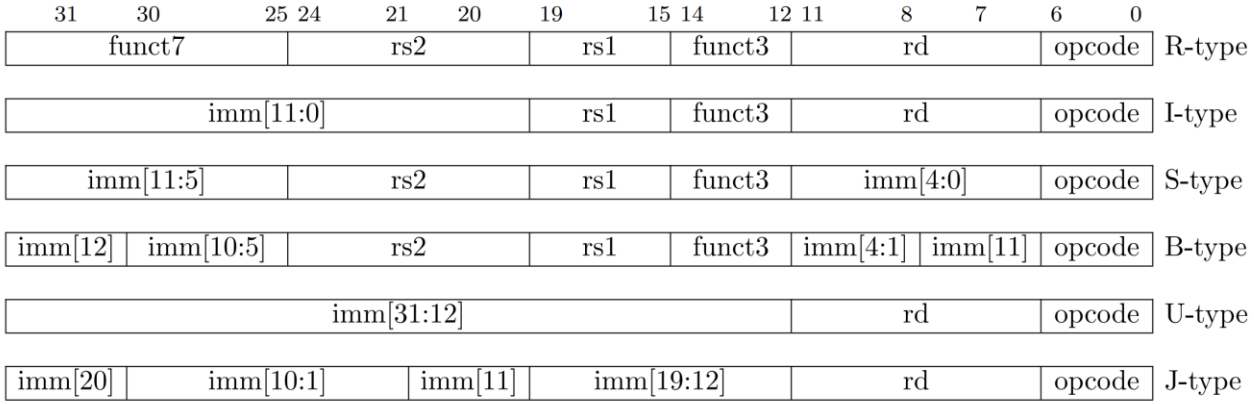
UCLA Samueli School of Engineering

CS-M151B
Computer Systems Architecture

Blaise Tine
UCLA Computer Science

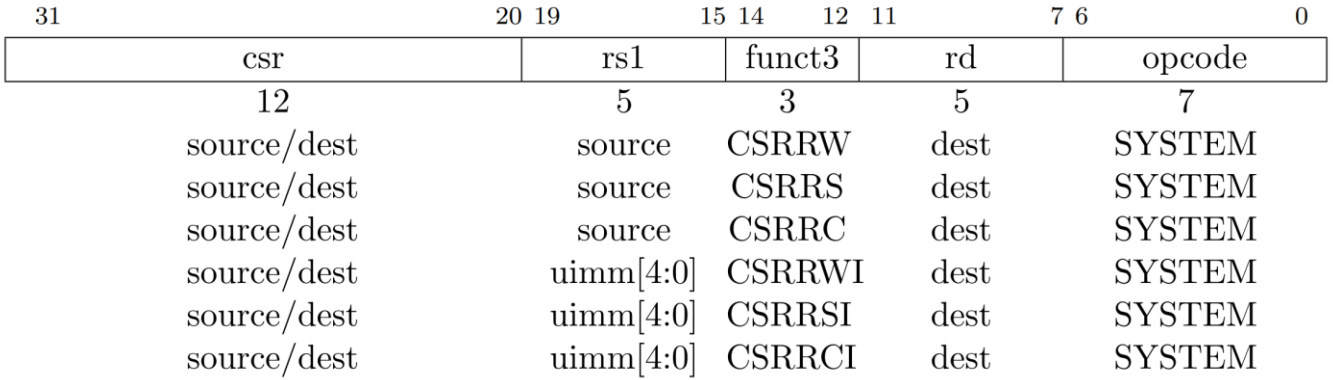
RISC-V Base Instruction Formats

- **Formats classified based on addressing mode and operation type**
  - **R-type: register**
  - **I-type: direct**
  - **S-type: store**
  - **B-type: branch**
  - **U-type: immediate**
  - **J-type: jump**



RISC-V Control and Status Instructions

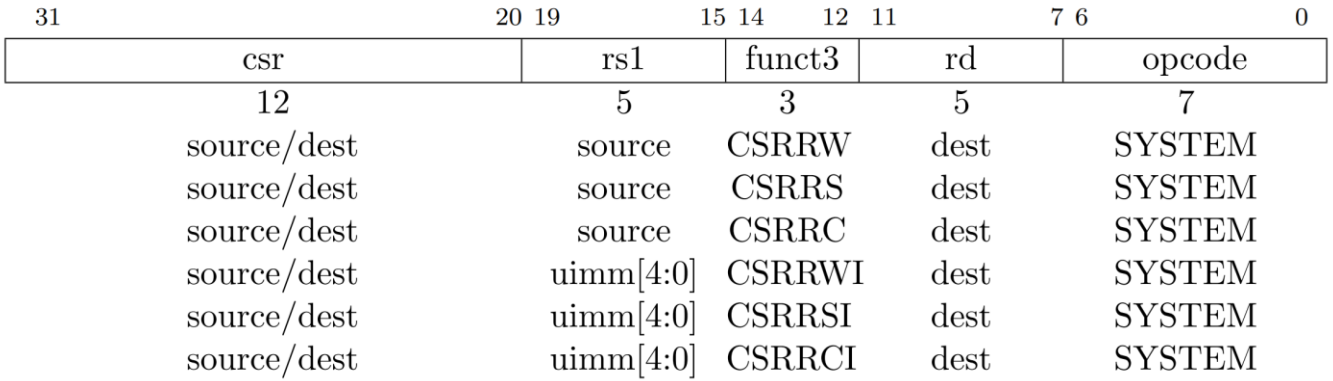
- **Read/Write to CSRs**
  - **#csr = address to 12-bit CSR table**
  - **CSRRW: atomic read/write:** *csr_new = source, dest = csr_old*
  - **CSRRS: atomic read set:** *csr_new = csr_old | source , dest = csr_old*
  - **CSRRC: atomic read & clear:** *csr_new = csr_old & ~source , dest = csr_old*



31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

RISC-V Control and Status Instructions

- **Read/Write to CSRs**
  - **#csr = address to 12-bit CSR table**
  - **CSRRWI: atomic read/write:** *csr_new = uimm, dest = csr_old*
  - **CSRRSI: atomic read set:** *csr_new = csr_old | uimm , dest = csr_old*
  - **CSRRCI: atomic read & clear:** *csr_new = csr_old & ~ uimm , dest = csr_old*



31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

RISC-V Control and Status Instructions

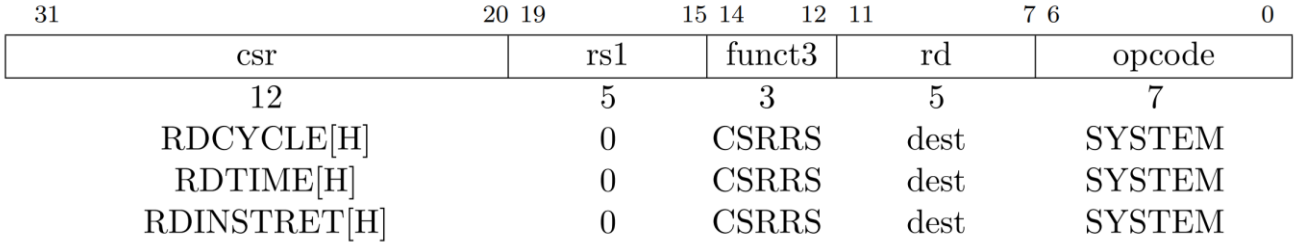
- User-level CSR Table
- Standard mapping
  - FPU flags
  - Cycle
  - Time
  - Instret
  - HPM Counters

Number	Privilege	Name	Description
Unprivileged Floating-Point CSRs			
0x001	URW	fflags	Floating-Point Accrued Exceptions.
0x002	URW	frm	Floating-Point Dynamic Rounding Mode.
0x003	URW	fcsr	Floating-Point Control and Status Register (frm + fflags).
Unprivileged Counter/Timers			
0xC00	URO	cycle	Cycle counter for RDCYCLE instruction.
0xC01	URO	time	Timer for RDTIME instruction.
0xC02	URO	instret	Instructions-retired counter for RDINSTRET instruction.
0xC03	URO	hpmcounter3	Performance-monitoring counter.
0xC04	URO	hpmcounter4	Performance-monitoring counter.
		⋮	
0xC1F	URO	hpmcounter31	Performance-monitoring counter.
0xC80	URO	cycleh	Upper 32 bits of cycle, RV32 only.
0xC81	URO	timeh	Upper 32 bits of time, RV32 only.
0xC82	URO	instreth	Upper 32 bits of instret, RV32 only.
0xC83	URO	hpmcounter3h	Upper 32 bits of hpmcounter3, RV32 only.
0xC84	URO	hpmcounter4h	Upper 32 bits of hpmcounter4, RV32 only.
		⋮	
0xC9F	URO	hpmcounter31h	Upper 32 bits of hpmcounter31, RV32 only.

Currently allocated RISC-V unprivileged CSR addresses.

RISC-V Control and Status Instructions

- Timers and counters
  - 64-bit registers
  - RD_CYCLE: total cycle time
  - RD_TIME: wall-clock time
  - RD_INSTRET: total instructions



31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
RDCYCLE[H]	0	CSRRS	dest	SYSTEM
RDTIME[H]	0	CSRRS	dest	SYSTEM
RDINSTRET[H]	0	CSRRS	dest	SYSTEM

RISC-V Control and Status Instructions

- Timers and counters
  - RD_CYCLE is a 64-bit counter
  - How to read RD_CYCLE on 32-bit machine?
  - **Why could this be challenging?**

RISC-V Control and Status Instructions

- Timers and counters
  - RD_CYCLE is a 64-bit counter
  - How to read RD_CYCLE on 32-bit machine?
  - Why could this be challenging?
    - Counter 32-bit overflow
  - Solution
    - Read Low 32-bit
    - Read high 32-bit

RISC-V Control and Status Instructions

- **Assembly code**
  - csrrs x2, cycle, x0
  - csrrs x3, cycleh, x0
- **What timing issue may emerge from the above code?**
  - ***x2* and *x3* may not match**
  - **Hundreds of CPU cycles can happen between the two calls**
  - **Reading *cycleh* could happen after *cycle* 32-bit overflow**

RISC-V Control and Status Instructions

- **Solution**
  - *loop*: csrrs x3, cycleh, x0
  - csrrs x2, cycle, x0
  - csrrs x4, cycleh, x0
  - bne x3, x4, *loop*
- **How does this work?**
  - It takes a while for cycleh to change
  - The loop handle the overflow cases

RISC-V Control and Status Instructions

- **Privilege Levels:** defines the level of access that software has to the processor and system resources. They enable:
  - **Access control:** control what an application can do
  - **Security:** run an application in isolation from others

RISC-V Control and Status Instructions

- **RIS-V Privilege Levels**
  - **User-level (U)**
  - **Hypervisor (H)**
  - **Supervisor (S)**
  - **Machine-level (M)**

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Hypervisor	H
3	11	Machine	M

RISC-V Control and Status Instructions

- **CSRs Privilege Mapping**

Why reserving dedicated bits for level?

