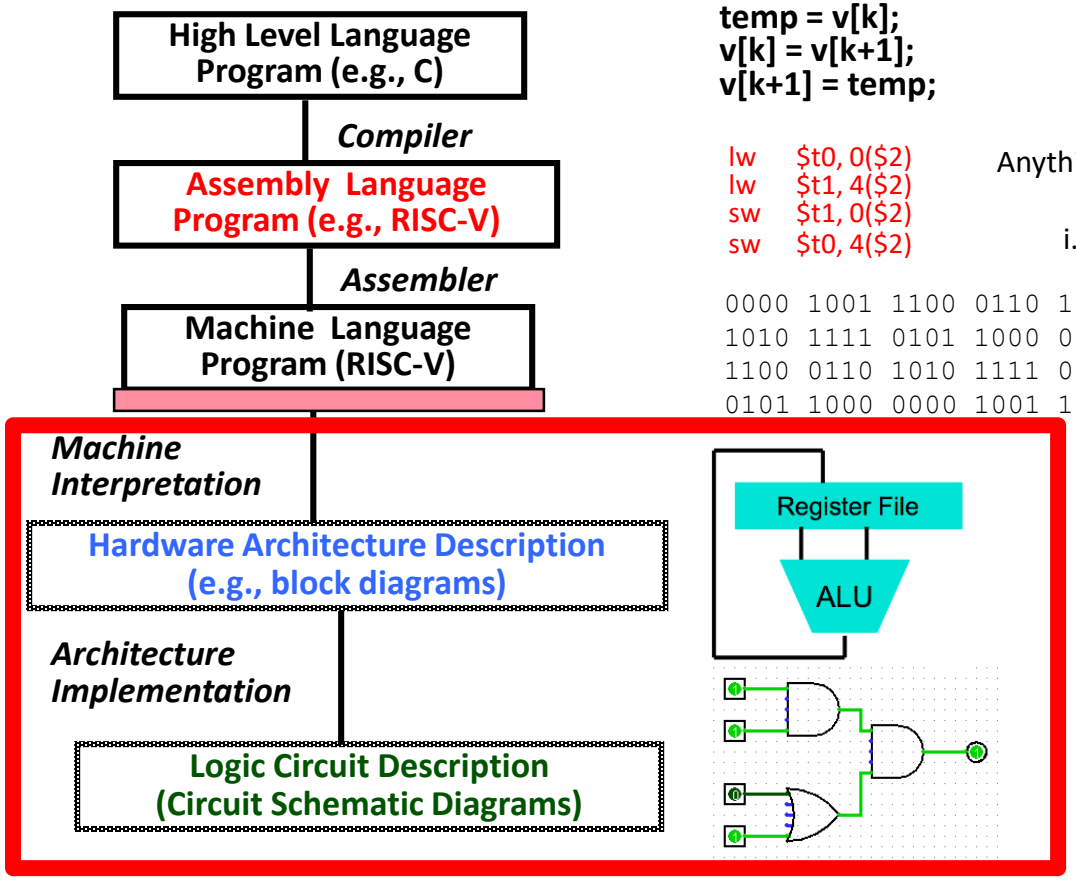
Lecture 3 – Hardware Implementation/ Simple Machine Implementations

Qiang Cao

Levels of Representation/Interpretation



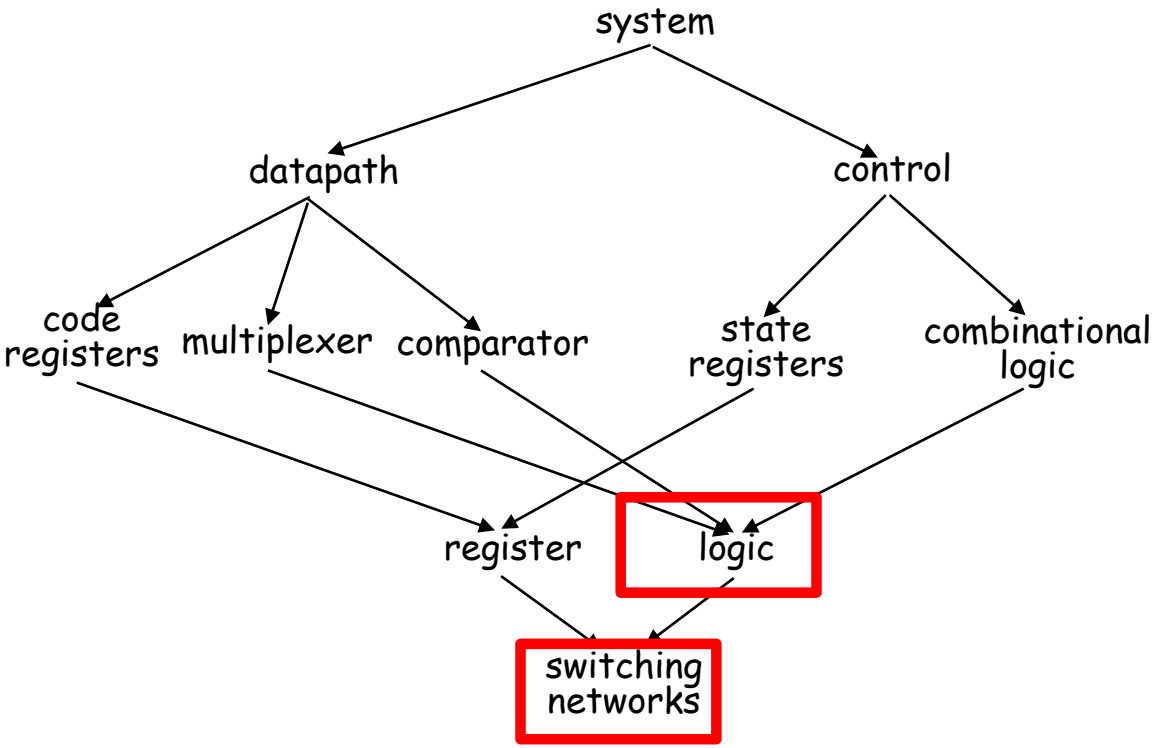

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Design Hierarchy



Hardware Implementation

- Boolean Algebra
- Timing and State Machines
- Datapath Elements: Mux + ALU
- RISC-V Lite Datapath
- And, in Conclusion, …

Hardware Implementation

- Boolean Algebra
- Timing and State Machines
- Datapath Elements: Mux + ALU
- RISC-V Lite Datapath
- And, in Conclusion, …

Boolean Algebra

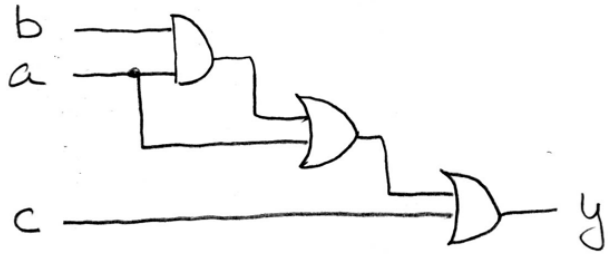
- Use plus for OR
  - "logical sum"
- Use product for AND (a•b or implied via ab)
  - "logical product"
- "Hat" to mean complement (NOT)
- Thus

  $ab + a + \overline{c}$

$= a \bullet b + a + \overline{c}$

= (a AND b) OR a OR (NOT c)

Boolean Algebra: Circuit & Algebraic Simplification



original circuit

$$\downarrow$$

$$y = ((ab) + a) + c$$

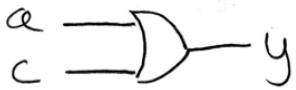
equation derived from original circuit

$$\downarrow$$

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

algebraic simplification

$$\downarrow$$



simplified circuit

Laws of Boolean Algebra

$X \overline{X} = 0$	$X + \overline{X} = 1$	Complementarity
$X\ 0 = 0$	$X + 1 = 1$	Laws of 0's and 1's
$X\ 1 = X$	$X + 0 = X$	Identities
$X\ X = X$	$X + X = X$	Idempotent Laws
$X\ Y = Y\ X$	$X + Y = Y + X$	Commutativity
$(X\ Y)\ Z = X\ (Y\ Z)$	$(X + Y) + Z = X + (Y + Z)$	Associativity
$X\ (Y + Z) = X\ Y + X\ Z$	$X + Y\ Z = (X + Y)\ (X + Z)$	Distribution
$X\ Y + X = X$	$(X + Y)\ X = X$	Uniting Theorem
$\overline{X}\ Y + X = X + Y$	$(\overline{X} + Y)\ X = X\ Y$	Uniting Theorem v. 2
$\overline{X\ Y} = \overline{X} + \overline{Y}$	$\overline{X + Y} = \overline{X}\ \overline{Y}$	DeMorgan's Law

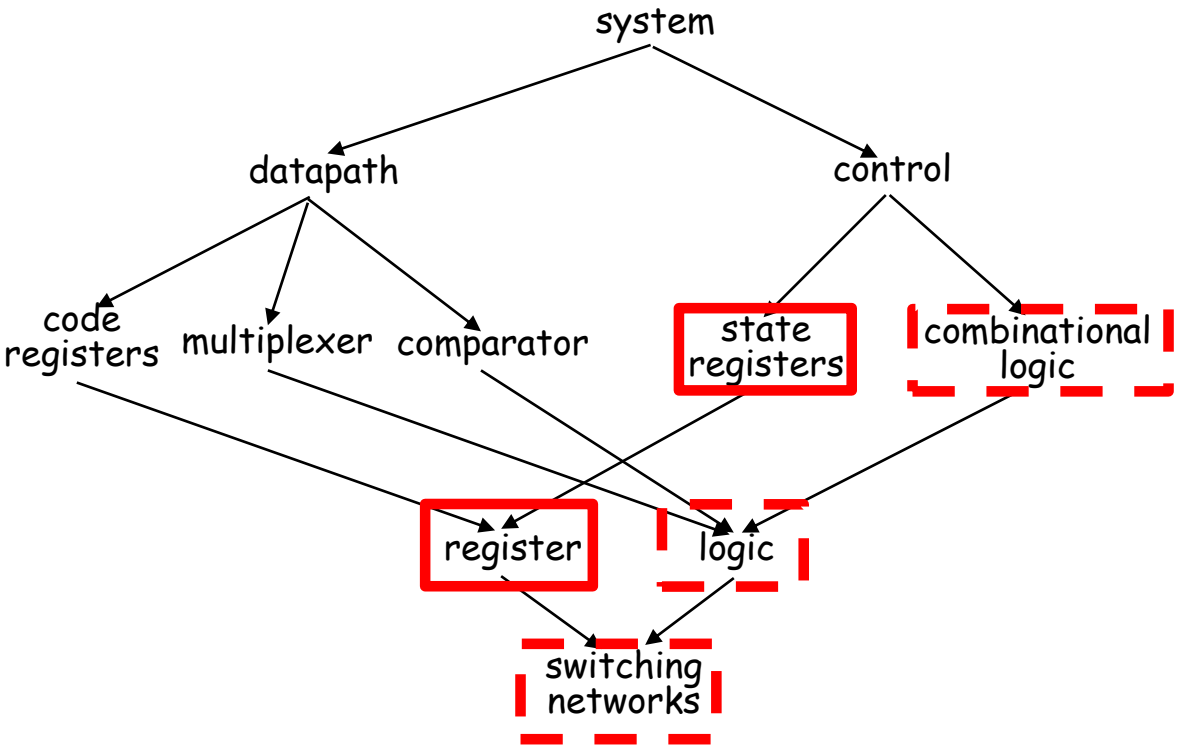
Hardware Implementation

- Boolean Algebra
- Timing and State Machines
- Datapath Elements: Mux + ALU
- RISC-V Lite Datapath
- And, in Conclusion, …

Types of Circuits

- *Synchronous Digital Systems* consist of two basic types of circuits:
  - Combinational Logic (CL) circuits
    - Output is a function of the inputs only, not the history of its execution
    - E.g., circuits to add A, B (ALUs)
    - Last lecture was comninational logic
  - Sequential Logic (SL)
    - Circuits that “remember” or store information
    - aka “State Elements”
    - E.g., memories and registers (Registers)
    - Rest of today’s lecture is *sequential logic*

Design Hierarchy

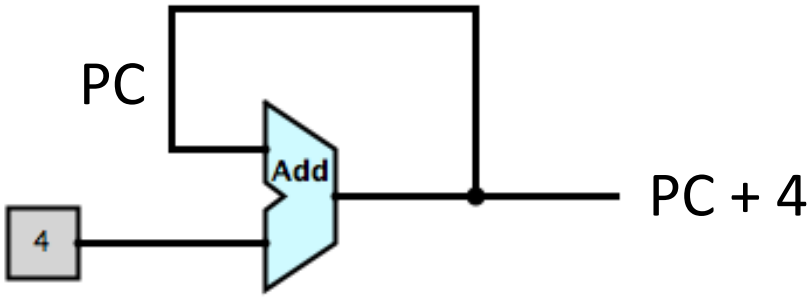


Uses for State Elements

- Place to store values for later re-use:
  - Register files (like $1-$31 on the RISC-V)
  - Memory (caches and main memory)
- *Help control flow of information between combinational logic blocks*
  - State elements hold up the movement of information at input to combinational logic blocks to allow for orderly passage

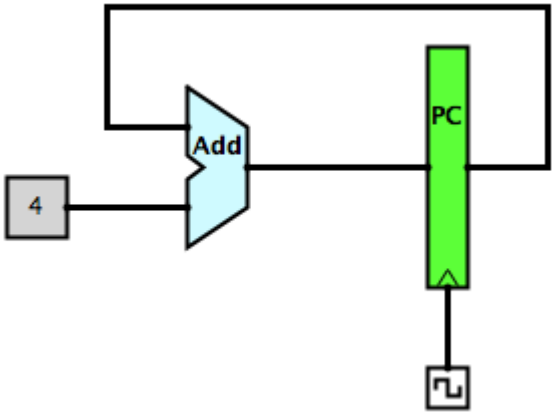
Program Counter: First Design

- Program Counter "PC"
  - Instruction address
  - Next PC: add 4 to current value
  - Let's try …



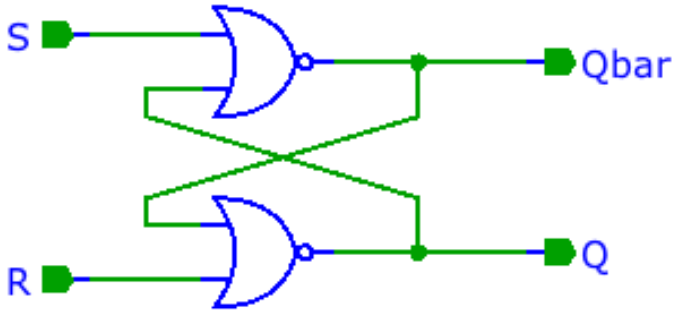
- Something is not quite right:
  - PC and PC+4 simutaneously on same wire???

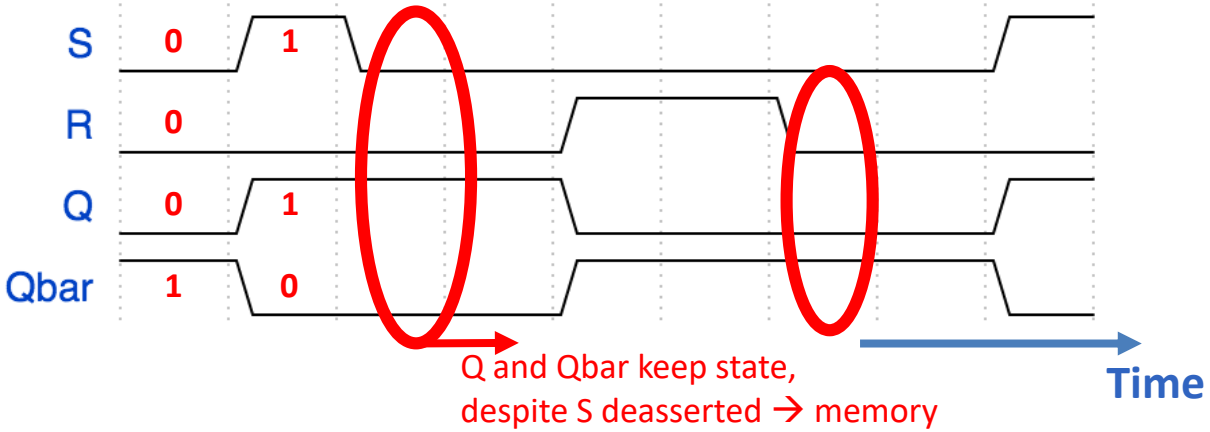
Fix



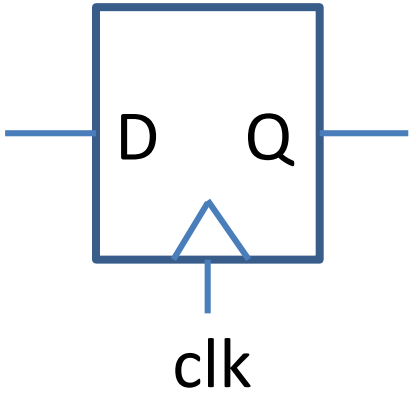
- Memory element breaks the feedback loop
- How does it work?

