Design Tradeoffs in Sequential Logic

Lecture Goals

- Finish discussion of pipelining and design tradeoffs in sequential logic

- Study how to generalize an FSM to solve multiple problems
  - First step towards building a general-purpose processor!

Software vs. Hardware Design
Timing is the key difference

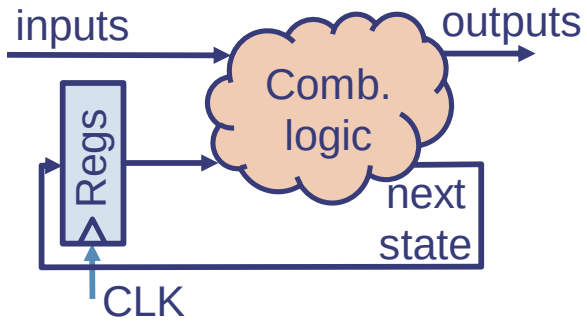
1. Software interfaces (even instructions) are timing-independent
   - Specify *what* should happen, not *when*

```
while (b != 0) {
     a = a * b;
     b = b - 1;
}
```

```
loop: mv a1, s0
      call mul
      addi s0, s0, -1
      beqz s0, loop
```

2. Hardware design is all about timing
   - Specify what happens on every clock cycle…
   - …which itself determines the length of the clock cycle

```
module Factorial;
  Reg#(Word) a(0);
  Reg#(Word) b(0);
  rule step;
    …
```



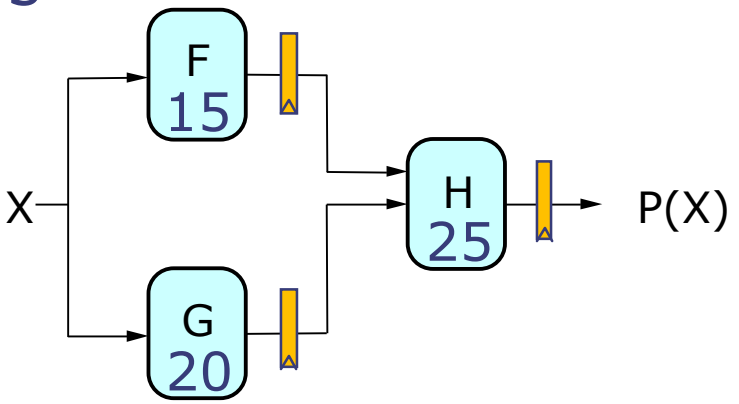
Recap: Benefits of Sequential Logic

- Sequential circuits can implement more computations than combinational circuits
  - Variable amount of input and/or output
  - Variable number of steps

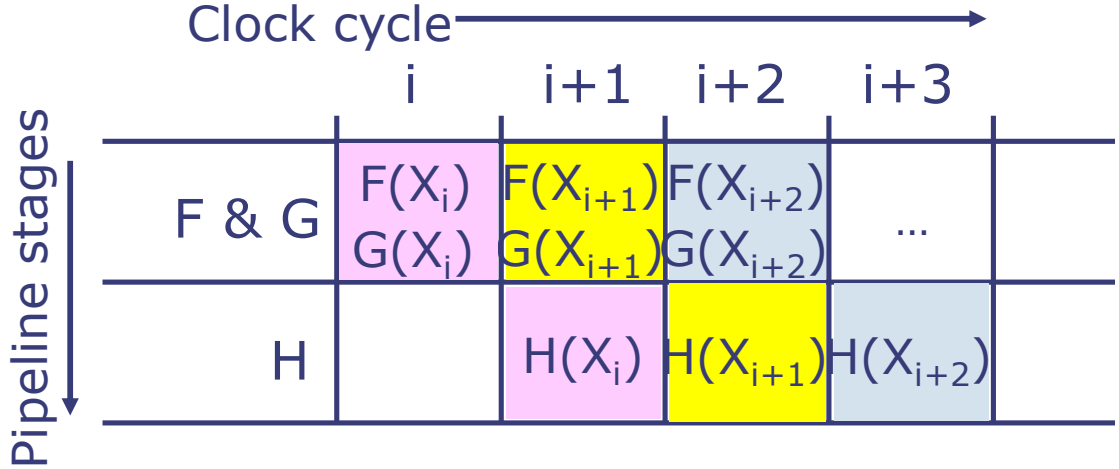
- Even when combinational circuits suffice, sequential circuits allow more design tradeoffs
  - Pipelined circuits *improve throughput* by increasing frequency and overlapping multiple computations
  - Folded circuits *reduce area* by reusing a small amount of combinational logic over multiple cycles

