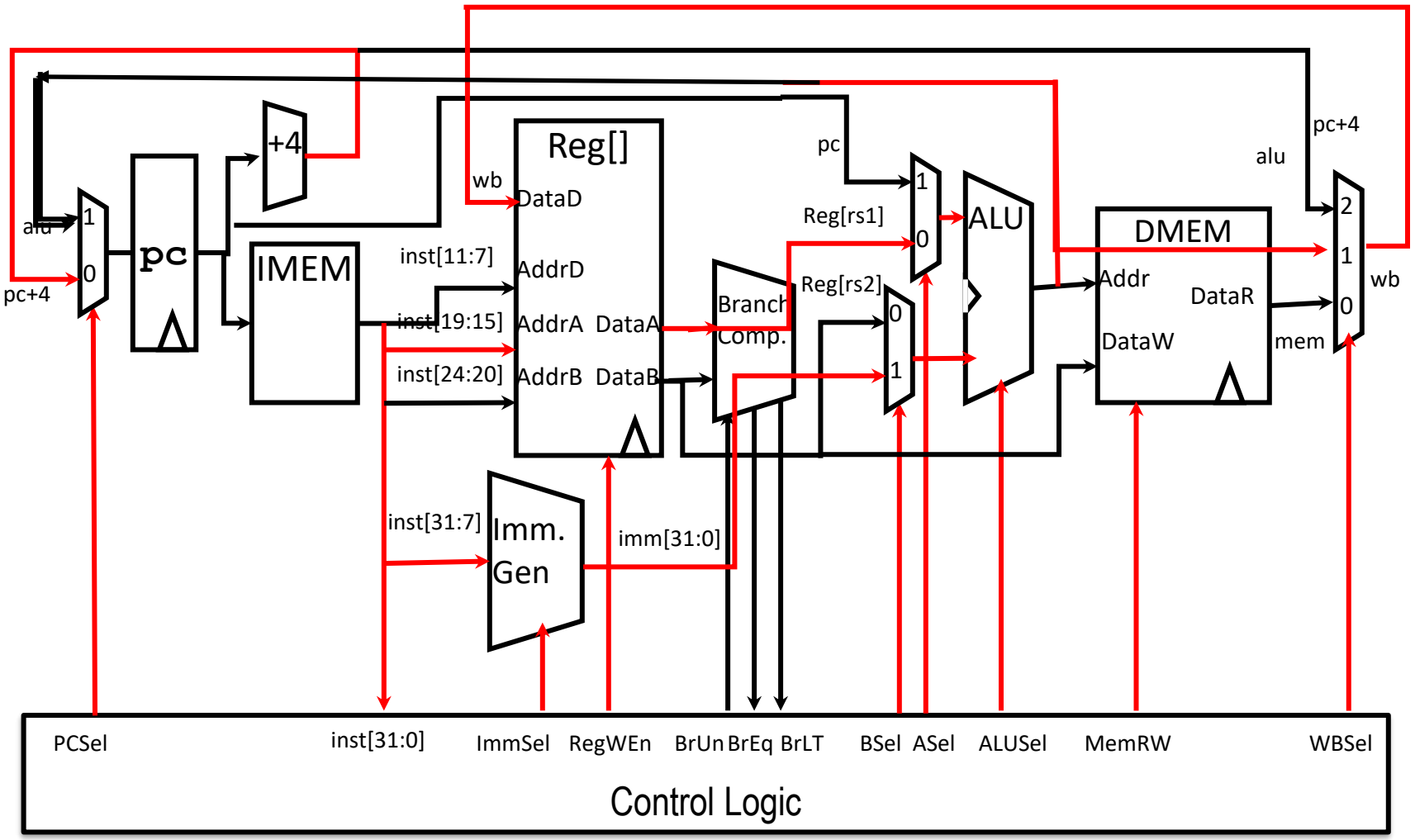
Here are the signals and values we've compiled for our ADD instruction:

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4	*	*	Reg	Reg	Add	Read	1 (Y)	ALU

(green = left 3 cols = control INPUTS)