CSR Address [11:10]	[9:8]	[7:4]	Hex	Use and Accessibility
Unprivileged and User-Level CSRs				
00	00	XXXX	0x000–0x0FF	Standard read/write
01	00	XXXX	0x400–0x4FF	Standard read/write
10	00	XXXX	0x800–0x8FF	Custom read/write
11	00	0XXX	0xC00–0xC7F	Standard read-only
11	00	10XX	0xC80–0xCBF	Standard read-only
11	00	11XX	0xCC0–0xCFF	Custom read-only
Supervisor-Level CSRs				
00	01	XXXX	0x100–0x1FF	Standard read/write
01	01	0XXX	0x500–0x57F	Standard read/write
01	01	10XX	0x580–0x5BF	Standard read/write
01	01	11XX	0x5C0–0x5FF	Custom read/write
10	01	0XXX	0x900–0x97F	Standard read/write
10	01	10XX	0x980–0x9BF	Standard read/write
10	01	11XX	0x9C0–0x9FF	Custom read/write
11	01	0XXX	0xD00–0xD7F	Standard read-only
11	01	10XX	0xD80–0xDBF	Standard read-only
11	01	11XX	0xDC0–0xDFF	Custom read-only
Hypervisor and VS CSRs				
00	10	XXXX	0x200–0x2FF	Standard read/write
01	10	0XXX	0x600–0x67F	Standard read/write
01	10	10XX	0x680–0x6BF	Standard read/write
01	10	11XX	0x6C0–0x6FF	Custom read/write
10	10	0XXX	0xA00–0xA7F	Standard read/write
10	10	10XX	0xA80–0xABF	Standard read/write
10	10	11XX	0xAC0–0xAFF	Custom read/write
11	10	0XXX	0xE00–0xE7F	Standard read-only
11	10	10XX	0xE80–0xEBF	Standard read-only
11	10	11XX	0xEC0–0xEFF	Custom read-only
Machine-Level CSRs				
00	11	XXXX	0x300–0x3FF	Standard read/write
01	11	0XXX	0x700–0x77F	Standard read/write
01	11	100X	0x780–0x79F	Standard read/write
01	11	1010	0x7A0–0x7AF	Standard read/write debug CSRs
01	11	1011	0x7B0–0x7BF	Debug-mode-only CSRs
01	11	11XX	0x7C0–0x7FF	Custom read/write
10	11	0XXX	0xB00–0xB7F	Standard read/write
10	11	10XX	0xB80–0xBBF	Standard read/write
10	11	11XX	0xBC0–0xBFF	Custom read/write
11	11	0XXX	0xF00–0xF7F	Standard read-only
11	11	10XX	0xF80–0xFBF	Standard read-only
11	11	11XX	0xFC0–0xFFF	Custom read-only

RISC-V Control and Status Instructions

- **Machine-level:** Machine-level CSRs are accessible from machine mode, which is the most privileged mode of operation. These registers are used for controlling low-level, sensitive aspects of the hardware, such as interrupt handling, exception handling, and access to physical memory.
  Note: default level for low-level embedded CPUs

- **User-level (U):** User-level CSRs are accessible from the user mode, which is the least-privileged mode of operation. These registers typically contain information or controls that are safe to expose to user applications, such as performance counters that can be read without compromising system integrity.

RISC-V Control and Status Instructions

- **Supervisor-level (S):** Supervisor mode is used by operating systems to control system resources and manage user applications. It has a lower privilege than M-mode but can still perform system-level tasks.
- **Hypervisor-level (H):** Hypervisor mode is an optional privilege level introduced in newer versions of the RISC-V specification, designed for virtualization. It sits between M-mode and S-mode and is used to run and manage virtual machines.

RISC-V Control and Status Instructions

Common CSRs used for privilege handling

- **mstatus** (Machine Status Register): Tracks the current operating state and privilege level. (e.g. FPU status, user interrupt enabled)
- **mepc** (Machine Exception Program Counter): Stores the address of the instruction that caused an exception.
- **mcause**: Contains the reason for the last exception or interrupt. e.g. (0) misaligned address, (2) illegal instruction, (3) breakpoint
- **mtvec**: Base address of the machine trap vector for handling exceptions.

RISC-V Control and Status Instructions

How is the privilege level set/modified?

- CPU starts at machine level (M)
- CPU uses CSR mstatus to lower the privilege level
- CPU uses exception handling to increase the privilege level.
  - e.g.

```
addi x10, zero, 10 # Load system call ID 10 into x10
ecall # Make an environment call (triggering a trap)
```

- RISC-V mret, sret, uret are system return instructions for switching back to previous privilege level after exception handling.

RISC-V Control and Status Instructions

How is the privilege level enforced?

- If software running at a lower privilege level attempts to execute a privileged instruction, or a protected address space, or privileged interrupt, the processor will raise an exception

RISC-V Registers

- 32 general-purpose registers
- Separate floating-point registers
- Register x0=0x0 - why?

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

RISC-V Registers

- x0 is useful for
  - Computing zero, a very common constant for cleanup
  - Implementing NOP
  - Controling writeback for some instruction (e.g JAL)

RISC-V Registers

- PC is a hidden register
- How is the PC accessed by S/W?
  - AUIPC instruction
    - e.g. AUIPC rd, 0
  - JAL instruction
    - e.g. JAL rd, 4

RISC-V Registers

- **ABI Calling convention:**
- Arguments: a0-7
- Return: a0
- Temporaries: t0-6
- Saved registers: s0-11
- Who back up registers during calls?
  - Caller/parent
  - or Callee/child

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

RISC-V Registers

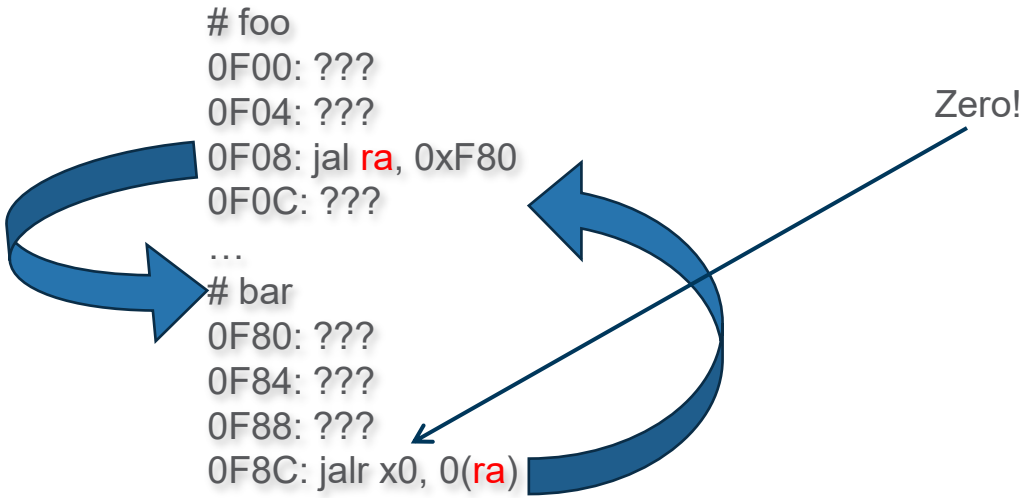
- **ABI function call example**
- Callee must save and restore saved registers
- RET Is not a native a RISC-V instruction

```
# Caller function
main:
   li s1, 100         # Load 100 into saved register s1
   jal ra, helloworld # Call the function "helloworld"
   ret

# Callee function
# This function wants to use s1 for its own calculations.
# It MUST save the original s1 first and restore it before returning.
helloworld:
   # 1. PROLOGUE: Back up registers to the stack
   addi sp, sp, -4     # Move stack pointer down (allocate 8 bytes)
   sw s1, 0(sp)        # Store caller's original s1 on the stack


   # 2. BODY:     Do work
   li s1, 500          # Now we can safely overwrite s1
   addi a0, s1, 5      # a0 = 505 (return value)


   # 3. EPILOGUE: Restore registers from the stack
   lw s1, 0(sp)        # Load original value back into s1
   addi sp, sp, 4      # Move stack pointer back up (deallocate)
   ret                 # Return to caller
```

RISC-V Registers

- **Return address (ra)**
  - Store the return address during function calls
  - Hold the next PC of the caller
  - Example: what is the value of *ra* when *bar()* is executing?

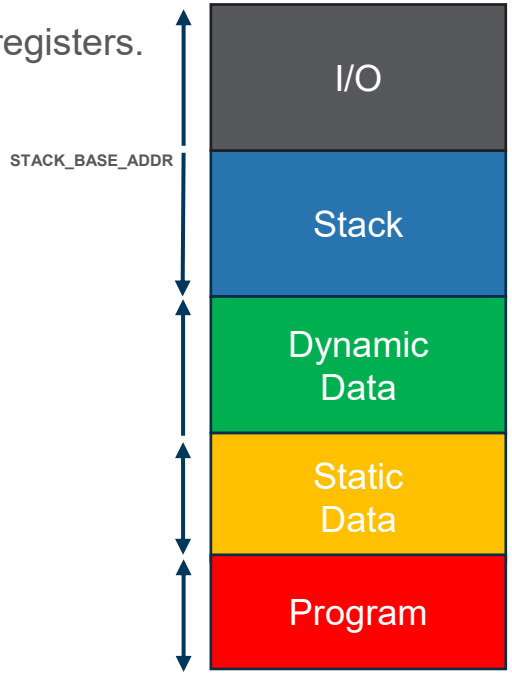
```
void bar() {
    <print("hello");>
}

void foo() {
   bar();  // call bar()
   <print("world\n");>
}
```



RISC-V Registers

- **Stack pointer (sp)**
  - Used for managing function call stack.
  - The stack holds function's temp variables that could not fit into registers.
  - Initialized to a starting location at boot time.
  - Stack base address decided by startup code.
  - **la** is a "load address" pseudo instruction
    - resolve to one or multiple RISC-V native instructions
    - depending on address value

```
# stack initialization
la sp, STACK_BASE_ADDR
```



RISC-V Registers

- **Stack pointer (sp)**
  - Example: A program uses 8 bytes of stack storage.

```
# Assume that x5 and x6 contain values that we want to preserve
# First, we'll adjust the stack pointer to allocate 8 bytes of space
addi sp, sp, -8   # Decrement the stack pointer by 8 bytes

# Now we'll store x5 and x6 onto the stack at the allocated space
sw x5, 0(sp)      # Store x5 at the beginning of the allocated space
sw x6, 4(sp)      # Store x6 right after x5

# ... the code can now use x5 and x6 for other operations…
add x5, x0, x1
add x6, x0, x1

# To restore the values of x5 and x6, we'll load them back from the stack
lw x5, 0(sp)      # Load x5 from the stack
lw x6, 4(sp)      # Load x6 from the stack

# Finally, we'll adjust the stack pointer back to free the 8 bytes
addi sp, sp, 8    # Increment the stack pointer by 8 bytes to restore its original value
```

RISC-V Registers

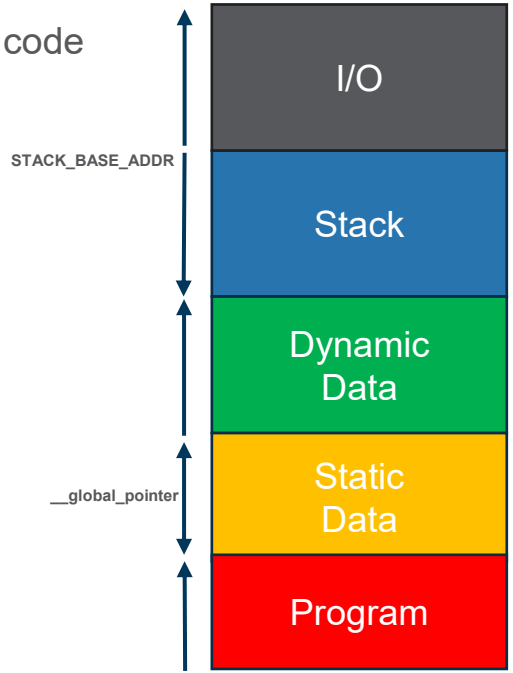
- **Global pointer (gp)**
  - Points to the middle of a global data segment
  - Why the middle?
    - Reduces instruction's immediate value when accessing global variables
    - Recall LD/ST instruction use signed immediate offsets
  - Example C code:

```
int counter = 10;

void increment_counter() {
  counter++;
}

int main() {
   increment_counter();
   return 0;
}
```