RS Latch


S
Qbar
R
Q

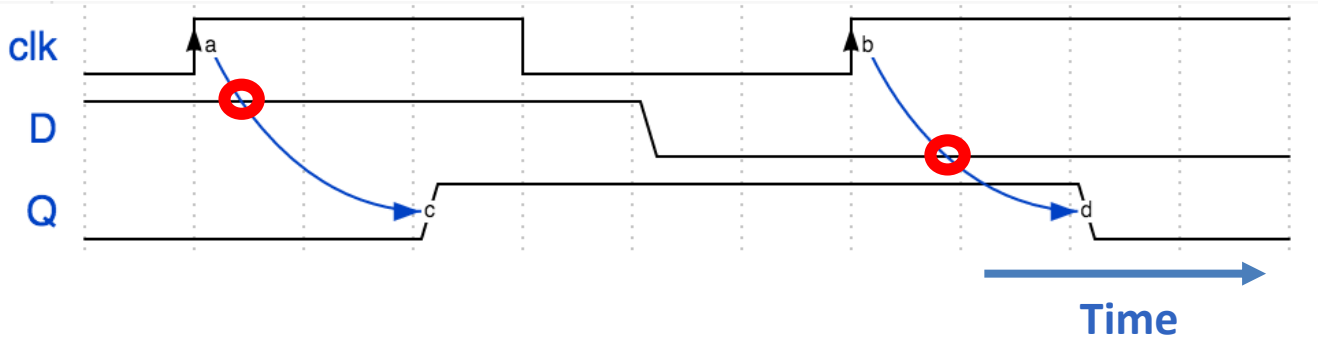

S
R
Q
Qbar
0
1
0
0
1
0
1
0
Q and Qbar keep state,
despite S deasserted → memory
Time

(Positive) Edge Triggered *Flip-Flop*

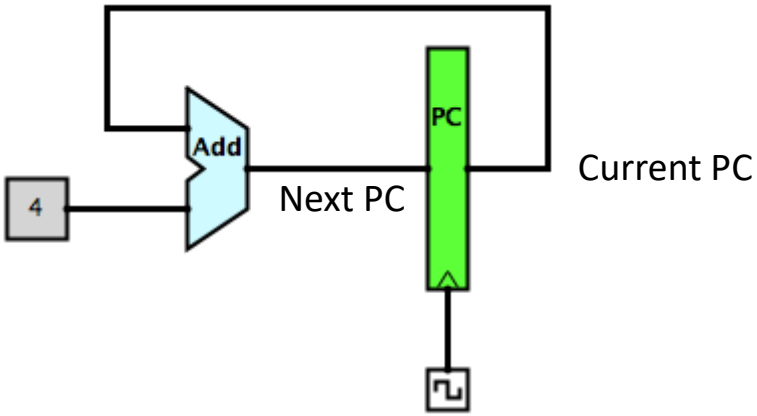


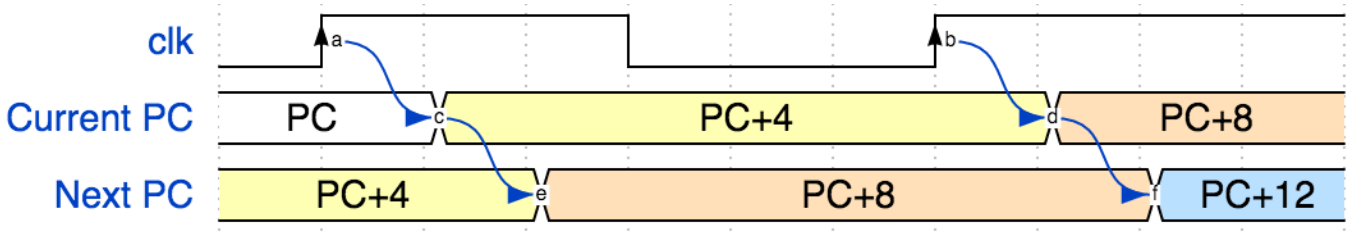
RS *latch* operates on input levels
D *FF* operates on (clock) edges

Clock "clk"



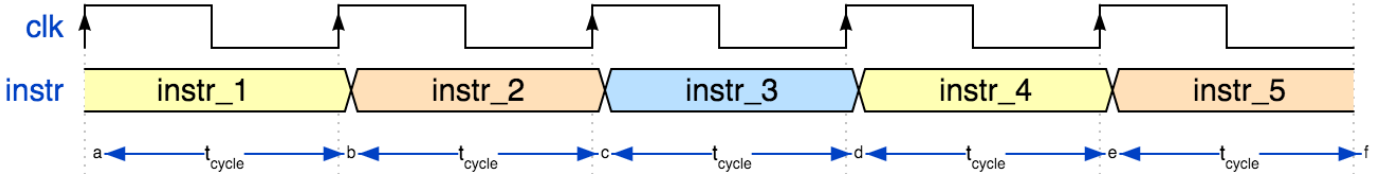
Program Counter: Improved Design





Clock

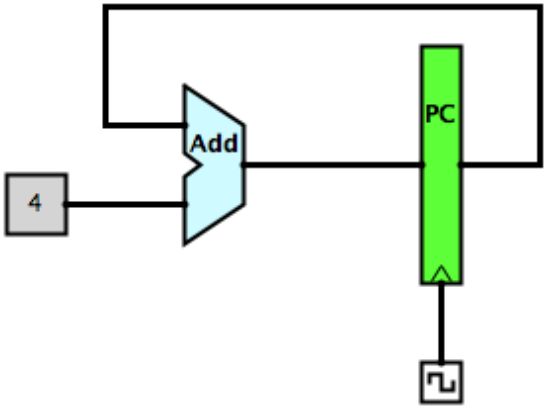
$$f_s = 1 / t_{cycle}$$



Clock frequency f_s	Period $t_{cycle} = 1/f_s$
CPU 2.5 GHz	400 ps
Drum/heartbeat, 1Hz	1 s

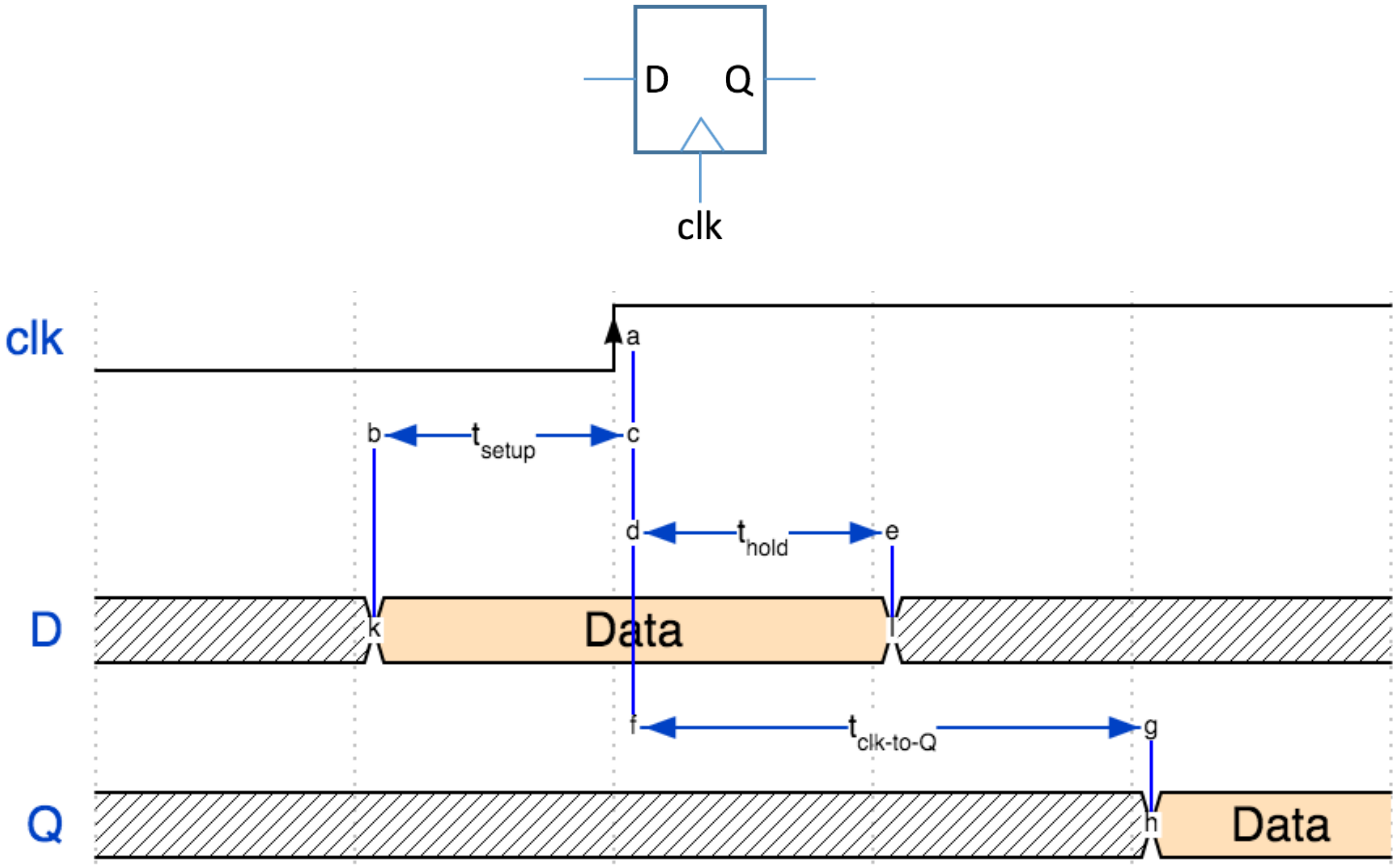
Maximum Clock Speed

- How fast can the PC circuit operate?
  - I.e. what is the maximum clock frequency?

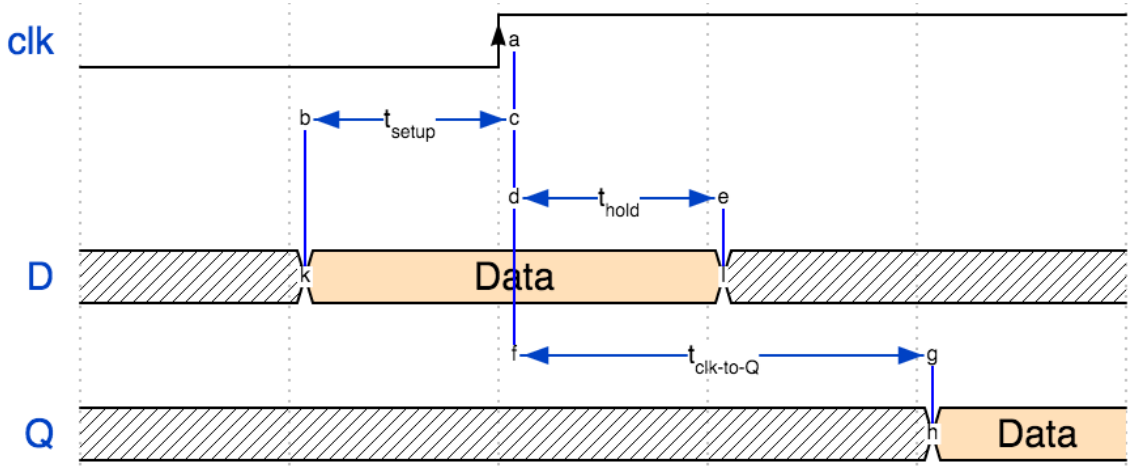


- Let's look at the timing requirements of each part
  - Let's start with the flip-flop

Flip-Flop Timing

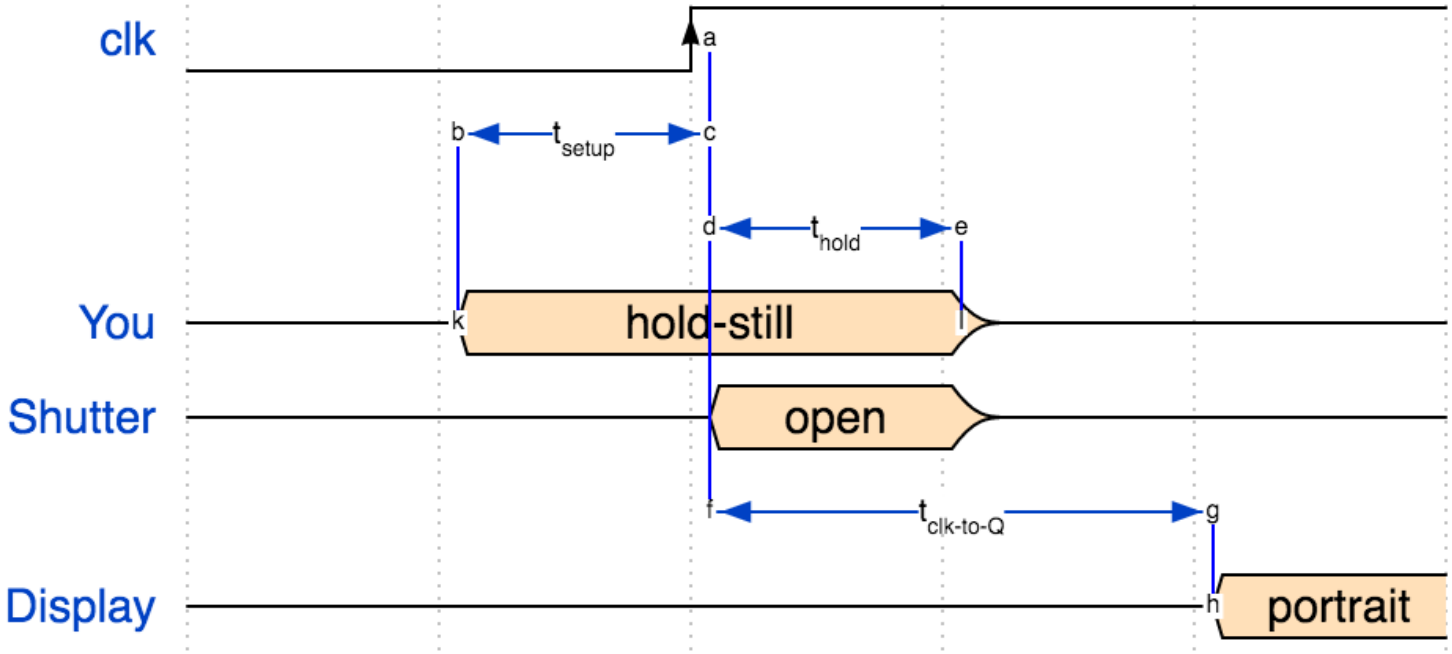


Flip-Flop Timing

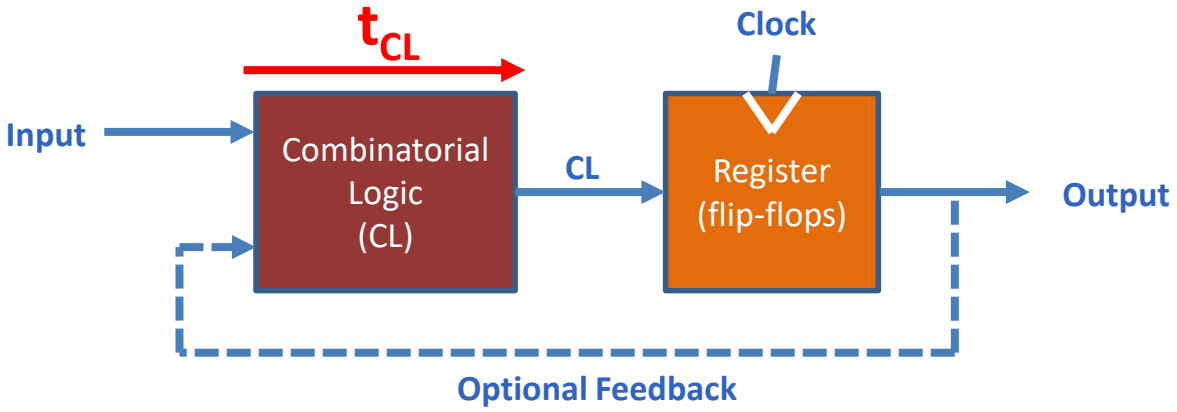


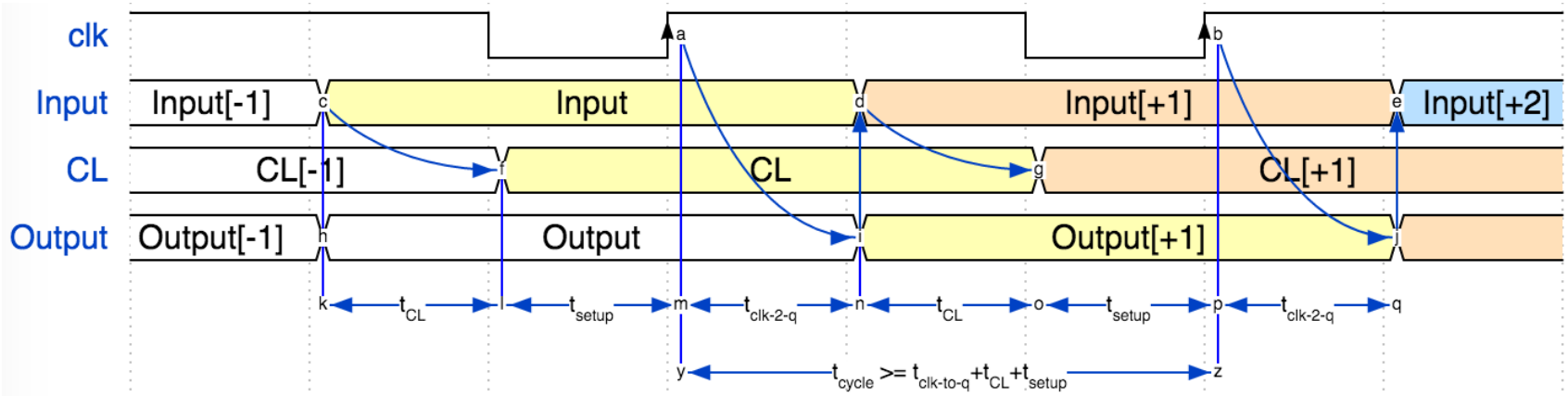
Time	Description
t_{setup}	time during which D must not change ***before*** positive clock edge
t_{hold}	time during which D must not change ***after*** positive clock edge
$t_{clk\text{-}to\text{-}Q}$	delay after positive clock edge after which D appears at Q output

Camera Analogy

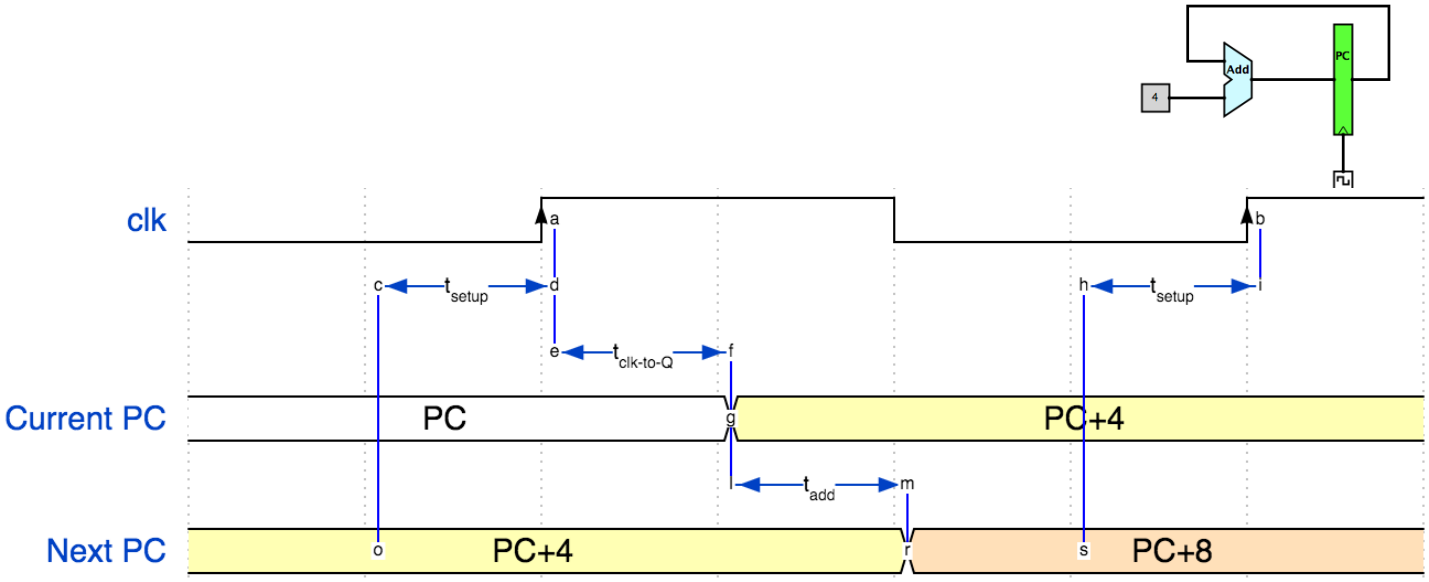


Maximum Clock Speed





Maximum PC Clock Speed



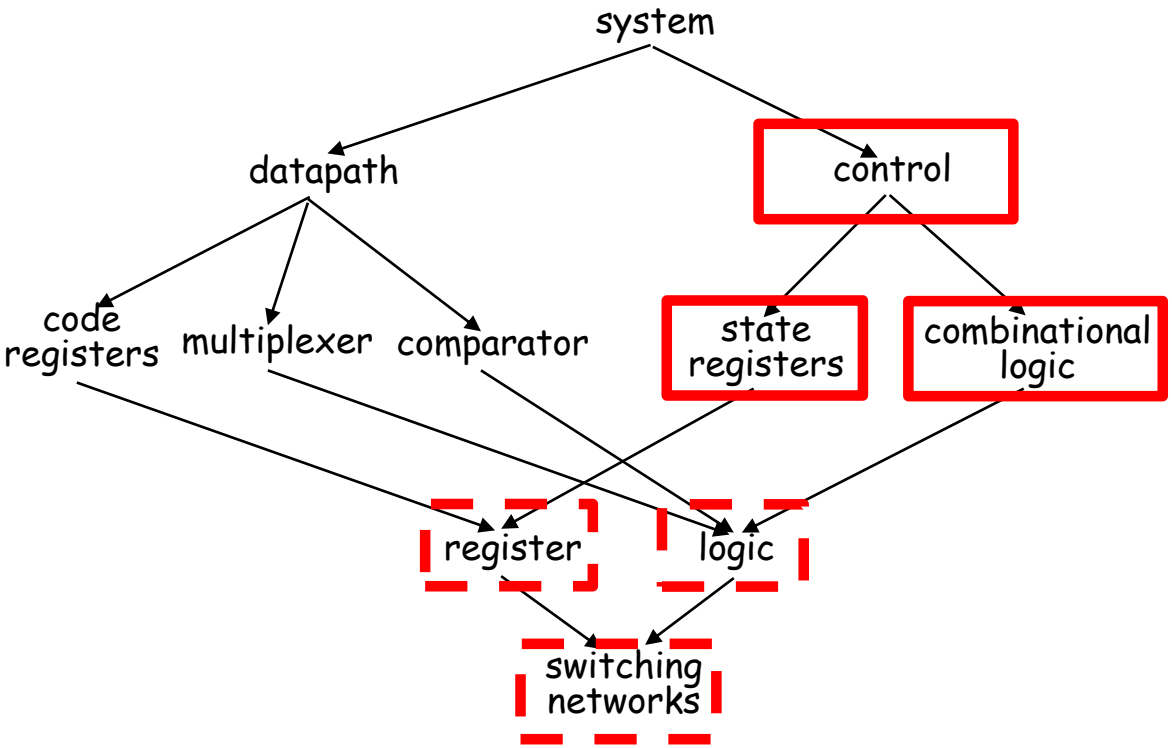
- Minimum cycle time: $t_{min} = t_{setup} + t_{add} + t_{clk-to-Q}$
- Maximum clock rate: $f_{clk-max} = \frac{1}{t_{min}}$