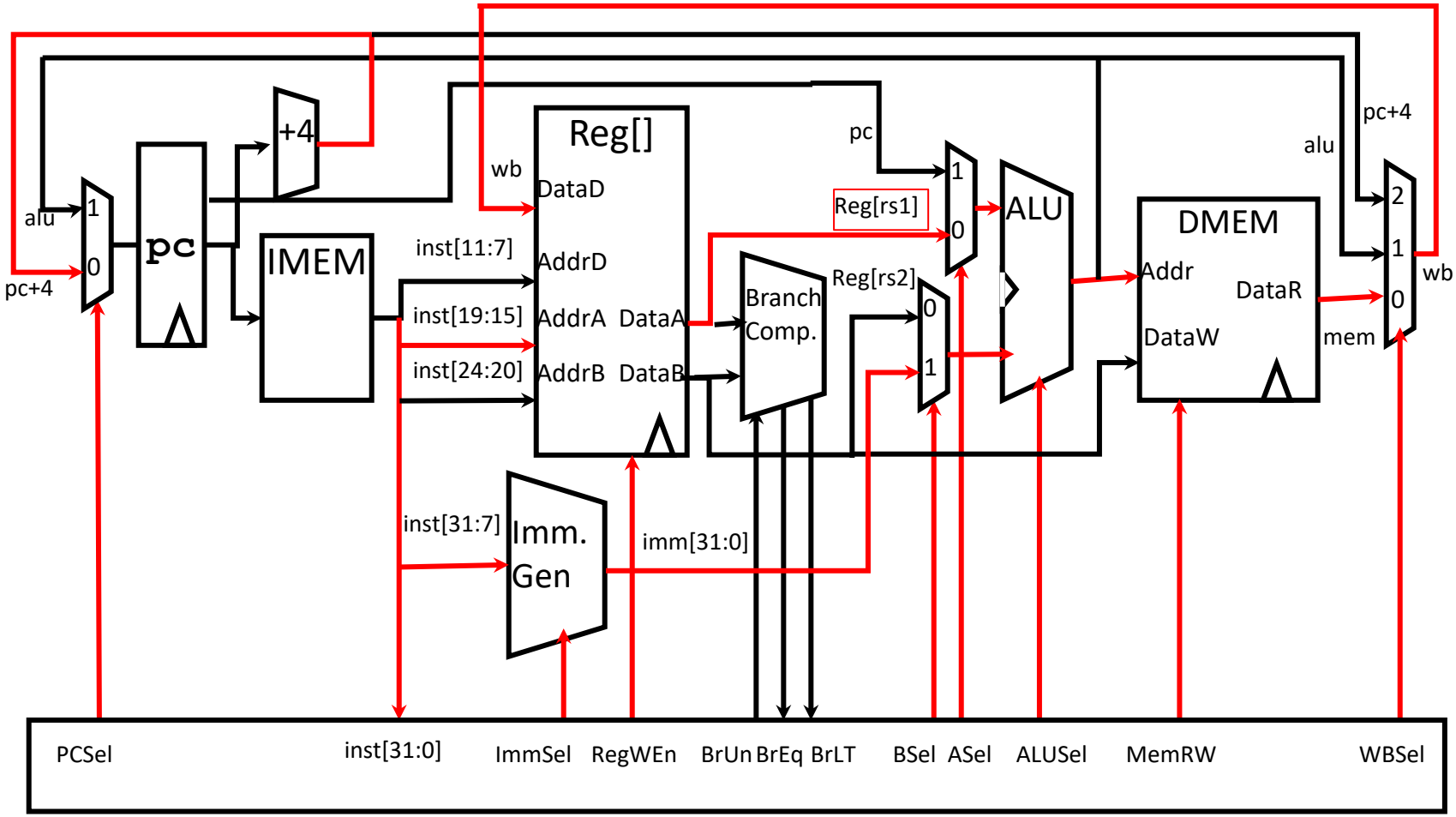
(orange = right 9 cols = control OUTPUTS)

addi datapath



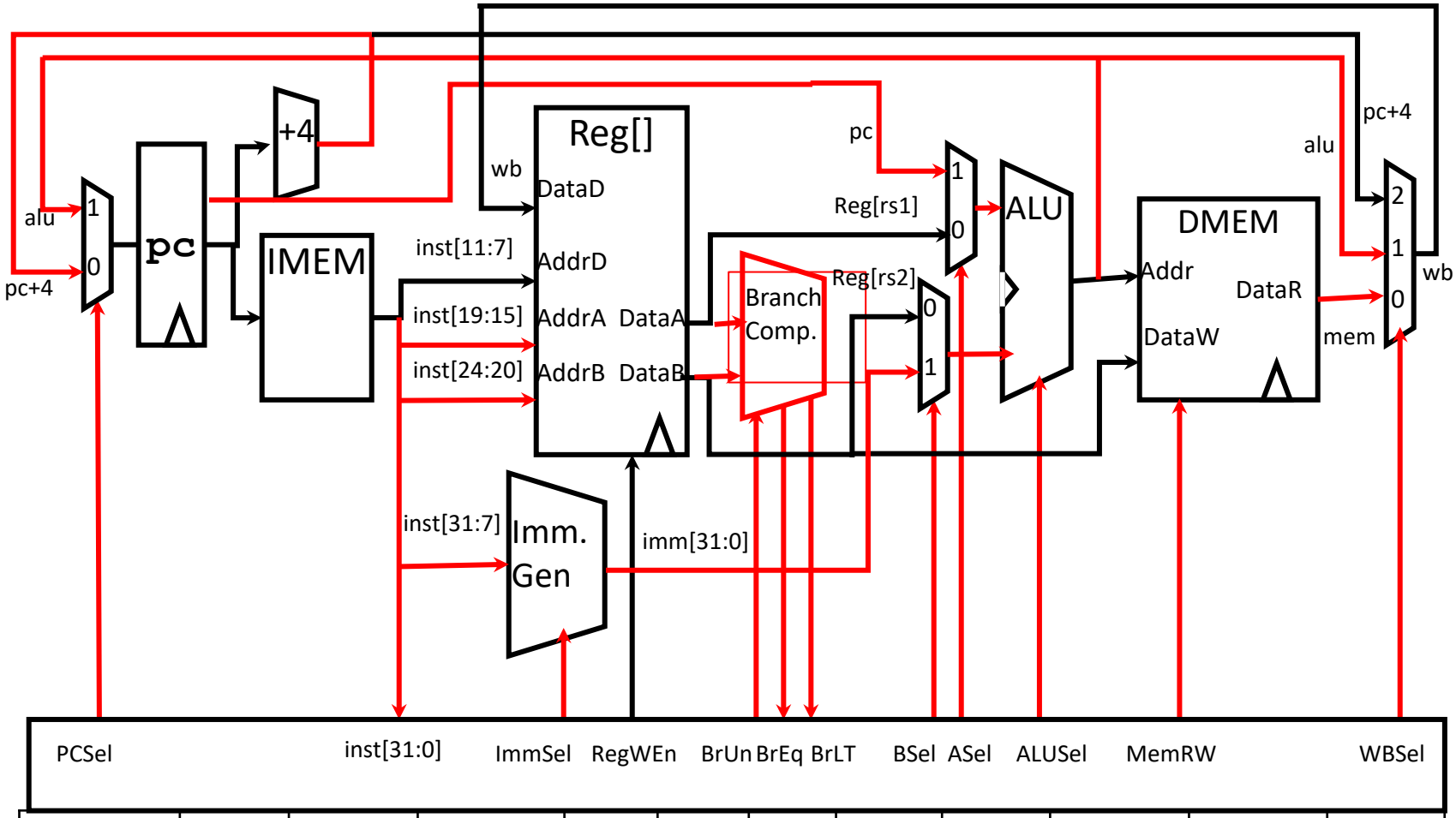
Inst[31:0]	PCSel	ImmSel	RegWEn	Br Un	Br LT	Br Eq	BSel	ASel	ALUSel	MemRW	WBSel
addi	+4	I	1	*	*	*	Imm	Reg	Add	Read	ALU