Reminder: Pipelined Circuits

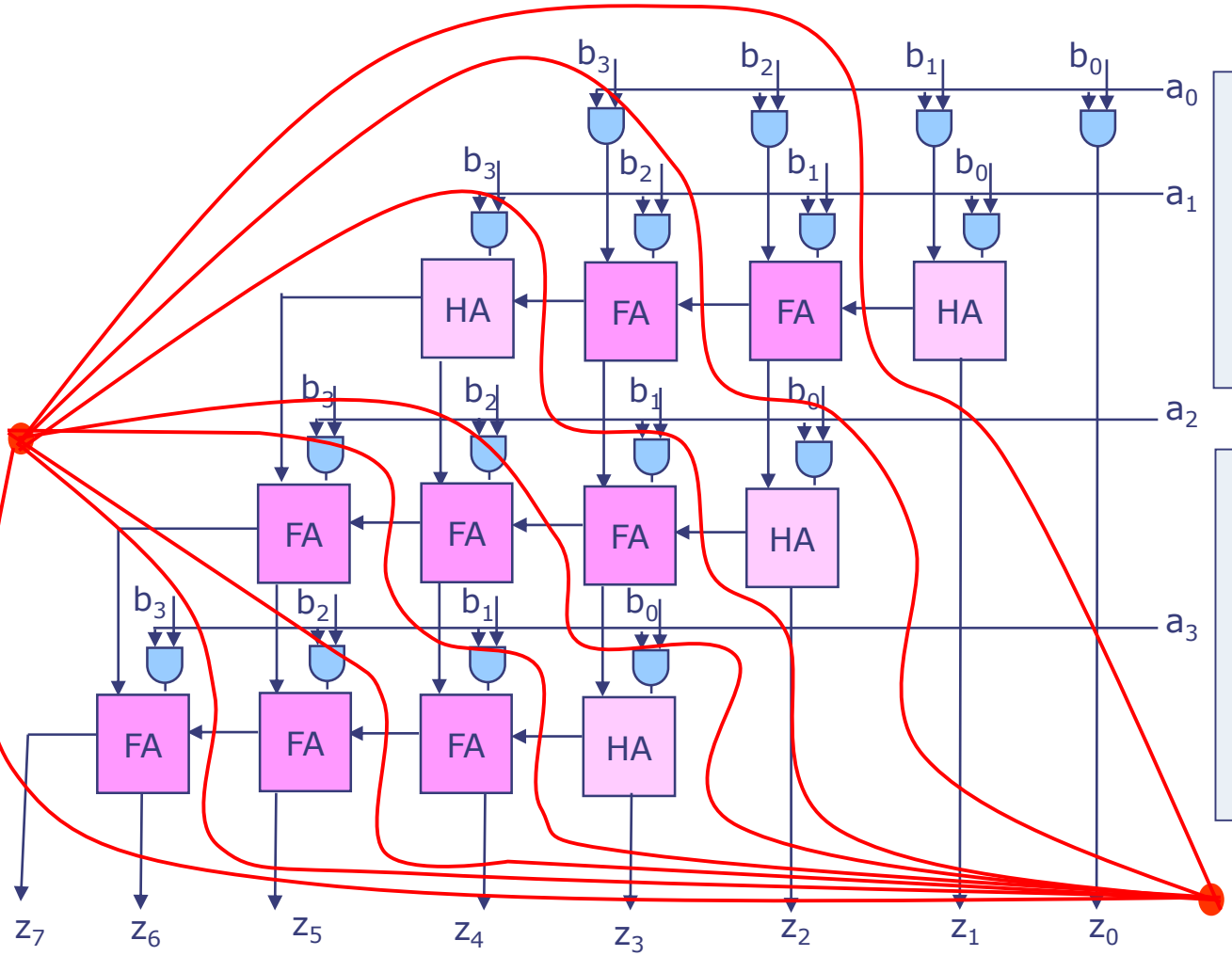
- Pipelining breaks a combinational circuit over multiple stages using registers
  - Each computation takes multiple cycles
  - On each cycle, each stage processes a different value
  - $t_{CLK} \downarrow \rightarrow$ Thoughput $\uparrow$



- Pipeline diagrams



Reminder: Pipelined Multiplier



Combinational

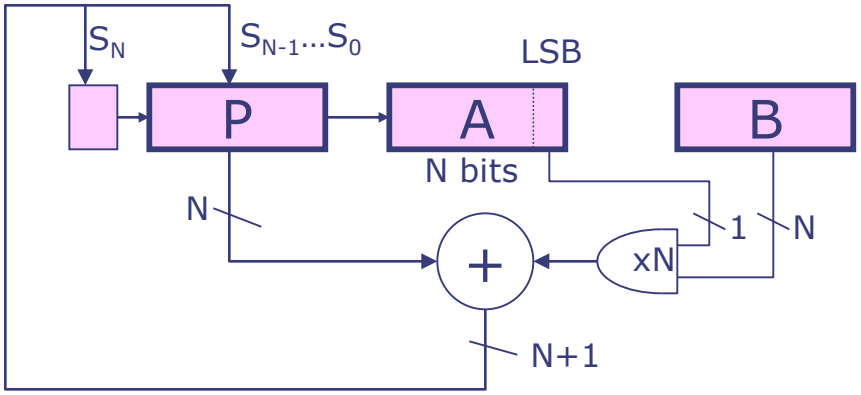
Area = $\Theta(N^2)$
$t_{PD} = \Theta(N)$
Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$

Pipelined

Area = $\Theta(N^2)$
$t_{CLK} = \Theta(1)$
#stages = $\Theta(N)$
Latency = $\Theta(N)$
Throughput = $\Theta(1)$

Reminder: Folded Circuits

- Combinational circuits often have repetitive logic
  - Example: N-bit multiplier has N-1 adders
- Folded circuits use less combinational logic, reuse it over multiple cycles
  - Example: Implement multiplication with one adder, taking ~N cycles to perform the additions

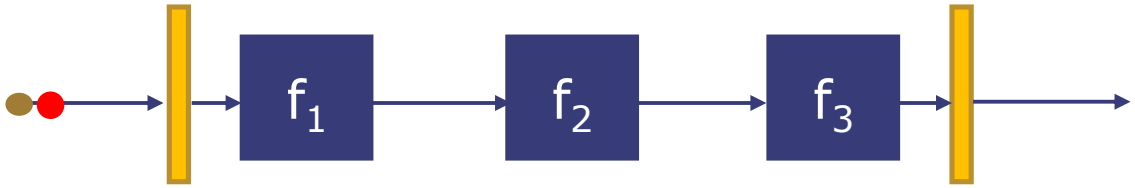



```
Init: P←0, load A&B

Repeat N times {
   P ← P + (A_LSB==1 ? B : 0)
   shift S_N,P,A right one bit
}

Done: 2N-bit result in P,A
```

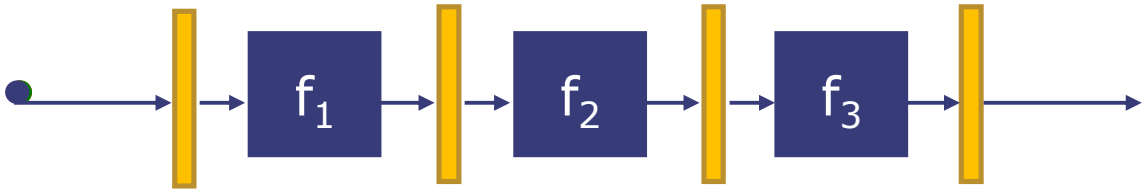
Tradeoff: reduced area, but lower throughput

Summary: Design Alternatives

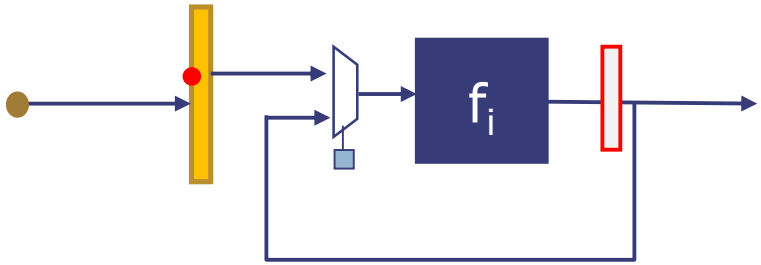
Several combinational blocks in one pipeline stage (A)