Example: $t_{setup} \geq 15ps$, $t_{add} = 75ps$, $t_{clk\text{-}toQ} \geq 10ps$

$t_{min} \geq 100ps$

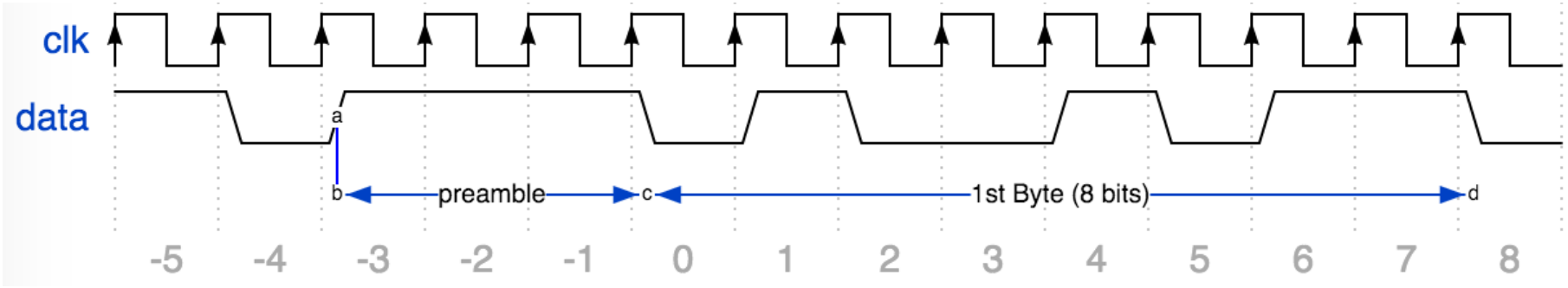
$f_{clk\text{-}max} \leq 10GHz$

Design Hierarchy



Example: Serial Communication

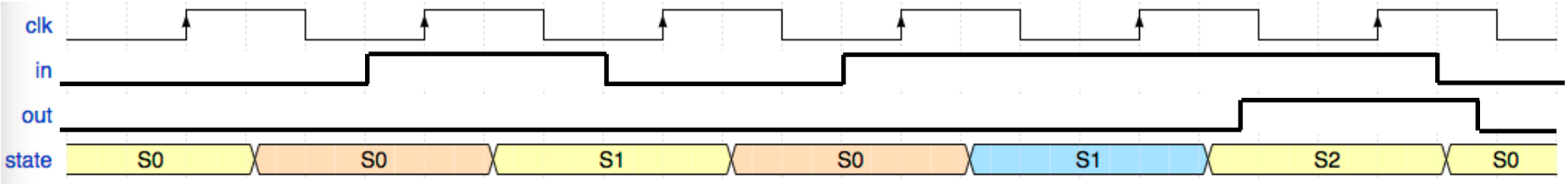
- Wifi sends data “1 bit at a time”
- How do we know where a byte “starts”?
- Send “preamble …”
  - E.g. 3 one’s in a row



FSM* to Detect 3 One's

* FSM = Finite State Machine

FSM to detect the occurrence of 3 consecutive 1's in the input.

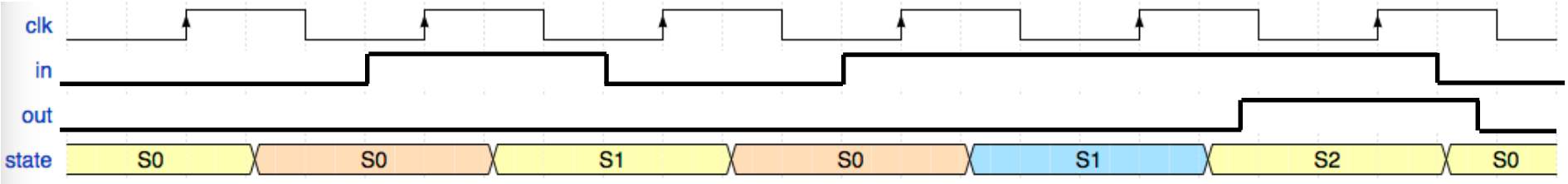


State transition diagram:

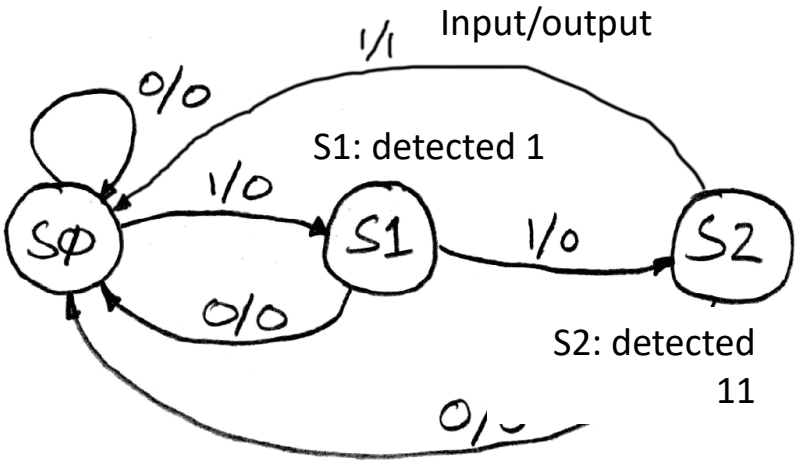
FSM* to Detect 3 One's

* FSM = Finite State Machine

FSM to detect the occurrence of 3 consecutive 1's in the input.



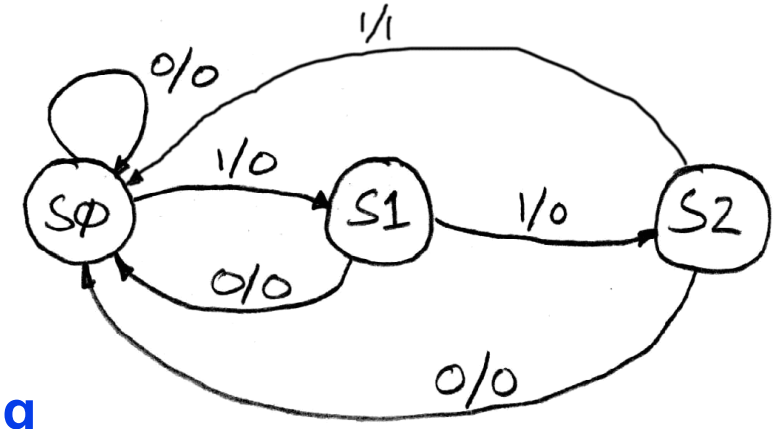
State transition diagram:



State transitions are controlled by the clock: on each clock cycle the FSM

- checks the inputs,
- transitions to a new state, and
- produces a new output …

FSM Combinatorial Logic



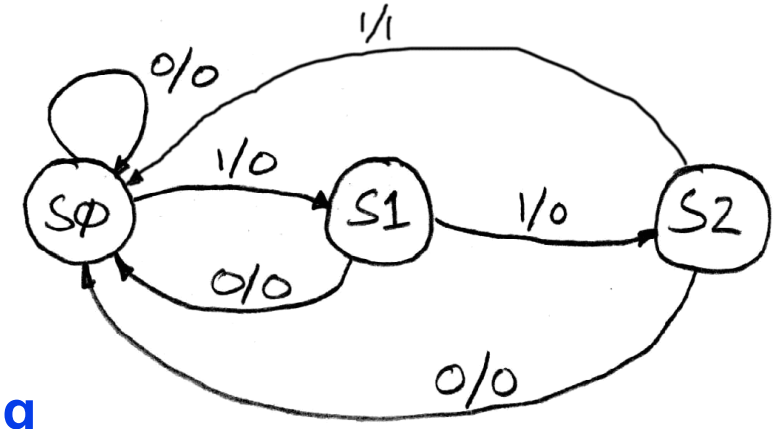
State Encoding

State	Code

Truth Table

Current State	Input	Next State	Output

FSM Combinatorial Logic



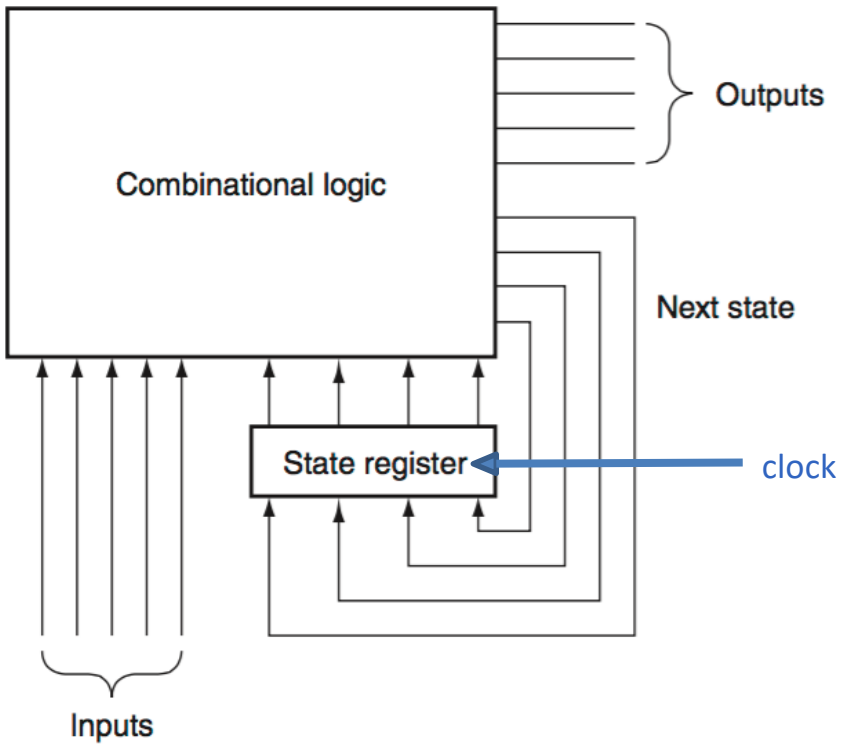
State Encoding

State	Code
S0	00
S1	01
S2	10
unused	11

Truth Table

PS	Input	NS	Output
XX	0	00	0
00	1	01	0
01	1	10	0
10	1	00	1
11	X	00	0

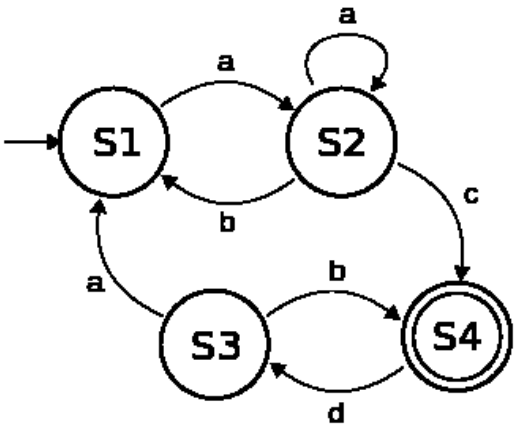
Hardware Implementation of FSM



Finite State Machines (FSM) - Summary

- Describe computation over time
- Represent FSM with “state transition diagram”
- Start at given state and input, follow some edge to next (or same) state
- With combinational logic and registers, any FSM can be implemented in hardware

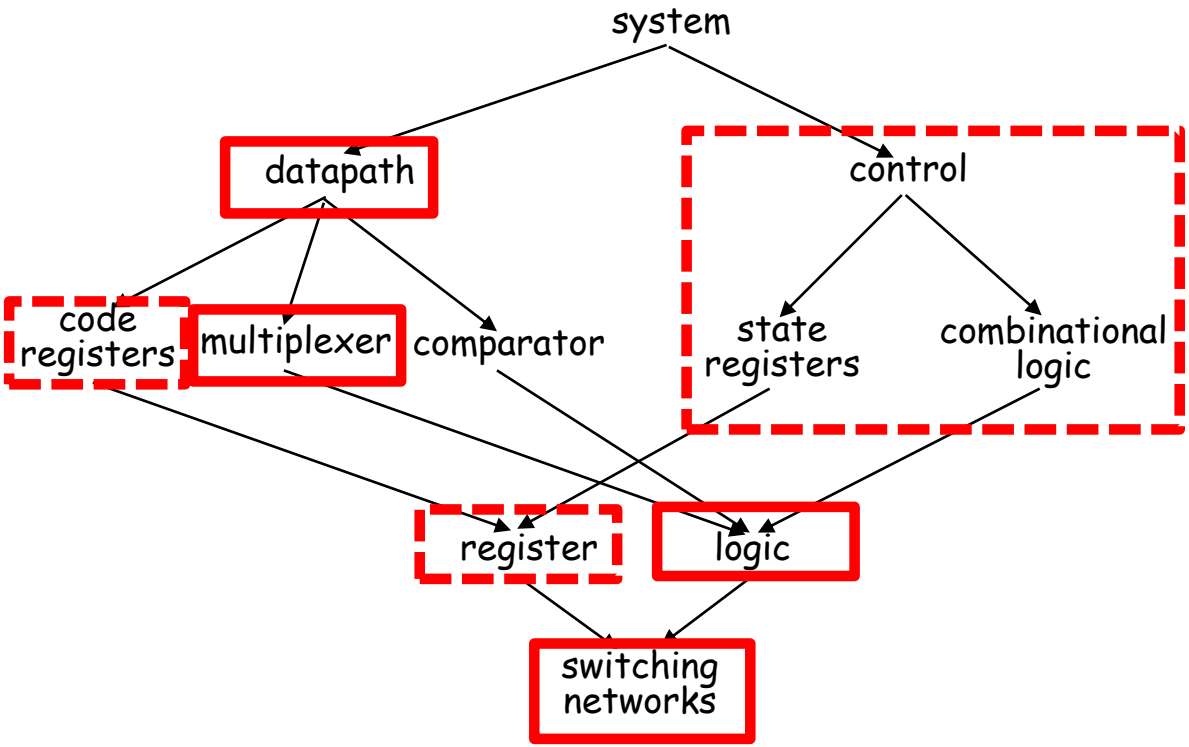
state transition diagram



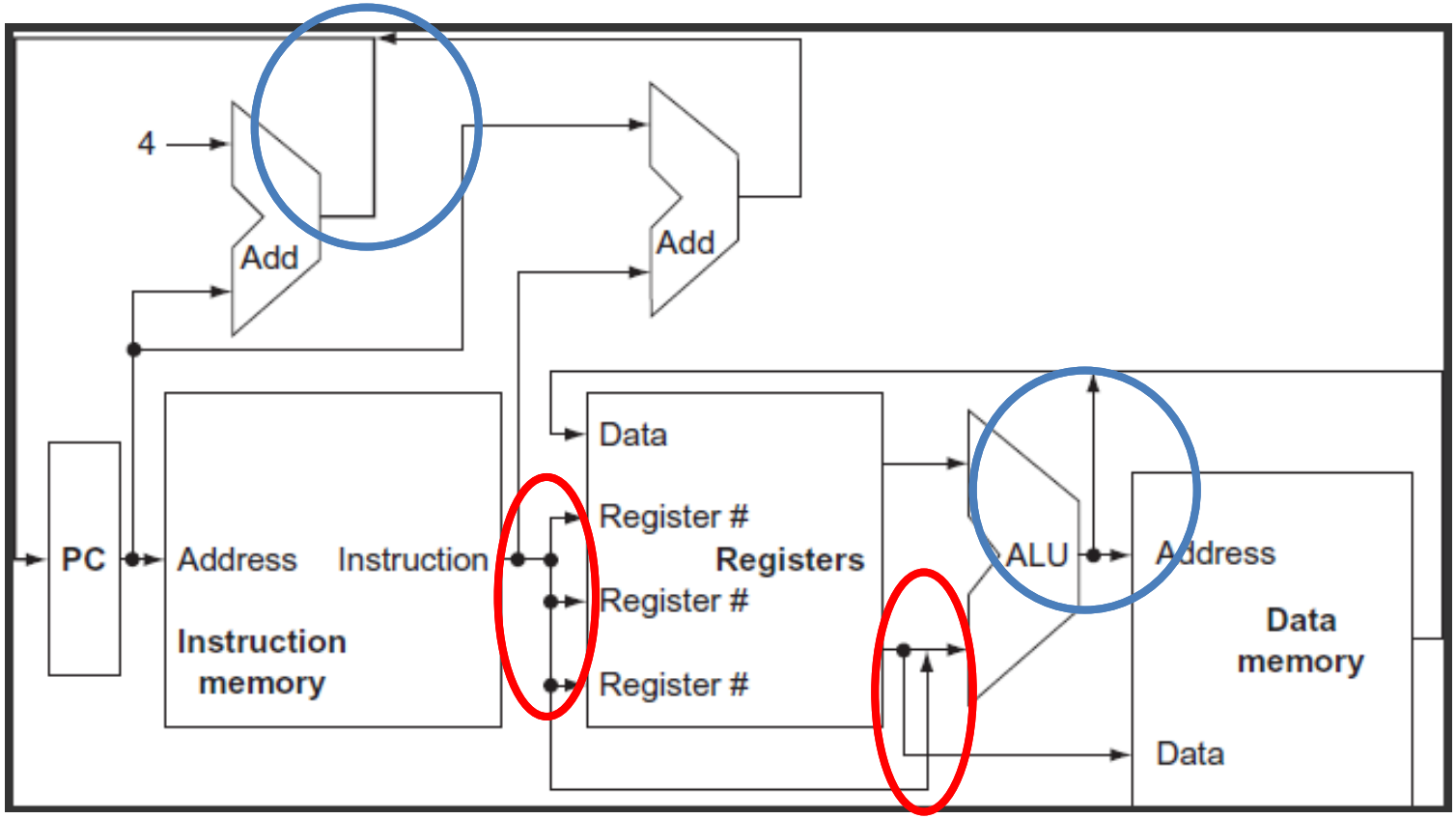
Hardware Implementation

- Boolean Algebra
- Timing and State Machines
- Datapath Elements: Mux + ALU
- RISC-V Lite Datapath
- And, in Conclusion, …