RISC-V Registers

- **Global pointer (gp)**
  - Initialized at boot time.
  - Linker computes and exports __global_pointer macro to startup code
  - **la** is a “load address” pseudo instruction
    - resolve to one or multiple RISC-V native instructions
    - depending on address value

```
# stack initialization
la gp, __global_pointer
```



RISC-V Registers

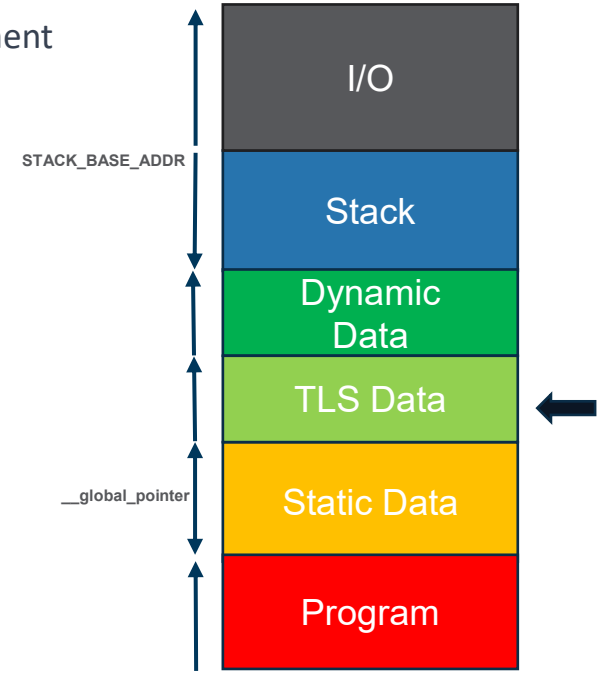
- **Global pointer (gp)**
  - Assembly program

```
increment_ counter :
   lw     t1, 0(gp)       # Load current value of counter
   addi    t1, t1, 1       # Increment the value
   sw     t1, 0(t0)       # Store the incremented value back to counter
   ret

main:
   call    increment_ counter
   li     a0, 0          # Set return value to 0
   ret
```

RISC-V Registers

- **Thread pointer (tp)**
  - Used to efficiently access **thread-local storage (TLS)**
  - TLS holds data unique to each thread in a multi-threading environment
  - Example:

```
#include <iostream>
#include <thread>
#include <vector>

thread_local int tl_counter = 0;

void incrementCounter() {
    tl_counter++;
}

int main() {
    std::vector<std::thread> threads;
    for(int i = 0; i < 5; ++i) {
        threads.push_back(std::thread(incrementCounter));
    }
    for(auto& t : threads) t.join();
    return 0;
}
```



RISC-V Registers

- **Thread pointer (tp)**
  - Initialized to a starting location at boot time.
  - Linker computes and exports __tcb_aligned_size macro to startup code.
  - Explain +63 and -64 below?

```
# initialize the stack pointer for each thread
csrr  t0, VX_CSR_MHARTID      # t0 will receive the thread identifier from mhartid CSR
sll   t1, t0, STACK_LOG2_SIZE  # shift by stack size
sub sp, sp, t1                 # set stack pointer

# set thread pointer register
# ensure cache line alignment
la    t1, __tcb_aligned_size  # allocated TLS size
mul   t0, t0, t1
la    tp, _end + 63            # _end holds the base address of TLS
add   tp, tp, t0
and   tp, tp, -64
ret
```

RISC-V Registers

- **Thread pointer (tp)**
  - Assembly code for incrementCounter()

```
# Pseudocode for RISC-V assembly to increment a TLS variable
# Assume that the offset of tl_counter within the TLS is known and is 'offset'

incrementCounter:
    # Load the address of tl_counter from the thread's TLS using the 'tp' register
    addi    t0, tp, offset       # t0 = address of tl_counter

    # Load the current value of tl_counter
    lw      t1, 0(t0)            # t1 = *t0 (the value of tl_counter)

    # Increment the value
    addi    t1, t1, 1            # t1 = t1 + 1

    # Store the incremented value back into tl_counter
    sw      t1, 0(t0)            # *t0 = t1

    # Return from the function
    ret
```

Attendance

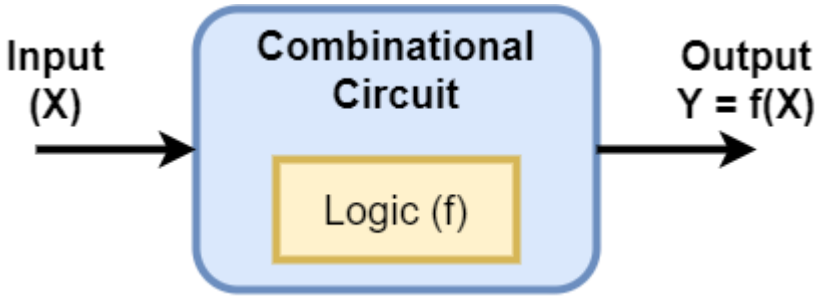
- Please Fill the form with today’s keyword to register your attendance.

Review: Digital Circuits

Combinational Circuits

Combinational circuits

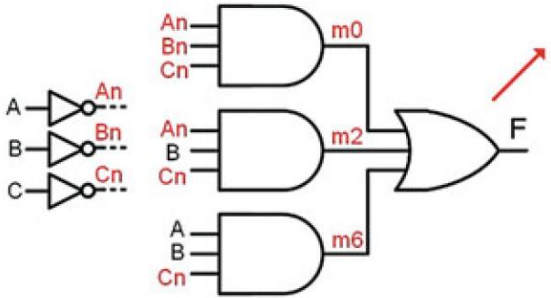
- Output depends on current inputs only
- Consist of a connection of multiple logic gates
  - And/Or/Not



Combinational Circuits

Combinational circuits

- **Example1**




```
module SystemX (output wire F,
                input  wire A, B, C);

   wire  An, Bn, Cn;   // internal nets
   wire  m0, m2, m6;

   assign An = ~A;             // Not's
   assign Bn = ~B;
   assign Cn = ~C;

   assign m0 = An & Bn & Cn;  // AND's
   assign m2 = An & B  & Cn;
   assign m6 = A  & B  & Cn;

   assign F  = m0 | m2 | m6;  // OR

endmodule
```

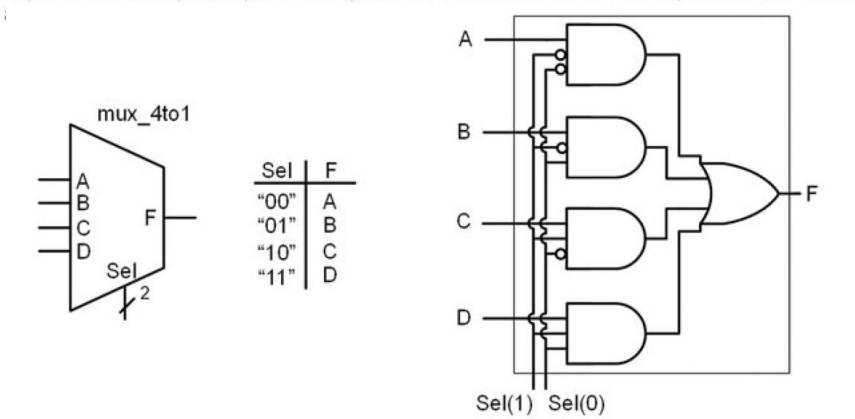
UCLA Samueli School of Engineering

From: Quick start Guide to Verilog (© springer Nature Switzerland)

Combinational Circuits

Combinational circuits

- **Example2: multiplexer**



The following shows how to model the behavior of the mux using continuous assignment and logical operators.

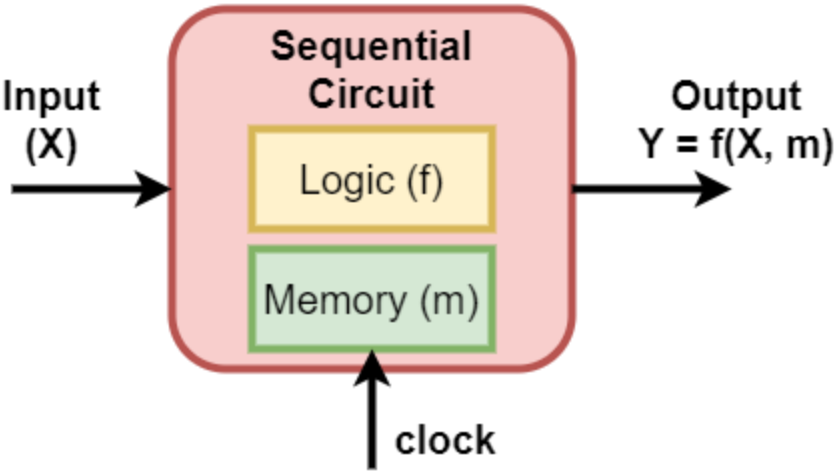
```
module mux_4to1 (output wire F,
                 input  wire A, B, C, D,
                 input  wire [1:0] Sel);

   assign F = (A & ~Sel[1] & ~Sel[0]) |
              (B & ~Sel[1] &  Sel[0]) |
              (C &  Sel[1] & ~Sel[0]) |
              (D &  Sel[1] &  Sel[0]);

endmodule
```

From: Quick start Guide to Verilog (© springer Nature Switzerland)

Sequential Circuits

Sequential circuits

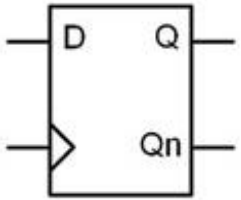
- Output depends on current inputs and prior state
- Need a storage/memory to hold prior state
- e.g. storage element
  - D-flip-flops
  - RAMs



Sequential Circuits

D-Flip-flop:

- Digital storage element
- Input is propagated to output at the rising edge of the clock
- **Registers are implemented using D-Flip-flops**



Clk	D	Q	Qn	
0	X	Last Q	Last Qn	Store
1	X	Last Q	Last Qn	Store
⨍	0	0	1	Update
⨍	1	1	0	Update

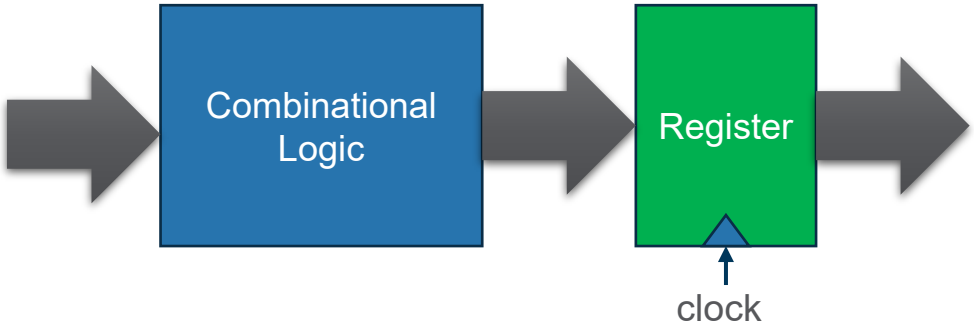
```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, D);

  always @ (posedge Clock)
     begin
        Q  <= D;
        Qn <= ~D;
     end

endmodule
```