One block per pipeline stage (B)



Folded: Reuse a single block, multicycle (C)



Clock: B ≈ C < A

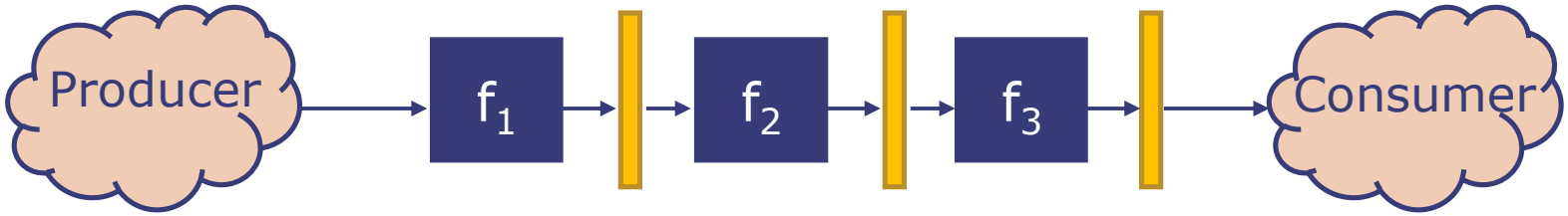
Area: C < A < B

Throughput: C < A < B

Clock Frequency Constraints

- To analyze latency and throughput, so far we've assumed t_{CLK} depends only on our circuit
  - So lower t_{PD} → lower t_{CLK} → lower latency & higher throughput
- In practice, other constraints may set t_{CLK}
  - Propagation delay of other circuits
  - Limits on power consumption
- When our own circuit is not limiting t_{CLK}, throughput and latency tradeoffs change
  - Example: 4-*stage vs. 2-stage pipeline*
    - *If* $t_{CLK,4stage} = t_{CLK,2stage}/2$? Throughput: 2x, Latency: 1x
    - *If* $t_{CLK,4stage} = t_{CLK,2stage}$? Throughput: 1x, Latency: 2x

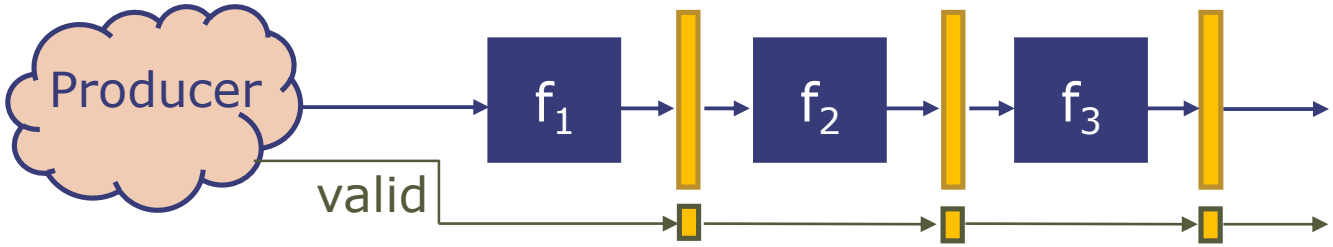
Pipeline Extensions



- Producer may not have input every cycle → Valid bits
- Consumer may not be able to accept output every cycle → Stall logic to freeze/pause the pipeline
- With large pipelines, may need to decouple stages further → Use queues instead of registers

Pipelines with Valid Bits

- If the producer won't give an input every cycle, tag each stage with a valid bit
  - In Minispec, use Maybe types

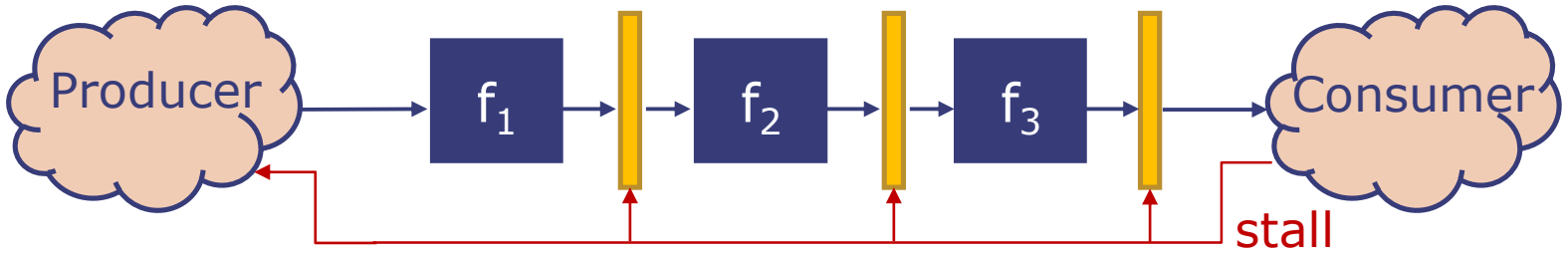


- Invalid inputs propagate through the pipeline, produce invalid outputs:

Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Stage 1	V1	V2	**Inv**	V3	**Inv**	**Inv**	V4	V5
Stage 2		V1	V2	**Inv**	V3	**Inv**	**Inv**	V4
Stage 3			V1	V2	**Inv**	V3	**Inv**	**Inv**

Pipelines with Stall Logic

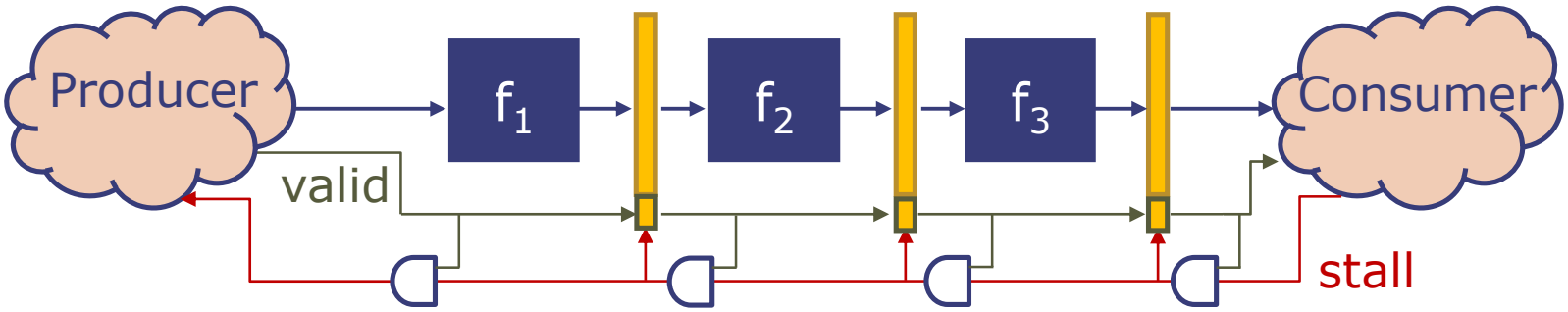
- If the consumer can't accept an output every cycle, we need to freeze the pipeline (and the producer!)
- Solution: Stall signal + registers with enable circuit
  - If stall is True, all pipeline registers retain their values



Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Stage 1	V1	V2	V3	V4	*V4*	V5	*V5*	*V5*
Stage 2		V1	V2	V3	*V3*	V4	*V4*	*V4*
Stage 3			V1	V2	*V2*	V3	*V3*	*V3*
Stall	False	False	False	*True*	False	*True*	*True*	False

Combining Valid Bits + Stall Logic

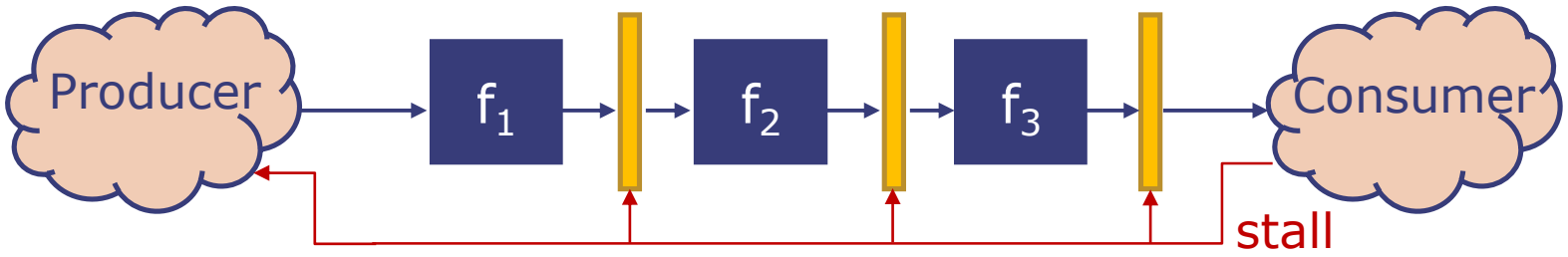
- If the consumer stalls, we can still let the pipeline make progress if a stage has an invalid value:



Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Stage 1	V1	V2	**Inv**	V3	V4	**Inv**	V5	*V5*
Stage 2	Inv	V1	V2	**Inv**	V3	V4	*V4*	*V4*
Stage 3	Inv	Inv	V1	V2	*V2*	V3	*V3*	*V3*
Output	Inv	Inv	Inv	V1	*V1*	V2	*V2*	*V2*
Stall	False	False	False	*True*	False	*True*	*True*	False

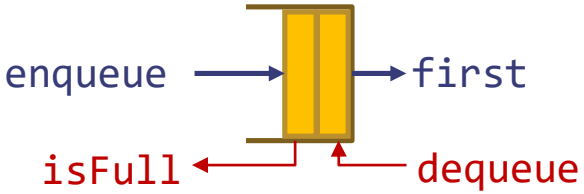
Stalling Delay in Large Pipelines

- We can't stall large pipelines immediately
  - Stall signal drives a huge number of register enables → excessive fan-out causes delay
  - Stall delay eventually sets t_{PD}, and limits t_{CLK}!

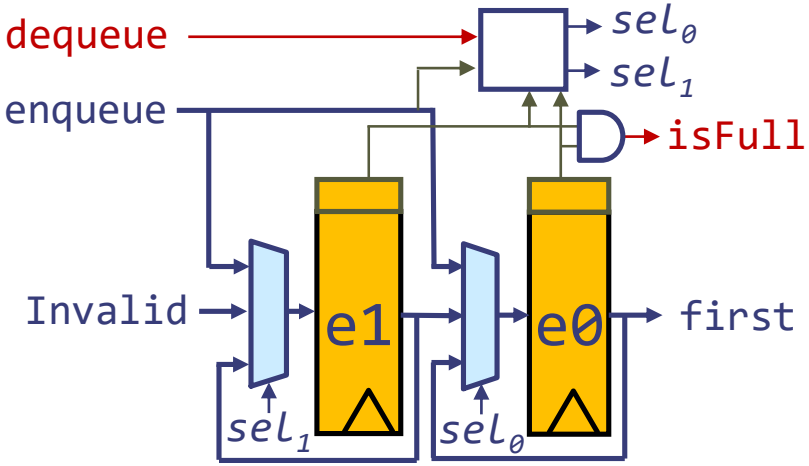


- Solution: Use queues with >1 element instead of registers to separate pipeline stages
  - Stages don't stall unless queue is full
  - Allows making stall decisions local

Example: 2-Element FIFO Queue
First-In, First-Out



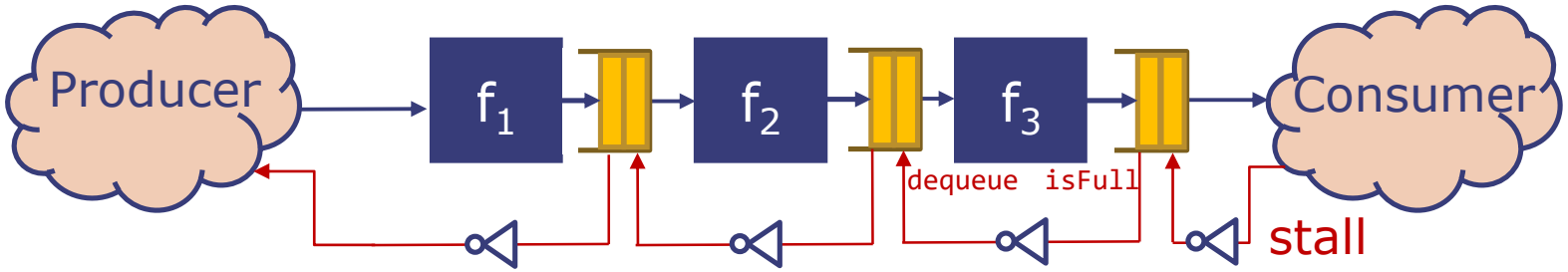
- Holds up to two values
- Outputs first enqueued value
- dequeue input controls whether to advance queue
- Possible implementation:




```
module FIFO2#(type T);
  Reg#(Maybe(T)) e0(Invalid);
  Reg#(Maybe(T)) e1(Invalid);

  method Maybe#(T) first = e0;
  method Bool isFull =
      isValid(e0) && isValid(e1);

  input Bool dequeue default = False;
  input Maybe#(T) enqueue default = Invalid;

  rule tick;
    if (!isValid(e0) && isValid(enqueue))
      e0 <= enqueue;
    else if (isValid(e0) && dequeue)
      e0 <= e1;
    if (isValid(e0) &&
        !isValid(e1) && isValid(enqueue))
      e1 <= enqueue;
    else if (isValid(e1) && dequeue)
      e1 <= Invalid;
  endrule
endmodule
```