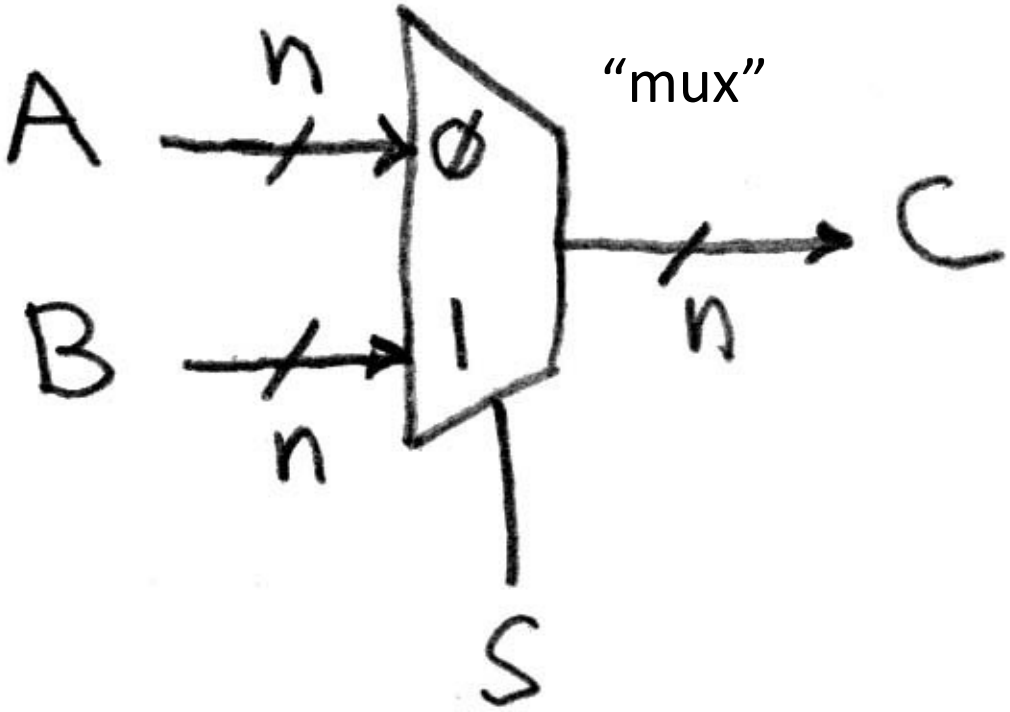
Design Hierarchy



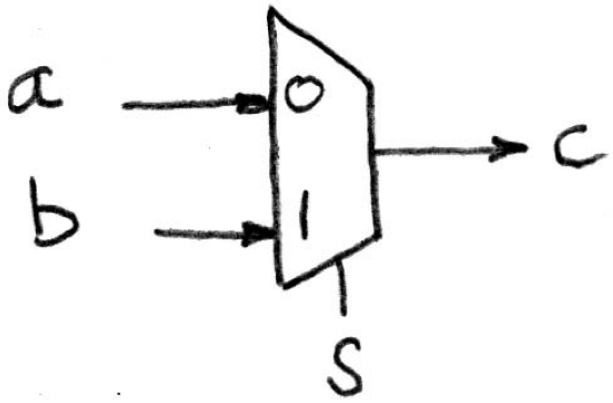
Conceptual RISC-V Datapath



Data Multiplexer
(e.g., 2-to-1 x n-bit-wide)



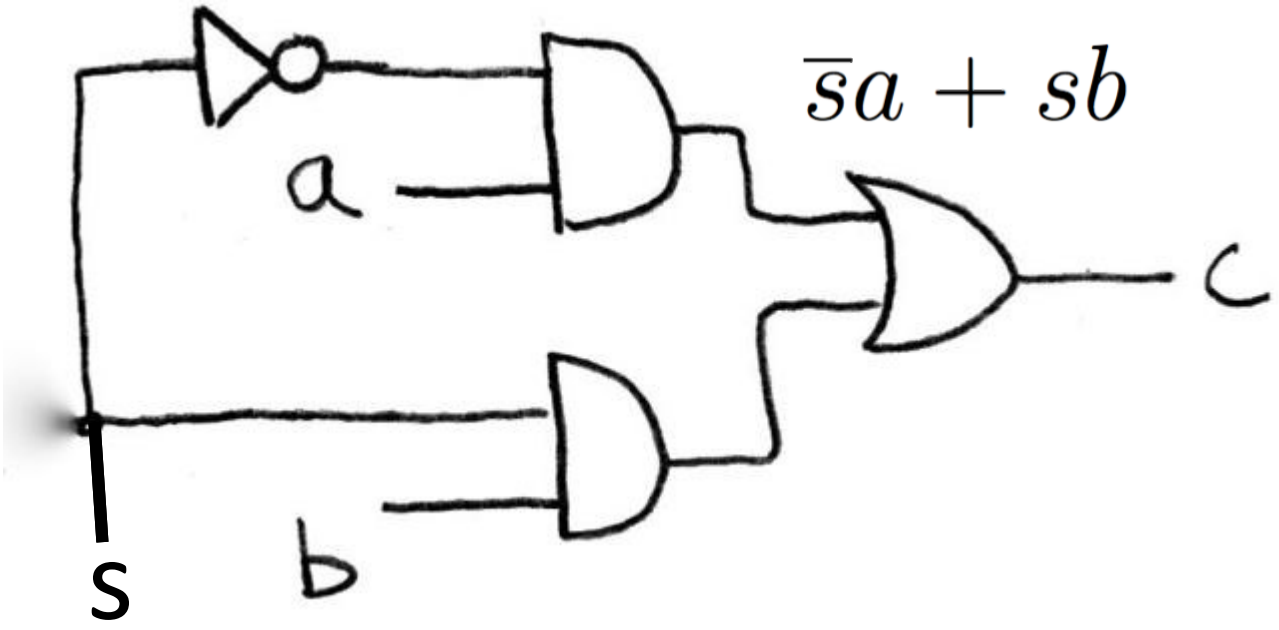
N Instances of 1-bit-Wide Mux



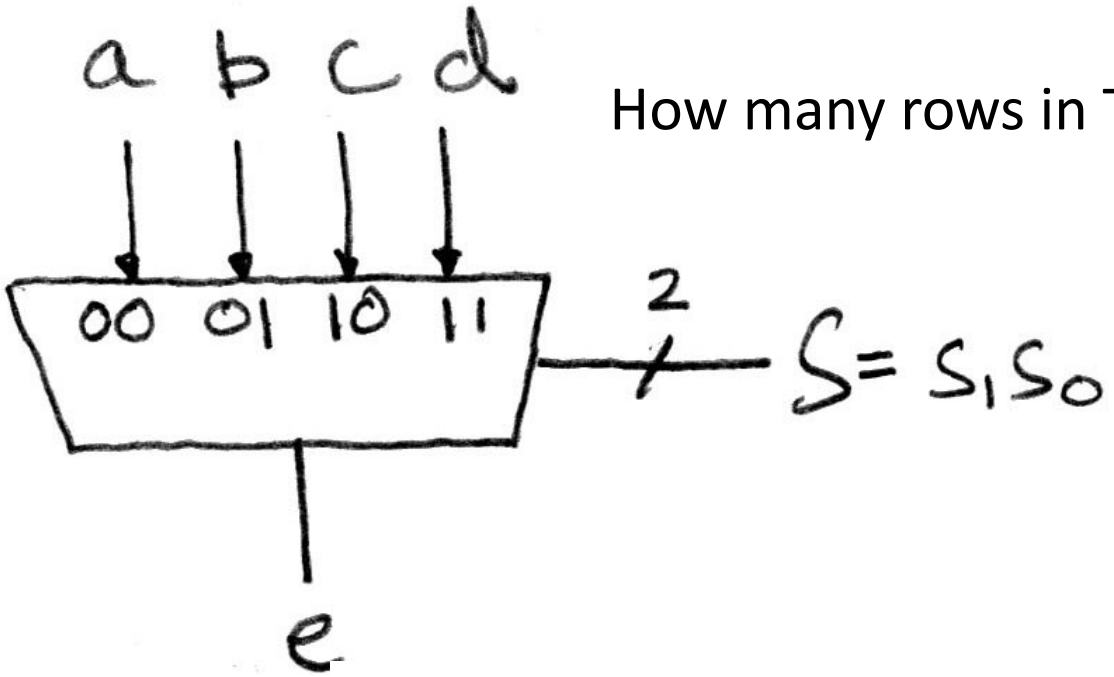
$$
\begin{aligned}
c &= \overline{s}a\overline{b} + \overline{s}ab + s\overline{a}b + sab \\
&= \overline{s}(a\overline{b} + ab) + s(\overline{a}b + ab) \\
&= \overline{s}(a(\overline{b} + b)) + s((\overline{a} + a)b) \\
&= \overline{s}(a(1) + s((1)b) \\
&= \overline{s}a + sb
\end{aligned}
$$

s	ab	c
0	00	0
0	01	0
0	10	1
0	11	1
1	00	0
1	01	1
1	10	0
1	11	1

How Do We Build a 1-bit-Wide Mux (in Logisim)?



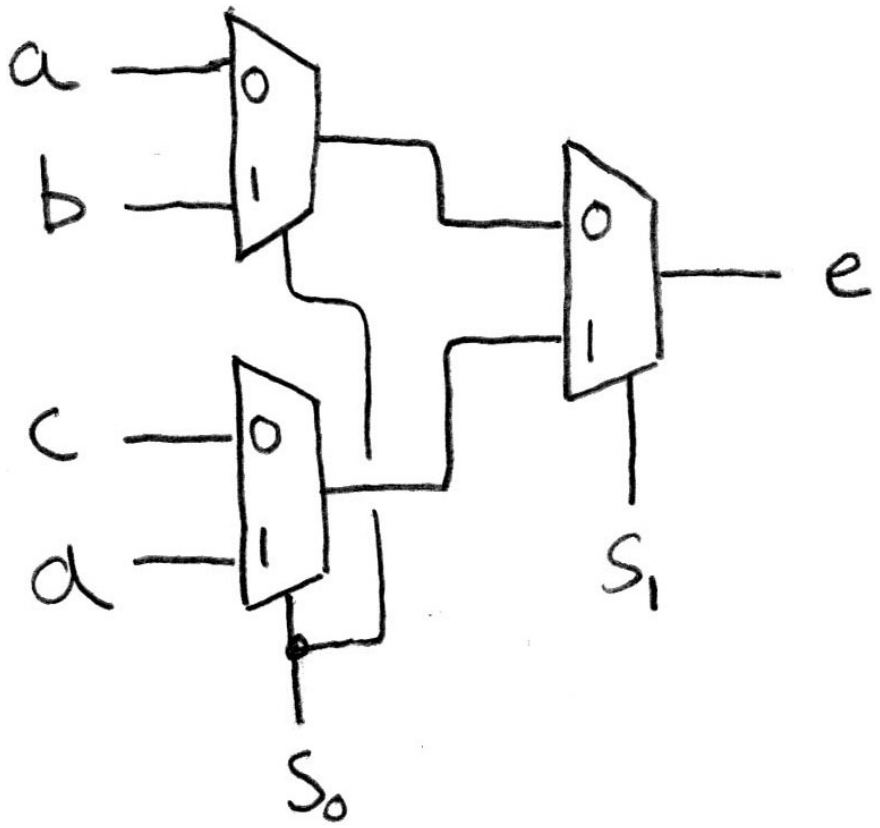
4-to-1 Multiplexer



How many rows in TT?

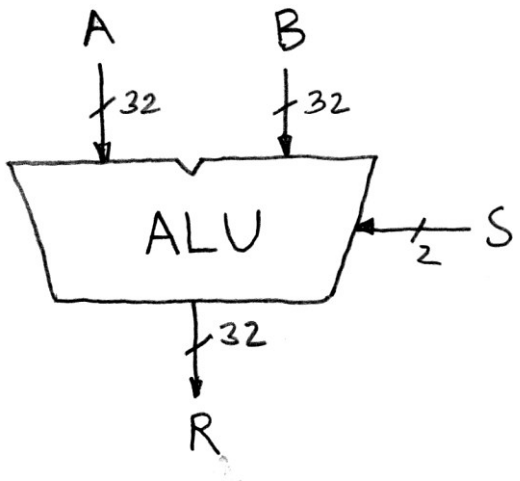
$$e = \overline{s_1}\,\overline{s_0}a + \overline{s_1}s_0b + s_1\overline{s_0}c + s_1s_0d$$

Alternative Hierarchical Approach (in Logisim)



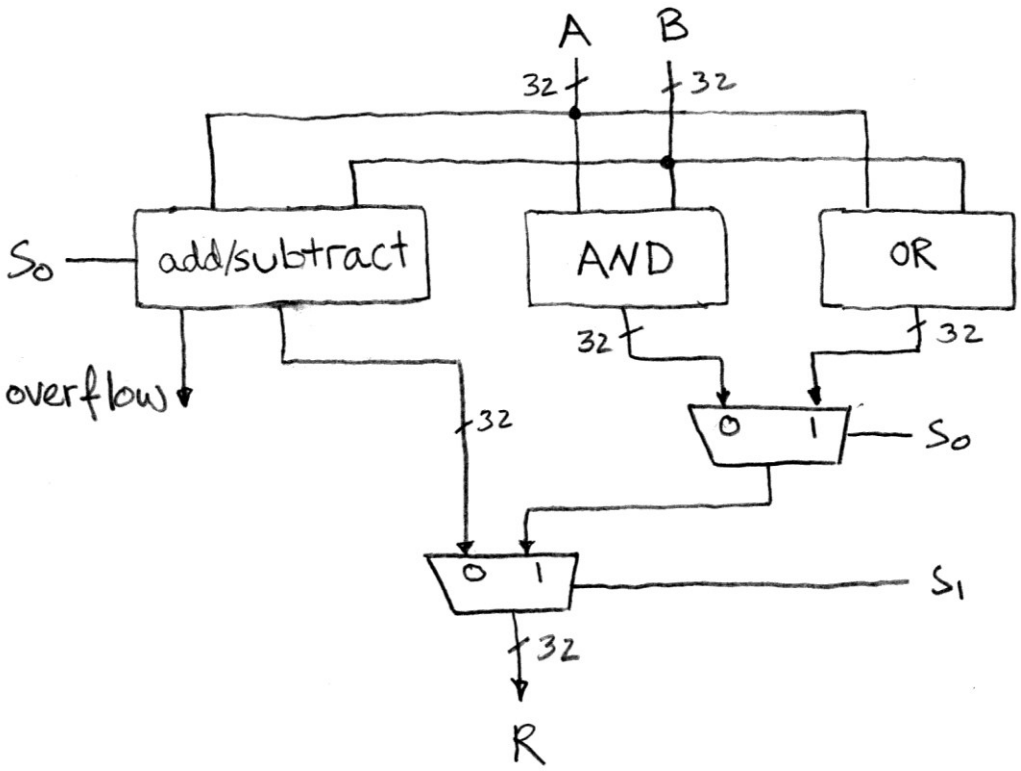
Arithmetic and Logic Unit

- Most processors contain a special logic block called “Arithmetic and Logic Unit” (ALU)
- We’ll show you an easy one that does ADD, SUB, bitwise AND, bitwise OR



when S=00, R=A+B
when S=01, R=A-B
when S=10, R=A AND B
when S=11, R=A OR B

Simple ALU



Adder/Subtractor: One-bit adder Least Significant Bit

	a_3	a_2	a_1	a_0
+	b_3	b_2	b_1	b_0
	s_3	s_2	s_1	s_0

a_0	b_0	s_0	c_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$s_0 = a_0 \text{ XOR } b_0$$
$$c_1 = a_0 \text{ AND } b_0$$

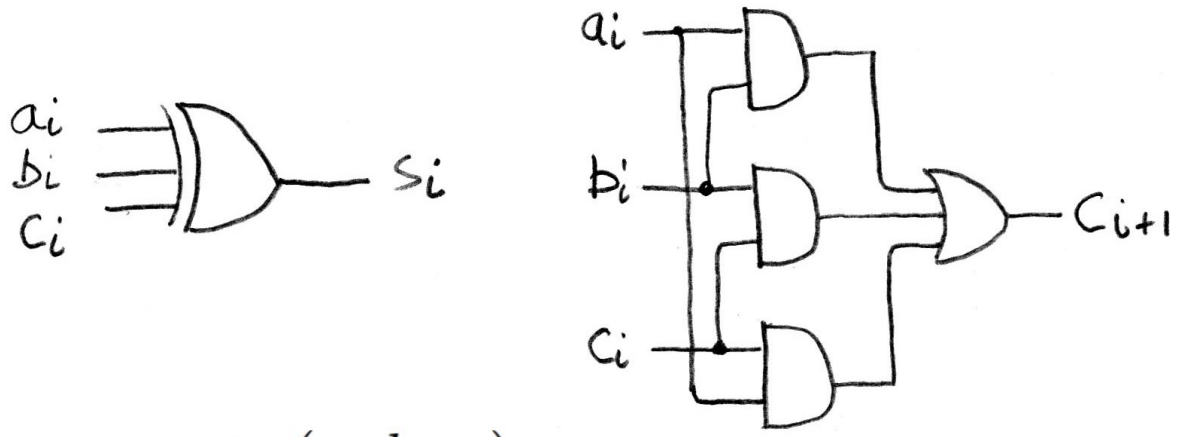
Adder/Subtractor: One-bit adder (1/2) …

$$
\begin{array}{cccc|c|c}
 & a_3 & a_2 & a_1 & a_0 \\
+ & b_3 & b_2 & b_1 & b_0 \\
\hline
 & s_3 & s_2 & s_1 & s_0
\end{array}
$$

a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$
\begin{aligned}
s_i &= \mathrm{XOR}(a_i, b_i, c_i) \\
c_{i+1} &=
\end{aligned}
$$