From: Quick start Guide to Verilog (© springer Nature Switzerland)

Applications of D-Flip-Flop

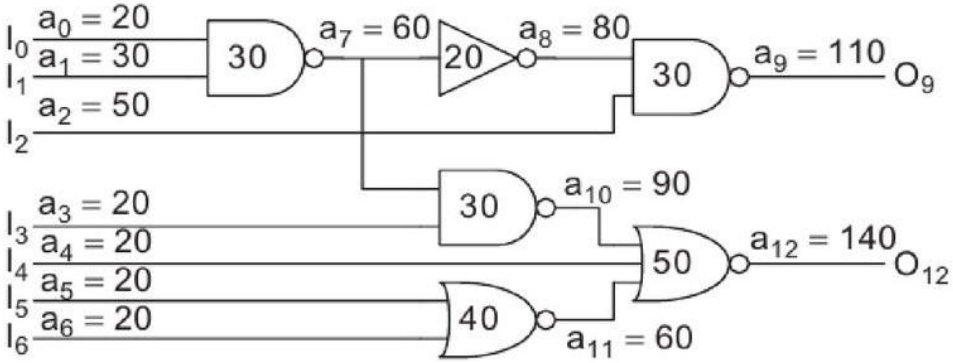
D-Flip-flop are use for:

- Data storage
- Data synchronization
- Finite state machines
- Pipelining
- **Timing resolution**
  - Reduced propagation delay of combinational blocks
  - Improve clock speed



Propagation Delay and Critical Path

- **Propagation delay** (Tp)
  - Time needed for the gate to respond to a change at its inputs
  - F = 1 / Tp
- **Critical path** is the longest propagation delay in the circuit between I/O or registers
  - Fmax = 1 / Tmax



UCLA Samueli School of Engineering

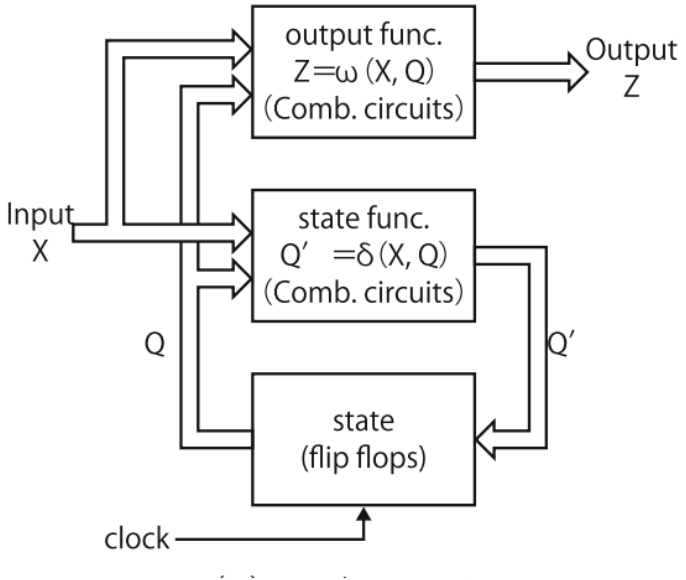
From Rudoff Hunter

Finite State Machines

Definition: State-driven model of computation to design sequential circuits.

Mealy type FSM

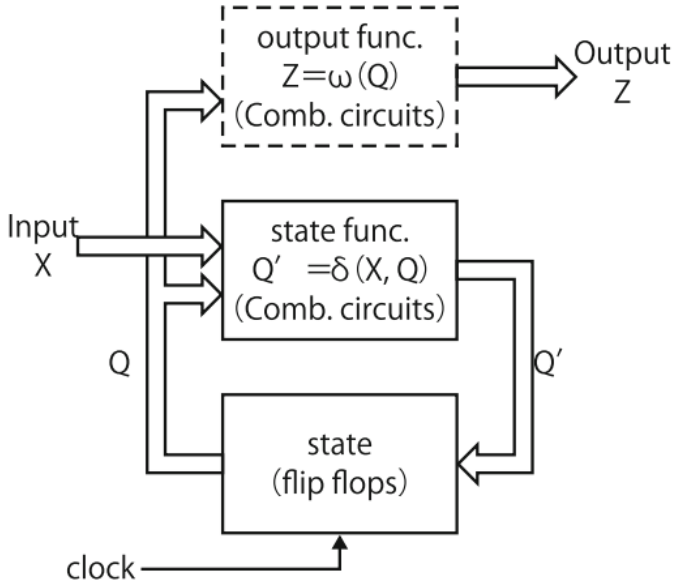
- (state, input) → output
- Pros: Smaller number of FSM states
  - Some input directly contribute to output
- Cons: Input critical path
  - Why?



Finite State Machines

Moore type FSM

- (state) → output
- Pros: High-speed
  - Why?
- Cons: Large number of states
  - due to full inputs handling
  - can affect circuit size



RISC-V CPU Implementation

RISC-V CPU Implementation

Basic RISC-V CPU Datapath

RISC-V ISA subset:

- R-type
- I-type
- LD/ST
- Branch

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI

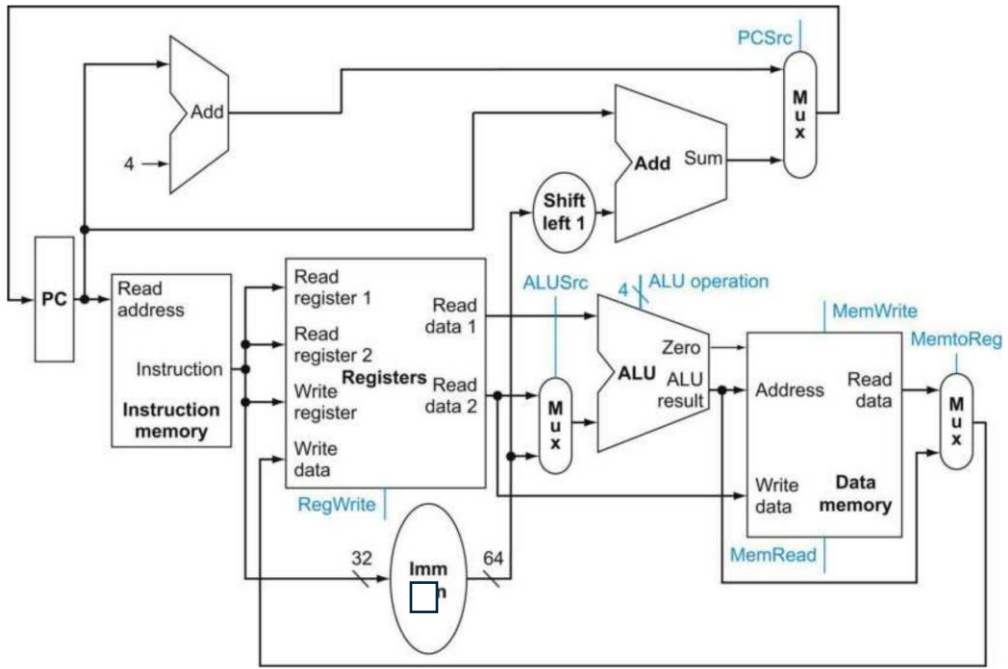
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ

Basic RISC-V CPU Datapath

Basic Steps:

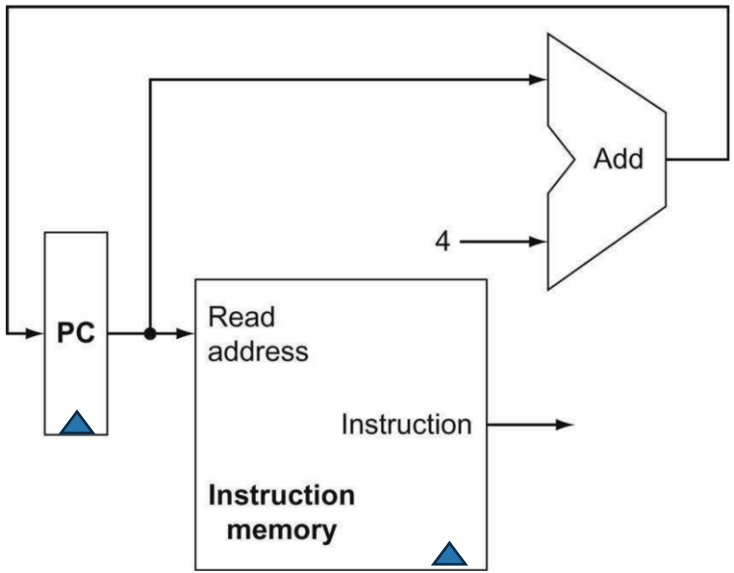
1. Fetch instruction from PC
2. Register file access
3. ALU execution
4. Memory access
5. Next PC update



Basic RISC-V CPU Datapath

Next PC calculation:

1. Program counter stored in PC
2. Can perform the addition in parallel
3. While reading the next instruction



Basic RISC-V CPU Datapath

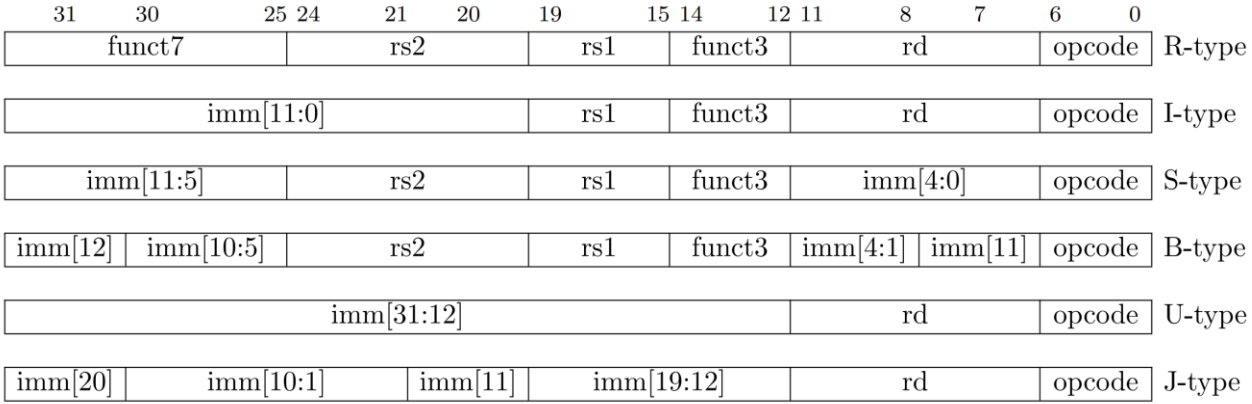
Register File:

- 32 total registers (5-bit addressing)
- 2x 5-bit source register addresses
- 1x 5-bit destination register address
- RegWrite: write enable
- How to decode register address?