Using Queues to Decouple Stages

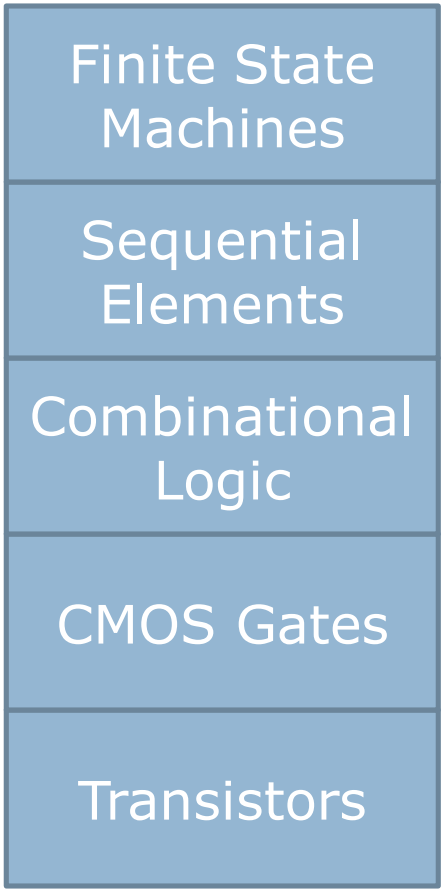
- Queues allow decoupling stall decisions:



  - A stage stalls only if its output queue is full
  - Tradeoff: Can't enqueue to full queue, even if an element is being dequeued on the same cycle
    - We could build a queue that allowed this, but this would add a combinational path from dequeue to isFull (so we'd still have the problem of high stall t_{PD}!)
- Queues also provide tolerance to variable latencies
  - Buffer multiple results without stalling producer when consumer takes variable number of cycles

From Special-Purpose FSMs to General-Purpose Procesors

6.004 So Far



- What can you do with these?
  - Take a (solvable) problem
  - Design a procedure (recipe) to solve the problem
  - Design a finite state machine that implements the procedure and solves the problem
- What you'll be able to do after this week:
  - Design a machine that can solve any solvable problem, given enough time and memory (a general-purpose computer)

Example: Factorial FSM

Let's design a circuit to compute factorial(N)

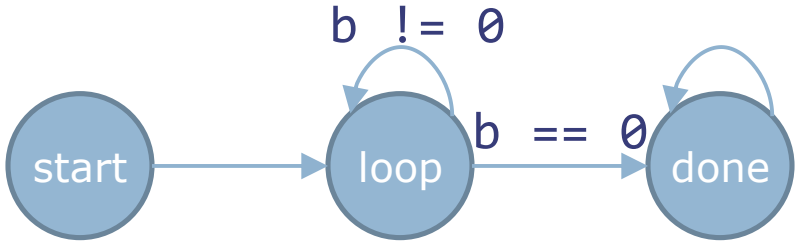
Python:

```
a = 1
b = N
while b != 0:
      a = a * b
      b = b - 1
```

C:

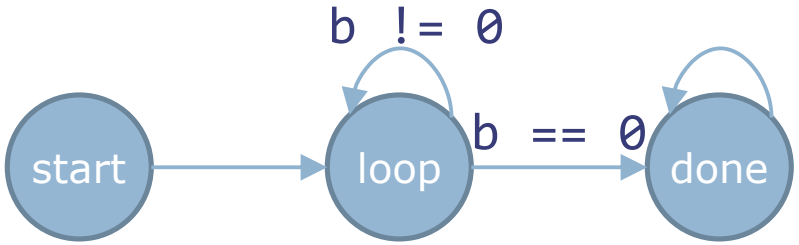
```
int a = 1;
int b = N;
while (b != 0) {
      a = a * b;
      b = b - 1;
}
```

High-level FSM:



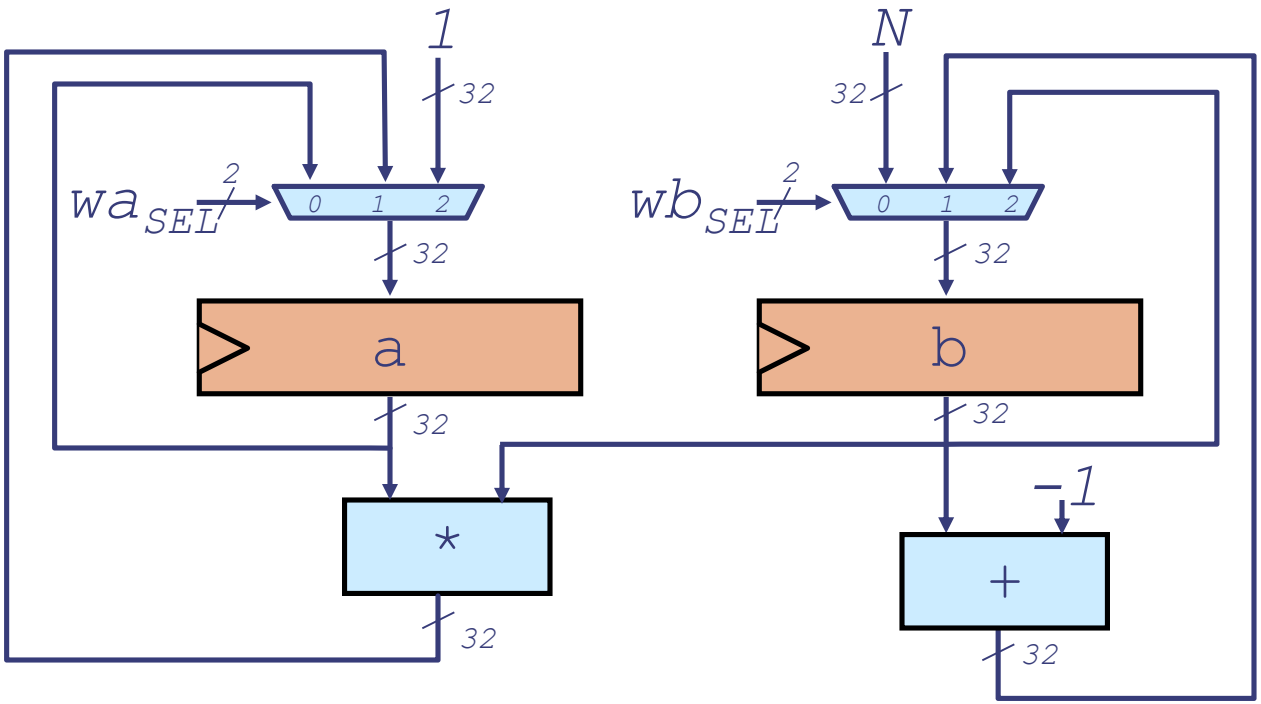
- Describes cycle-by-cycle behavior
- Registers (a, b)
- States (start, loop, done)
- Boolean transitions (b==0, b!=0)
- Register assignments in states (e.g., a ← a * b)

