Combinational Logic
Sequential Logic
CPU Datapath

CMPT 295 Week 10

Synchronous Digital Systems (SDS)

Hardware of a processor (e.g., RISC-V) is an example of a Synchronous Digital System

Synchronous:

- All operations coordinated by a central clock
  - "Heartbeat" of the system (processor frequency)

Digital:

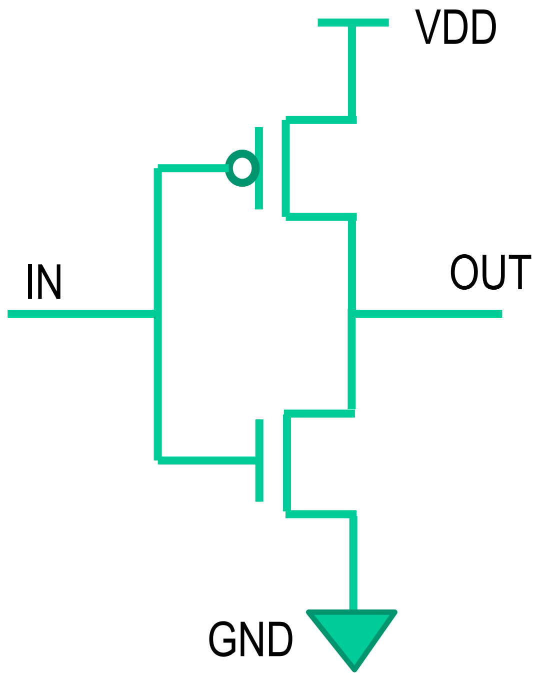
- Represent all values with two discrete values
- Electrical signals are treated as 1's and 0's
  - High/Low voltage represent True/False, 1/0

Moore's Law

- **Original Version (1965):** Since the integrated circuit was invented, the number of transistors in an integrated circuit has roughly doubled every year; this trend would continue for the foreseeable future
- 1975: Revised - circuit complexity doubles every two years
- **Hardware Trend:** Hardware gets more powerful every year (due to technology advancement and the hard work of many engineers)
- **Software Trend:** Software gets faster and uses more resources (And has to keep up with ever-changing hardware)
- Digital circuits are used to build hardware

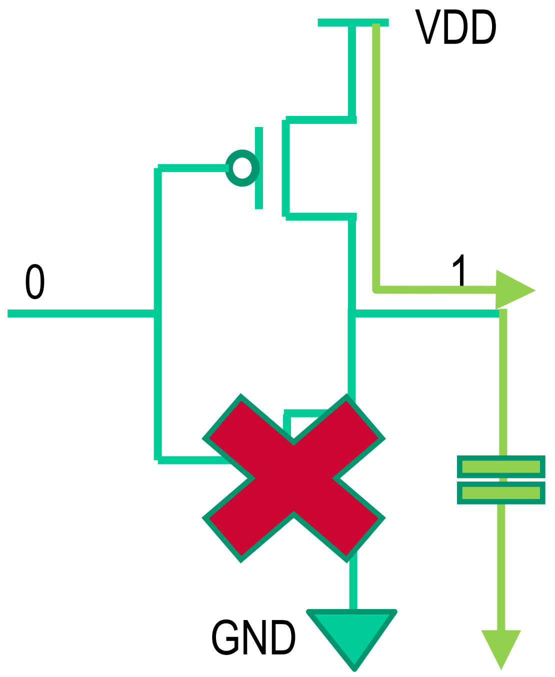
Transistors to Gates Example: Inverter

- CMOS technology
- Two transistors:
  - NMOS (top): turns on when input is 0 (low V)
  - PMOS (bottom): turns on when input is 1 (high V)



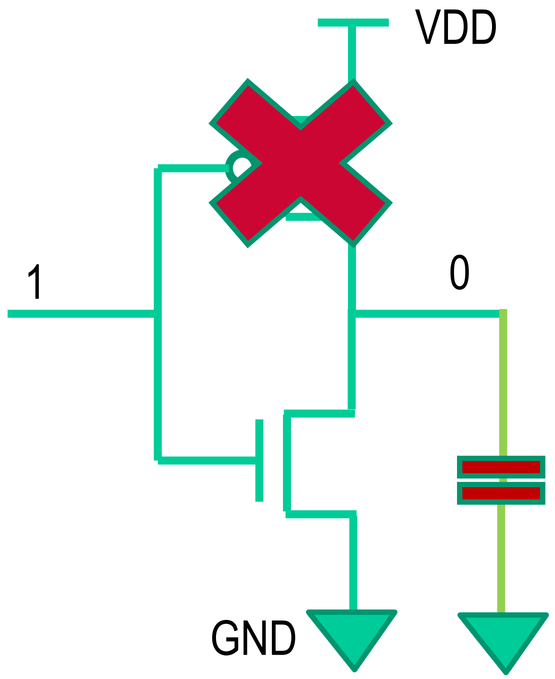
Transistors to Gates Example: Inverter

- Input = 0
- Top transistor turned on, bottom transistor turned off -> Output connected to VDD, capacitor charged
- Output = 1



Transistors to Gates Example: Inverter

- Input = 1
- Top transistor turned off, bottom transistor turned on -> Output connected to GND, capacitor discharged
- Output = 0



Inverter is commonly called “NOT gate”

Combinational vs. Sequential Logic

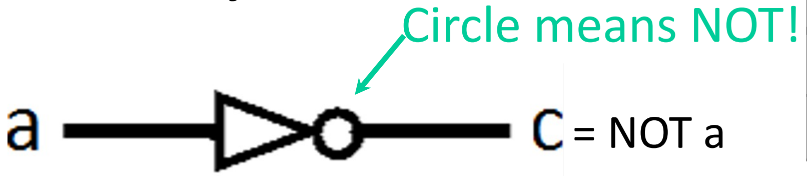
- *Digital Systems* consist of two basic types of circuits:
  - Combinational Logic (CL)
    - Output is a function of the inputs only, not the history of its execution
    - Example: add A, B (ALUs)
  - Sequential Logic (SL)
    - Circuits that "remember" or store information
    - Also called "State Elements"
    - Example: Memory and registers

Simple Logic Gates

Truth Table

- Special names and symbols:

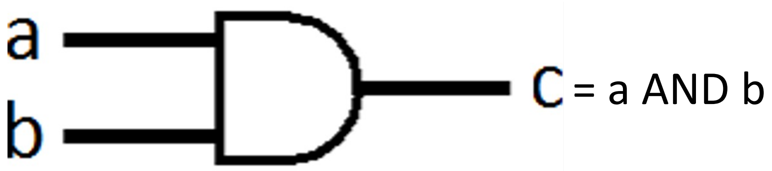
NOT



a	NOT a
0	1
1	0

True if input is false

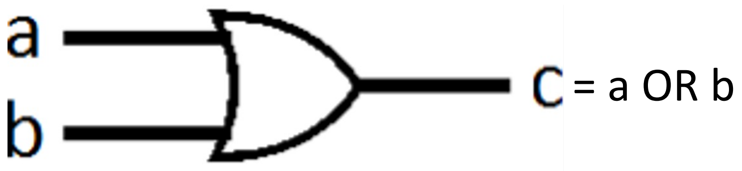
AND



a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

True if both inputs are true

OR



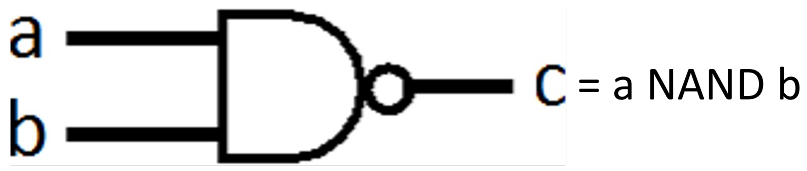
a	b	A OR b
0	0	0
0	1	1
1	0	1
1	1	1

True if at least one input is true

More Simple Logic Gates

Inverted versions are easier to implement in CMOS

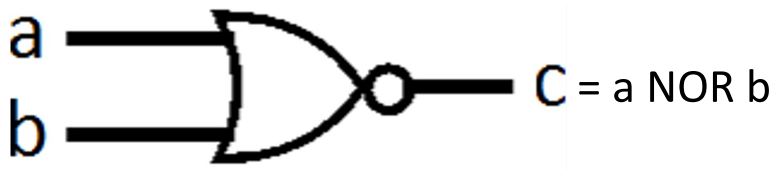
NAND



True if at least one input is false

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

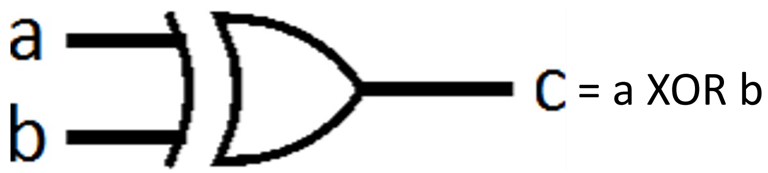
NOR



True if both inputs are false

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

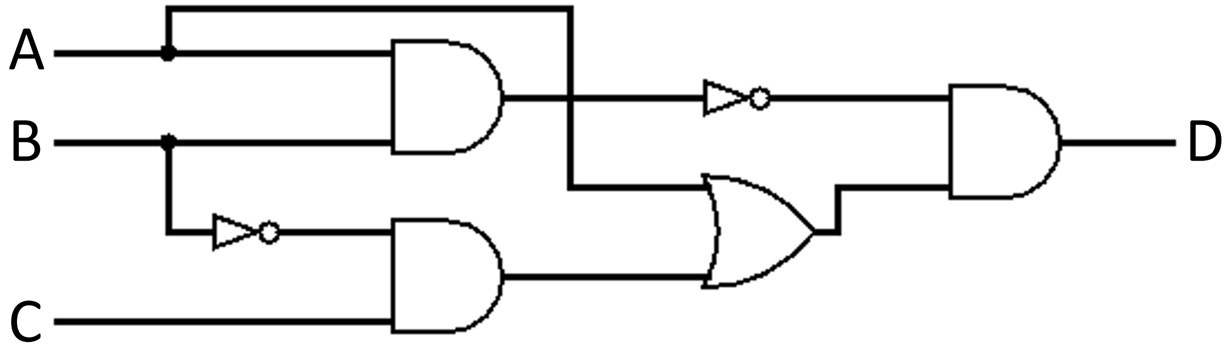
XOR



True if exactly one input is true
(or if odd number of inputs are true for > 2 inputs)

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Combining Multiple Logic Gates



D = (NOT(**A** AND **B**)) AND (**A** OR (NOT **B** AND **C**))

How to Represent Combinational Logic?