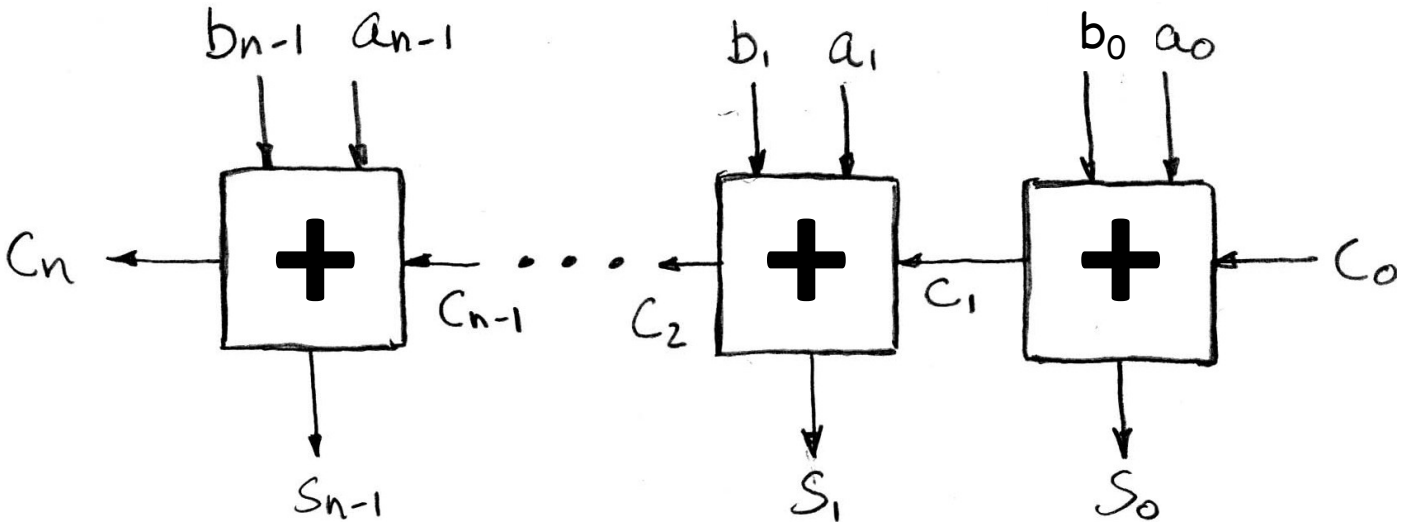
Adder/Subtractor: One-bit Adder (2/2) …



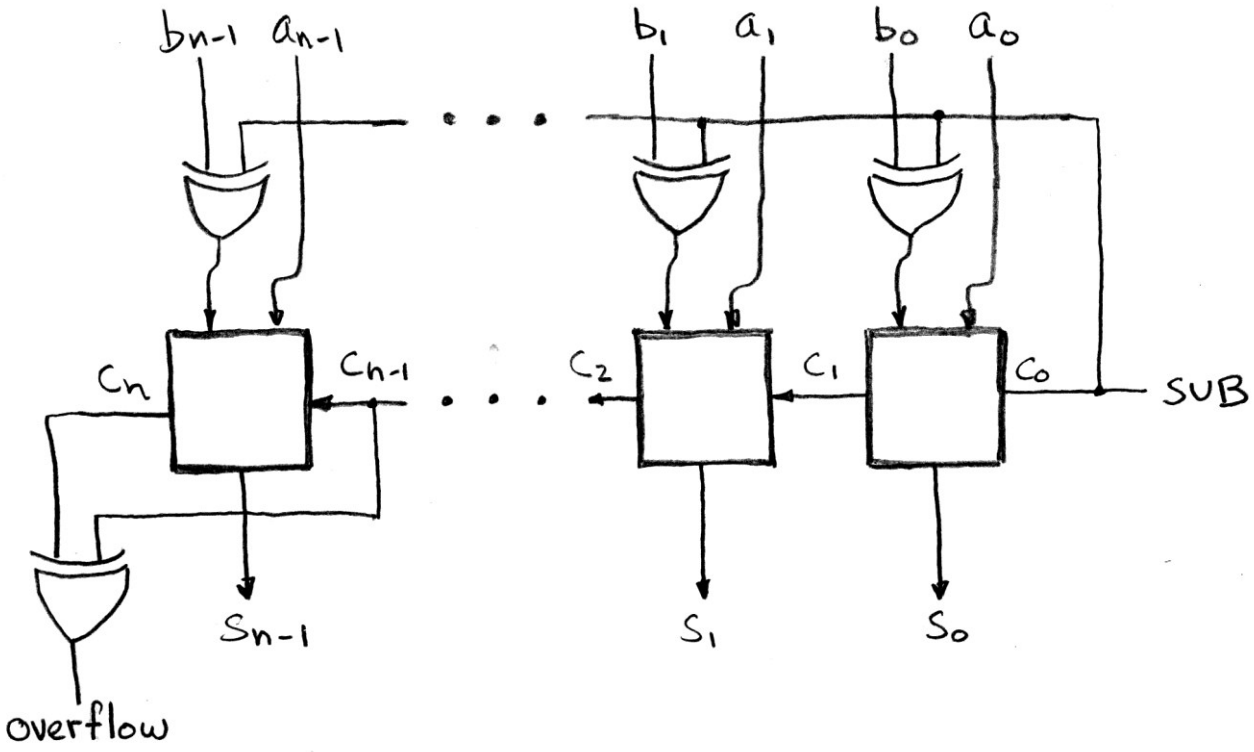
$$s_i = \mathbf{XOR}(a_i, b_i, c_i)$$
$$c_{i+1} = \mathbf{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i$$

N x 1-bit Adders ⇒ 1 N-bit Adder

Connect Carry Out i-1 to Carry in i:



Twos Complement Adder/Subtractor



Critical Path

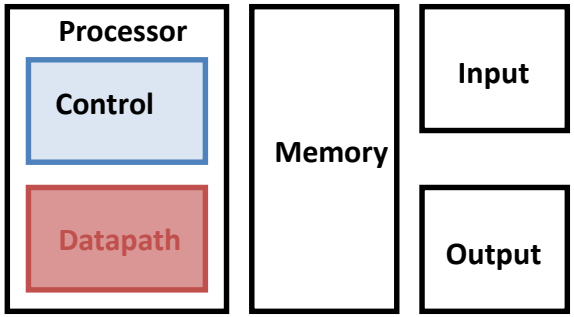
- When setting clock period in synchronous systems, must allow for worst case
- Path through combinational logic that is worst case called "critical path"
  - Can be estimated by number of "gate delays": Number of gates must go through in worst case
- Idea: Doesn't matter if speedup other paths if don't improve the critical path
- What might critical path of ALU?

Hardware Implementation

- Boolean Algebra
- Timing and State Machines
- Datapath Elements: Mux + ALU
- RISC-V Lite Datapath
- And, in Conclusion, …

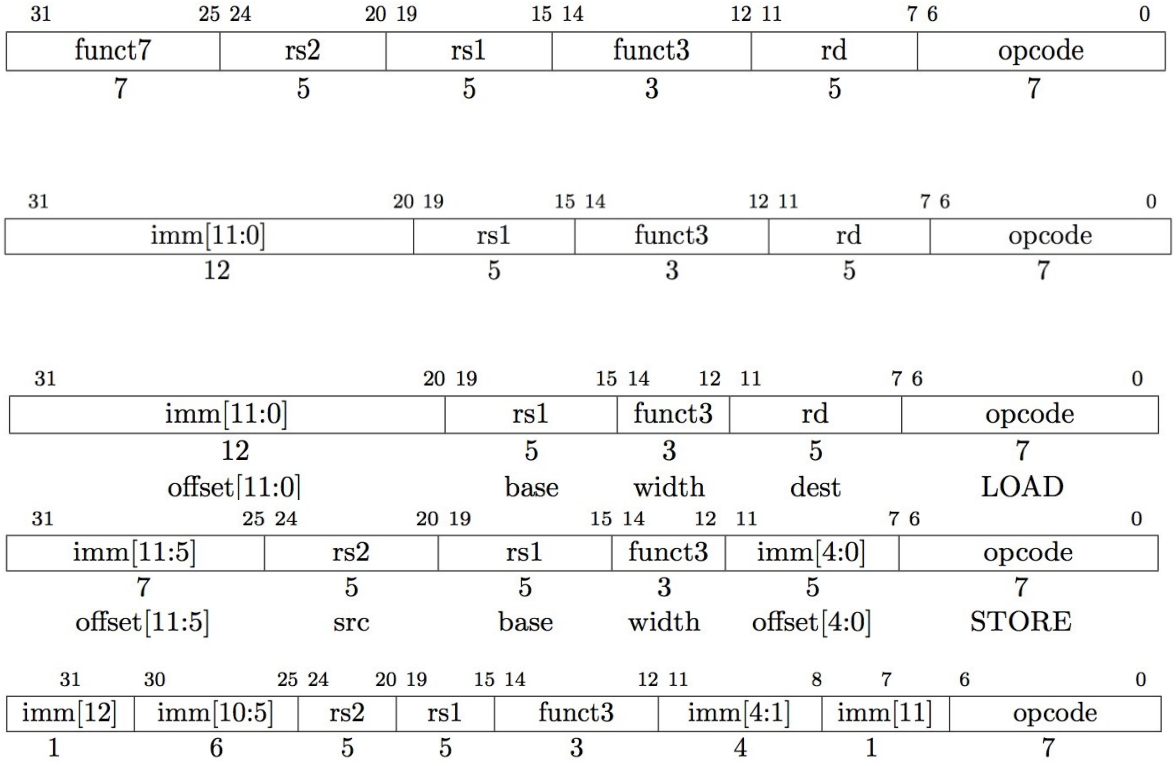
Processor Design Process

- 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



The RISC-V Lite Subset

- ADD and SUB
  - `add rd,rs1,rs2`
  - `sub rd,rs1,rs2`
- OR Immediate:
  - `ori rd,rs1,imm12`
- LOAD and STORE Word
  - `lw rd,rs1,imm12`
  - `sw rs2,rs1,imm12`
- BRANCH:
  - `beq rs1,rs2,imm12`



Register Transfer Language (RTL)

- RTL gives the meaning of the instructions

  {op , rs1 , rs2 , rd , funct3} ← MEM[PC]

  {op , rs1 , rs2 , Imm12} ← MEM[PC]

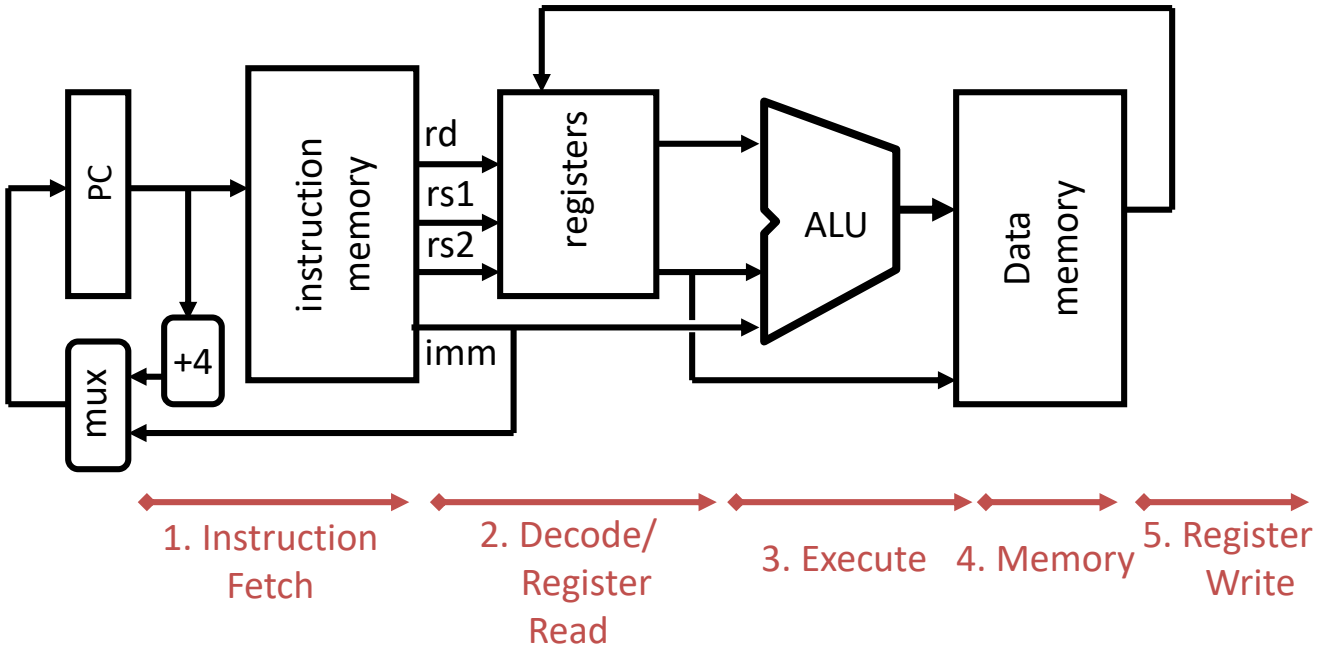
- All start by fetching the instruction

Inst	Register Transfers
ADD	R[rd] ← R[rs1] + R[rs2]; PC ← PC + 4
SUB	R[rd] ← R[rs1] – R[rs2]; PC ← PC + 4
ORI	R[rd] ← R[rs1] | sign_ext(Imm12); PC ← PC + 4
LW	R[rd] ← MEM[R[rs1] + sign_ext(Imm12)]; PC ← PC + 4
SW	MEM[R[rs1] + sign_ext(Imm12)] ← R[rs2]; PC ← PC + 4
BEQ	if (R[rs] == R[rt]) then PC ← PC + (sign_ext(Imm12) || 0) else PC ← PC + 4

Step 1: Requirements of the Instruction Set

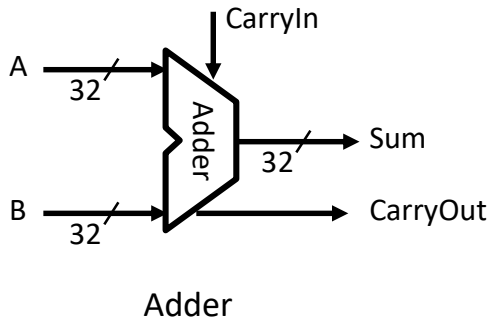
- Memory (MEM)
  - Instructions & data (will use one for each: really caches)
- Registers (R: 32 x 32)
  - Read *rs1*
  - Read *rs2*
  - Write *rd*
- PC
- Sign Extender
- Add/Sub/OR unit for operation on register(s) or sign extended immediate
- Add 4 (or maybe sign extended immediate) to PC
- Compare if registers equal?

Generic Steps of Datapath

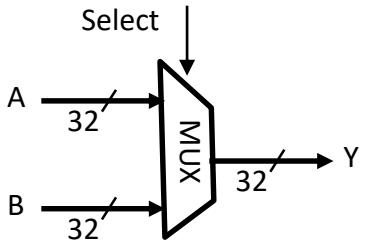


Step 2: Components of the Datapath

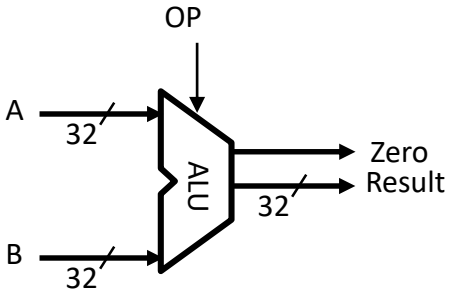
- Combinational Elements
- State Elements + Clocking Methodology
- Building Blocks



Adder



Multiplexer



ALU

ALU Needs for RISC-V Lite + Rest of RISC-V

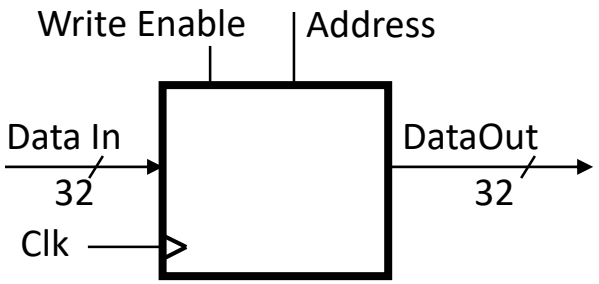
- Addition, subtraction, logical OR, ==:

```
ADD   R[rd] = R[rs1] + R[rs2]; ...
SUB   R[rd] = R[rs1] - R[rs2]; ...
ORI   R[rt] = R[rs1] | sign_ext(Imm12)...
BEQ   if ( R[rs] == R[rt] )...
```

- Test to see if output == 0 for any ALU operation gives == test. How?
- P&H Ch. 4 also adds AND, 64 bit LD/SD instructions
- ALU from Appendix A, Section A.5

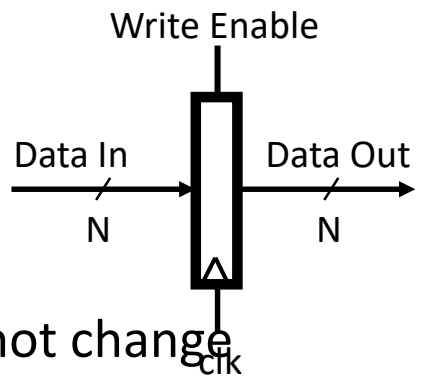
Storage Element: Idealized Memory

- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is found by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block: Address valid $\Rightarrow$ Data Out valid after "access time"



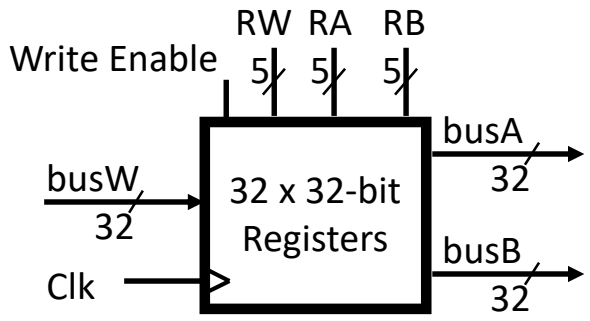
Storage Element: Register (Building Block)

- Similar to D Flip Flop except
  - N-bit input and output
  - Write Enable input
- Write Enable:
  - Negated (or deasserted) (0): Data Out will not change
  - Asserted (1): Data Out will become Data In on rising edge of clock



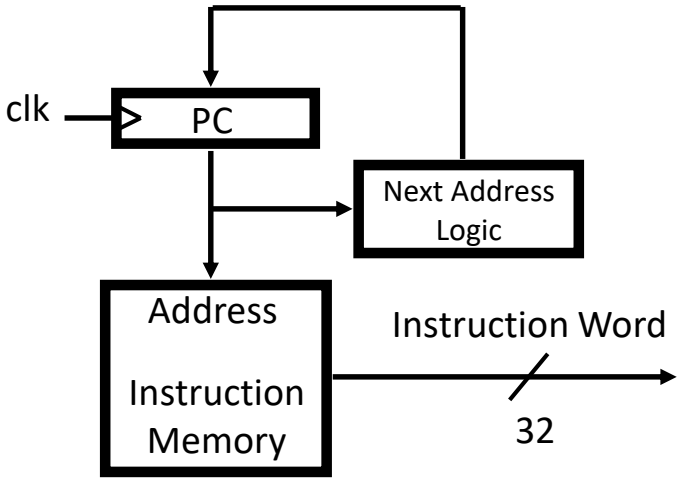
Storage Element: Register File

- Register File consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (clk)
  - Clk input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid ⇒ busA or busB valid after "access time."