lw datapath



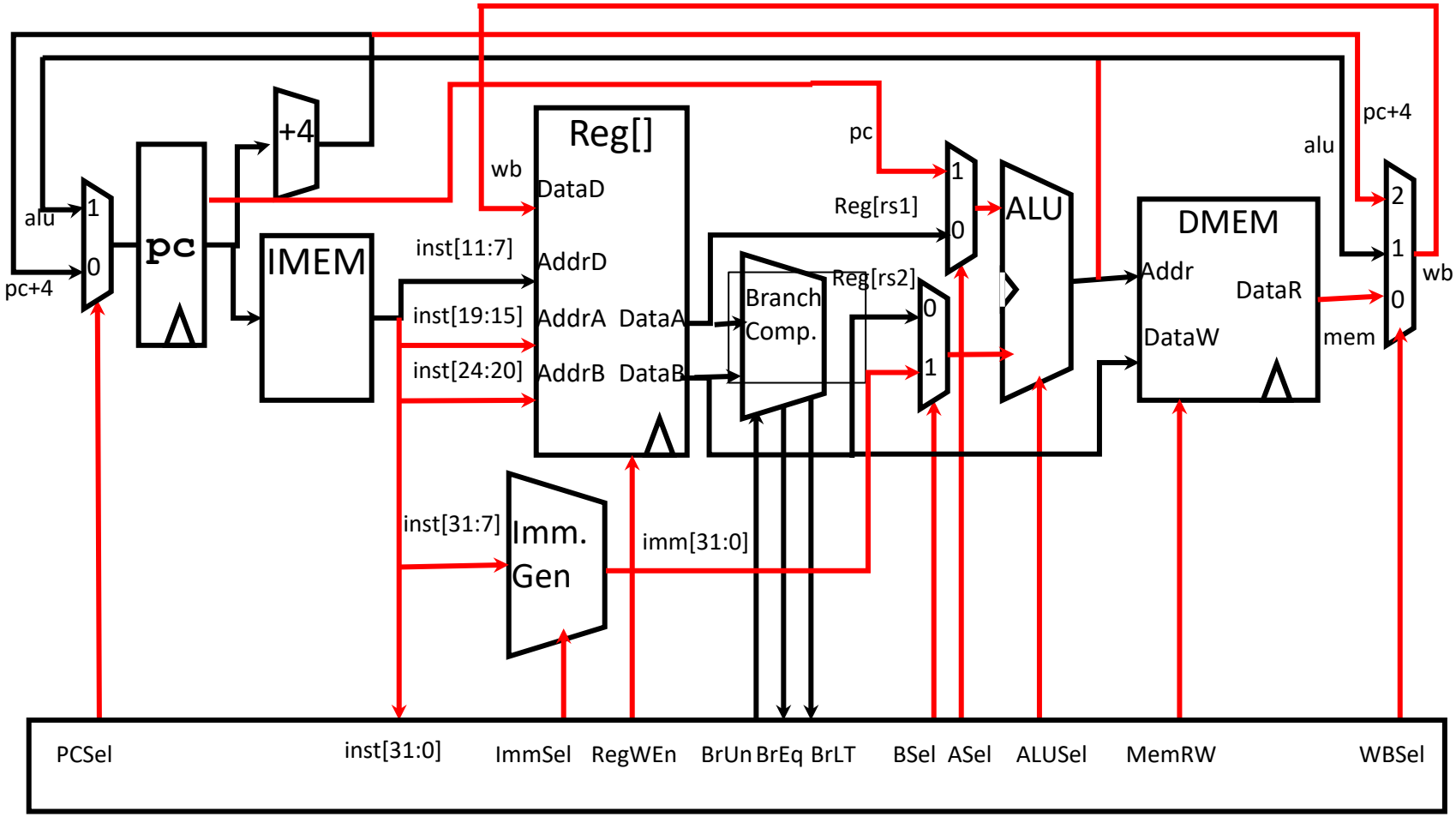
Inst[31:0]	PCSel	ImmSel	RegWEn	Br Un	Br Eq	Br LT	BSel	ASel	ALUSel	MemRW	WBSel
lw	+4	I	1	*	*	*	Imm	Reg	Add	Read	Mem