Basic RISC-V CPU Datapath

Register File:

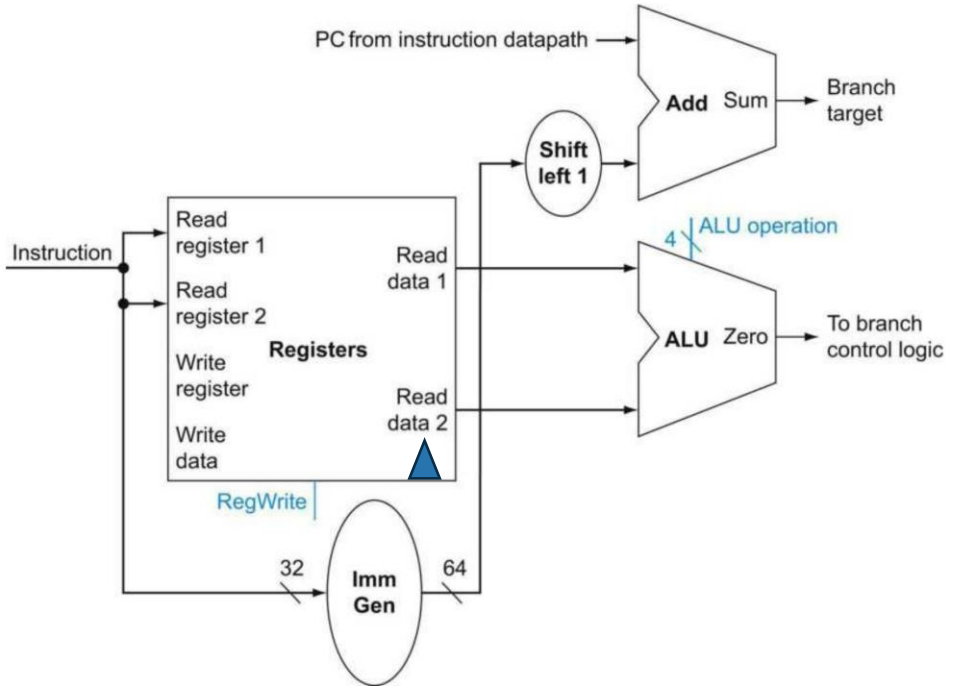
- How to decode register address?
  - No logic needed
  - why?



Basic RISC-V CPU Datapath

Branch operations:

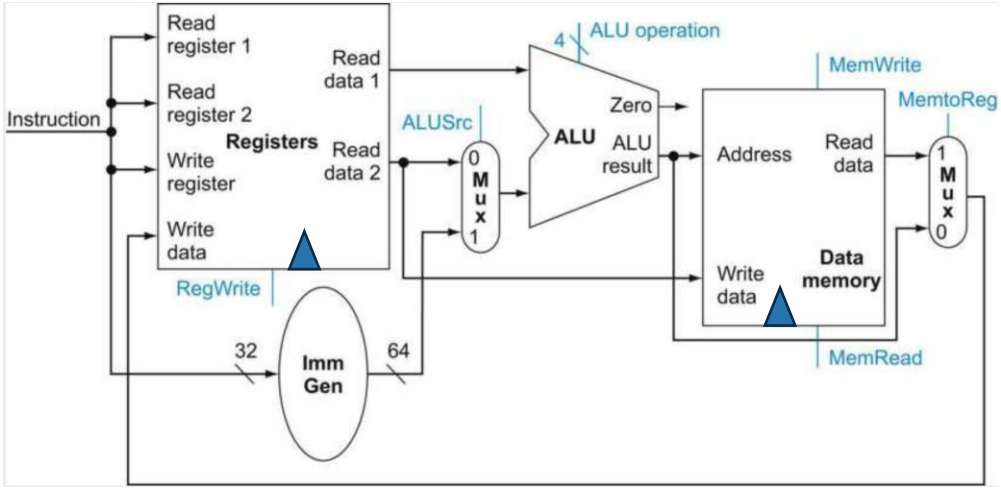
1. Register access
2. Immediate generation
3. Branch target calculation
4. Branch condition evaluation
5. Parallelism?
   - 2-3, 4
6. Critical path?
   - 2-3



Basic RISC-V CPU Datapath

Memory Operations:

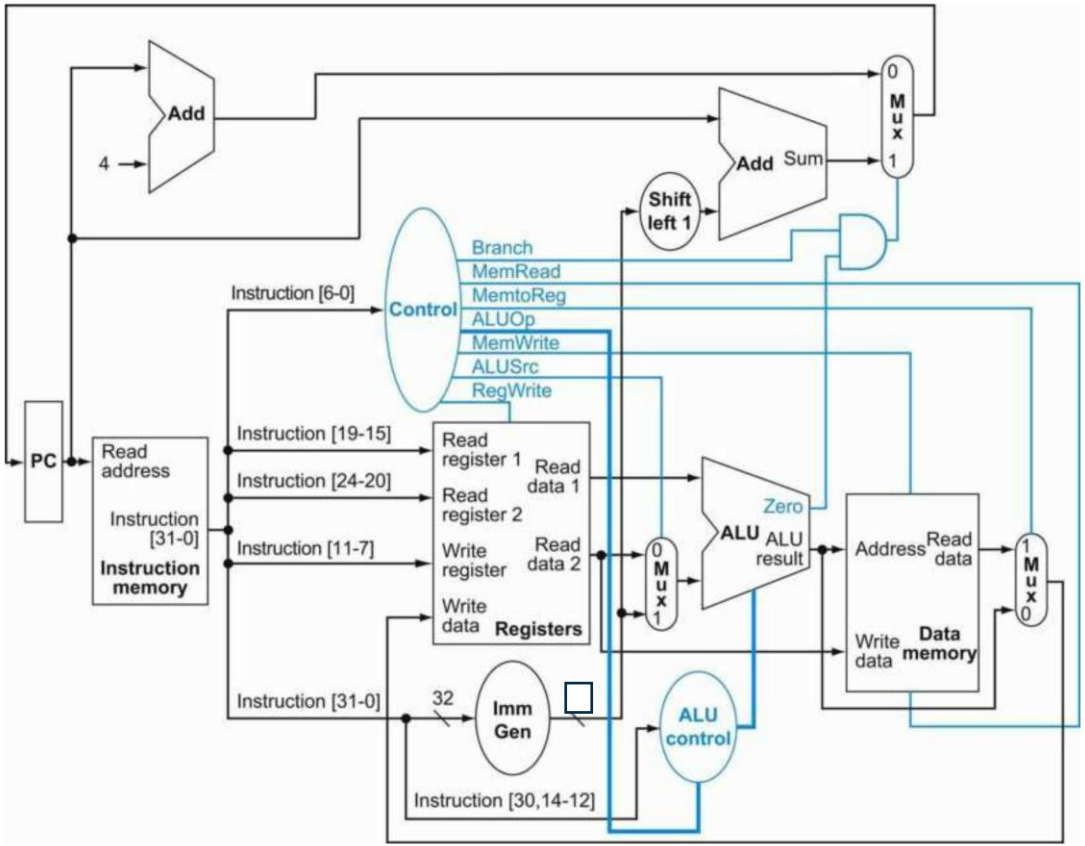
1. Address calculation
2. Memory access
3. ALUSrc select
   - 0: register
   - 1: immediate



Basic RISC-V CPU Control Path

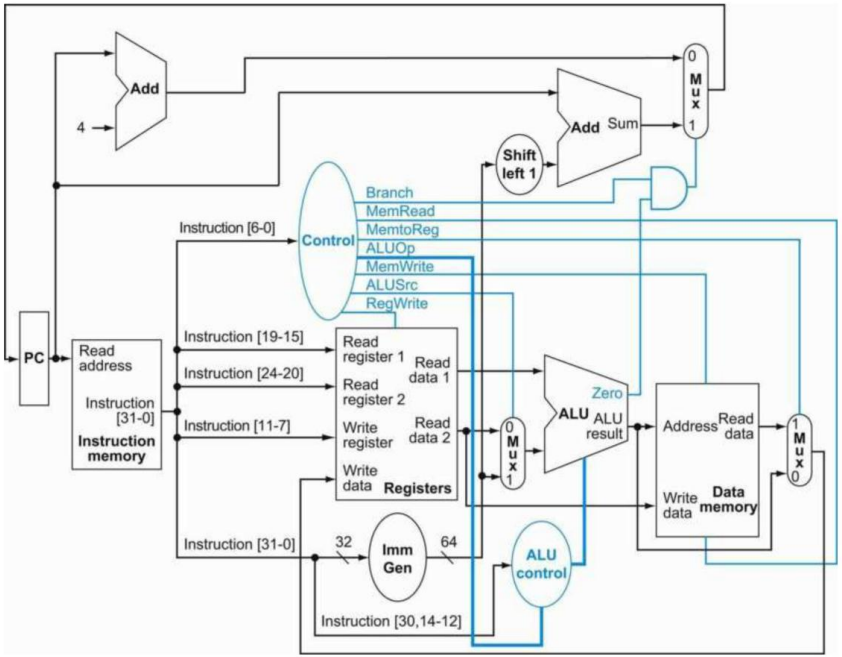
CPU control logic:

- Main control
  - Register File
    - RegWrite
    - MemtoReg
  - ALU
    - ALUOp
    - ALUSrc
  - Data memory
    - MemRead
    - MemWrite
  - Branch
- ALU control
  - ALU operation



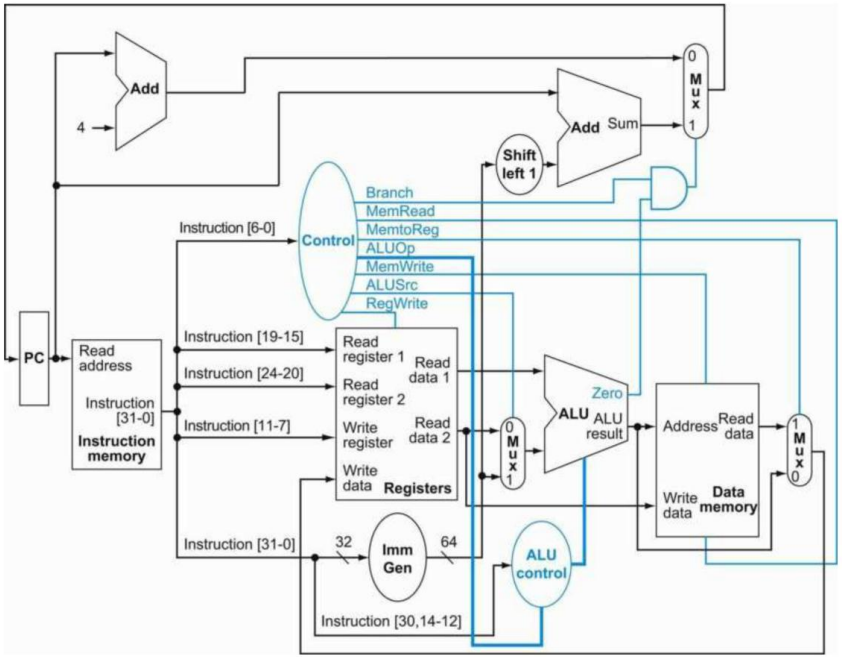
Basic RISC-V CPU Control Path

Instruction	Opcode	RegWrite	AluSrc	Branch	MemRead	MemWrite	MemtoReg
R-Type	0110011						