✓ Text Description

✓ Circuit Diagram
  – Transistors and wires
  – Logic Gates

✓ Truth Table

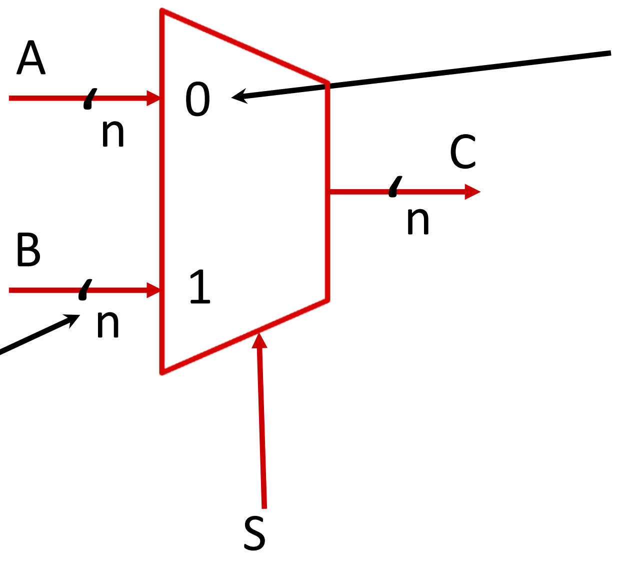
✓ Boolean Expression

✓ *All are equivalent*

Useful Combinational Circuits

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

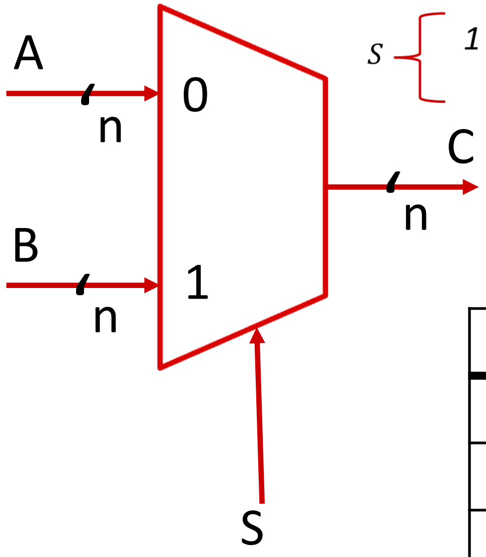


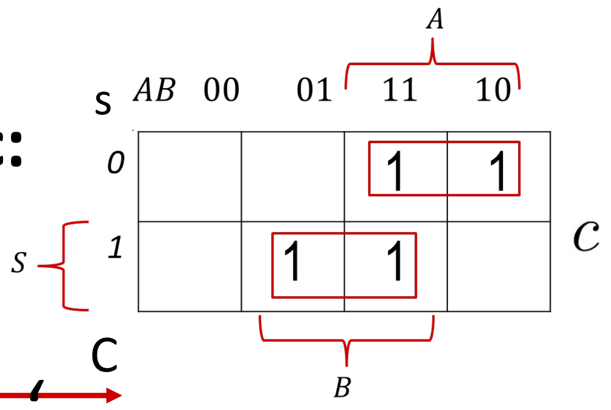
This input is passed to output if selector bits match shown value

Represents that this input has n bits

Implementing a 1-bit 2-to-1 MUX

- **Schematic:**





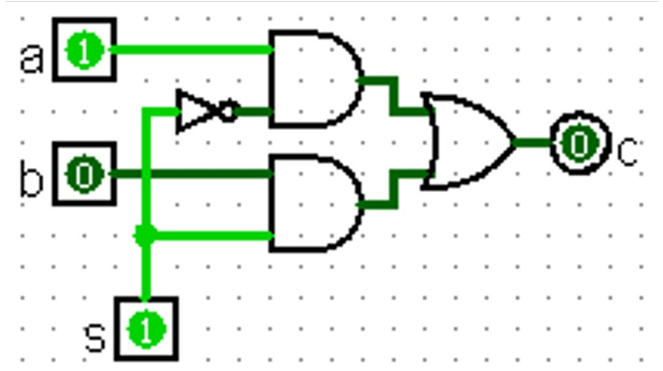
- **Boolean Algebra:**

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

- **Truth Table:**

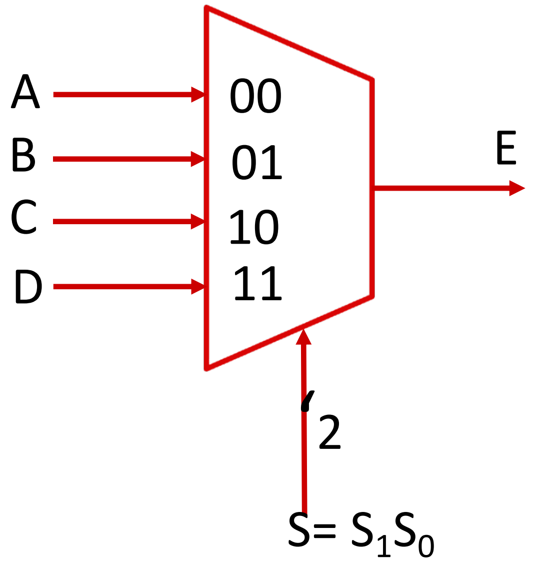
s	a	b	c
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- **Circuit Diagram:**



1-bit 4-to-1 MUX

- **Schematic:**

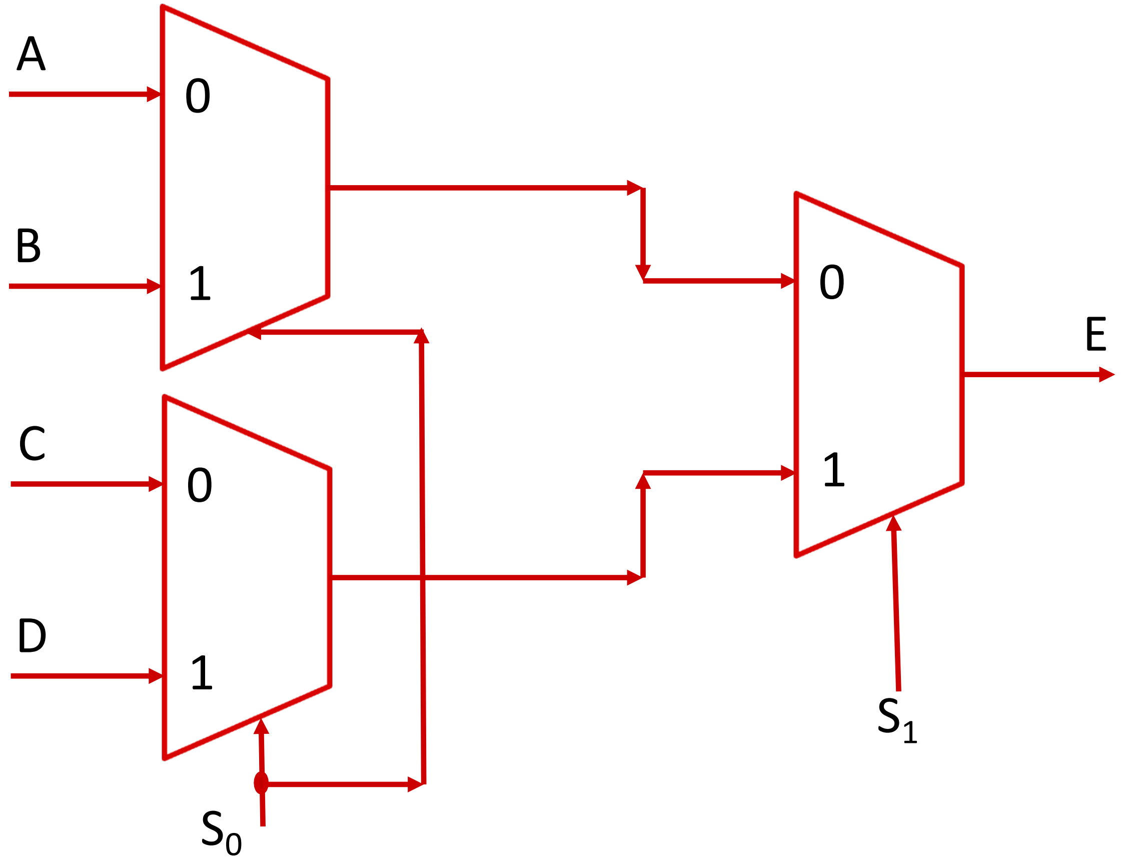


- **Truth Table:** How many rows? 2^6
- **Boolean Expression:**

$$E = \overline{S_1S_0}A + \overline{S_1}S_0B + S_1\overline{S_0}C + S_1S_0D$$

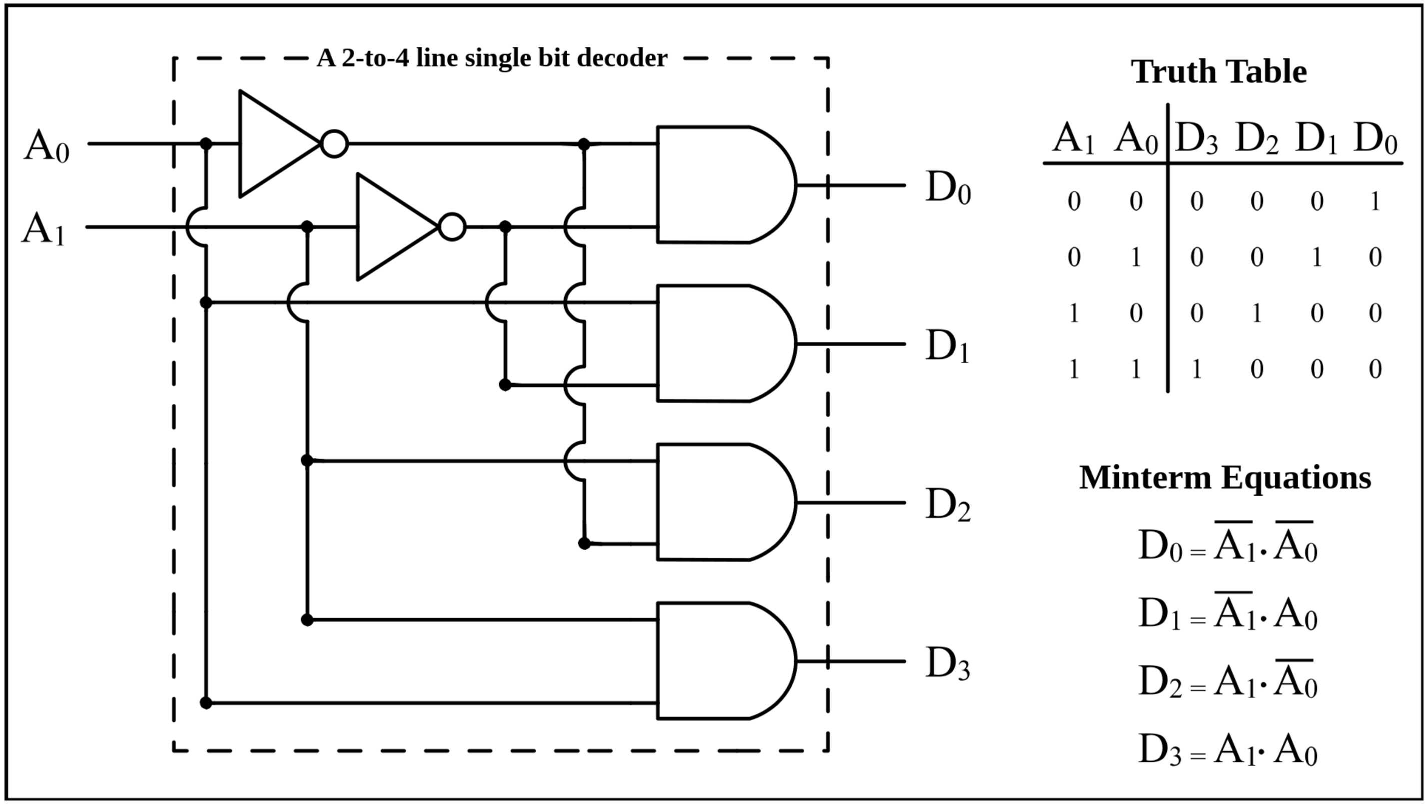
Another Design for 4-to-1 MUX

- Can we leverage what we've previously built?
  - Alternative hierarchical approach:



Decoder

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



Truth Table

A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Minterm Equations

$$D_0 = \overline{A_1} \cdot \overline{A_0}$$

$$D_1 = \overline{A_1} \cdot A_0$$

$$D_2 = A_1 \cdot \overline{A_0}$$

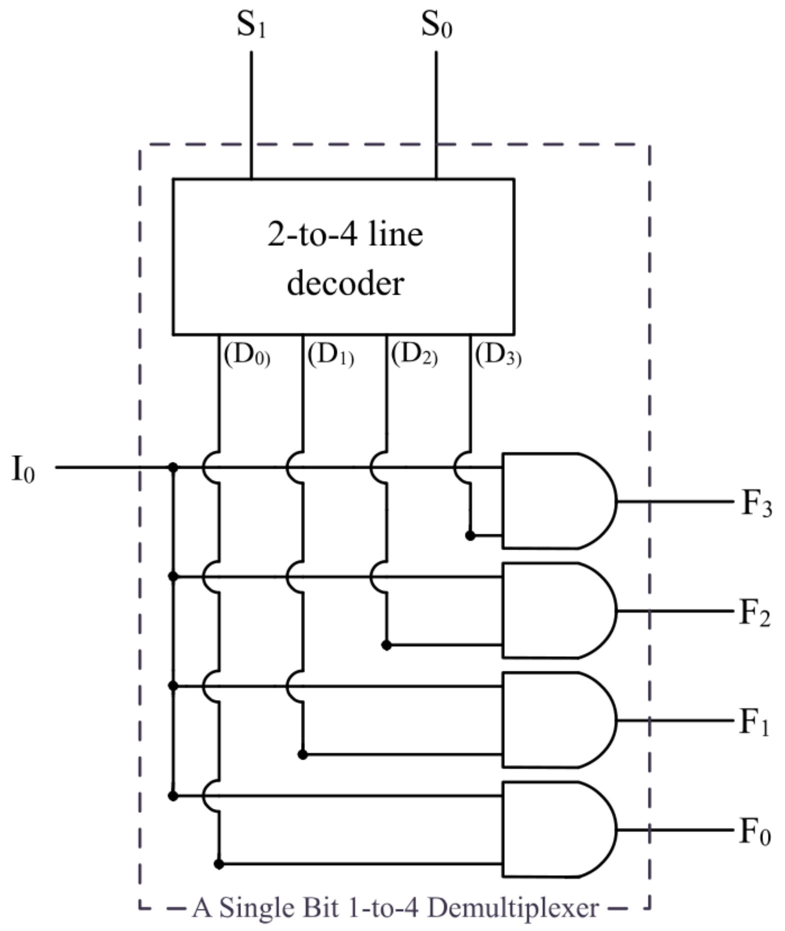
$$D_3 = A_1 \cdot A_0$$

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

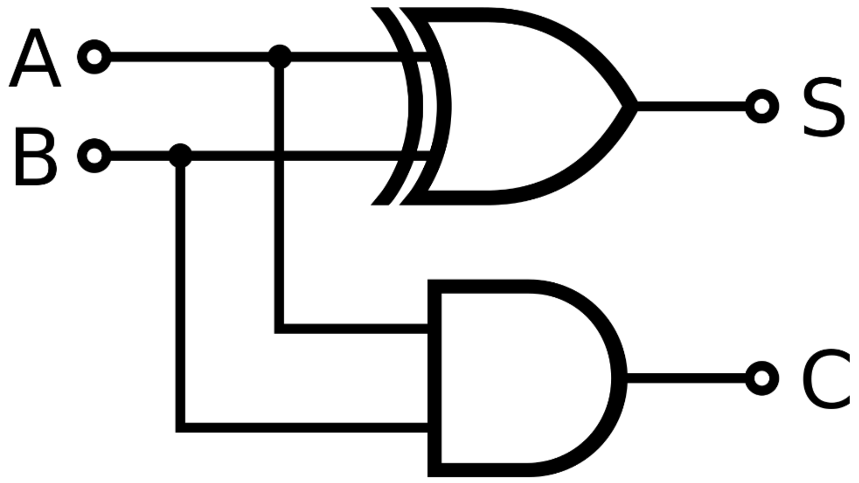
S_1	S_0	I_0	F_3	F_2	F_1	F_0
0	0	0	0	0	0	0
		1	0	0	0	1
0	1	0	0	0	0	0
		1	0	0	1	0
1	0	0	0	0	0	0
		1	0	1	0	0
1	1	0	0	0	0	0
		1	1	0	0	0

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

Single-Bit Binary Adder (Half Adder)

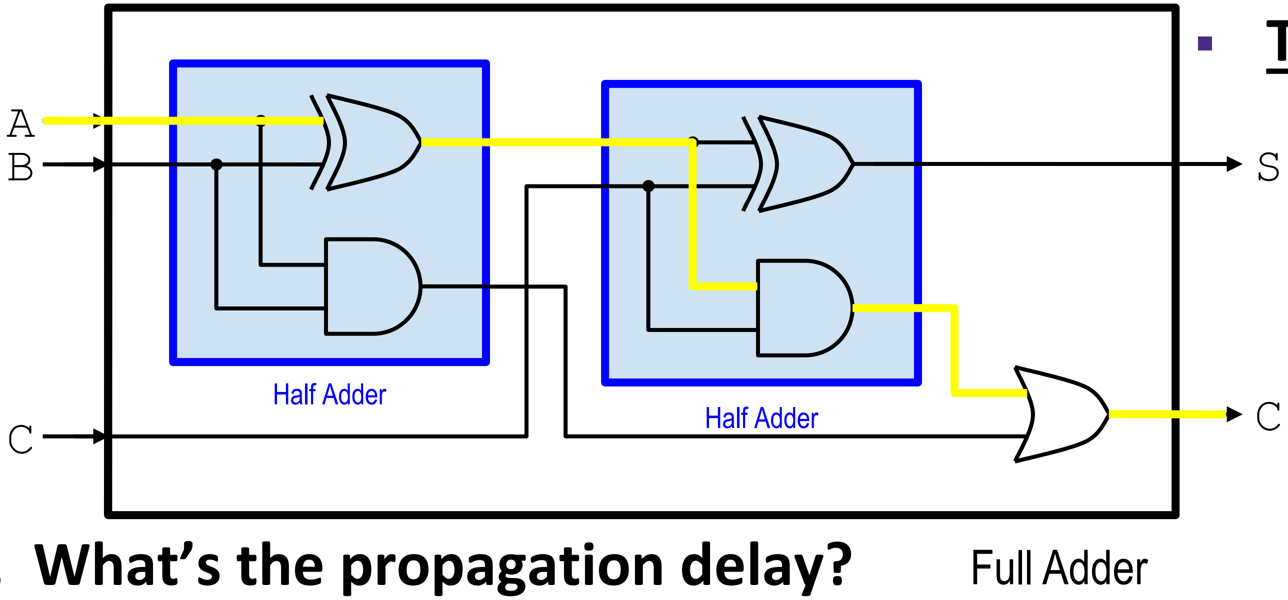
- Add A + B to get Sum (S) and Carry (C)
- Truth Table:
- Boolean Expressions:
  - S = A⊕B; C = AB
- Circuit:

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



By inductiveload - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=1023090

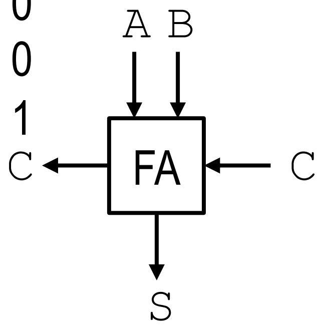
What is this Circuit?



- **Truth table:**

A	B	C	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

3-bit Addition!



Q. What's the propagation delay?

- **3 gate delays (highlighted)**

Q. What does the circuit accomplish?

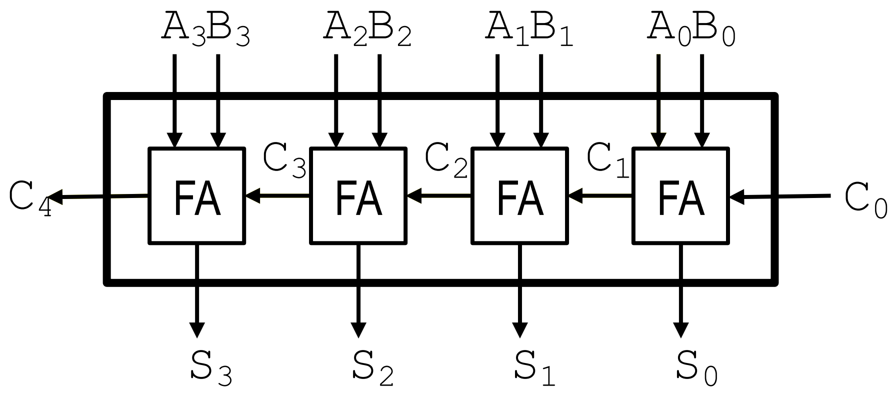
- **Algebra:**

$$S = A \oplus B \oplus C \ ; \quad C = AB + C\,(A \oplus B)$$

Computing with Combinational Circuits

Definition: A combinational circuit computes a pure function, i.e., its outputs react only based on its inputs. There are no feedback loops and no state information (memory) is maintained.

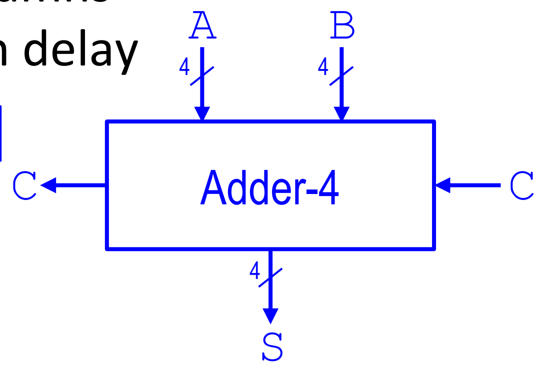
Theorem: Every Boolean function can be implemented with NAND and NOT. Circuits are modular



… a 4-bit ripple carry adder!

- Adds by columns
- Propagation delay

= 9 **(2*n* + 1)**

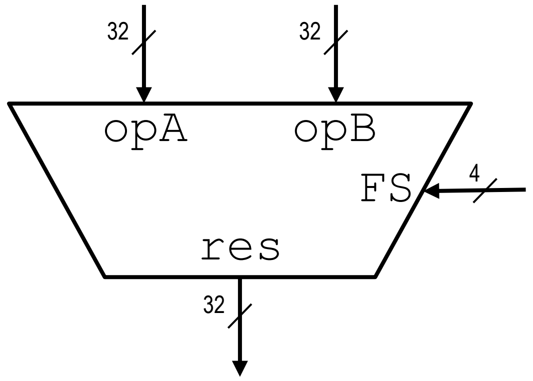


Functional Unit

Hardware circuits are fixed

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

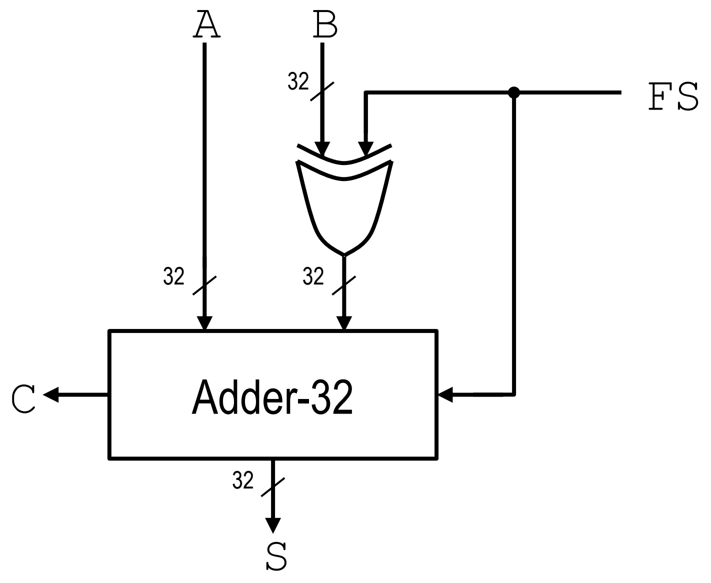
Function Unit:



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 + $\overline{B}$ + 1
  = A − B

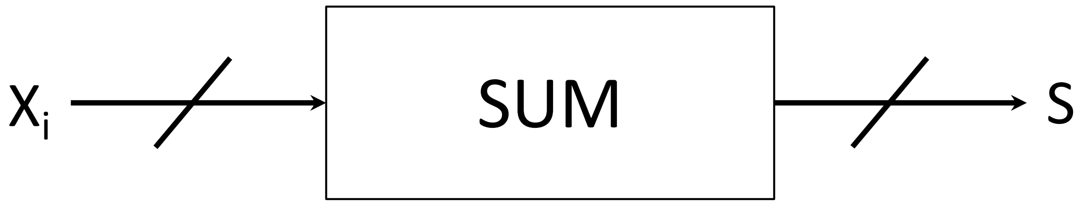
Combinational vs. Sequential Logic

- *Digital Systems* consist of two basic types of circuits:
  - Combinational Logic (CL)
    - Output is a function of the inputs only, not the history of its execution
    - Example: add A, B (ALUs)
  - Sequential Logic (SL)
    - Circuits that "remember" or store information
    - Also called "State Elements"
    - Example: Memory and registers

Sequential Logic

Accumulator Example

An example of why we would need sequential logic

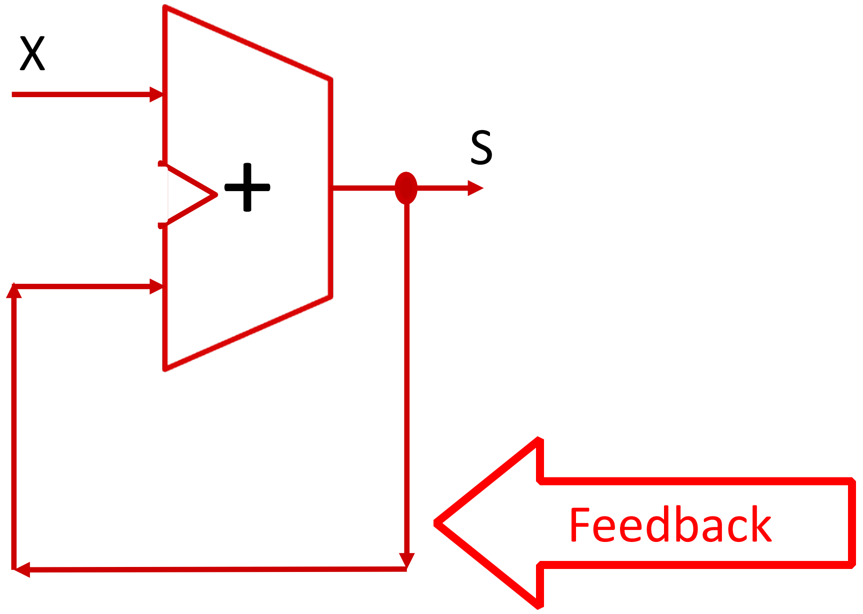


Want:
```
S=0;
for X1,X2,X3 over time...
        S = S + Xi
```

Assume:

- Each X value is applied in succession, one per cycle
- The sum since time 1 (cycle) is present on S

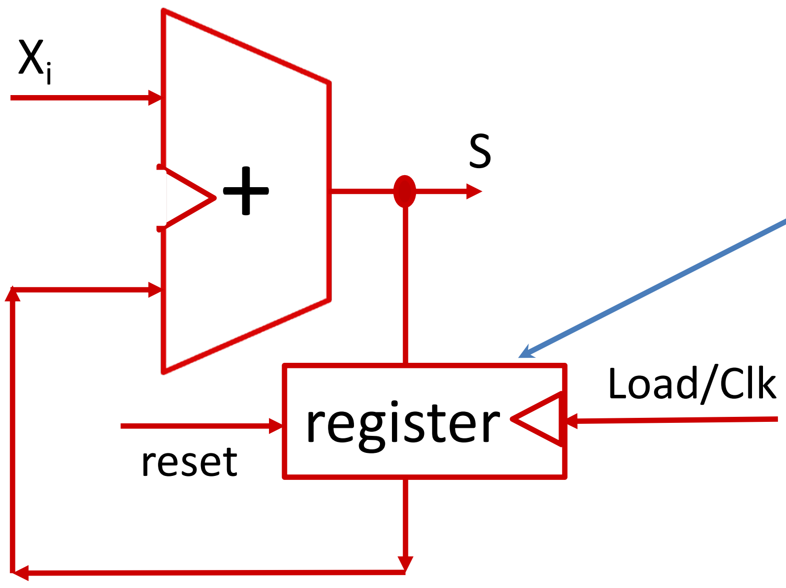
First Try: Does this work?



No!

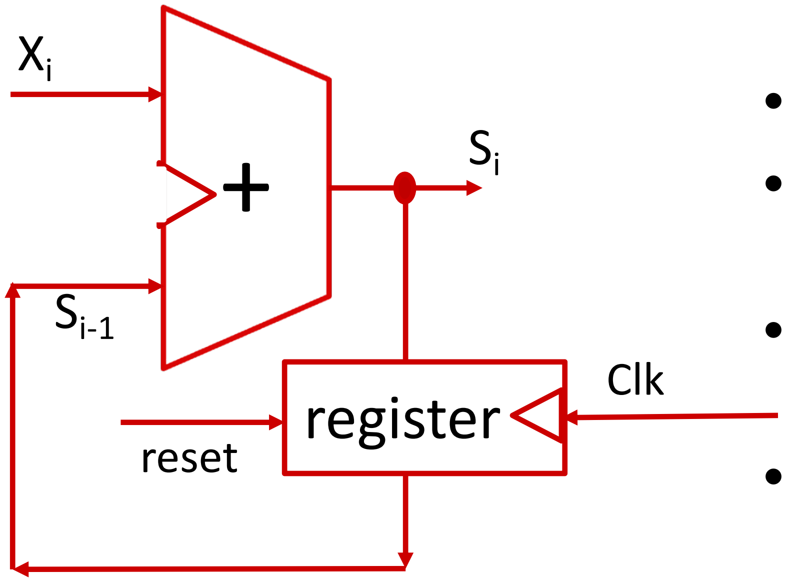
1) How to control the next iteration of the ‘for’ loop?
2) How do we say: ‘S=0’?

Second Try: How About This?

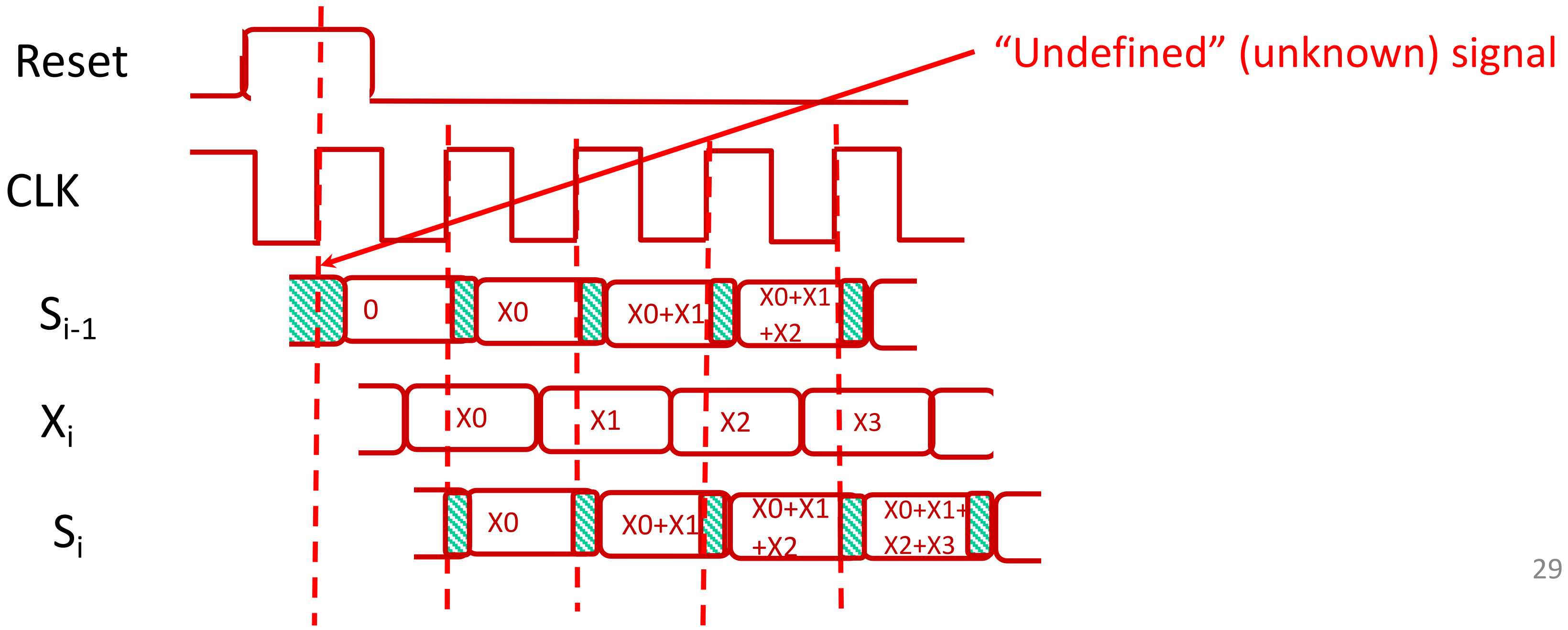


A *Register* is the state element that is used here to hold up the transfer of data to the adder

Accumulator Revisited: Proper Timing



- Reset signal shown
- In practice X_i might not arrive to the adder at the same time as S_{i-1}
- S_i temporarily is wrong, but register always captures correct value
- In good circuits, instability never happens around rising edge of CLK



Uses for State Elements

- Place to store values for some amount of time:
  - Register files (like in RISCV)
  - Memory (caches and main memory)
- *Help control flow of information between combinational logic blocks*
  - State elements are used to hold up the movement of information at the inputs to combinational logic blocks and allow for orderly passage

CPU Hardware

Goal: Given an instruction set architecture, construct a machine that reliably executes instructions.