Br datapath



Inst[31:0]	PCSel	ImmSel	RegWEn	BrUn	BrEq	BrLT	BSel	ASel	ALUSel	MemRW	WBSel
beq	+4	B	0	*	0	*	Imm	PC	Add	Read	*
beq	ALU	B	0	*	1	*	Imm	PC	Add	Read	*

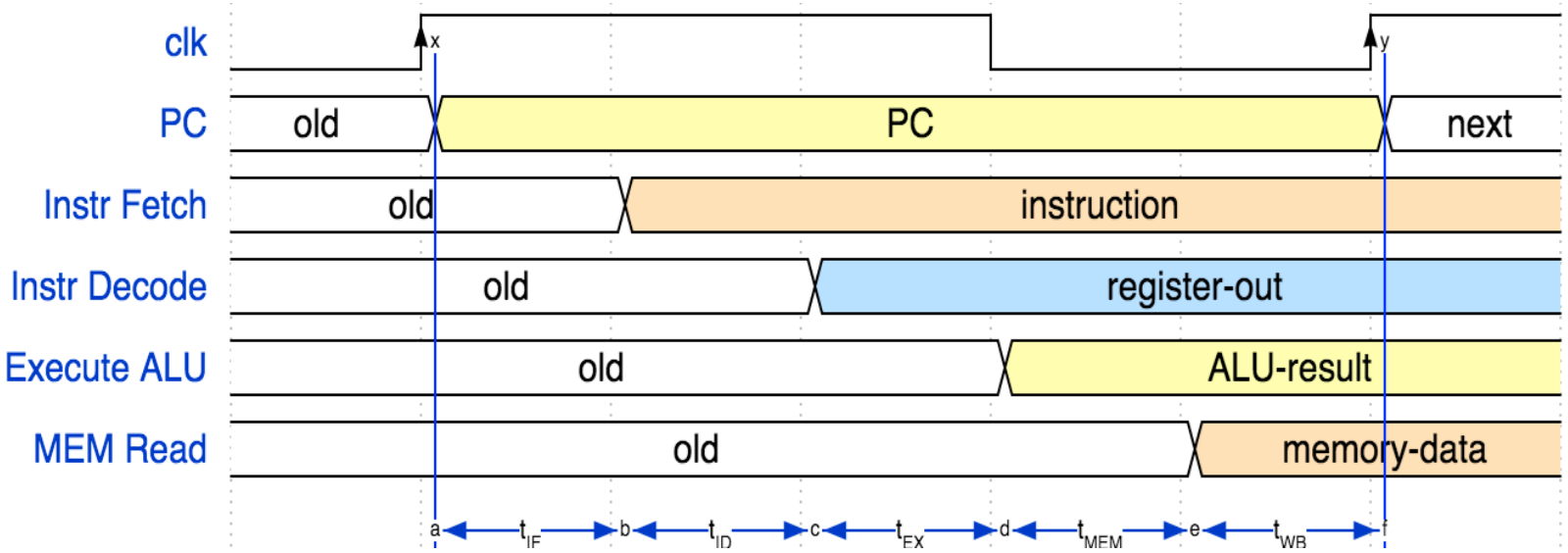
jal datapath



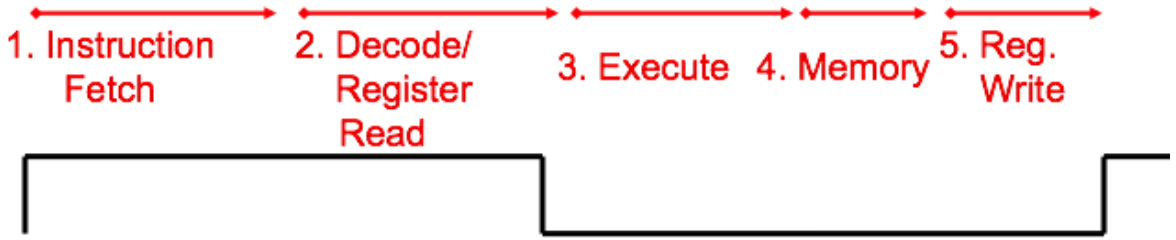
Inst[31:0]	PCSel	ImmSel	RegWEn	Br Un	Br Eq	BrLT	BSel	ASel	ALUSel	MemRW	WBSel
jal	ALU	J	1	*	*	*	Imm	PC	Add	Read	PC+4

Inst[31:0]	PCSel	ImmSel	RegWEn	Br Un	Br Eq	Br LT	BSel	ASel	ALUSel	MemRW	WBSel
add	+4	*	1 (Y)	*	*	*	Reg	Reg	Add	Read	ALU
sub	+4	*	1	*	*	*	Reg	Reg	Sub	Read	ALU
(R-R Op)	+4	*	1	*	*	*	Reg	Reg	*(Op)*	Read	ALU
addi	+4	I	1	*	*	*	Imm	Reg	Add	Read	ALU
lw	+4	I	1	*	*	*	Imm	Reg	Add	Read	Mem
sw	+4	S	0 (N)	*	*	*	Imm	Reg	Add	Write	*
beq	+4	B	0	*	0	*	Imm	PC	Add	Read	*
beq	ALU	B	0	*	1	*	Imm	PC	Add	Read	*
bne	ALU	B	0	*	0	*	Imm	PC	Add	Read	*
bne	+4	B	0	*	1	*	Imm	PC	Add	Read	*
blt	ALU	B	0	0	*	1	Imm	PC	Add	Read	*
bltu	ALU	B	0	1	*	1	Imm	PC	Add	Read	*
jalr	ALU	I	1	*	*	*	Imm	Reg	Add	Read	PC+4
jal	ALU	J	1	*	*	*	Imm	PC	Add	Read	PC+4
auipc	+4	U	1	*	*	*	Imm	PC	Add	Read	ALU