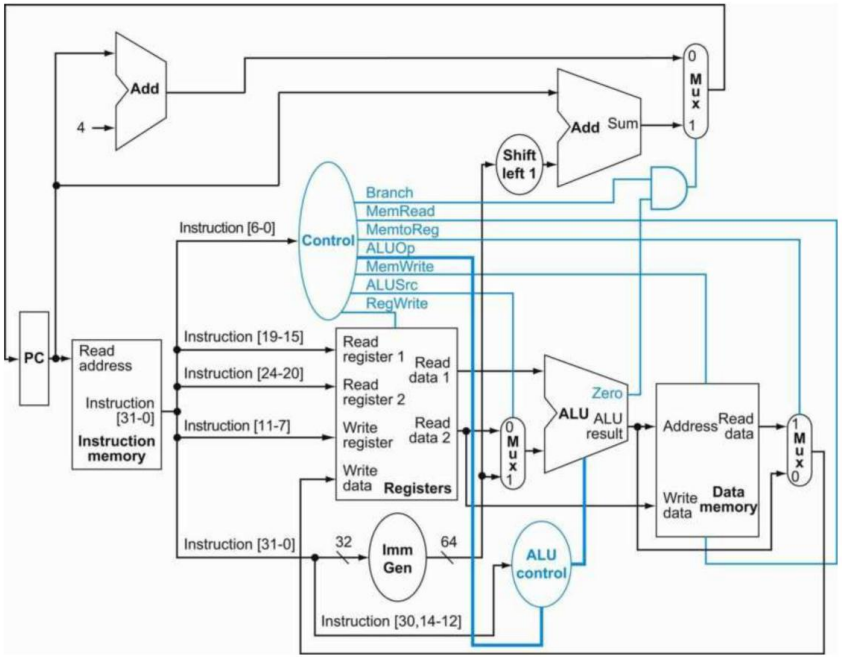
Basic RISC-V CPU Control Path

Instruction	Opcode	RegWrite	AluSrc	Branch	MemRead	MemWrite	MemtoReg
R-Type	0110011	1	0	0	0	0	0



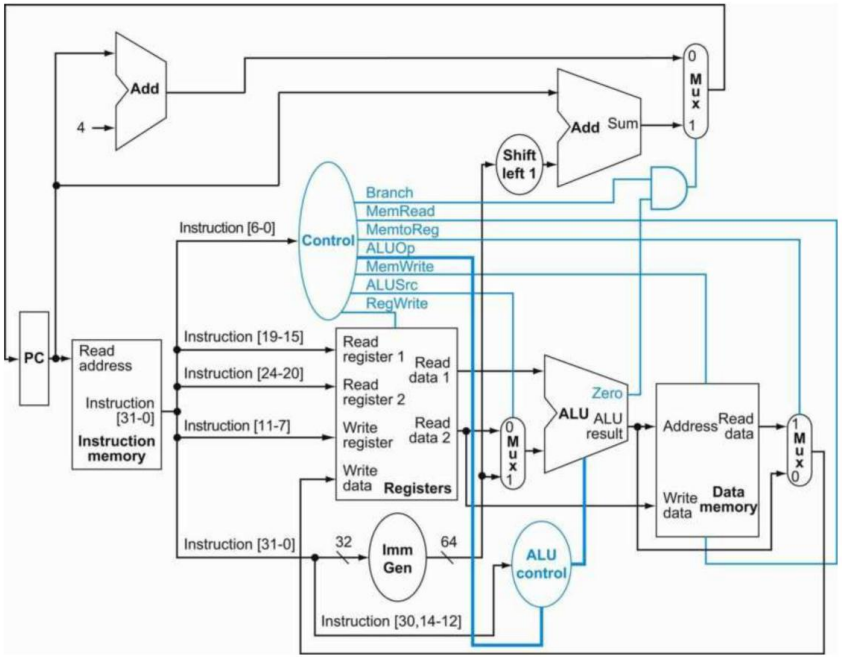
Basic RISC-V CPU Control Path

Instruction	Opcode	RegWrite	AluSrc	Branch	MemRead	MemWrite	MemtoReg
I-Type	0010011						



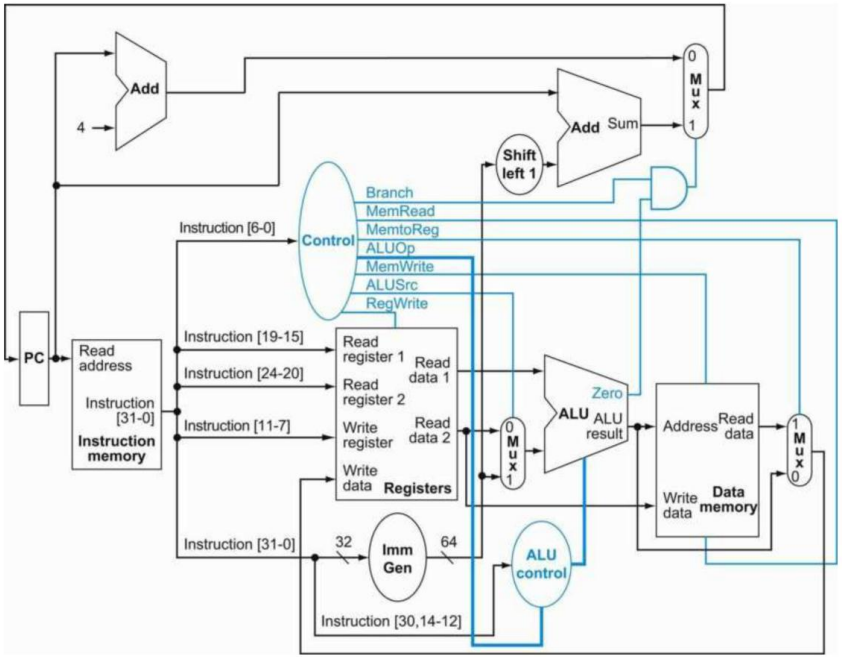
Basic RISC-V CPU Control Path

Instruction	Opcode	RegWrite	AluSrc	Branch	MemRead	MemWrite	MemtoReg
I-Type	0010011	1	1	0	0	0	0



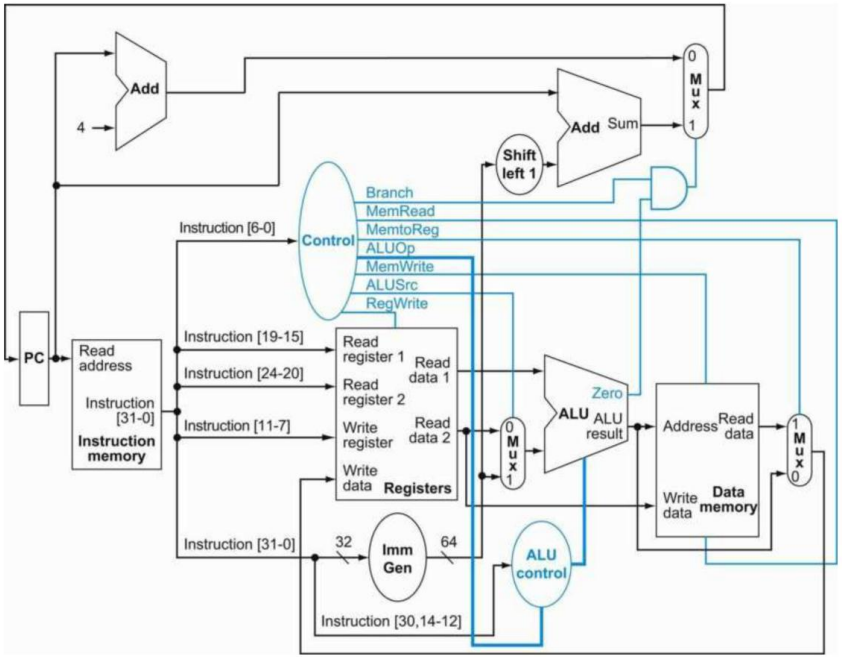
Basic RISC-V CPU Control Path

Instruction	Opcode	RegWrite	AluSrc	Branch	MemRead	MemWrite	MemtoReg
LW	0000011						



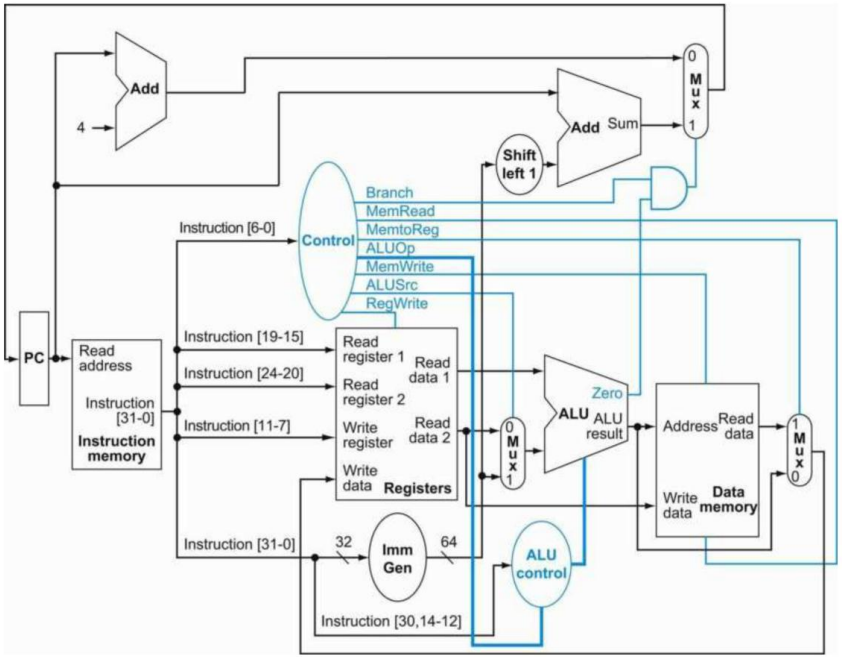
Basic RISC-V CPU Control Path

Instruction	Opcode	RegWrite	AluSrc	Branch	MemRead	MemWrite	MemtoReg
LW	0000011	1	1	0	1	0	1