Design choices will influence speed of instructions:

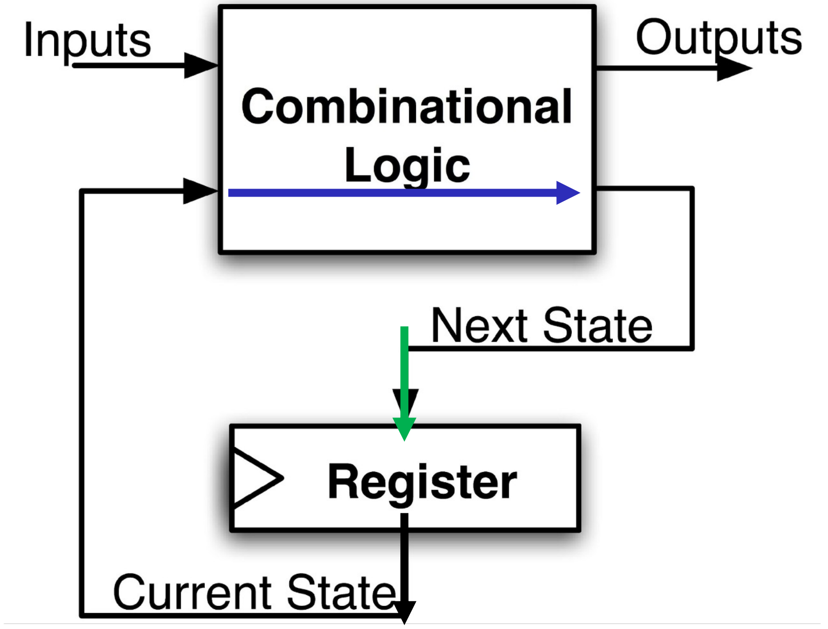
- **some instructions will be faster than others**
- **order of instructions may matter**
- **order of memory accesses may matter**

"conflicts" or "hazards"

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



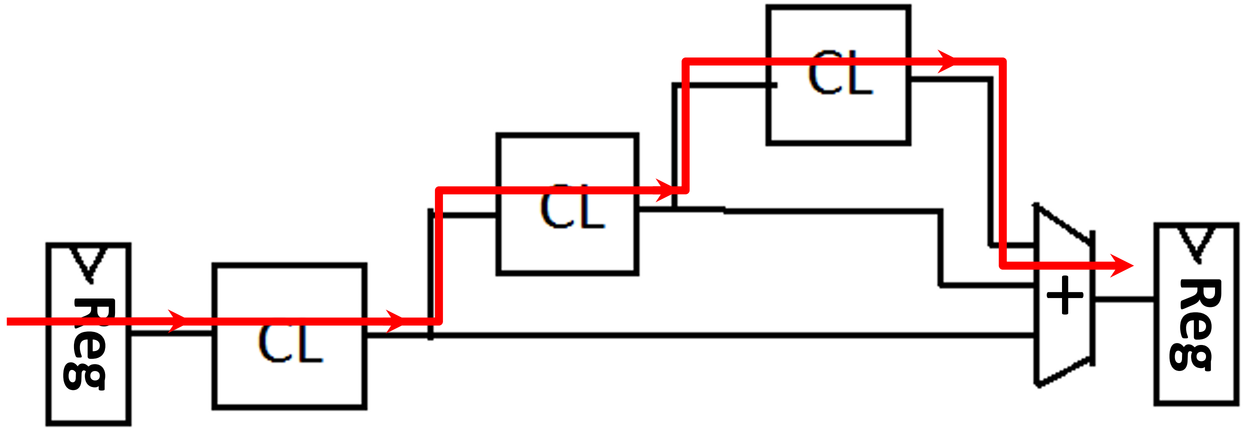
Max Delay = CLK-to-Q Delay
+ CL Delay
+ Setup Time

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



How do we go faster?

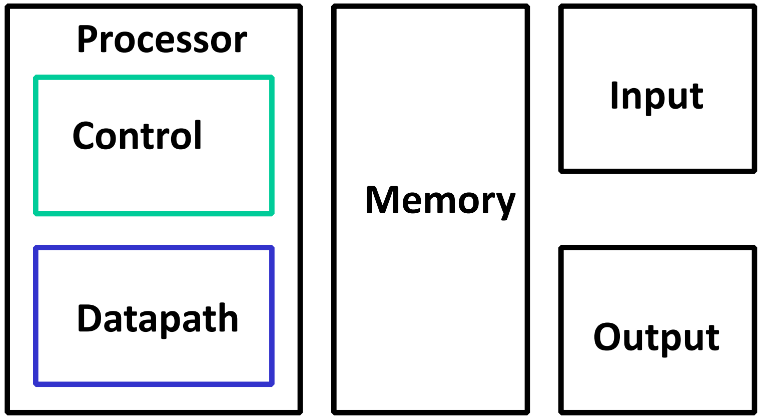
Pipelining!

- Split operation into smaller parts and add a register between each one.

RISC-V *CPU Datapath, Control Intro*

Design Principles

- Five steps to design a processor:
  1) Analyze instruction set → datapath requirements
  2) Select set of datapath components & establish clock methodology
  3) Assemble datapath meeting the requirements
  4) Analyze implementation of each instruction to determine setting of control points that effects the register transfer
  5) Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits



Summary !

- Universal datapath
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
- 5 Phases of execution
  - IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory), WB (Write Back)
  - Not all instructions are active in all phases (except for loads!)
- Controller specifies how to execute instructions
  - Worth thinking about: what new instructions can be added with just most control?

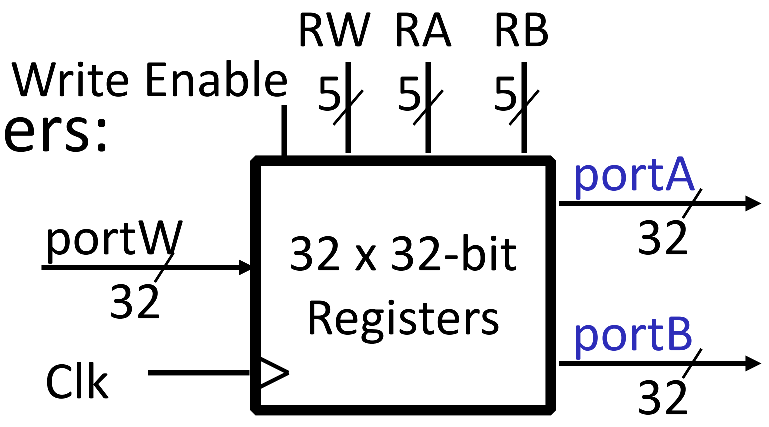
Your CPU in two parts

- ***Central Processing Unit (CPU):***
  - *Datapath:* contains the hardware necessary to <u>perform</u> operations required by the processor
    - Reacts to what the controller tells it! (ie. "I was told to do an add, so I"ll feed these arguments through an adder)
  - *Control:* <u>decides</u> what each piece of the datapath should do
    - What operation am I performing? Do I need to get info from memory? Should I write to a register? Which register?
    - Has to make decisions based on the input instruction only!

Design Principles

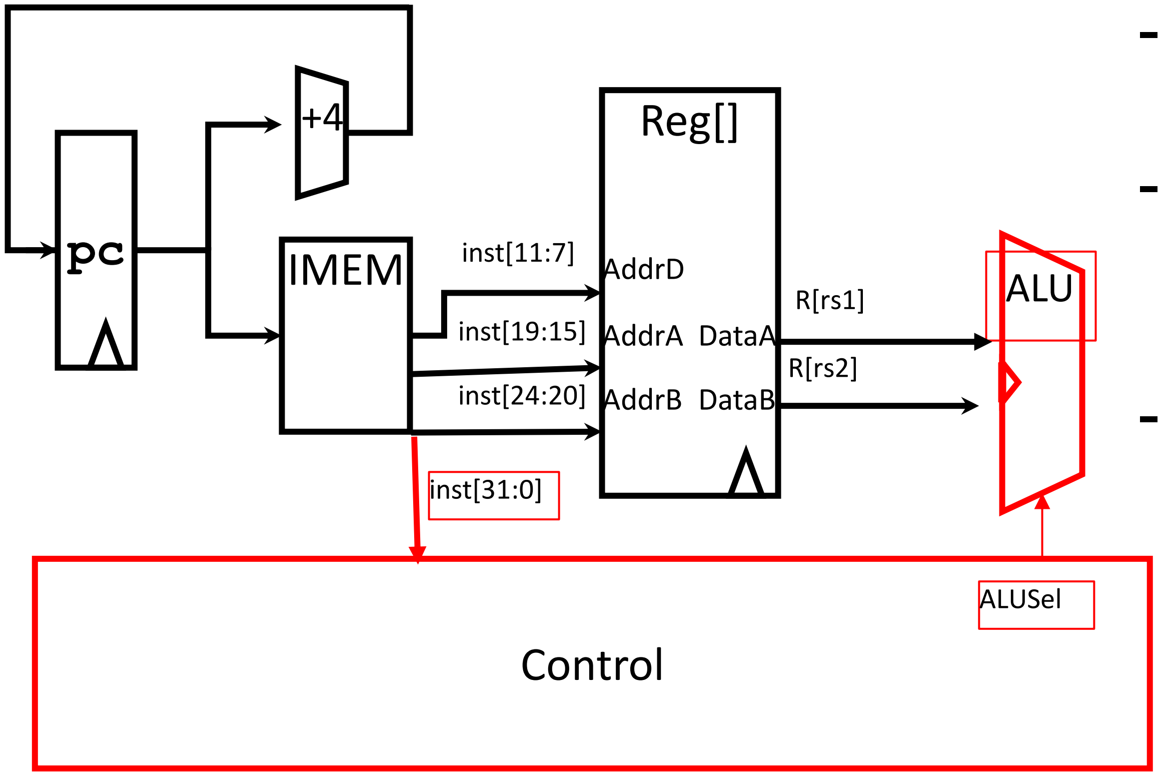
- Determining control signals
  - Any time a datapath element has an input that changes behavior, it requires a control signal (e.g. ALU operation, read/write)
  - Any time you need to pass a different input based on the instruction, add a **MUX** with a control signal as the selector (e.g. next PC, ALU input, register to write to)
- Your control signals will change based on your exact datapath
- Your datapath will change based on your ISA

Storage Element: Register File



- *Register File* consists of 31 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

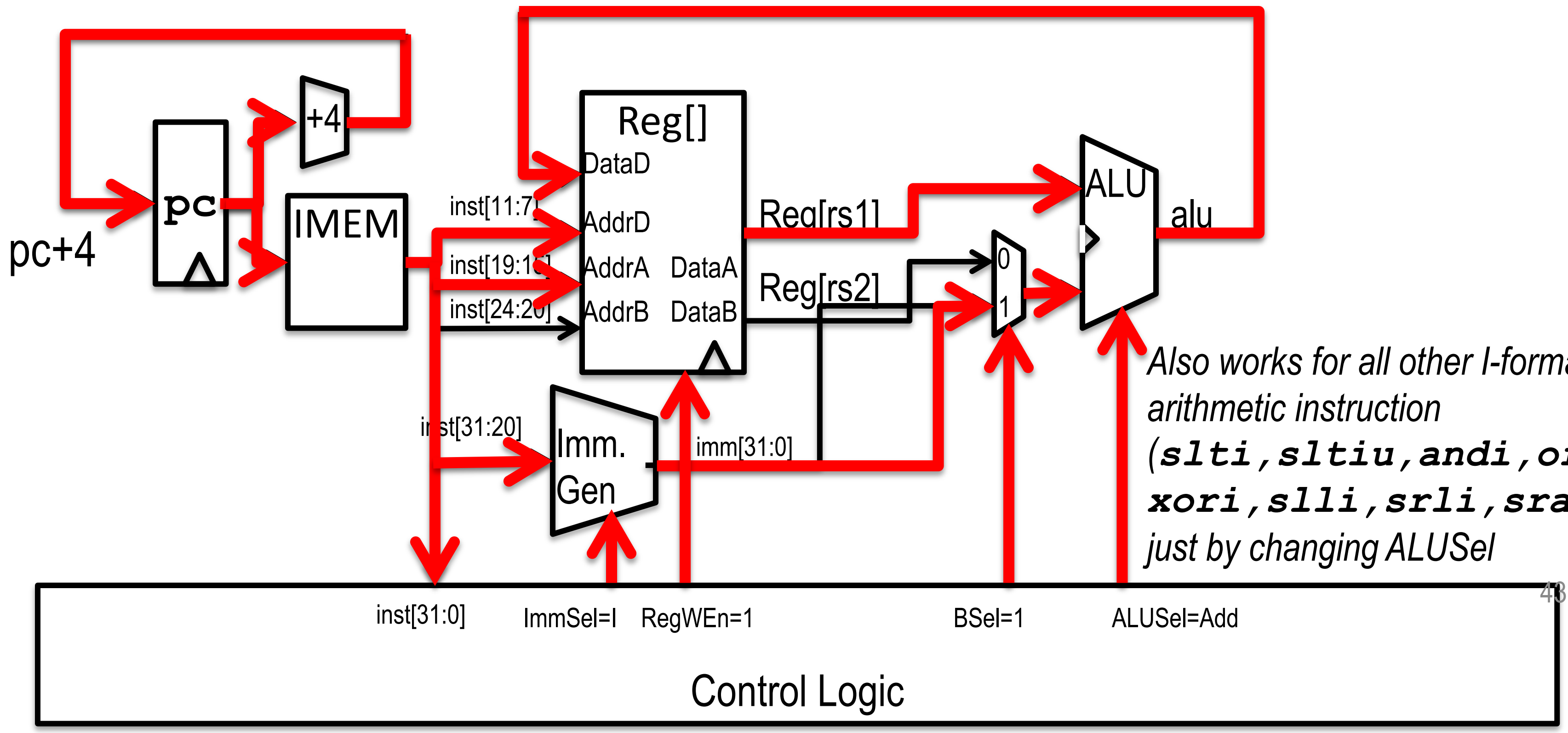


(4) Perform operation

- 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)