Instruction Timing



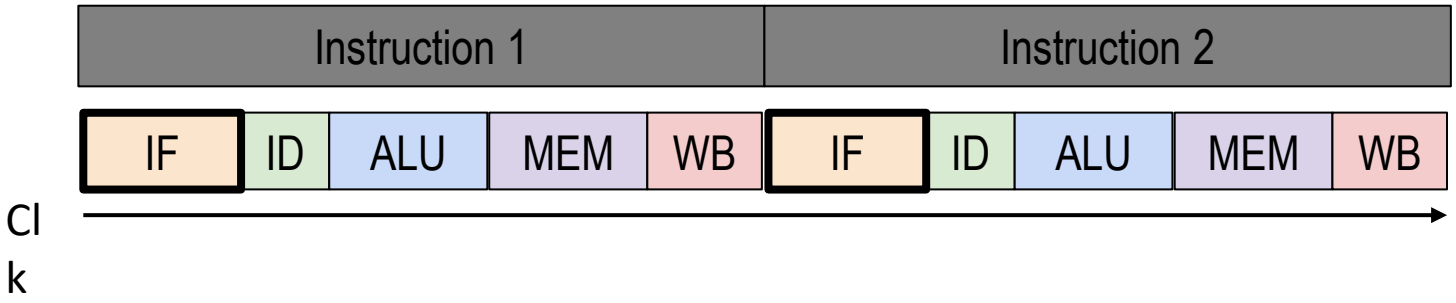
IF	ID	EX	MEM	WB	Total
IMEM	Reg Read	ALU	DMEM	Reg W	
200 ps	100 ps	200 ps	200 ps	100 ps	800 ps



Instruction Timing

Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X		X	600ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

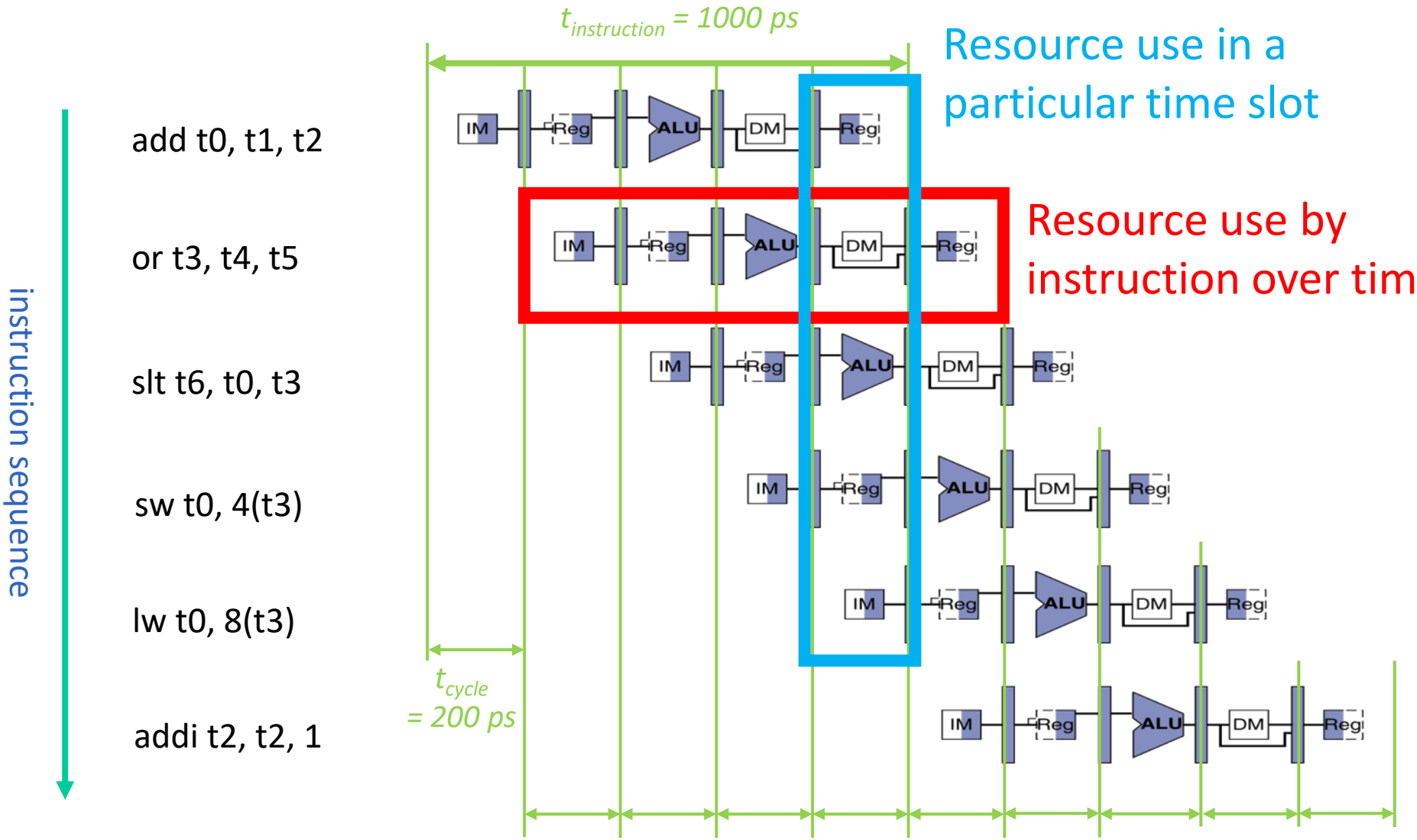
- Maximum clock frequency
  - f_{max} = 1/800ps = 1.25 GHz
- Most blocks idle most of the time! ex. "IF" active every 600ps