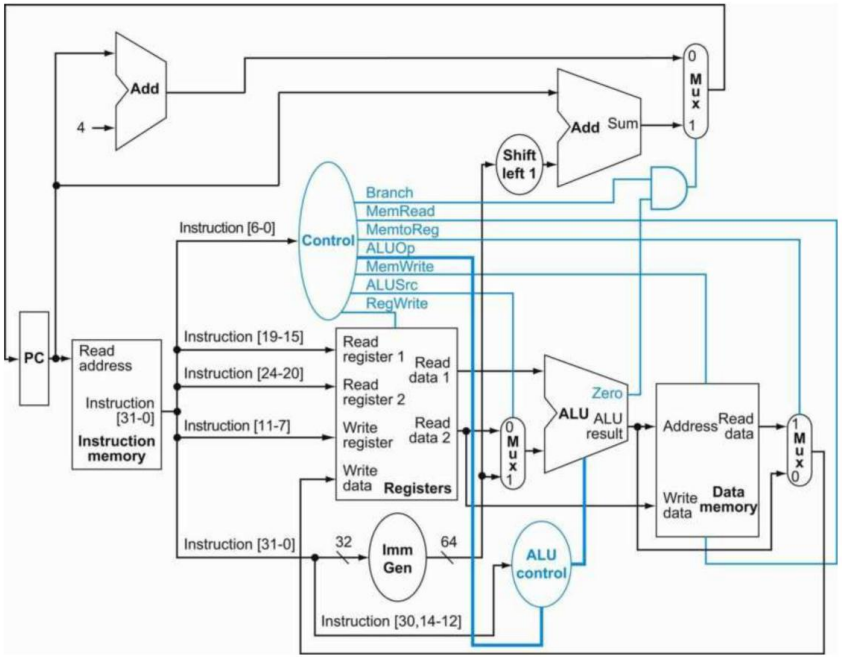
Basic RISC-V CPU Control Path

Instruction	Opcode	RegWrite	AluSrc	Branch	MemRead	MemWrite	MemtoReg
SW	0100011						



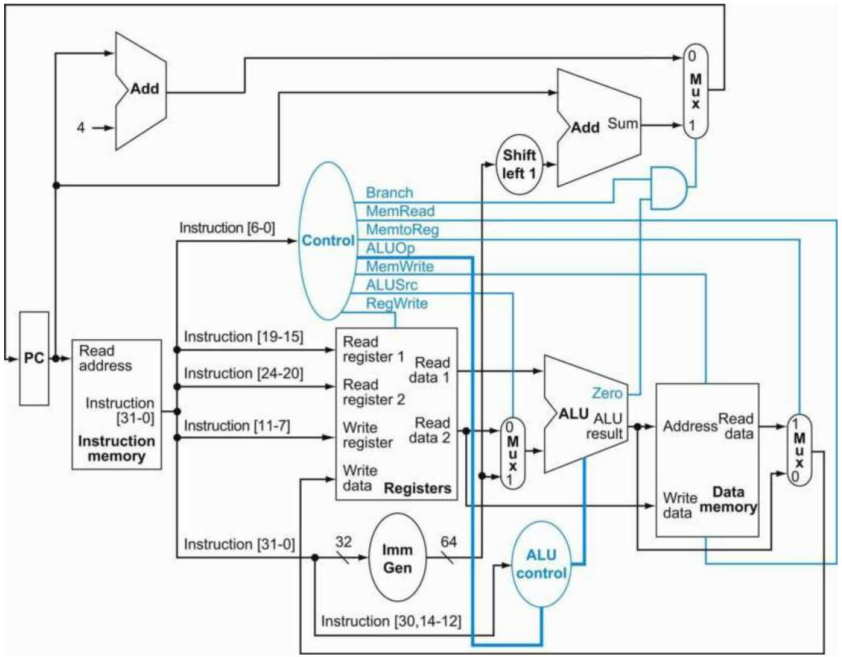
Basic RISC-V CPU Control Path

Instruction	Opcode	RegWrite	AluSrc	Branch	MemRead	MemWrite	MemtoReg
SW	0100011	0	1	0	0	1	0



Basic RISC-V CPU Control Path

Instruction	Opcode	RegWrite	AluSrc	Branch	MemRead	MemWrite	MemtoReg
BEQ	1100011	0	0	1	0	0	0



Basic RISC-V CPU Control Path

How to determine the type of instruction with minimal logic?

Instruction	Opcode	RegWrite	AluSrc	Branch	MemRead	MemWrite	MemtoReg
R-Type	0110011	1	0	0	0	0	0
I-Type	0010011	1	1	0	0	0	0
LW	0000011	1	1	0	1	0	1
SW	0100011	0	1	0	0	1	0
BEQ	1100011	0	0	1	0	0	0

Basic RISC-V CPU Control Path

- R-Type := intrs[4] & instrs[5]
- I-Type := intr[4] & ~instr[5]
- LW := ~instr[4] & ~instr[5]
- SW := ?
- BEQ := ?

Instruction	Opcode	RegWrite	AluSrc	Branch	MemRead	MemWrite	MemtoReg
R-Type	0110011	1	0	0	0	0	0
I-Type	0010011	1	1	0	0	0	0
LW	0000011	1	1	0	1	0	1
SW	0100011	0	1	0	0	1	0
BEQ	1100011	0	0	1	0	0	0

Basic RISC-V CPU Control Path

ALUOp?

Only 4 operations:

- AND
- OR
- ADD
- SUB

BEQ := (rs1 – rs2)

Instruction	Opcode	Func3	Func7	ALU Fn	ALU Op
AND	0110011	000	0000000	ADD	10
SUB	0110011	000	0100000	SUB	11
OR	0110011	110	0000000	OR	01
AND	0110011	111	0000000	AND	00
ADDI	0010011	000	xxxxxxx	ADD	10
ORI	0010011	110	xxxxxxx	OR	01
ANDI	0010011	111	xxxxxxx	AND	00
LW	0000011	010	xxxxxxx	ADD	10
SW	0100011	010	xxxxxxx	ADD	10
BEQ	1100011	000	xxxxxxx	SUB	11

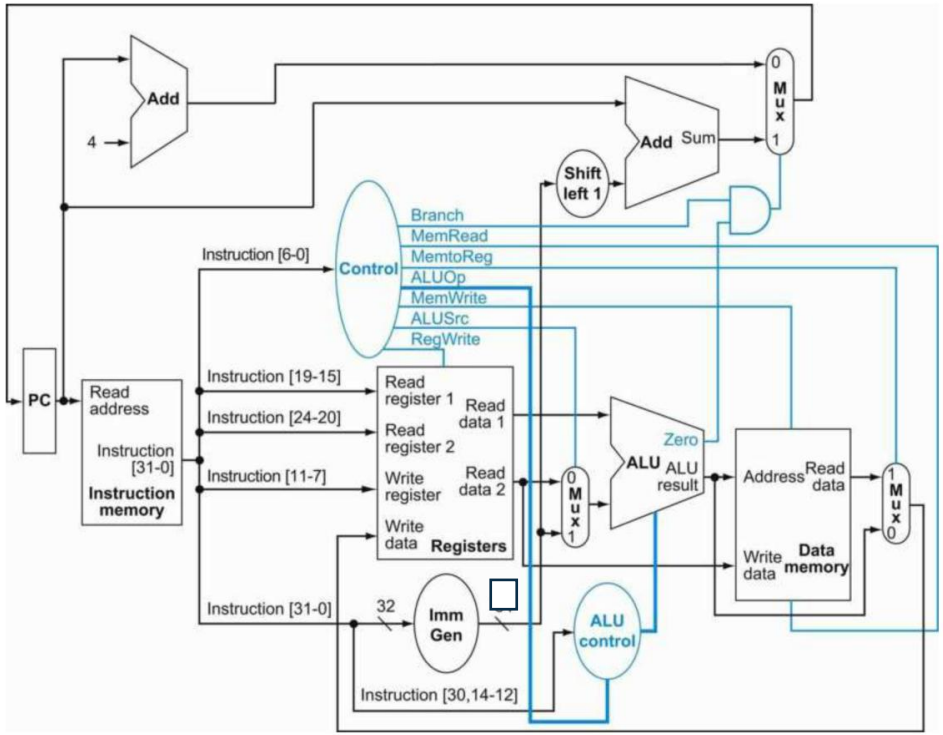
Single-Cycle Processor

Single-cycle Processor:

- Cycle I
  - Instruction memory read
  - Control Units produce output
  - ALU produces output
  - Data memory read
  - Next PC calculation

Note: Assume Instruction and data memory read is asynchronous (same cycle), however memory updates are only visible in next cycle.

Advantages?

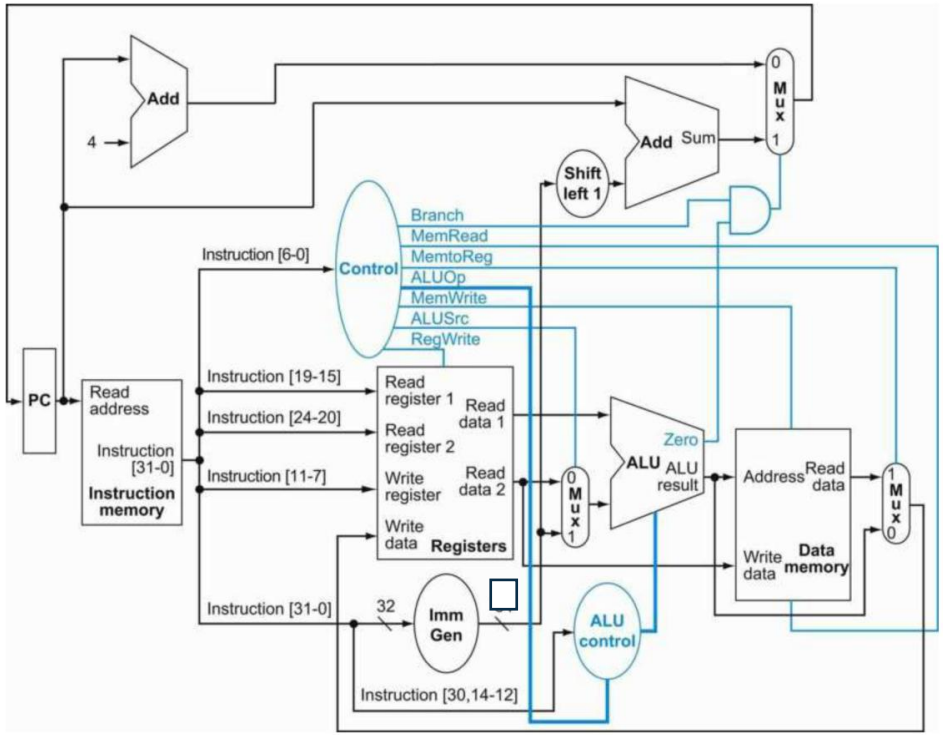


Single-Cycle Processor

Single-cycle Processor Advantages:

- Simple to implement
- Efficient design (resource usage)

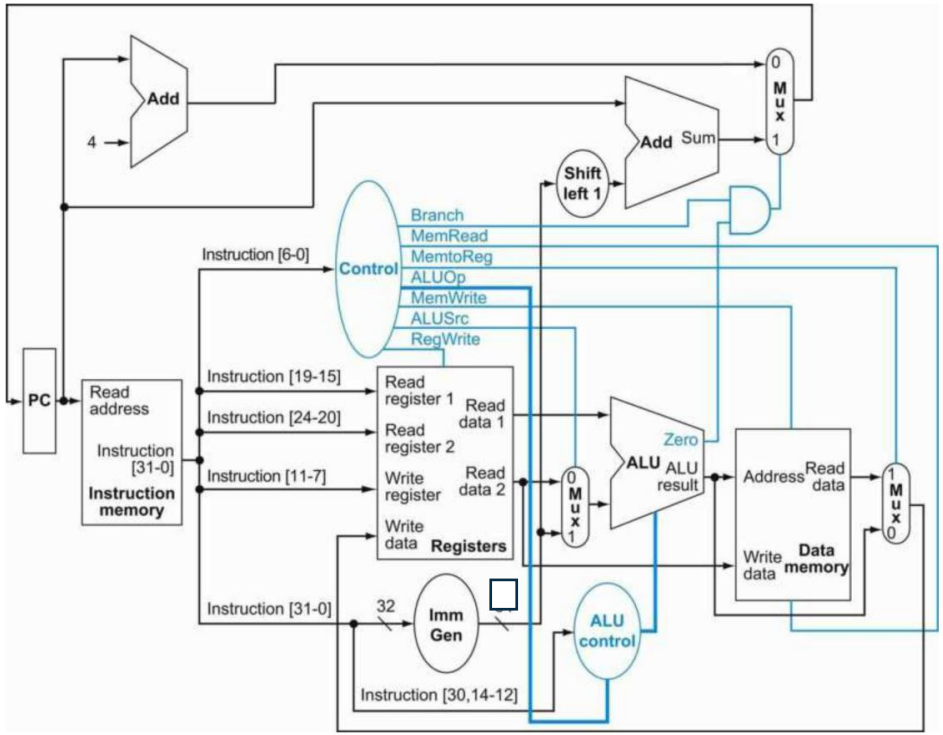
Limitations?