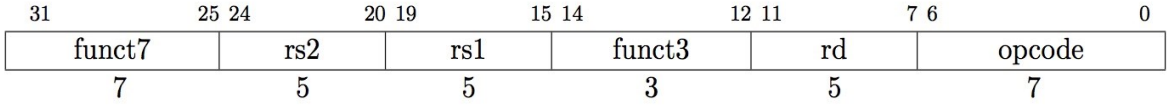
Step 3: Assemble DataPath Meeting Requirements

- Register Transfer Requirements ⇒ Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation
- Common RTL operations
  - Fetch the Instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC ← PC + 4
    - Branch and Jump: PC ← "something else"

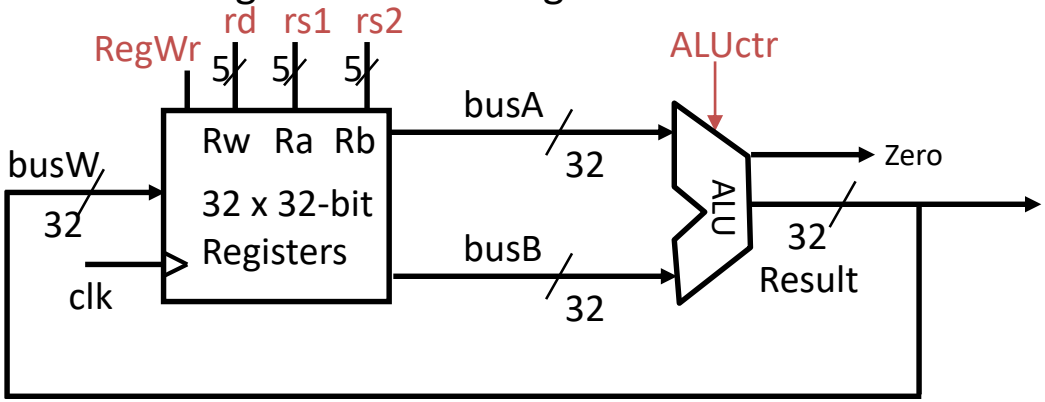


Step 3: Add & Subtract

- `R[rd] = R[rs] op R[rt] (addu rd,rs,rt)`
  - Ra, Rb, and Rw come from instruction's Rs1, Rs2, and Rd fields



  - ALUctr and RegWr: control logic after decoding the instruction



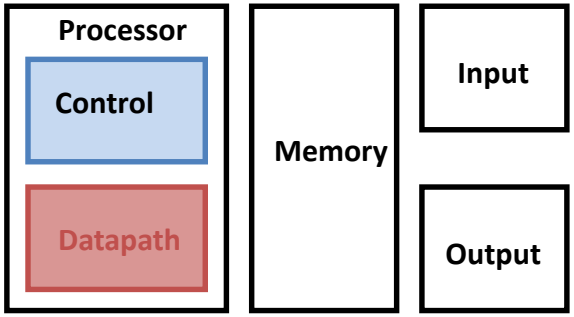
- … Already defined the register file & ALU

Hardware Implementation

- Boolean Algebra
- Timing and State Machines
- Datapath Elements: Mux + ALU
- RISC-V Lite Datapath
- And, in Conclusion, …

And in Conclusion, …

- State Machines
  - Finite State Machines: made from *Stateless* combinational logic and *Stateful* "Memory" Logic (aka Registers)
  - Clocks synchronize D-FF change (Setup and Hold times important!)
  - Pipeline long-delay CL for faster clock cycle— Split the *critical path*
- Use muxes to select among inputs
  - S input bits selects 2^S inputs
  - Each input can be n-bits wide, independent of S
- Can implement muxes hierarchically
- Arithmetic circuits are a kind of combinational logic



- Five steps to processor design:
  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

Instruction Set Architecture (ISA)

- The contract between software and hardware
- Typically described by giving all the programmer-visible state (registers + memory) plus the semantics of the instructions that operate on that state
- IBM 360 was first line of machines to separate ISA from implementation (aka. *microarchitecture*)
- Many implementations possible for a given ISA
  - E.g., Soviets built code-compatible clones of the IBM360, as did Amdahl after he left IBM.
  - E.g.2., AMD, Intel, VIA processors run the AMD64 ISA
  - E.g.3: many cellphones use the ARM ISA with implementations from many different companies including Apple, Qualcomm, Samsung, Huawei, etc.
- We use RISC-V as standard ISA in class (www.riscv.org)
  - Many companies and open-source projects build RISC-V implementations

ISA to Microarchitecture Mapping

- ISA often designed with particular microarchitectural style in mind, e.g.,

  Accumulator $\Rightarrow$ hardwired, unpipelined
  CISC $\Rightarrow$ microcoded
  RISC $\Rightarrow$ hardwired, pipelined
  VLIW $\Rightarrow$ fixed-latency in-order parallel pipelines
  JVM $\Rightarrow$ software interpretation

- But can be implemented with any microarchitectural style
  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Apple M1 (native ARM ISA, emulates x86 in software)
  - Spike: Software-interpreted RISC-V machine
  - ARM Jazelle: A hardware JVM processor
  - **This lecture: a microcoded RISC-V machine**

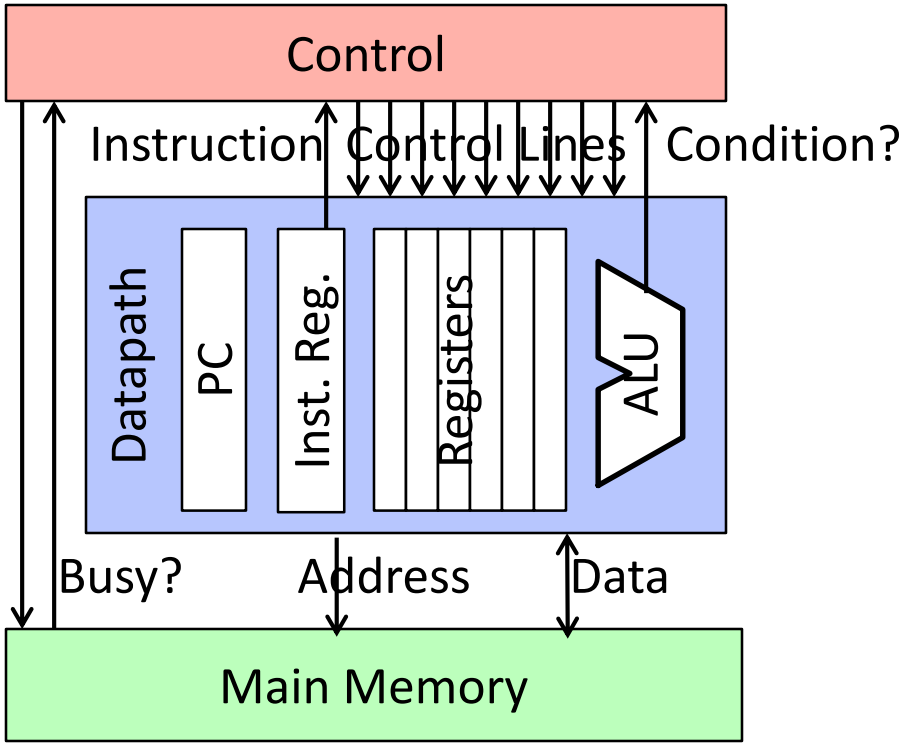
Why Learn Microprogramming?

- To show how to build very small processors with complex ISAs
- To help you understand where CISC* machines came from
- Because still used in common machines (x86, IBM360, PowerPC)
- As a gentle introduction into machine structures
- To help understand how technology drove the move to RISC*

** "CISC"/"RISC" names much newer than style of machines they refer to.*

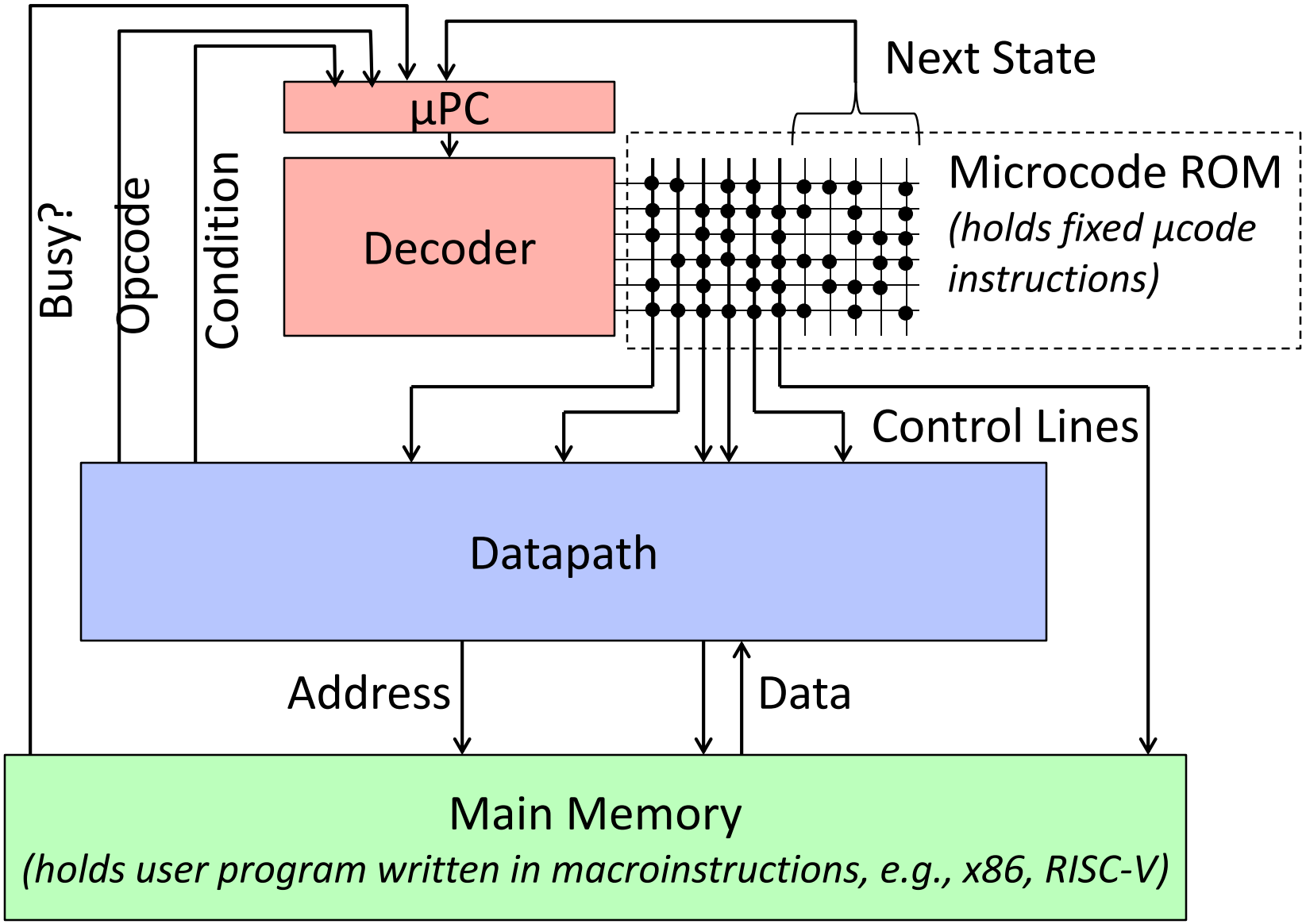
Control versus Datapath

- Processor designs can be split between *datapath*, where numbers are stored and arithmetic operations computed, and *control*, which sequences operations on datapath



- Biggest challenge for early computer designers was getting control circuitry correct
- Maurice Wilkes invented the idea of microprogramming to design the control unit of a processor for EDSAC-II, 1958
  - Foreshadowed by Babbage's "Barrel" and mechanisms in earlier programmable calculators

Microcoded CPU



Technology Influence

- When microcode appeared in 1950s, different technologies for:
  - Logic: Vacuum Tubes
  - Main Memory: Magnetic cores
  - Read-Only Memory: Diode matrix, punched metal cards, …
- Logic very expensive compared to ROM or RAM
- ROM cheaper than RAM
- ROM much faster than RAM

RISC-V ISA

- New fifth-generation RISC design from UC Berkeley
- Realistic & complete ISA, but open & small
- Not over-architected for a certain implementation style
- Both 32-bit (RV32) and 64-bit (RV64) address-space variants
- Designed for multiprocessing
- Efficient instruction encoding
- Easy to subset/extend for education/research
- RISC-V spec available on Foundation website and github
- Increasing momentum with industry adoption

- Please see CS61C Fall 2017, Lectures 5-7 for RISC-V ISA review:
  `http://inst.eecs.berkeley.edu/~cs61c/fa17/`

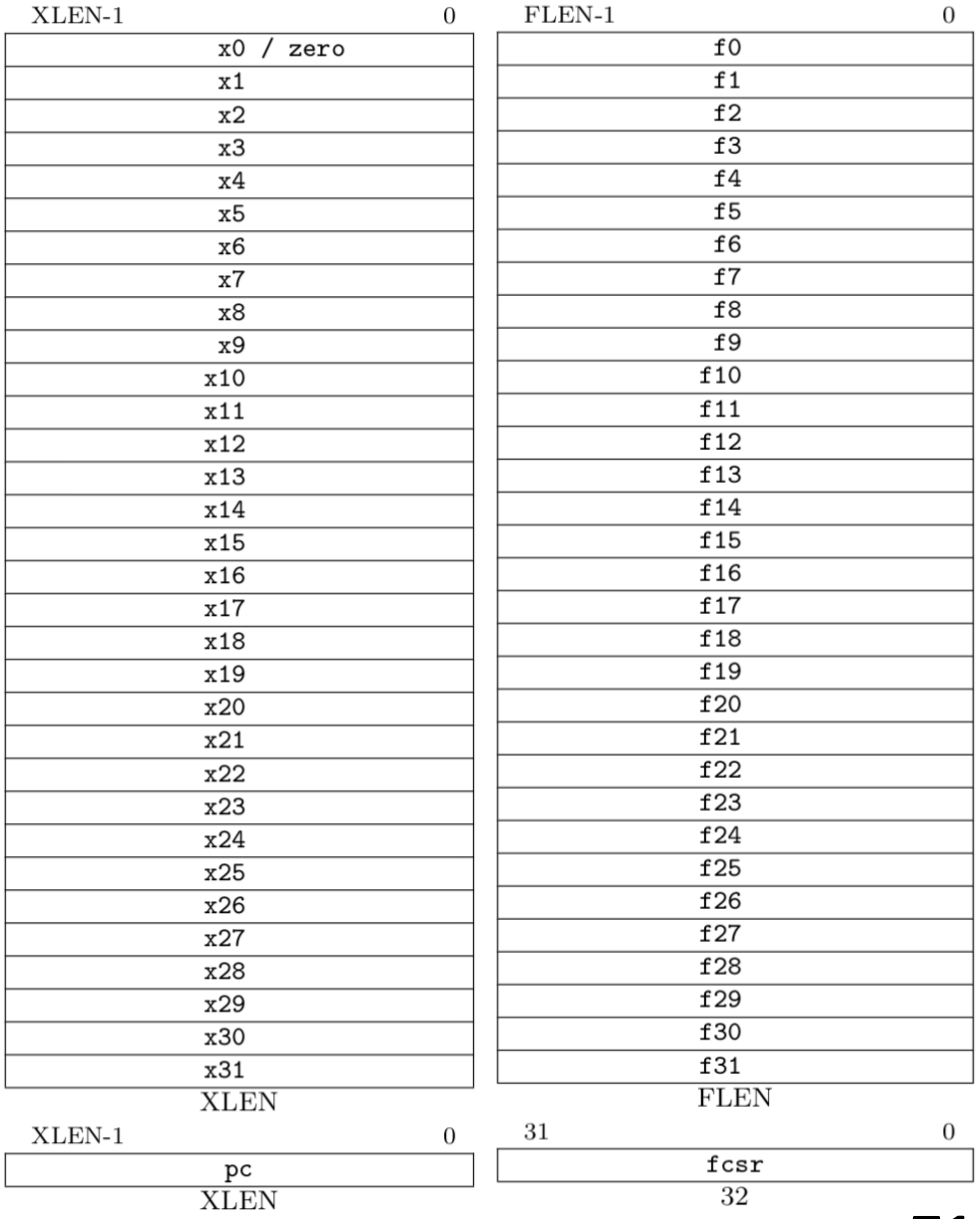
RV32I Processor State

Program counter (**pc**)

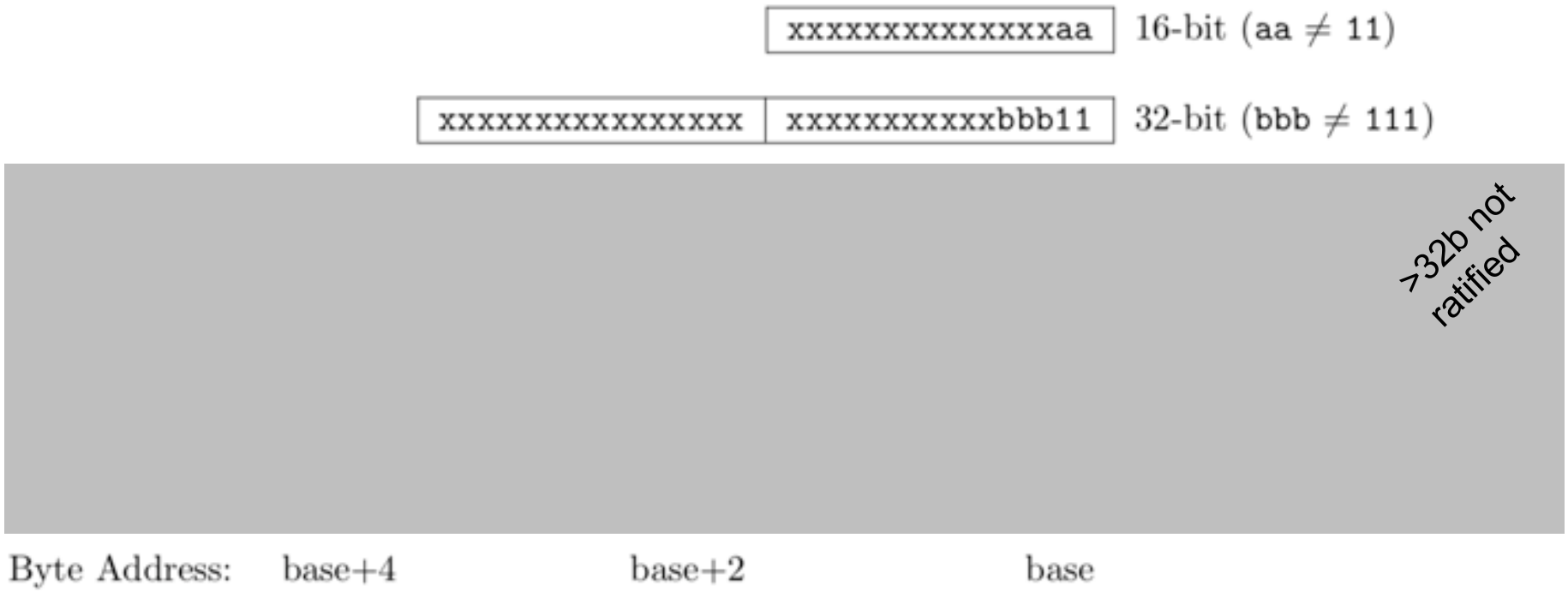
32x32-bit integer registers (**x0-x31**)
• **x0** always contains a 0