Datapath for Factorial

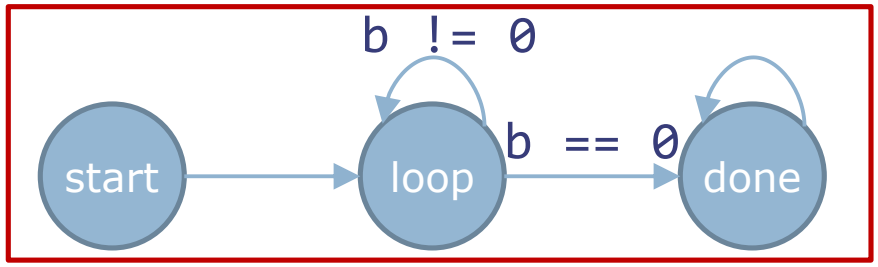



```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

- Implement registers
- Implement combinational circuit for each assignment
- Connect to input muxes

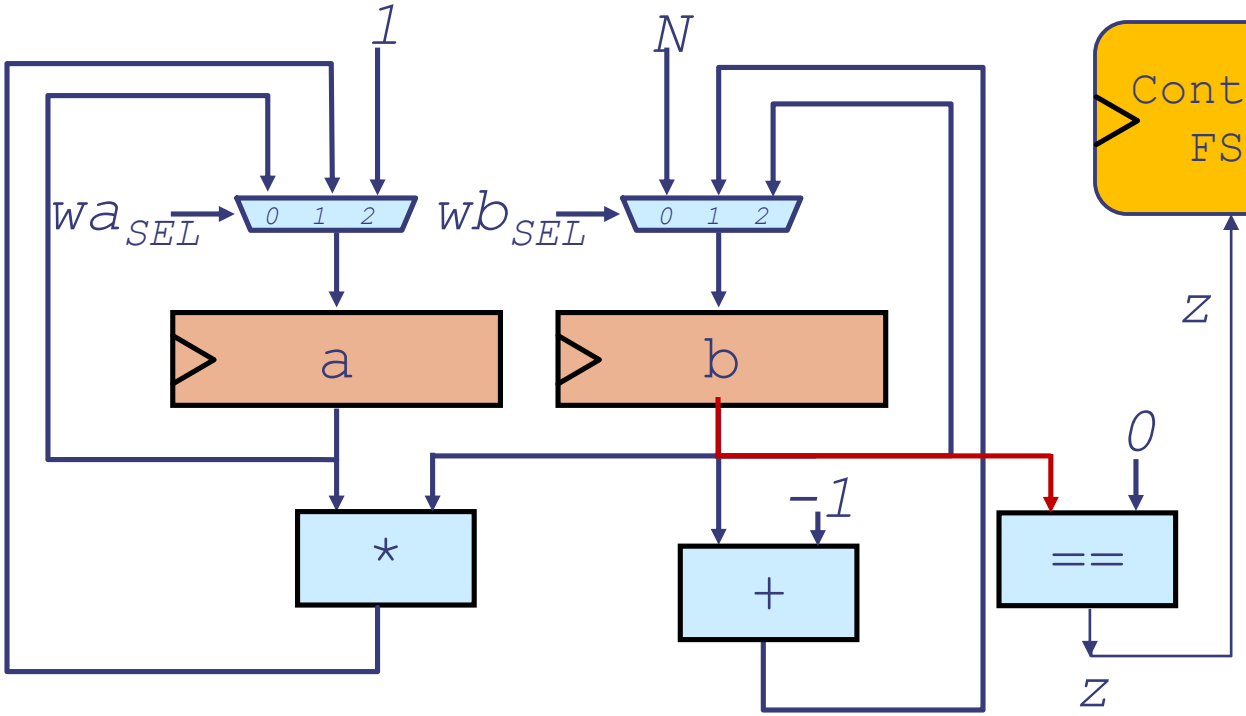


Control FSM for Factorial




```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

- Implement combinational logic for transition conditions
- Implement control FSM:
  - States: High-level FSM states
  - Inputs: Transition conditions
  - Outputs: Mux select signals