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

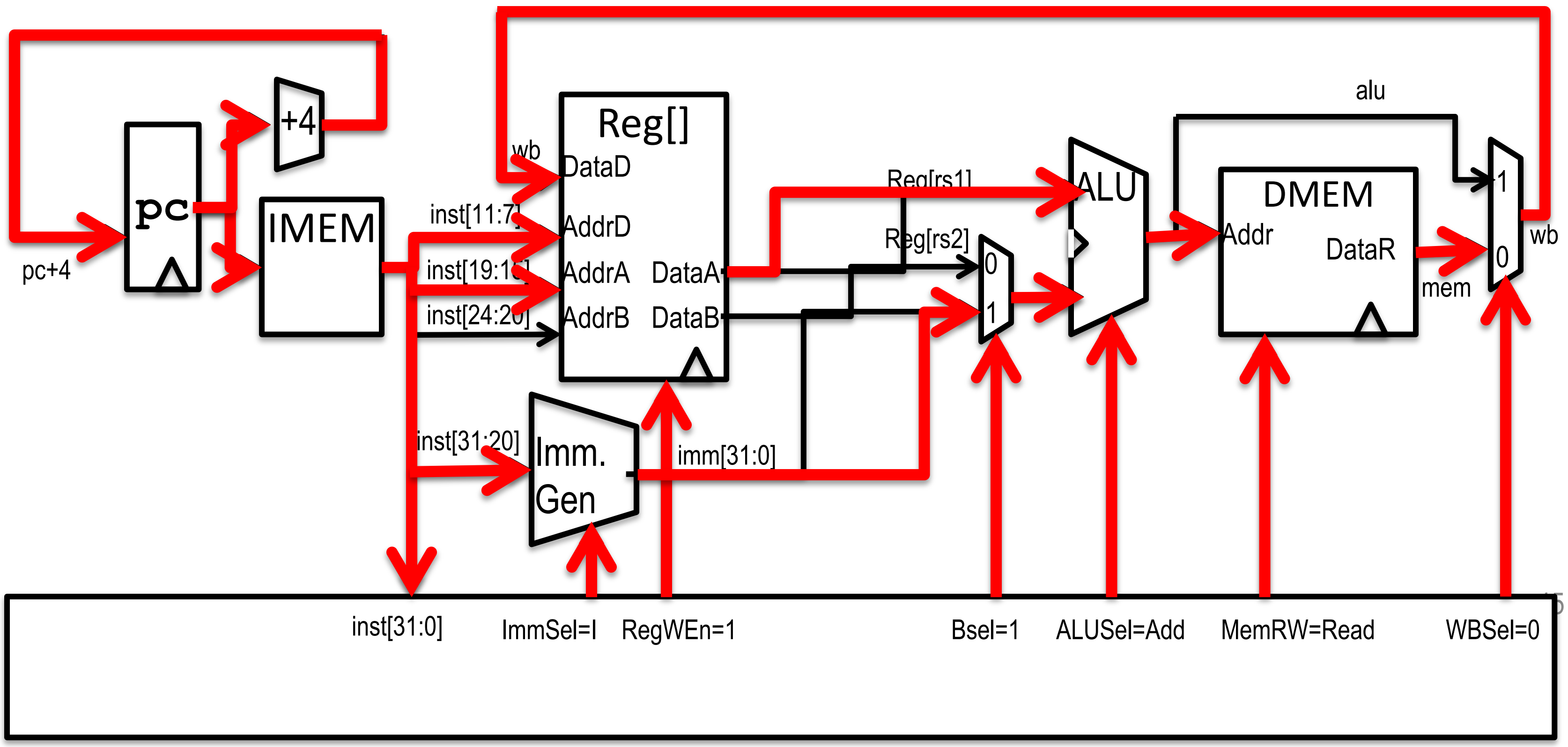
Adding `addi` to datapath



*Also works for all other I-format arithmetic instruction (**`slti,sltiu,andi,ori,xori,slli,srli,srai`**) just by changing ALUSel*

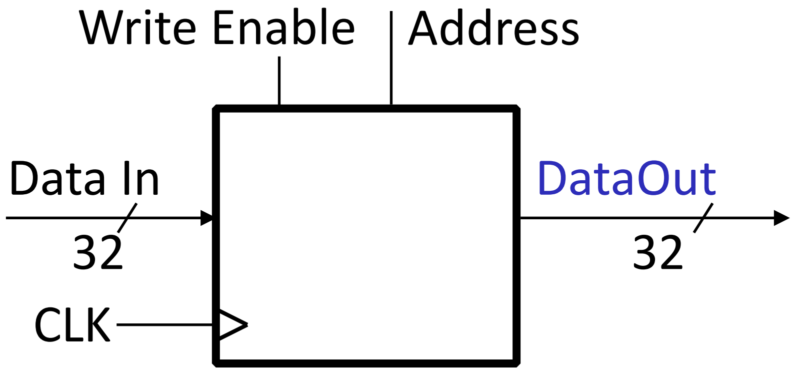
Inst[31:0]	PCSel	ImmSel	RegWEn	Br Un	Br Eq	Br LT	BSel	ASel	ALUSel	MemRW	WBSel
add	+4	*	1 (Y)	*	*	*	Reg	Reg	Add	Read	ALU
sub	+4	*	1	*	*	*	Reg	Reg	Sub	Read	ALU
(R-R Op)	+4	*	1	*	*	*	Reg	Reg	*(Op)*	Read	ALU
addi	+4	I	1	*	*	*	Imm	Reg	Add	Read	ALU

Adding `lw` to datapath



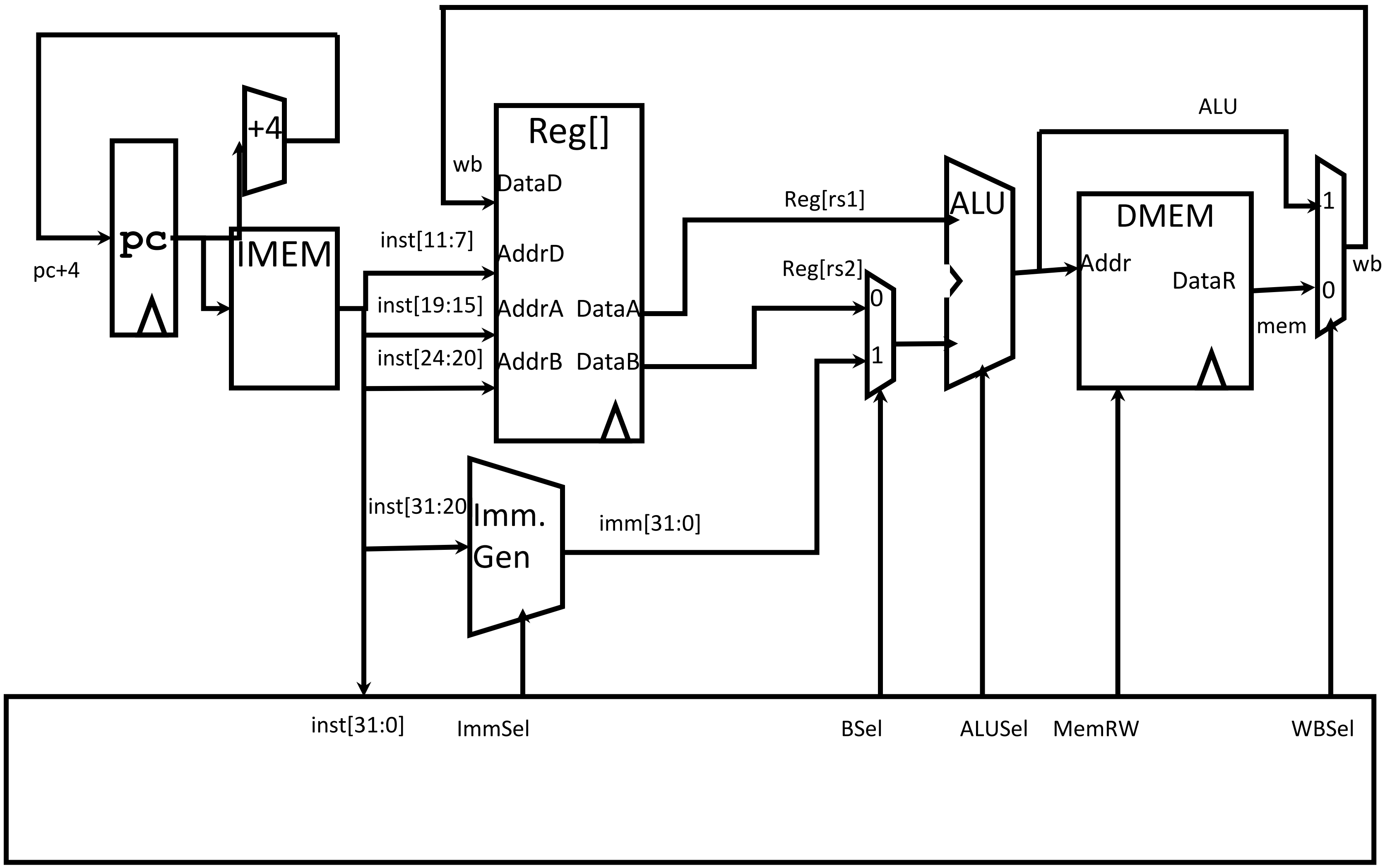
Inst[31:0]	PCSel	ImmSel	RegWEn	Br Un	Br Eq	Br LT	BSel	ASel	ALUSel	MemRW	WBSel
`add`	+4	*	1 (Y)	*	*	*	Reg	Reg	Add	Read	ALU
`sub`	+4	*	1	*	*	*	Reg	Reg	Sub	Read	ALU
`(R-R Op)`	+4	*	1	*	*	*	Reg	Reg	*(Op)*	Read	ALU
`addi`	+4	I	1	*	*	*	Imm	Reg	Add	Read	ALU
`lw`	+4	I	1	*	*	*	Imm	Reg	Add	Read	Mem