32 floating-point (FP) registers (**f0-f31**)
• each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)

FP status register (**fcsr**), used for FP rounding mode & exception reporting

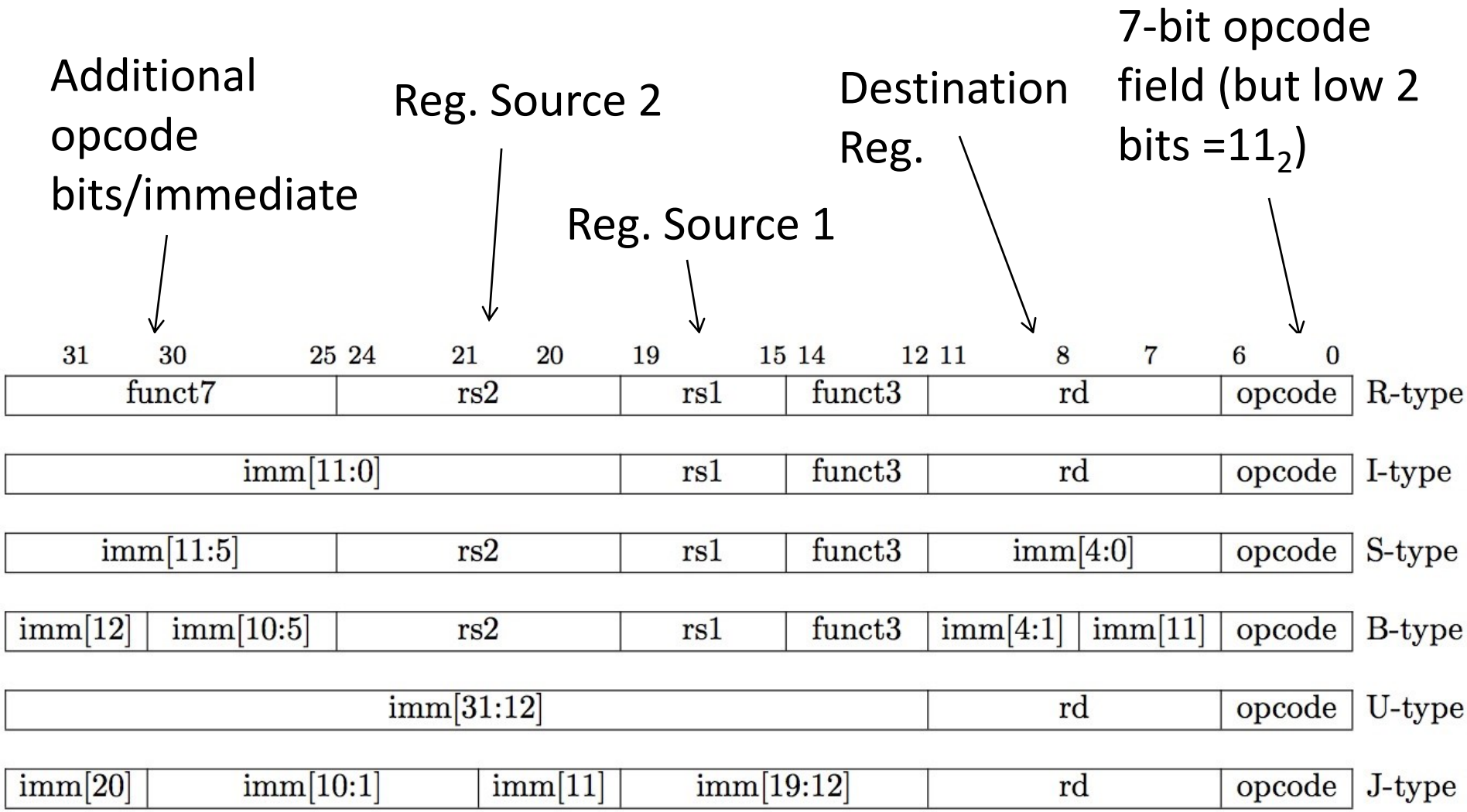


RISC-V Instruction Encoding

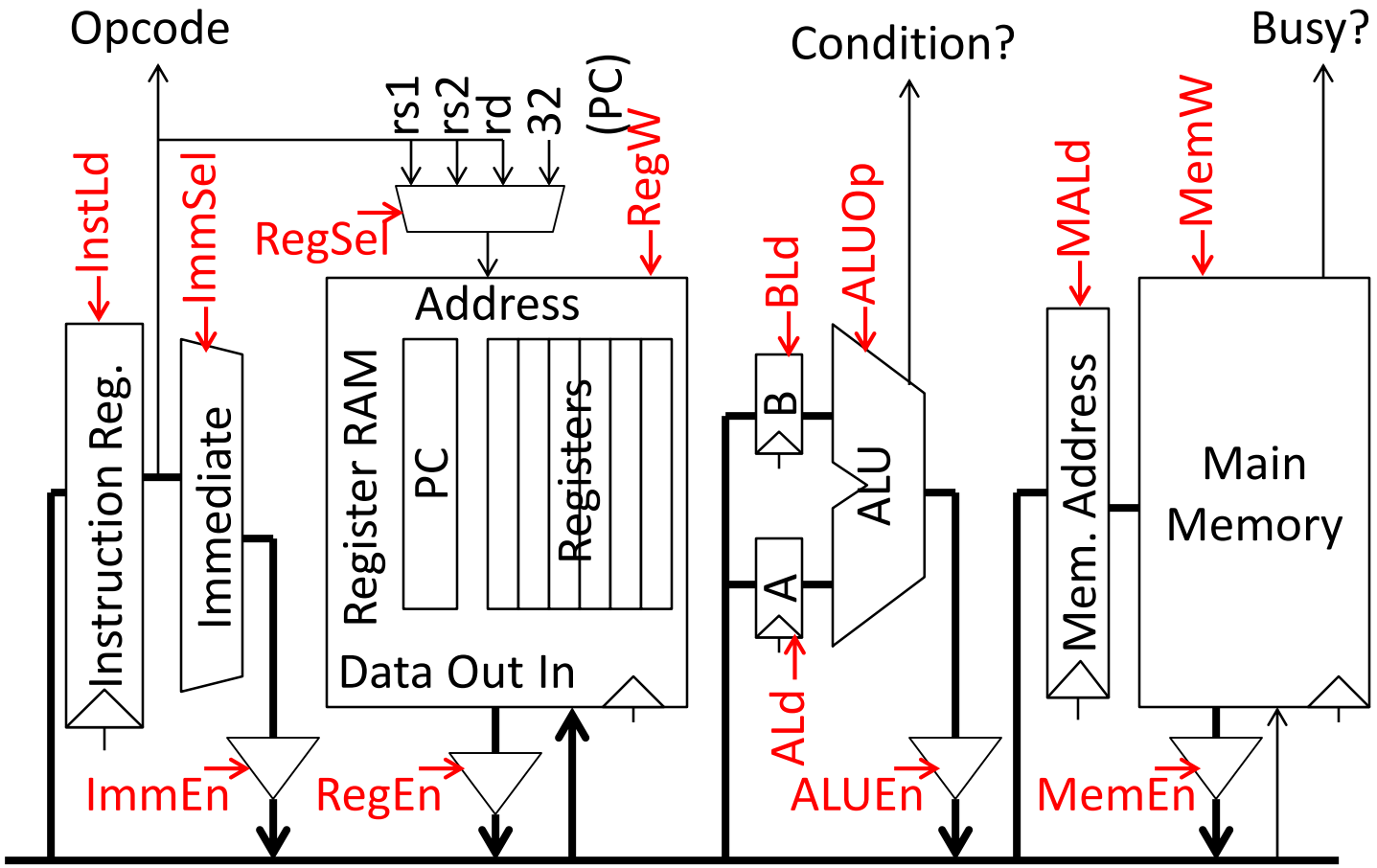


- Can support variable-length instructions.
- Base instruction set (RV32) always has fixed 32-bit instructions lowest two bits = 11_2
- All branches and jumps have targets at 16-bit granularity (even in base ISA where all instructions are fixed 32 bits)

RISC-V Instruction Formats



Single-Bus Datapath for Microcoded RISC-V



Microinstructions written as register transfers:

- MA:=PC means RegSel=PC; RegW=0; RegEn=1; MALd=1
- B:=Reg[rs2] means RegSel=rs2; RegW=0; RegEn=1; BLd=1
- Reg[rd]:=A+B means ALUop=Add; ALUEn=1; RegSel=rd; RegW=1

RISC-V Instruction Execution Phases

- Instruction Fetch
- Instruction Decode
- Register Fetch
- ALU Operations
- *Optional* Memory Operations
- *Optional* Register Writeback
- Calculate Next Instruction Address

Microcode Sketches (1)

Instruction Fetch:

MA,A:=PC
PC:=A+4
wait for memory
IR:=Mem
dispatch on opcode

ALU:

A:=Reg[rs1]
B:=Reg[rs2]
Reg[rd]:=ALUOp(A,B)
goto instruction fetch

ALUI:

A:=Reg[rs1]
B:=ImmI //Sign-extend 12b immediate
Reg[rd]:=ALUOp(A,B)
goto instruction fetch

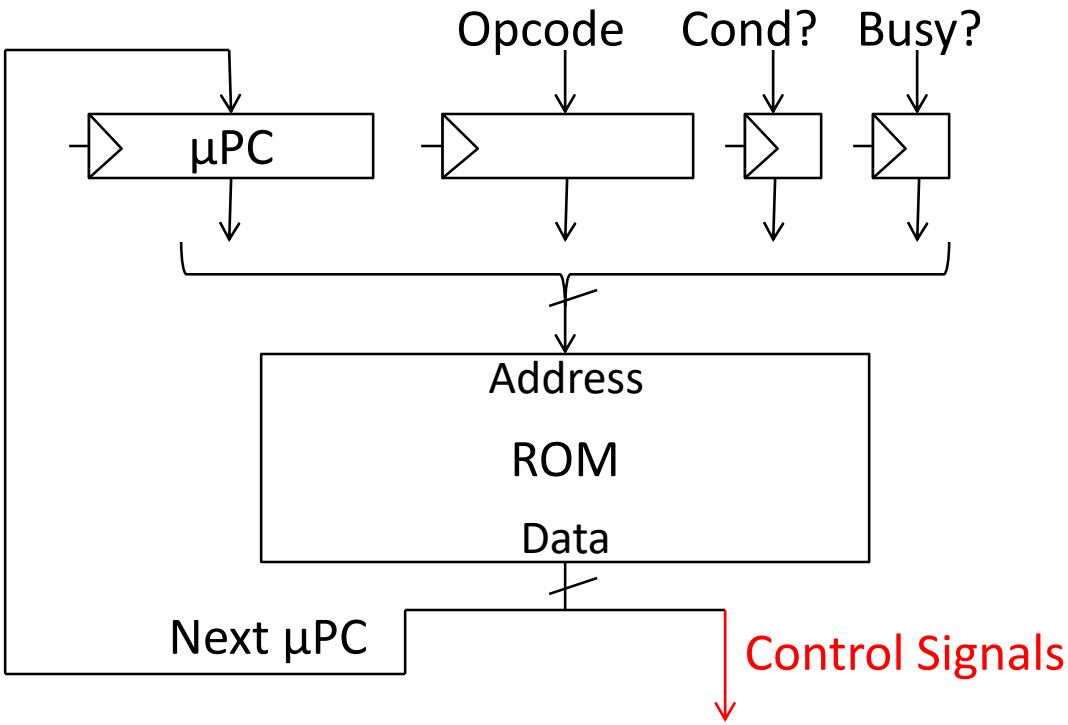
Microcode Sketches (2)

LW:
A:=Reg[rs1]
B:=ImmI //Sign-extend 12b immediate
MA:=A+B
wait for memory
Reg[rd]:=Mem
goto instruction fetch

JAL:
Reg[rd]:=A // Store return address
A:=A-4 // Recover original PC
B:=ImmJ // Jump-style immediate
PC:=A+B
goto instruction fetch

Branch:
A:=Reg[rs1]
B:=Reg[rs2]
if (!ALUOp(A,B)) *goto instruction fetch* //Not taken
A:=PC //Microcode fall through if branch taken
A:=A-4
B:=ImmB// Branch-style immediate
PC:=A+B
goto instruction fetch

Pure ROM Implementation



- How many address bits?

  |µaddress| = |µPC|+|opcode|+ 1 + 1

- How many data bits?

  |data| = |µPC|+|control signals| = |µPC| + 18

- Total ROM size = $2^{|\mu address|} x |data|$

Pure ROM Contents

	Address			Data	
µPC	Opcode	Cond?	Busy?	Control Lines	Next µPC
fetch0	X	X	X	MA,A:=PC	fetch1
fetch1	X	X	1		fetch1
fetch1	X	X	0	IR:=Mem	fetch2
fetch2	ALU	X	X	PC:=A+4	ALU0
fetch2	ALUI	X	X	PC:=A+4	ALUI0
fetch2	LW	X	X	PC:=A+4	LW0
….					
ALU0	X	X	X	A:=Reg[rs1]	ALU1
ALU1	X	X	X	B:=Reg[rs2]	ALU2
ALU2	X	X	X	Reg[rd]:=ALUOp(A,B)	fetch0

Single-Bus Microcode RISC-V ROM Size

- Instruction fetch sequence 3 common steps
- ~12 instruction groups
- Each group takes ~5 steps (1 for dispatch)
- Total steps 3+12*5 = 63, needs 6 bits for µPC

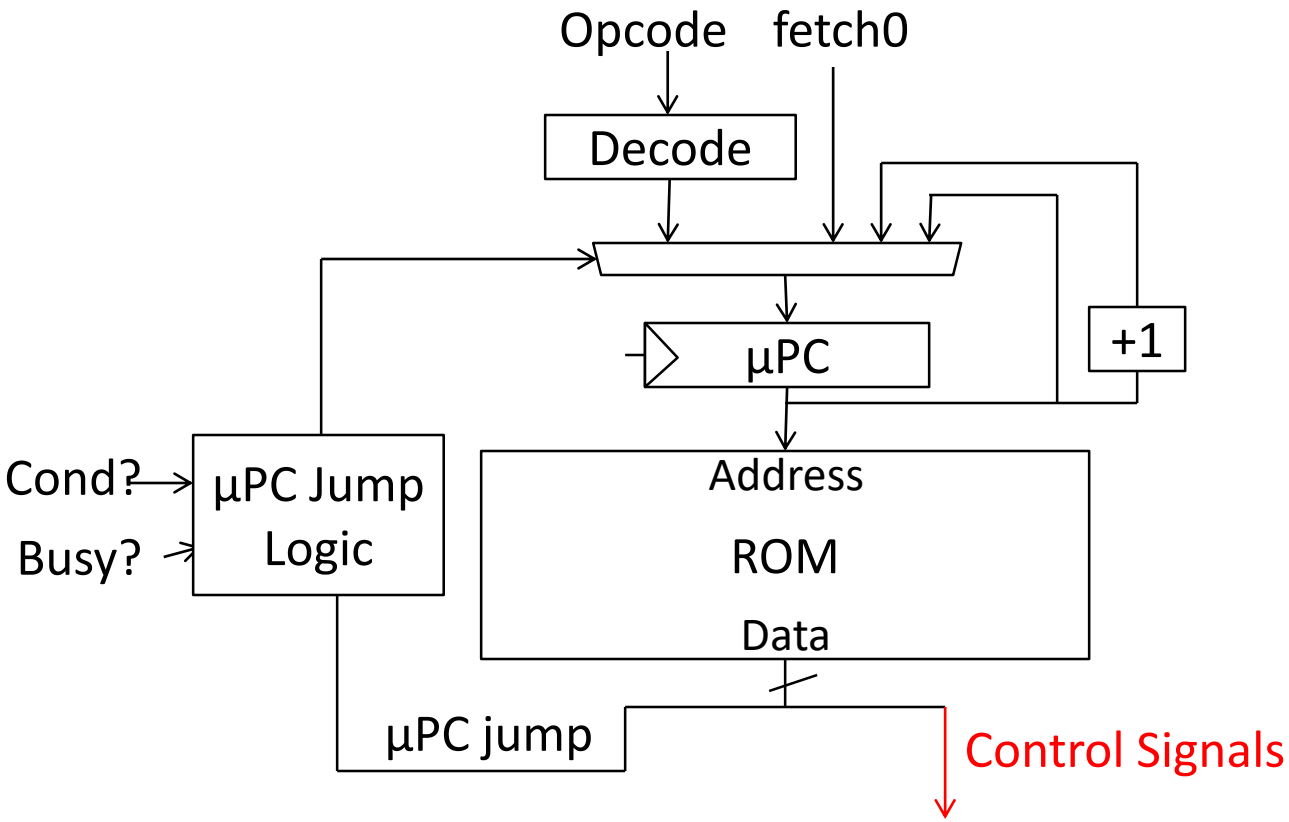
- Opcode is 5 bits, ~18 control signals

- Total size = $2^{(6+5+2)}$x(6+18)=2^{13}x24 = ~25KiB!