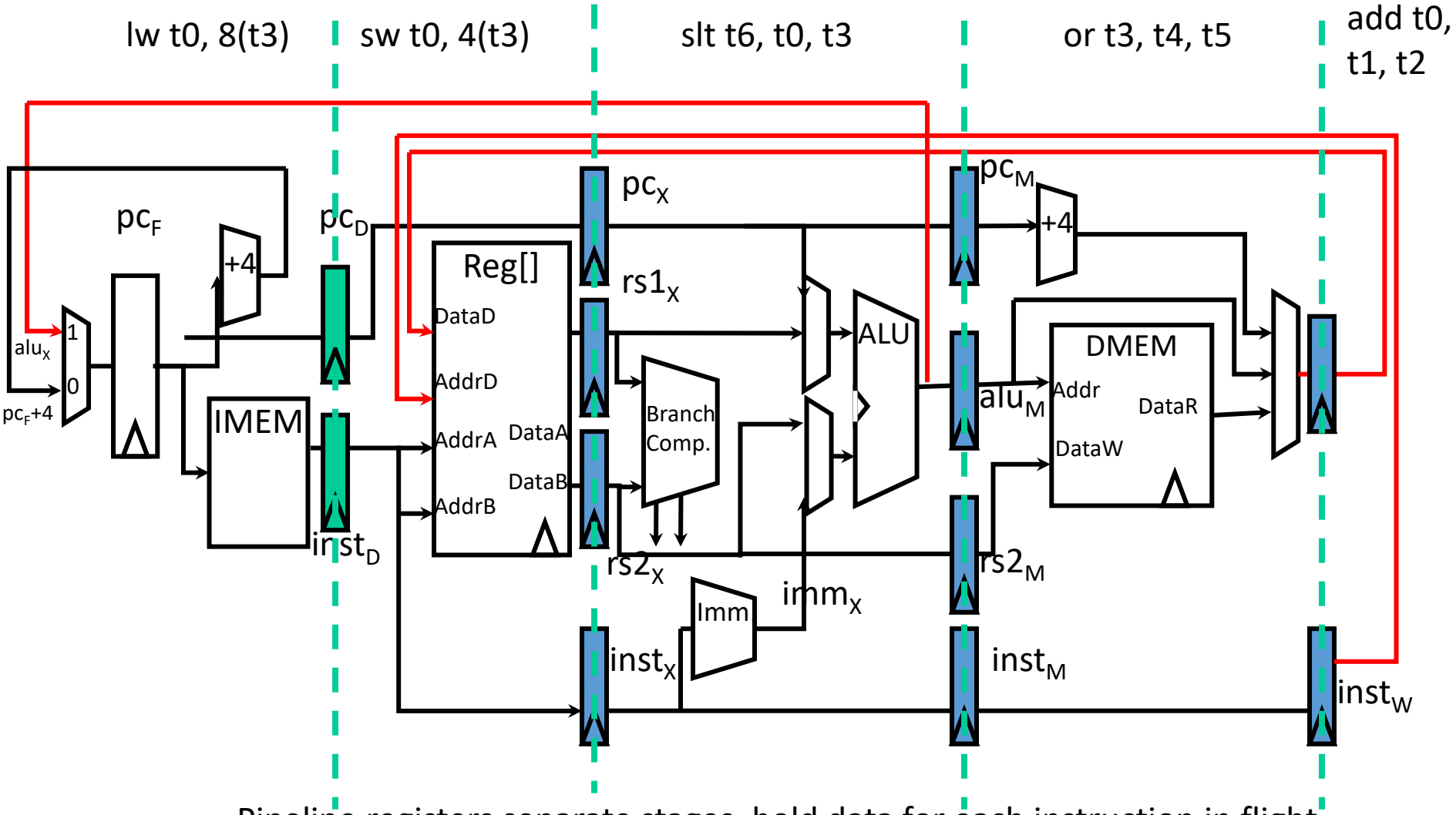
"Iron Law" of Processor Performance

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

RISC-V Pipeline



Each stage operates on different instruction



Pipeline registers separate stages, hold data for each instruction in flight

RISC-V Pipeline Example

Address	Inst | Cycle	0	1	2	3	4	5	6	7
0x00	add a1,a2,a3	IF	ID	EX	MEM	WB			
0x04	addi a4,a5,0x2f7		IF	ID	EX	MEM	WB		
0x08	sub s4,s0,s3			IF	ID	EX	MEM	WB	
0x0C	or s1,s2,s5				IF	ID	EX	MEM	WB

Question: Assume the stage times shown below. Suppose we *remove loads and stores* from our ISA. Consider going from a single-cycle implementation to a **4-stage** pipelined version.

Instr Fetch	Reg Read	ALU Op	Mem Access	Reg Write
200ps	100 ps	200ps	200ps	100 ps

1) The *latency* will be 1.33x slower.
2) The *throughput* will be 3x faster.