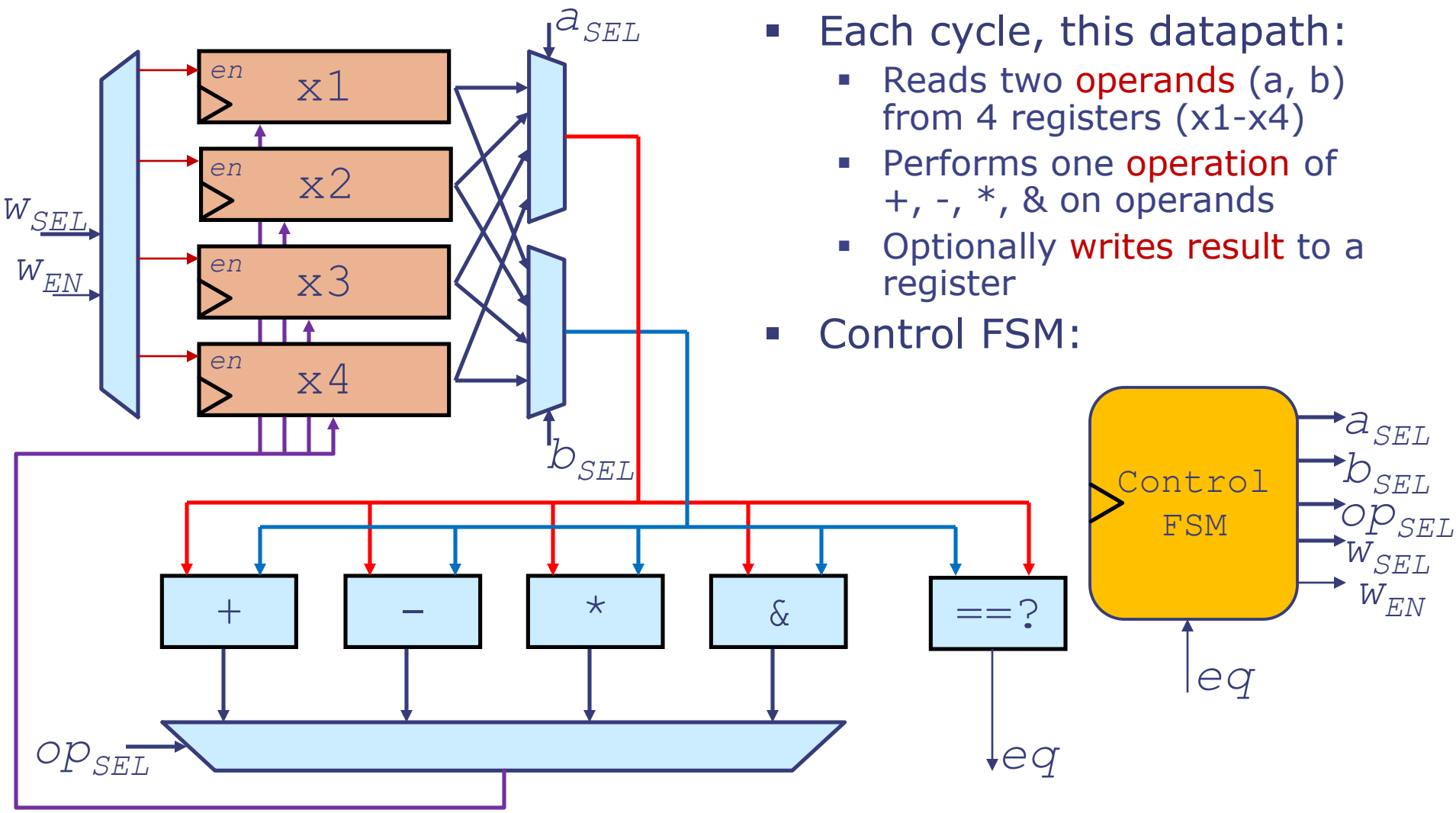
S	z	wa_{SEL}	wb_{SEL}	S′
0	0	2	0	1
0	1	2	0	1
1	0	1	1	1
1	1	1	1	2
2	0	0	2	2
2	1	0	2	2

Programming the Datapath

- We can use our factorial datapath and change the control FSM to solve other problems! Examples:
  - Multiplication
  - Squaring

- But very limited problems. Reasons:
  - Limited storage (only two registers!)
  - Limited set of operations, and inputs to those operations
  - Limited inputs to the control FSM

A Simple Programmable Datapath



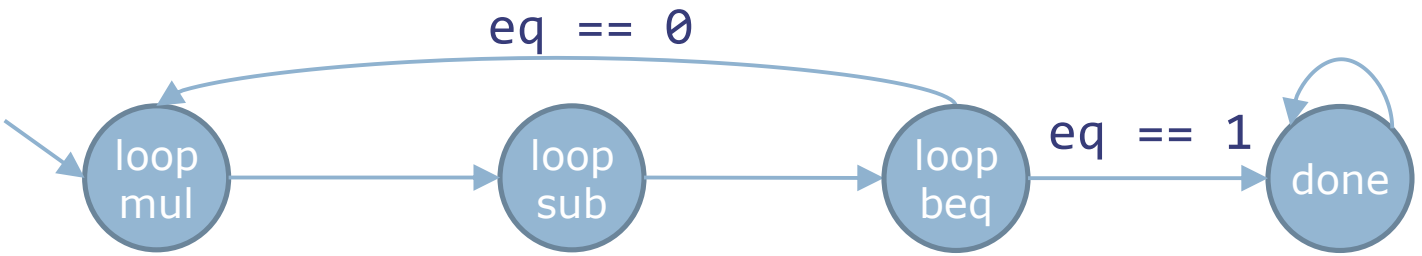
- Each cycle, this datapath:
  - Reads two operands (a, b) from 4 registers (x1-x4)
  - Performs one operation of +, -, *, & on operands
  - Optionally writes result to a register
- Control FSM:

A Control FSM for Factorial

- Assume initial register contents:

```
x1 value = 1
x2 value = N
x3 value = -1
x4 value = 0
```

- Control FSM:

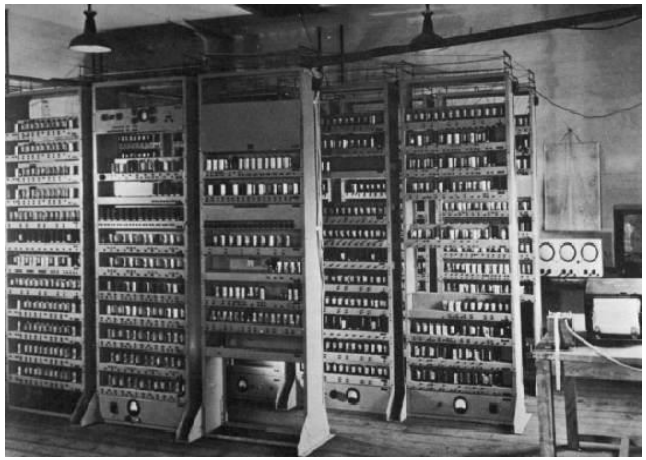


loop mul	loop sub	loop beq	done
asel = x1	asel = x2	asel = x2	asel = X
bsel = x2	bsel = x3	bsel = x3	bsel = X
opsel = 2 (*)	opsel = 0 (+)	opsel = X	opsel = X
wen = 1	wen = 1	wen = 0	wen = 0
wsel = x1	wsel = x2	wsel = X	wsel = X
x1 <= x1 * x2	x2 <= x2 + x3		N! in x1
mul x1, x1, x2	add x2, x2, x3	beq x2, x3, loopmul	j done