Reducing Control Store Size

- Reduce ROM height (#address bits)
  - Use external logic to combine input signals
  - Reduce #states by grouping opcodes
- Reduce ROM width (#data bits)
  - Restrict µPC encoding (next,dispatch,wait on memory,…)
  - Encode control signals (vertical µcoding, nanocoding)

Single-Bus RISC-V Microcode Engine



µPC jump = next | spin | fetch | dispatch | ftrue | ffalse

µPC Jump Types

- *next* increments µPC
- *spin* waits for memory
- *fetch* jumps to start of instruction fetch
- *dispatch* jumps to start of decoded opcode group
- *ftrue/ffalse* jumps to fetch if Cond? true/false

Encoded ROM Contents

Address	Data	
µPC	Control Lines	Next µPC
fetch0	MA,A:=PC	next
fetch1	IR:=Mem	spin
fetch2	PC:=A+4	dispatch
ALU0	A:=Reg[rs1]	next
ALU1	B:=Reg[rs2]	next
ALU2	Reg[rd]:=ALUOp(A,B)	fetch
Branch0	A:=Reg[rs1]	next
Branch1	B:=Reg[rs2]	next
Branch2	A:=PC	ffalse
Branch3	A:=A-4	next
Branch4	B:=ImmB	next
Branch5	PC:=A+B	fetch

Implementing Complex Instructions

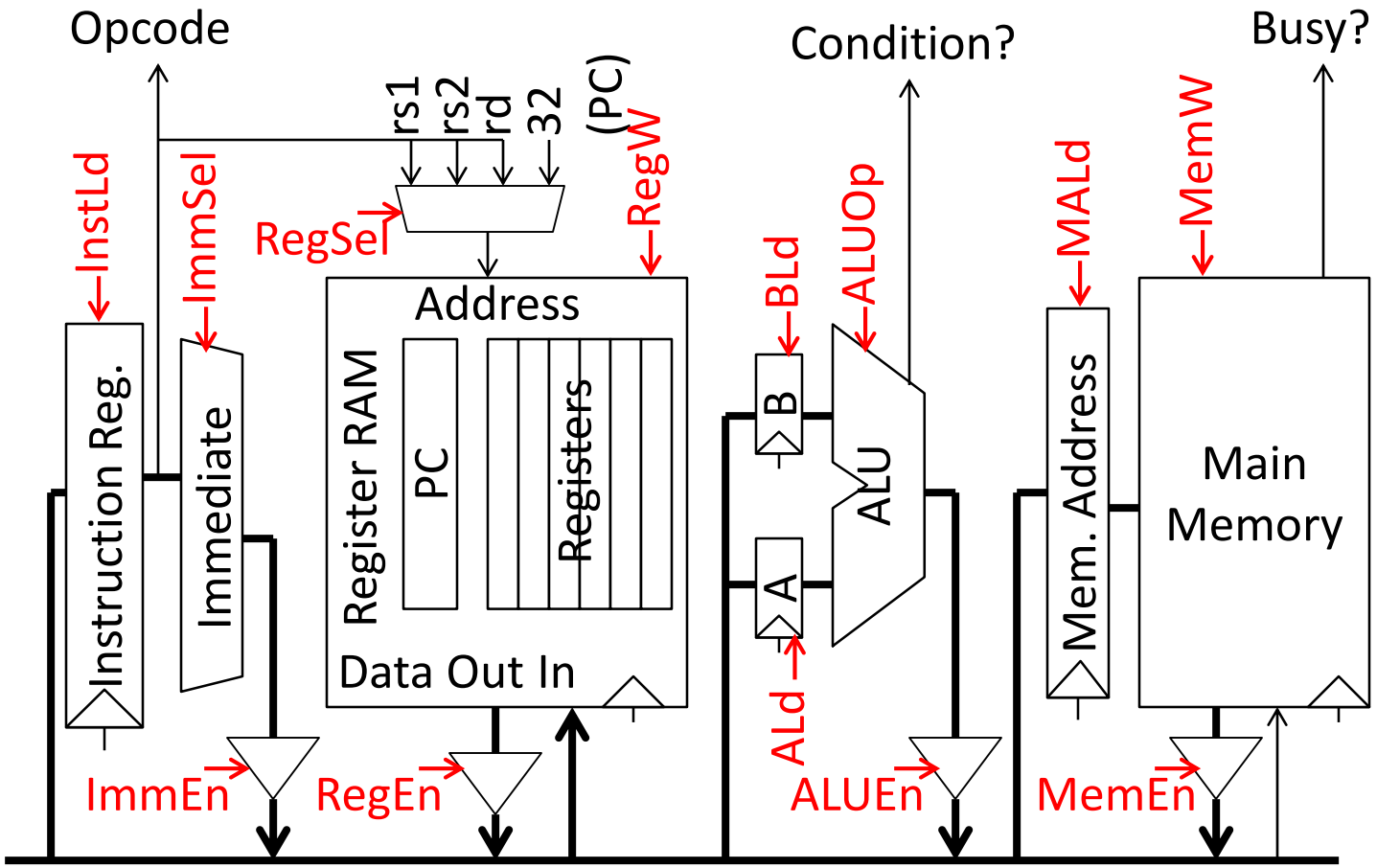
Memory-memory add: M[rd] = M[rs1] + M[rs2]

Address	Data	
µPC	Control Lines	Next µPC
MMA0	MA:=Reg[rs1]	next
MMA1	A:=Mem	spin
MMA2	MA:=Reg[rs2]	next
MMA3	B:=Mem	spin
MMA4	MA:=Reg[rd]	next
MMA5	Mem:=ALUOp(A,B)	spin
MMA6		fetch

Complex instructions usually do not require datapath modifications, only extra space for control program

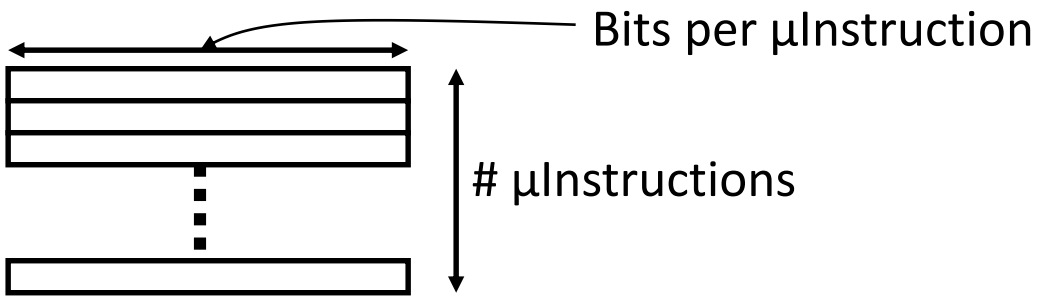
Very difficult to implement these instructions using a hardwired controller without substantial datapath modifications

Single-Bus Datapath for Microcoded RISC-V



Datapath unchanged for complex instructions!

Horizontal vs Vertical µCode

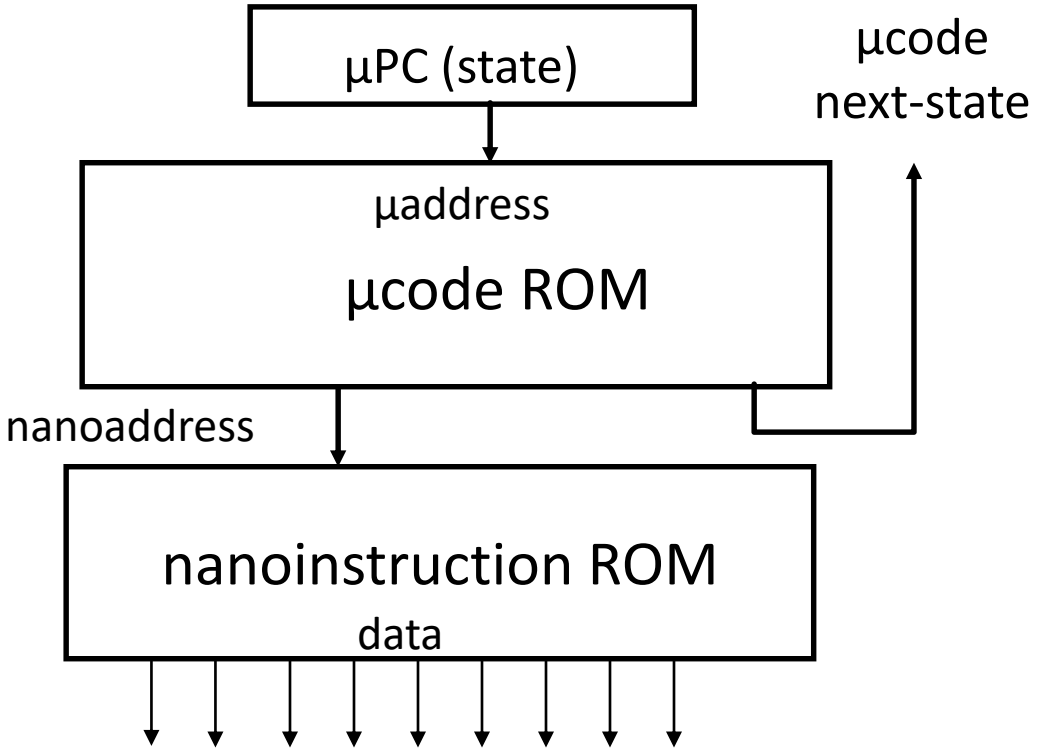


- Horizontal µcode has wider µinstructions
  - Multiple parallel operations per µinstruction
  - Fewer microcode steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical µcode has narrower µinstructions
  - Typically a single datapath operation per µinstruction
    - separate µinstruction for branches
  - More microcode steps per macroinstruction
  - More compact ⇒ less bits
- Nanocoding
  - Tries to combine best of horizontal and vertical µcode

Nanocoding

Exploits recurring control signal patterns in µcode, e.g.,

ALU0 A ← Reg[rs1]
...
ALUI0 A ← Reg[rs1]
...



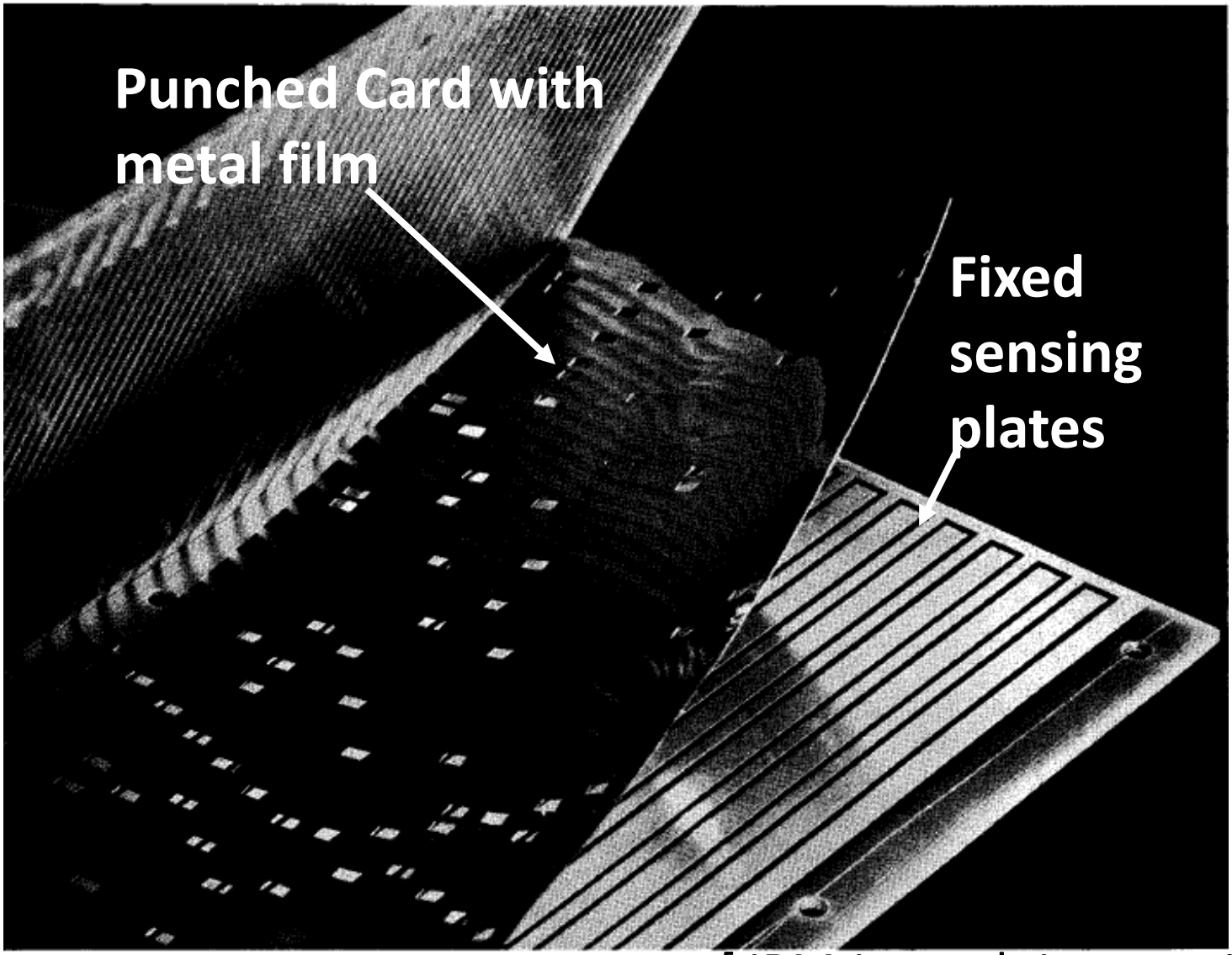
- Motorola 68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

Microprogramming in IBM 360

	M30	M40	M50	M65
Datapath width (bits)	8	16	32	64
µinst width (bits)	50	52	85	87
µcode size (K µinsts)	4	4	2.75	2.75
µstore technology	CCROS	TCROS	BCROS	BCROS
µstore cycle (ns)	750	625	500	200
memory cycle (ns)	1500	2500	2000	750
Rental fee ($K/month)	4	7	15	35

- Only the fastest models (75 and 95) were hardwired

IBM Card-Capacitor Read-Only Storage



[IBM Journal, January 1961]

Microcode Emulation

- IBM initially miscalculated the importance of software compatibility with earlier models when introducing the 360 series
- Honeywell stole some IBM 1401 customers by offering translation software (“Liberator”) for Honeywell H200 series machine
- IBM retaliated with optional additional microcode for 360 series that could emulate IBM 1401 ISA, later extended for IBM 7000 series
  - one popular program on 1401 was a 650 simulator, so some customers ran many 650 programs on emulated 1401s
  - i.e., 650 simulated on 1401 emulated on 360

Microprogramming thrived in ‘60s and ‘70s

- Significantly faster ROMs than DRAMs were available
- For complex instruction sets, datapath and controller were cheaper and simpler
- New instructions , e.g., floating point, could be supported without datapath modifications
- Fixing bugs in the controller was easier
- ISA compatibility across various models could be achieved easily and cheaply

Except for the cheapest and fastest machines, all computers were microprogrammed

Microprogramming: early 1980s

- Evolution bred more complex micro-machines
  - Complex instruction sets led to need for subroutine and call stacks in µcode
  - Need for fixing bugs in control programs was in conflict with read-only nature of µROM
  - →Writable Control Store (WCS) (B1700, QMachine, Intel i432, …)
- With the advent of VLSI technology assumptions about ROM & RAM speed became invalid →more complexity
- Better compilers made complex instructions less important.
- Use of numerous micro-architectural innovations, e.g., pipelining, caches and buffers, made multiple-cycle execution of reg-reg instructions unattractive

VAX 11-780 Microcode

```
; P1WFUD.M [600,1205]     MICRO2  1F(12)    26-May-81  14:58:1     VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122      Page  771
; CALL2 .MIC [600,1205]   Procedure call       : CALLG, CALLS

                                                ;29744  ;HERE FOR CALLG OR CALLS, AFTER PROBING THE EXTENT OF THE STACK
                                                ;29745
                                                ;29746  =0      ;-------------------------------;CALL SITE FOR MPUSH
                                                ;29747  CALL.7: D_Q,AND,RC[T2],                 ;STRIP MASK TO BITS 11-0
6557K    0  U 11F4, 0811,2035,0180,F910,0000,0CD8 ;29748        CALL,J/MPUSH                    ;PUSH REGISTERS
                                                ;29749
                                                ;29750          ;-------------------------------;RETURN FROM MPUSH
                                                ;29751          CACHE_D[LONG],                  ;PUSH PC
6557K 7763K U 11F5, 0000,003C,0180,3270,0000,134A ;29752        LAB_R[SP]                       ; BY SP
                                                ;29753
                                                ;29754          ;-------------------------------;
6856K    0  U 134A, 0018,0000,0180,FAF0,0200,134C ;29755  CALL.8: R[SP]&VA_LA-K[,8]              ;UPDATE SP FOR PUSH OF PC &
                                                ;29756
                                                ;29757          ;-------------------------------;
6856K    0  U 134C, 0800,003C,0180,FA68,0000,11F8 ;29758        D_R[FP]                         ;READY TO PUSH FRAME POINTER
                                                ;29759
                                                ;29760  =0      ;-------------------------------;CALL SITE FOR PSHSP
                                                ;29761          CACHE_D[LONG],                  ;STORE FP,
                                                ;29762          LAB_R[SP],                      ; GET SP AGAIN
                                                ;29763          SC_K[,FFF0],                    ;-16 TO SC
6856K   21M U 11F8, 0000,003D,6D80,3270,0084,6CD9 ;29764        CALL,J/PSHSP
                                                ;29765
                                                ;29766          ;-------------------------------;
                                                ;29767          D_R[AP],                        ;READY TO PUSH AP
6856K    0  U 11F9, 0800,003C,3DF0,2E60,0000,134D ;29768        Q_ID[PSL]                       ; AND GET PSW FOR COMBINATIO
                                                ;29769
                                                ;29770          ;-------------------------------;
                                                ;29771          CACHE_D[LONG],                  ;STORE OLD AP
                                                ;29772          Q_Q,ANDNOT,K[,1F],              ;CLEAR PSW<T,N,Z,V,C>
6856K   21M U 134D, 0019,2024,8DC0,3270,0000,134E ;29773        LAB_R[SP]                       ;GET SP INTO LATCHES AGAIN
                                                ;29774
                                                ;29775          ;-------------------------------;
6856K    0  U 134E, 2010,0038,0180,F909,4200,1350 ;29776        PC&VA_RC[T1], FLUSH,IB          ; LOAD NEW PC AND CLEAR OUT
                                                ;29777
                                                ;29778          ;-------------------------------;
                                                ;29779          D_DAL,SC,                       ;PSW TO D<31:16>
                                                ;29780          Q_RC[T2],                       ;RECOVER MASK
                                                ;29781          SC_SC+K[,3],                    ;PUT -13 IN SC
6856K    0  U 1350, 0D10,0038,0DC0,6114,0084,9351 ;29782        LOAD,IB, PC_PC+1                ;START FETCHING SUBROUTINE I
                                                ;29783
                                                ;29784          ;-------------------------------;
                                                ;29785          D_DAL,SC,                       ;MASK AND PSW IN D<31:03>
                                                ;29786          Q_RC[T4],                       ;GET LOW BITS OF OLD SP TO Q<1:0>
6856K    0  U 1351, 0D10,0038,F5C0,F920,0084,9352 ;29787        SC_SC+K[,A]                     ;PUT -3 IN SC
                                                ;29788
```

Writable Control Store (WCS)

- Implement control store in RAM not ROM
  - MOS SRAM memories now almost as fast as control store (core memories/DRAMs were 2-10x slower)
  - Bug-free microprograms difficult to write
- User-WCS provided as option on several minicomputers
  - Allowed users to change microcode for each processor
- User-WCS failed
  - Little or no programming tools support
  - Difficult to fit software into small space
  - Microcode control tailored to original ISA, less useful for others
  - Large WCS part of processor state - expensive context switches
  - Protection difficult if user can change microcode
  - Virtual memory required restartable microcode

Microprogramming is far from extinct

- Played a crucial role in micros of the Eighties
  - DEC uVAX, Motorola 68K series, Intel 286/386
- Plays an assisting role in most modern micros
  - e.g., AMD Zen, Intel Sky Lake, Intel Atom, IBM PowerPC, …
  - Most instructions executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke microcode
- Patchable microcode common for post-fabrication bug fixes, e.g. Intel processors load µcode patches at bootup
  - Intel had to scramble to resurrect microcode tools and find original microcode engineers to patch Meltdown/Spectre security vulnerabilites

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Krste Asanovic(UCB)
  - Onur Mutlu(ETH Zürich)

- MIT material derived from course 6.823
- UCB material derived from course CS252