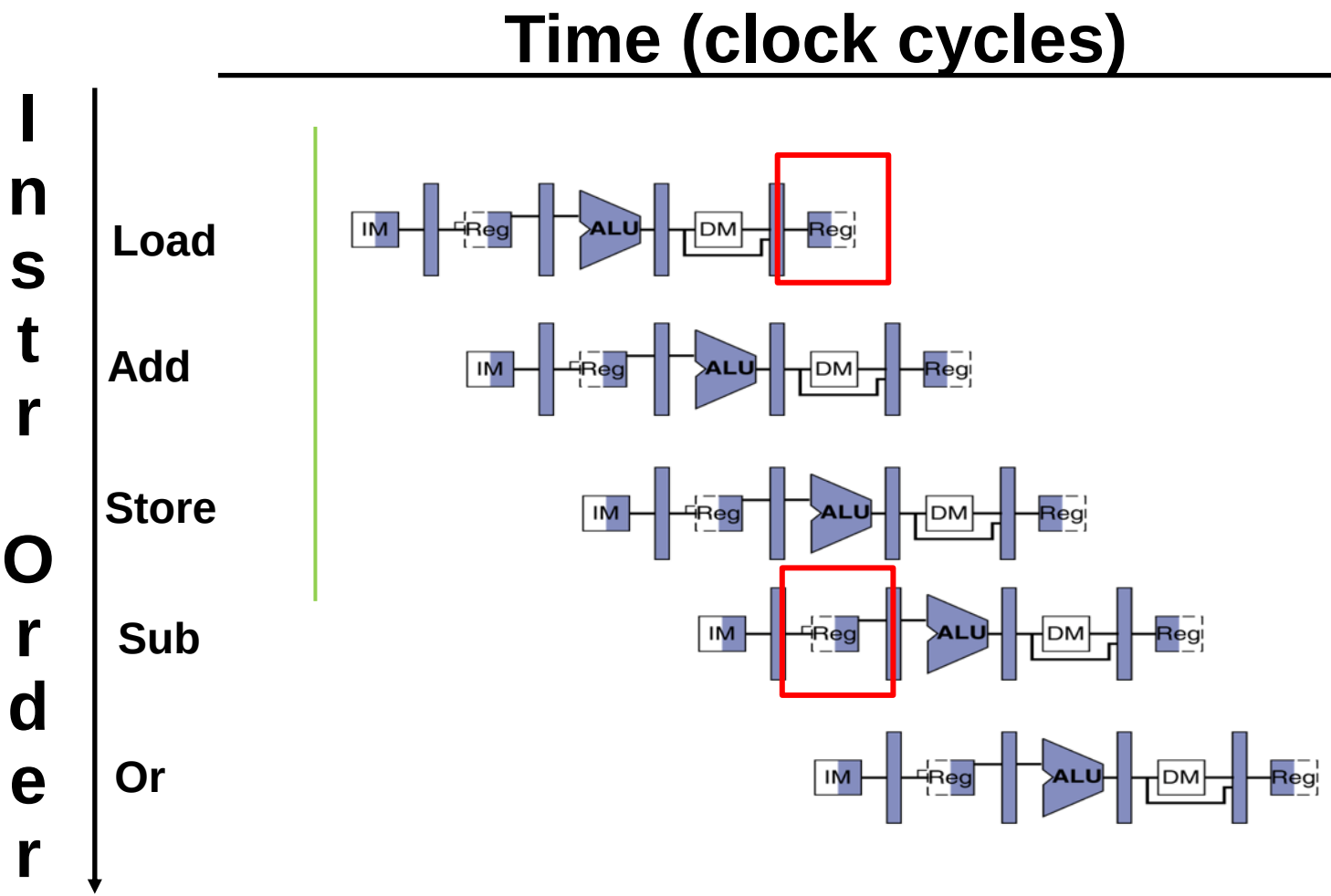
Pipeline Hazards

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

1) *Structural hazard*
   - A required resource is busy (e.g. needed in multiple stages)
2) *Data hazard*
   - Data dependency between instructions
   - Need to wait for previous instruction to complete its data write
3) *Control hazard*
   - Flow of execution depends on previous instruction

Structural Hazard: Register File

- RegFile: Used in ID and WB



RISC-V Pipeline: Regfile Structural Hazard

Addr	Inst | Cycle	0	1	2	3	4	5	6	7	8	9	10
0x00	addi a0, zero, 5	IF	ID	EX	MM	WB						
0x04	addi a1, a4, 5		IF	ID	EX	MM	WB					
0x08	addi a2, a5, 5			IF	ID	EX	MM	WB				
0x0C	addi a3, a6, 5				IF	ID	ID	EX	MM	WB		

Data Hazard: Read after Write

Addr	Inst | Cycle	0	1	2	3	4	5	6	7	8	9	10
0x00	add s0, s1, s2	IF	ID	EX	MM	WB						
0x04	sub s4, s0, s3		IF	ID	-	-	EX	MM	WB			
0x08	and s5, s0, s6			IF	IF	IF	ID	EX	MM	WB		
0x0C	or s7, s0, s8						IF	ID	EX	MM	WB	

Data Hazard Solution: Forwarding

- Forward result as soon as it is available, even though it's not stored in RegFile yet