New Problem → New Control FSM

- You can solve many problems with this datapath!
  - GCD, Fibonacci, exponentiation, division, square root, …
  - But nothing that requires more than four registers

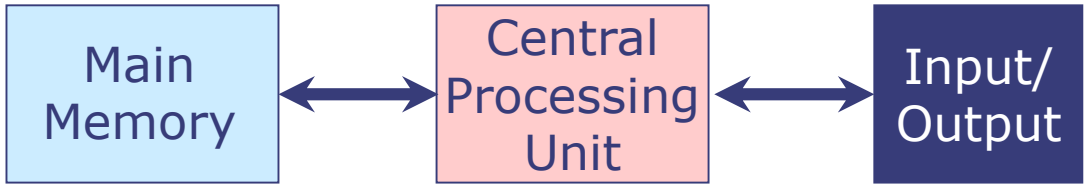
- By designing a control FSM, we are programming the datapath



- Early digital computers were programmed this way!
  - ENIAC (1943):
    - First general-purpose digital computer
    - Programmed by setting huge array of dials and switches
    - Reprogramming it took about 3 weeks

The von Neumann Model

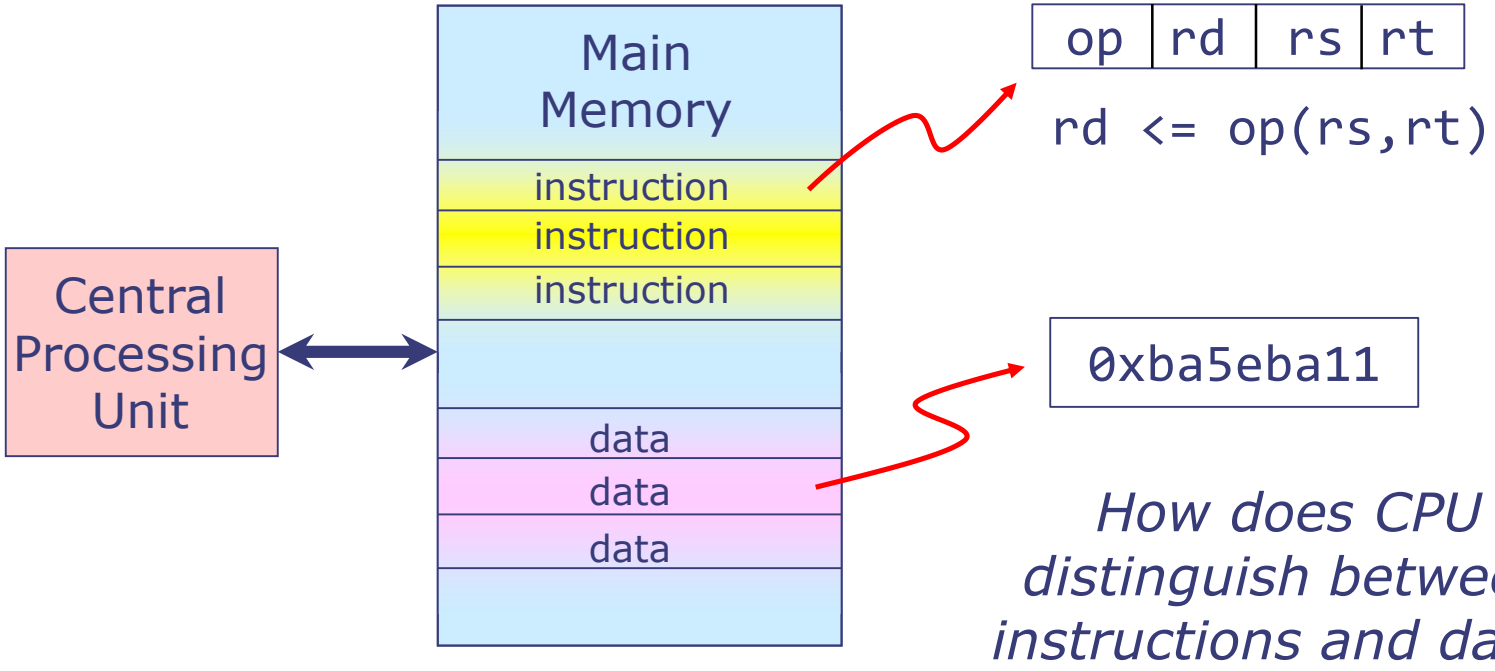
- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:



  - Main memory holds programs and their data
  - Central processing unit accesses and processes memory values
  - Input/output devices to communicate with the outside world

Key Idea: Stored-Program Computer

- Express program as a sequence of coded instructions
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program



Anatomy of a von Neumann Computer

Internal storage

Datapath

control

status

Control Unit

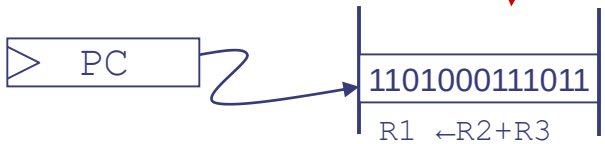
address

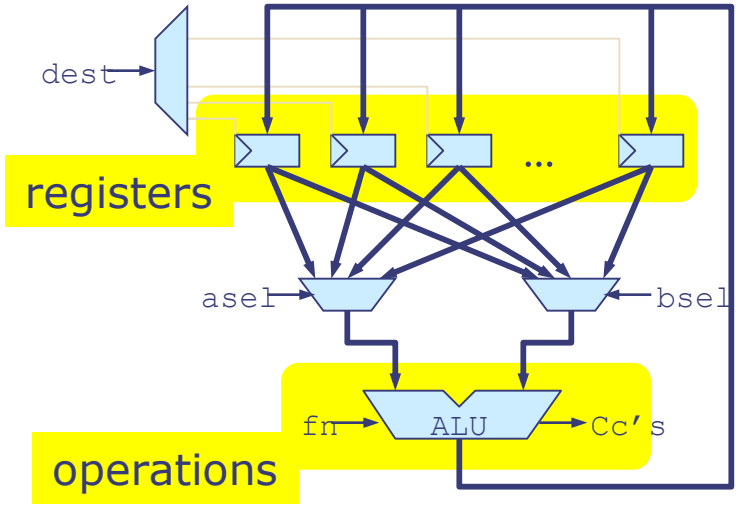
data

address

instructions

Main Memory





- Instructions coded as binary data
- Program Counter or PC: Address of the instruction to be executed
- Logic to translate instructions into control signals for datapath

Thank you!

Next lecture: Building a RISC-V processor