Storage Element: Idealized Memory

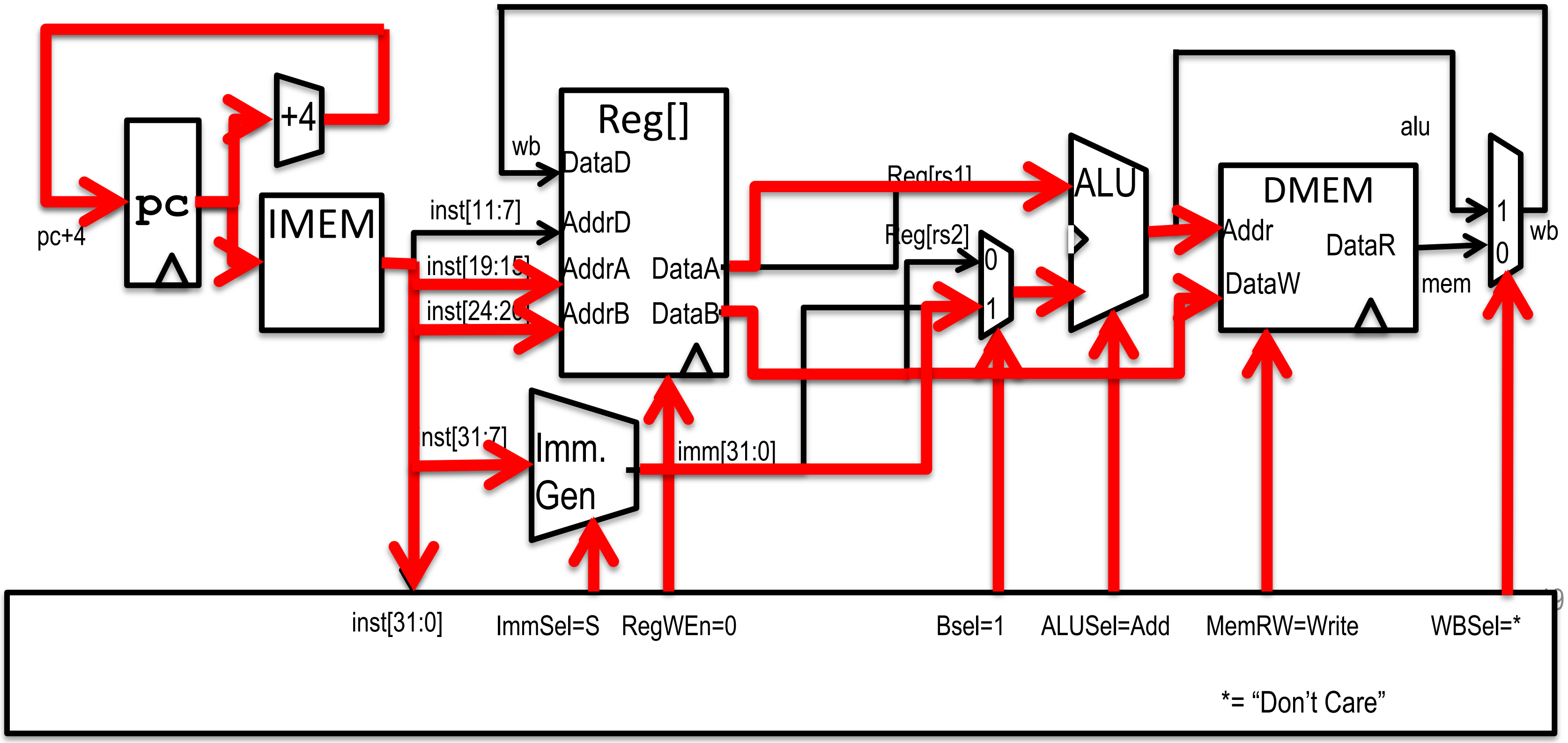


- Memory (idealized)
  - One input port: Data In
  - One output port: Data Out
- Memory access:
  - Read: Write Enable = 0, data at Address is placed on Data Out
  - Write: Write Enable = 1, Data In written to Address
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read, behaves as a combinational logic block: Address valid → Data Out valid after "access time"

Current Datapath



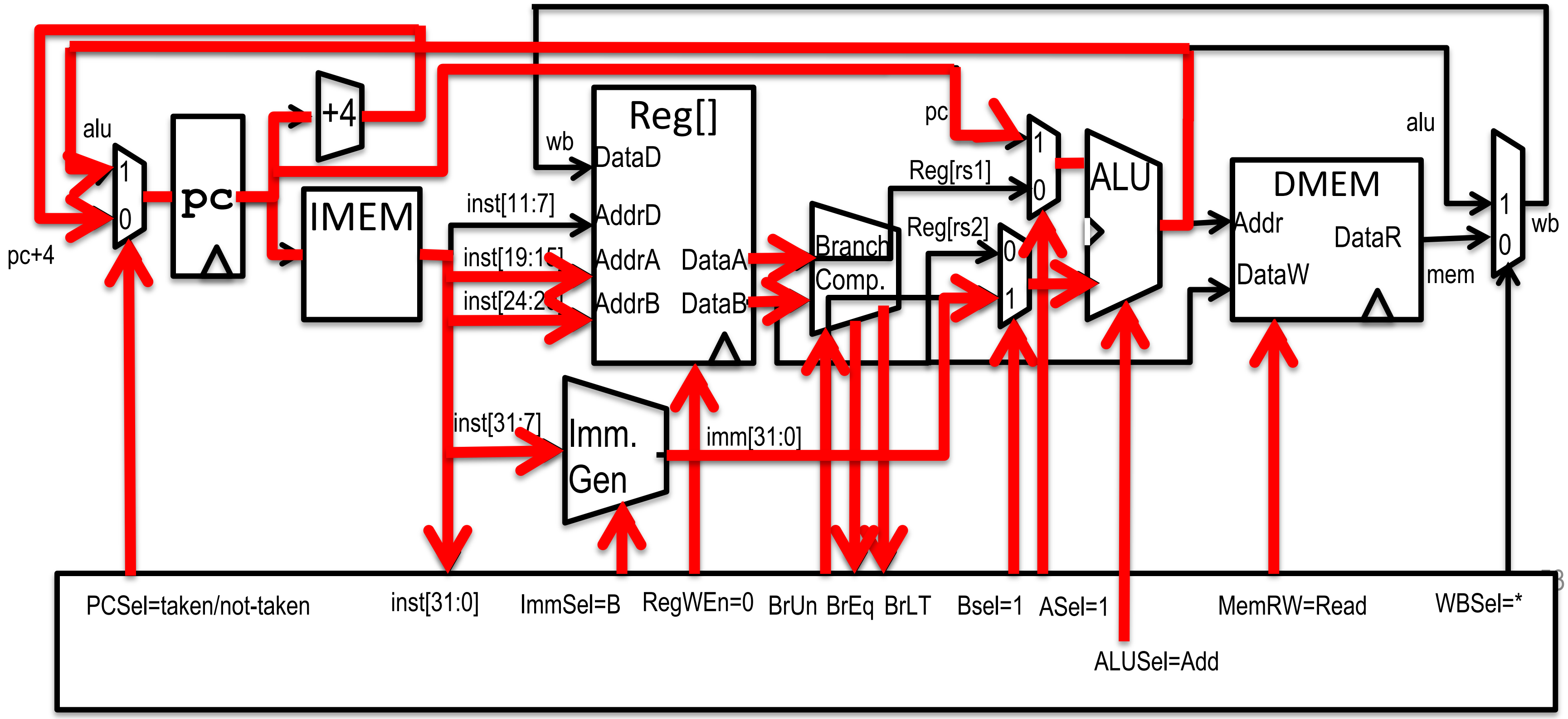
Adding sw to datapath



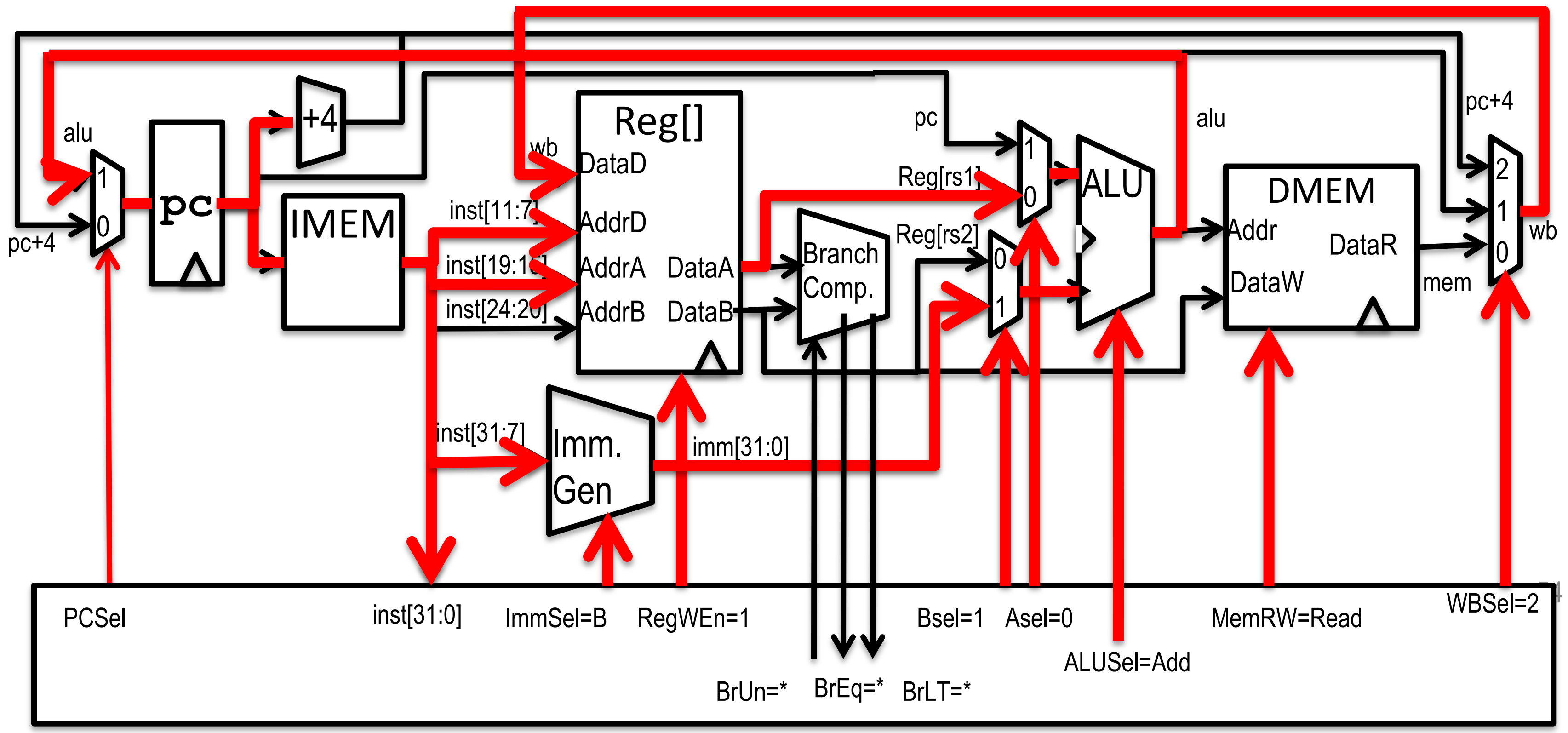
Inst[31:0]	PCSel	ImmSel	RegWEn	Br Un	Br Eq	Br LT	BSel	ASel	ALUSel	MemRW	WBSel
add	+4	*	1 (Y)	*	*	*	Reg	Reg	Add	Read	ALU
sub	+4	*	1	*	*	*	Reg	Reg	Sub	Read	ALU
(R-R Op)	+4	*	1	*	*	*	Reg	Reg	*(Op)*	Read	ALU
addi	+4	I	1	*	*	*	Imm	Reg	Add	Read	ALU
lw	+4	I	1	*	*	*	Imm	Reg	Add	Read	Mem
sw	+4	S	0 (N)	*	*	*	Imm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