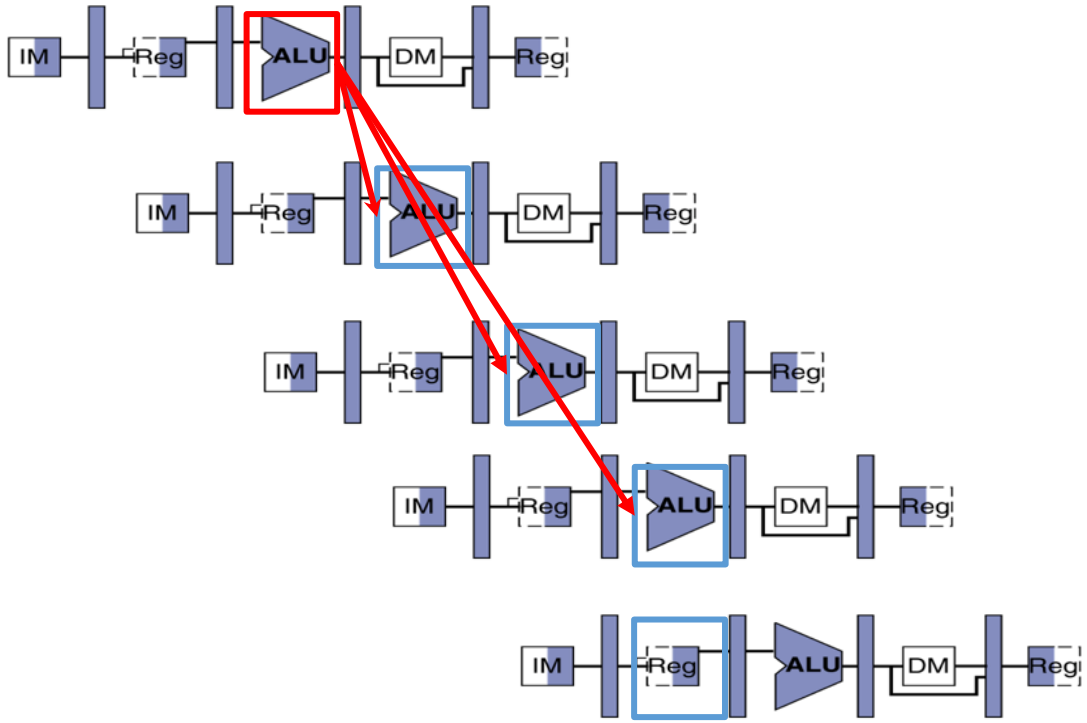
add s0, s1, s2

sub s4, s0, s3

and s5, s0, s6

or s7, s0, s8

xor s9, s0, s10



Forwarding: get operand from pipeline stage, rather than register file

Data Hazard with Forwarding

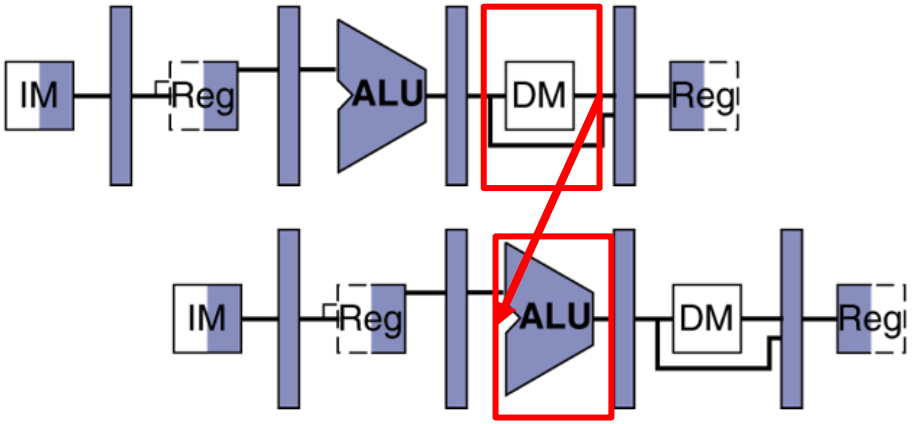
Addr	Inst | Cycle	0	1	2	3	4	5	6	7	8	9	10
0x00	add s0, s1, s2	IF	ID	EX	MM	WB						
0x04	sub s4, s0, s3		IF	ID	EX	MM	WB					
0x08	and s5, s0, s6			IF	ID	EX	MM	WB				
0x0C	or s7, s0, s8				IF	ID	EX	MM	WB			

Data Hazard: Loads

- **Recall:** Dataflow backwards in time are hazards

lw t0, 0(t1)

sub t3, t0, t2



- Can't solve all cases with forwarding
  - Must *stall* instruction dependent on load (sub), then forward after the load is done (more hardware)

Control Hazards: Taken Branch & ecall

Address	Ins-Cycle	0	1	2	3	4	5	6	7	8	9	10	11
0x00	add a2, a1, a0	IF	ID	EX	MM	WB							
0x04	bne a2, zero, 0x00000010		IF	ID	EX	MM	WB						
0x08	addi a3, zero, 1			IF	ID								
0x0c	jal zero, 0x00000014				IF								
0x10	addi a3, zero, 0					IF	ID	EX	MM	WB			
0x14	ecall						IF	ID	EX	-	-	MM	WB

Not-Taken Branch

Address	Ins-Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12
0x00	add a2, a1, a0	IF	ID	EX	MM	WB								
0x04	beq a2, zero, 0x00000010		IF	ID	EX	MM	WB							
0x08	addi a3, zero, 1			IF	ID	EX	MM	WB						
0x0c	jal zero, 0x00000014				IF	ID	EX	MM	WB					
0x10	addi a3, zero, 0					IF	ID							
0x14	ecall						IF	IF	ID	EX	-	-	MM	WB