Single-Cycle Processor

Single-cycle Processor Limitations:

- Weak performance
  - Long latency

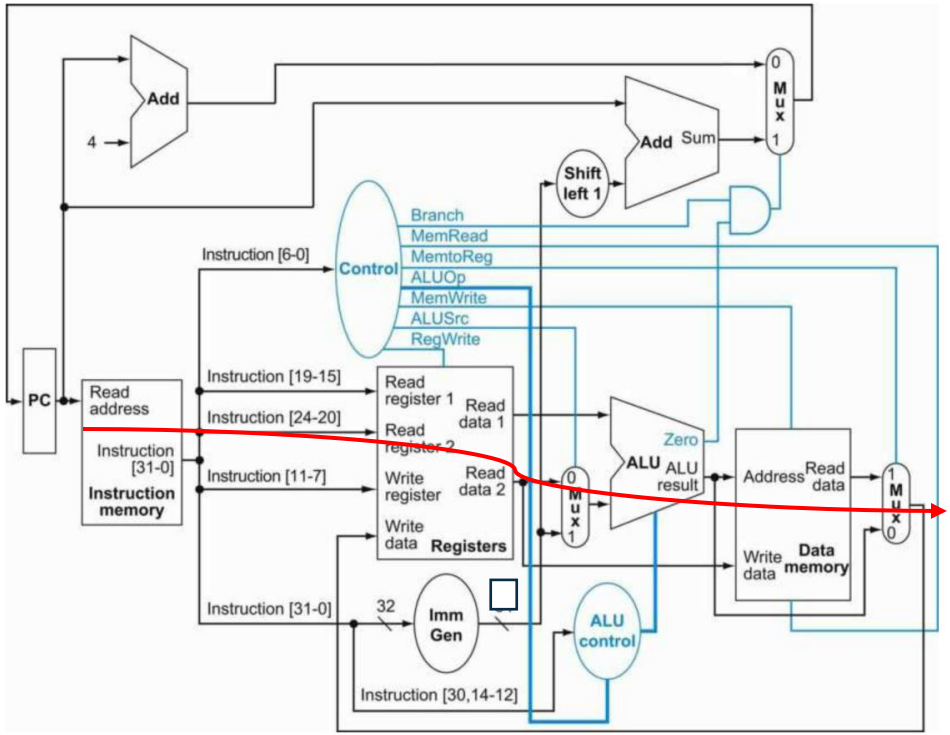


Single-Cycle Processor

Single-cycle Processor Latency:

- Long propagation delay
  - Signal propagation from source to destination
  - **Source**: input or register
  - **Destination**: output or register
- Imbalanced datapath
- Which path is the longest?

Why it affects latency?



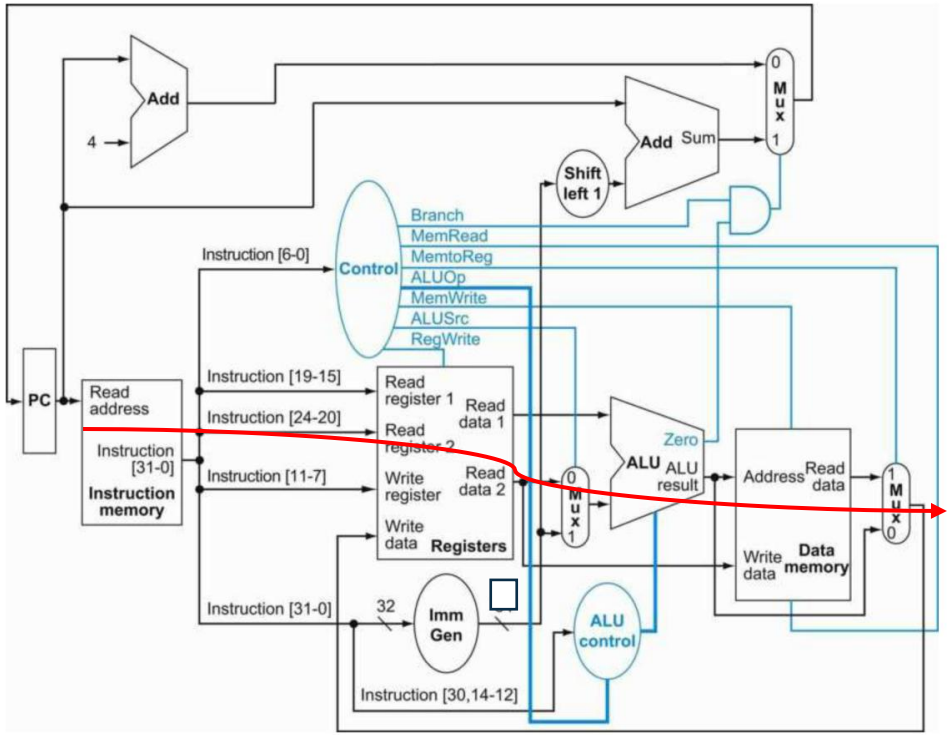
Single-Cycle Processor

Single-cycle Processor Latency:

- How this affect CPU latency?
- Longest delay determines the clock speed
- CPU clock affects performance

CPU Time = InstCount * CPI * **CycleTime**

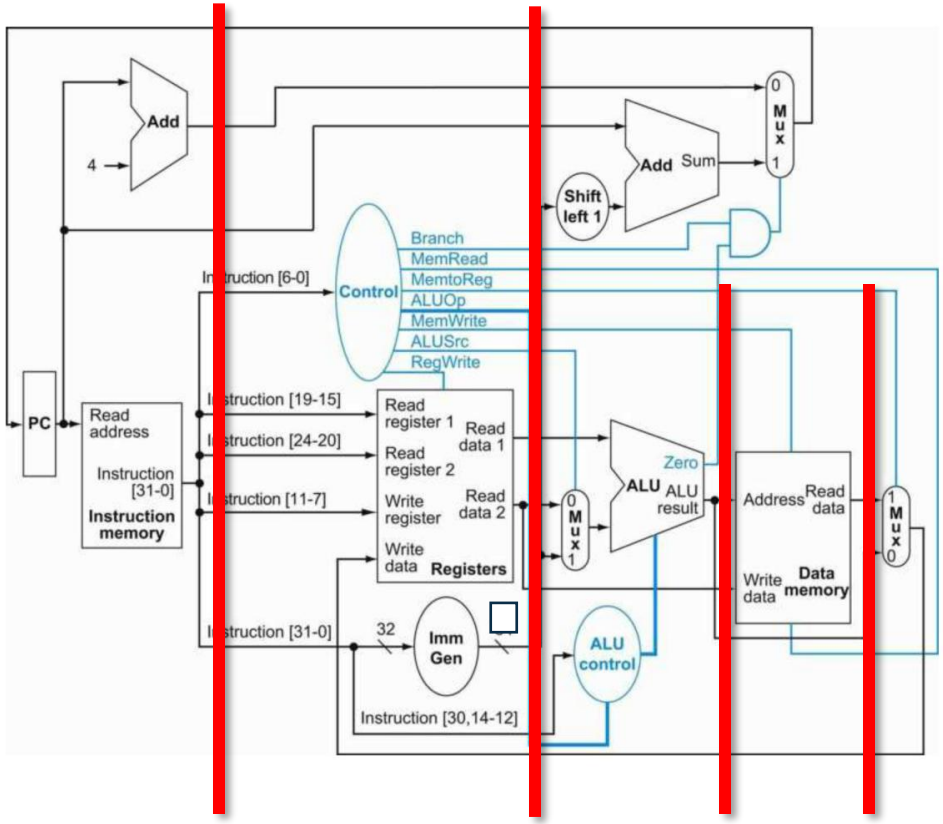
How to reduce propagation delay?



Single-Cycle Processor

Reducing Propagation Delay

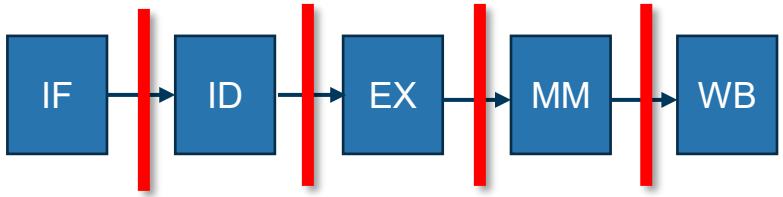
- Split the datapath into smaller chunks
- Insert registers in between
- Update control logic
  - Control registers inputs/outputs

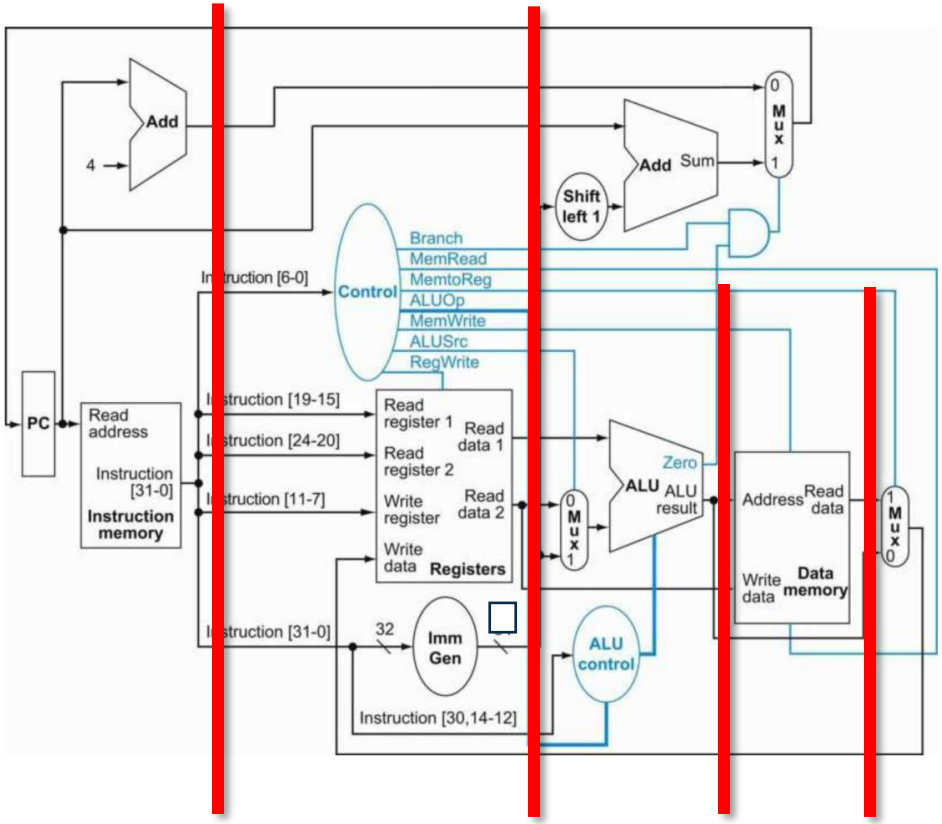


Multi-Cycle Processor

Change to multi-cycle architecture

- 5 stages
  - IF: Instruction Fetch
  - ID: Register access
  - EX: ALU operation
  - MEM: Memory access
  - WB: Write back





Q&A