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

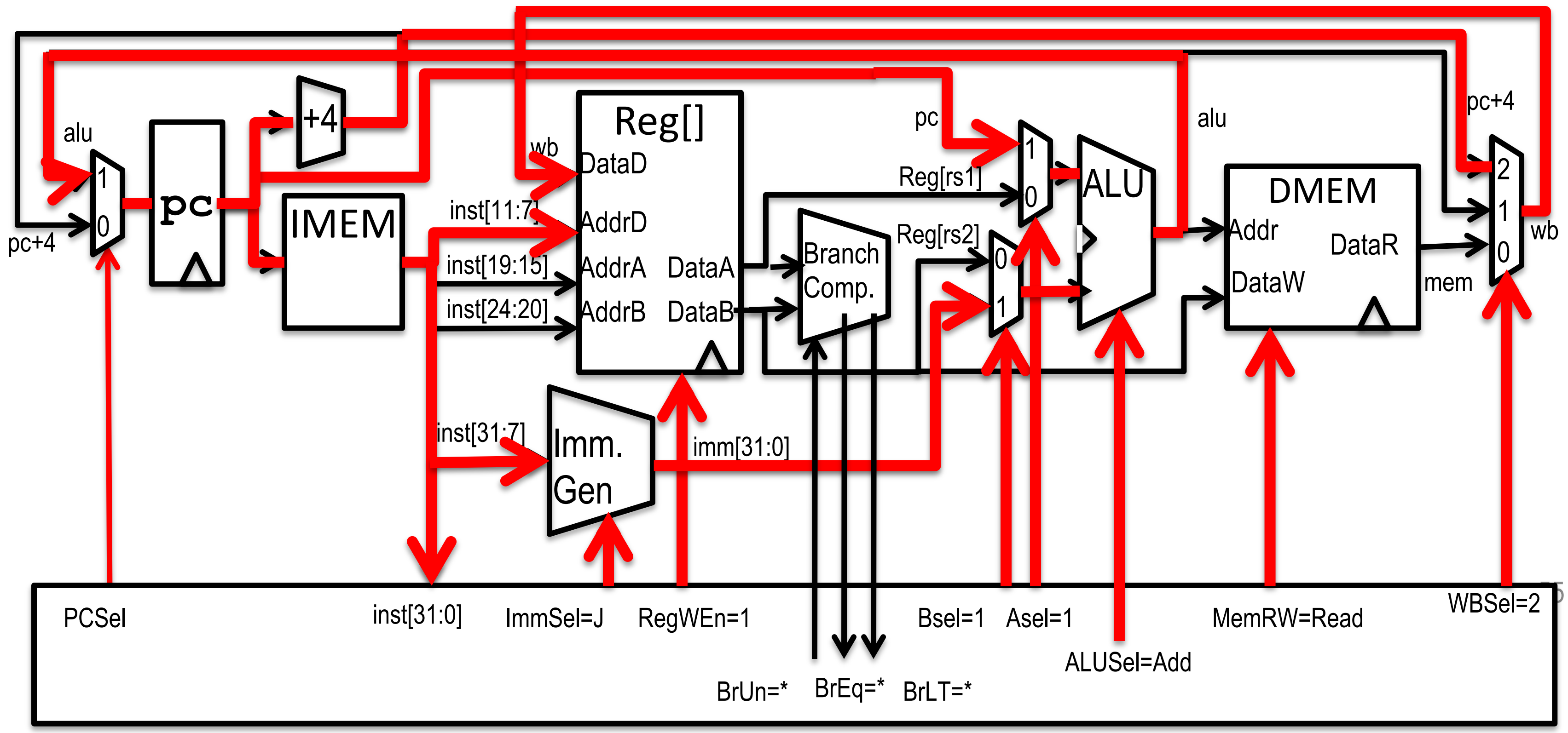
Adding branches to datapath



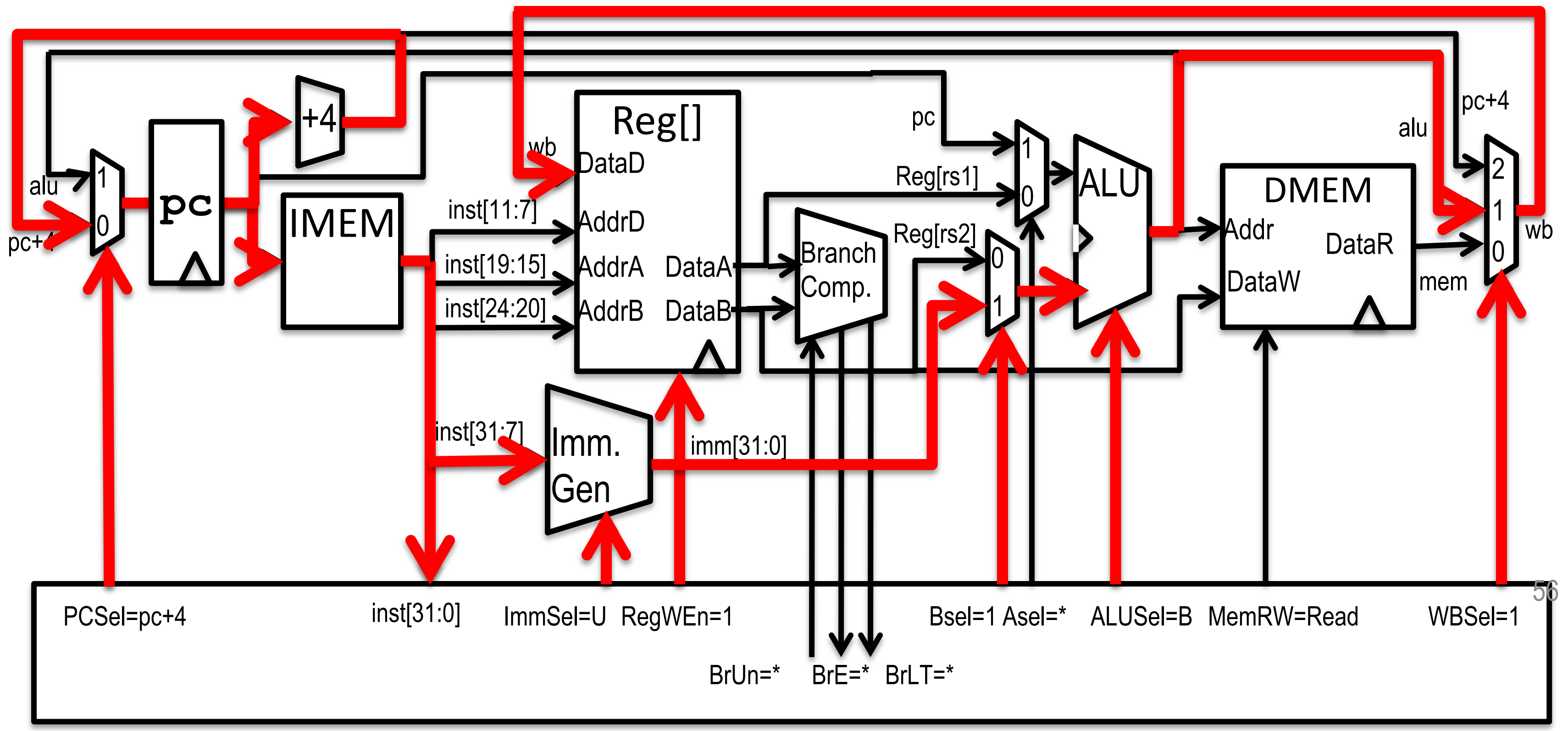
Adding `jalr` to datapath



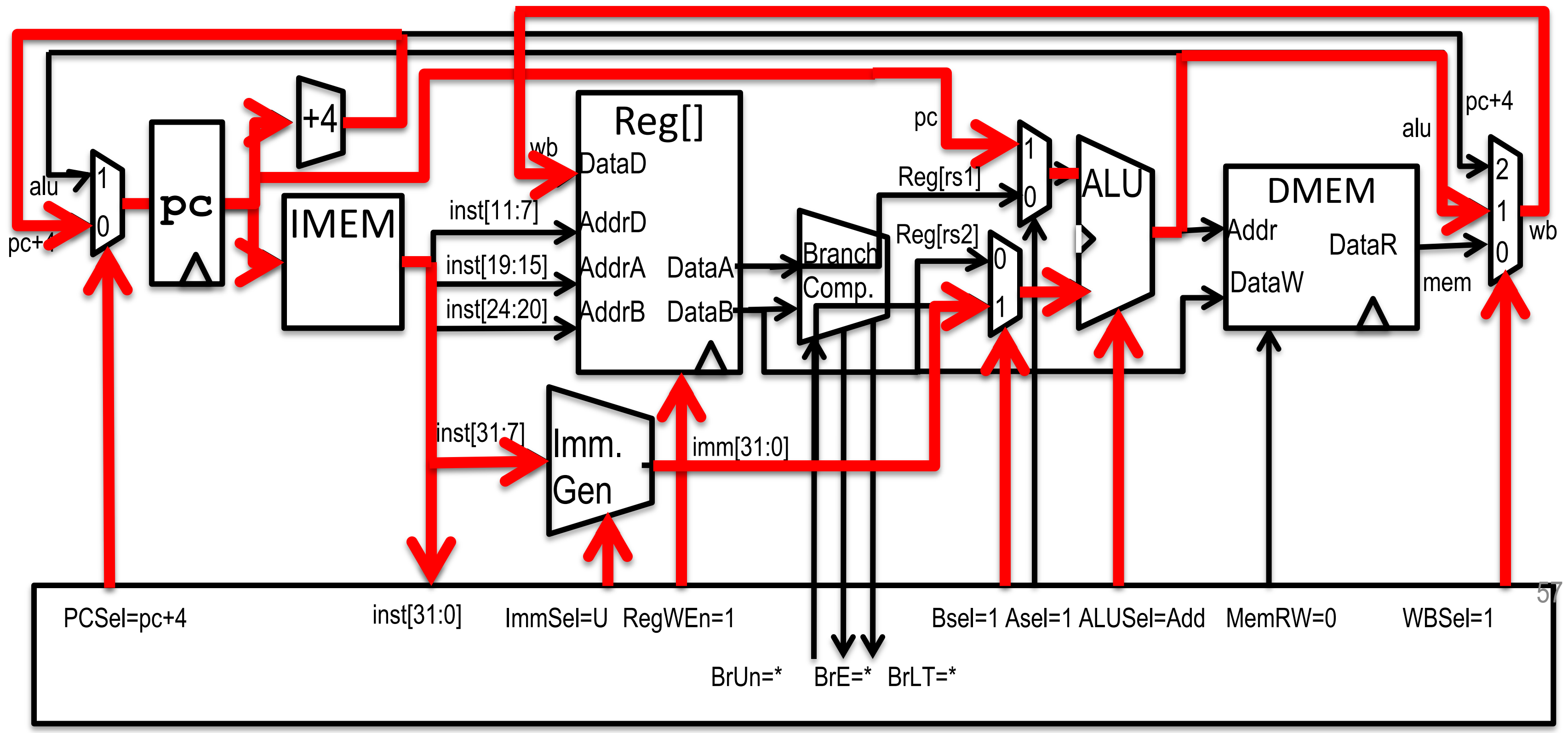
Adding `jal` to datapath



Implementing `lui`



Implementing auipc


alu
pc+4
1
0
pc
+4
IMEM
Reg[]
wb
DataD
inst[11:7]
AddrD
inst[19:15]
AddrA
DataA
inst[24:20]
AddrB
DataB
inst[31:7]
Imm. Gen
imm[31:0]
Branch Comp.
pc
Reg[rs1]
Reg[rs2]
ALU
DMEM
Addr
DataR
DataW
alu
pc+4
mem
2
wb
PCSel=pc+4
inst[31:0]
ImmSel=U
RegWEn=1
BrUn=*
BrE=*
BrLT=*
Bsel=1
Asel=1
ALUSel=Add
MemRW=0
WBSel=1