Lecture 5: Single-Cycle CPU

CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

Acknowledgements and Disclaimer

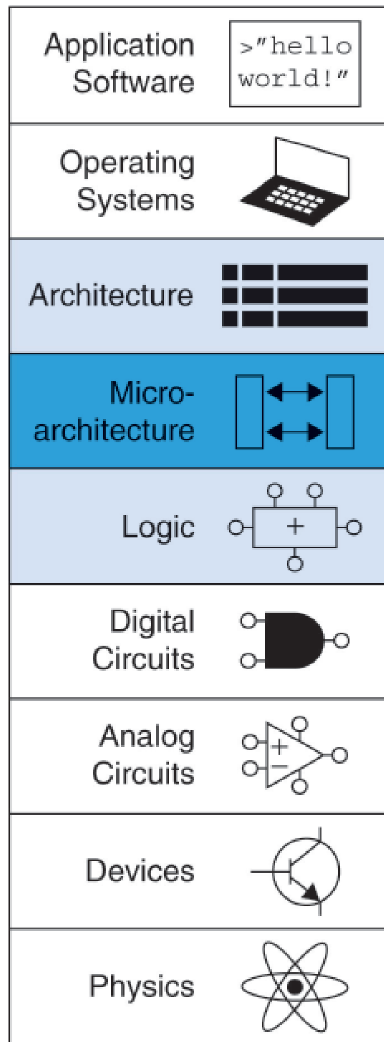
- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS 252 at UC Berkeley
    - https://people.eecs.berkeley.edu/~culler/courses/cs252-s05/
  - E85 at HMC
    - https://pages.hmc.edu/harris/class/e85/old/fall21/

Outline

- ALU
- State Element
- Single-Cycle CPU
- Design the Datapath

Introduction

- **Microarchitecture**
  - How to implement an architecture in hardware
- **Processor**
  - Datapath: functional blocks
  - Control: control signals

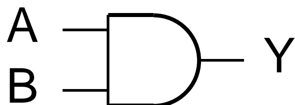


Logic Design Basics

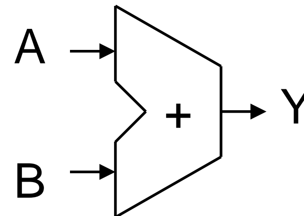
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

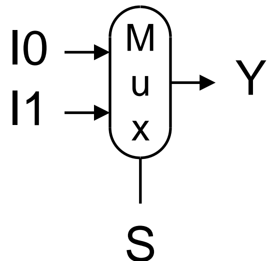
- AND-gate
  - Y = A & B



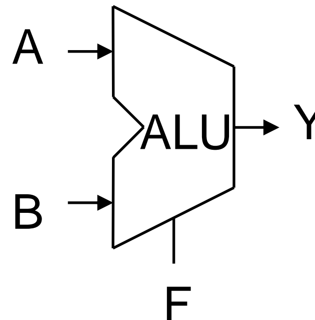
- Adder
  - Y = A + B



- Multiplexer
  - Y = S ? I1 : I0



- Arithmetic/Logic Unit
  - Y = F(A, B)

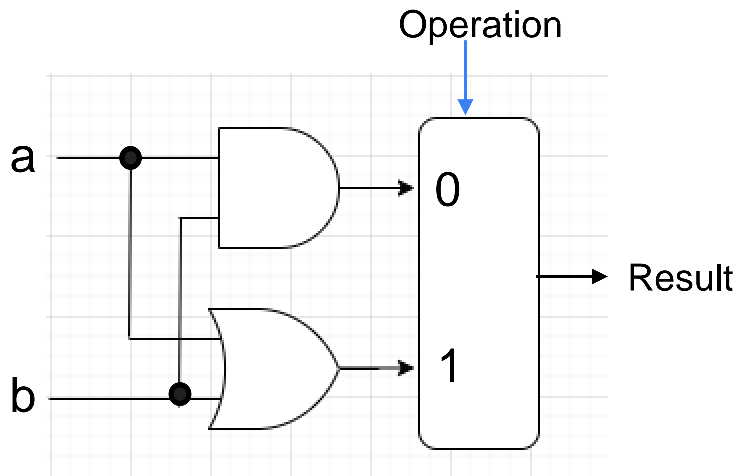


ALU: Arithmetic Logic Unit

- ALU combines a variety of mathematical and logical operations into a single unit
  - Addition
  - Subtraction
  - AND
  - OR
- ALU used for
  - Load/Store: compute memory address
  - Branch: test the branch condition
  - R-type: depends on opcode

ALU: Arithmetic Logic Unit

- A 1-bit ALU for AND and OR
  - The MUX selects a AND b or a OR b depending on whether the value of Operation is 0 or 1



ALU in RISC-V

- ALU control (4 bits)
  - Specifies which function to perform

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	Subtract
1010	SLT

- ALUOp (2-bit) control field
  - Indicates whether the operation to be performed

ALU in RISC-V

- A Verilog behavioral definition of a RISC-V ALU

```
module RISCVALU (ALUctl, A, B, ALUOut, 
Zero);
    input [3:0] ALUCtrl;
    input [31:0] A, B;
    output reg [31:0] ALUOut;
    output Zero;
    assign Zero = (ALUOut == 0); // 
zero is true if ALUOut is 0
    always @(ALUCtrl, A, B) begin
        case (ALUCtrl)
            0: ALUOut <= A & B;
            ……
        endcase
    end
endmodule
```

```
module ALUControl (ALUOp, FuncCode, 
ALUCtrl);
    input [1:0] ALUOp;
    input [5:0] FuncCode;
    input [3:0] reg ALUCtrl;
    always case (FuncCode)
        ….
        default: ALUOp <= 15;
     endcase
endmodule
```

ALU in RISC-V

- ALUOp (2-bit) control field
  - Indicates whether the operation to be performed
    - add (00) for loads and store
    - Determined by the operation encoded in the funct7 and funct3 field

opcode	ALUOp	Operation	Desired ALU action	ALU control
lw	00	Load word	Add	0010
sw	00	Store word	Add	0010
beq	01	Branch if equal	Subtract and test if zero	0110
R-type	10	add	Add	0010
R-type	10	sub	Subtract	0110
R-type	10	and	AND	0000
R-type	10	or	OR	0001

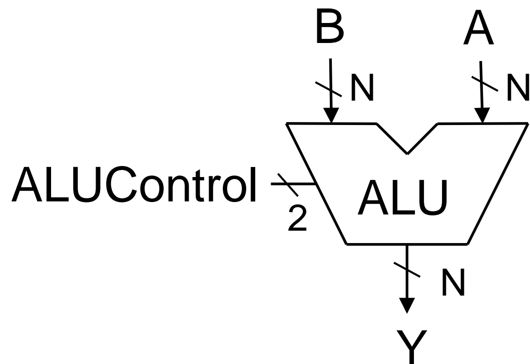
Simple ALU Implementation

- Simple ALU
  - N-bit ALU with N-bit inputs and outputs
  - ALU receives a 2-bit control signal *ALUControl*
    - When ALUControl = 11, ALU performs A OR B

$ALUControl_1$

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

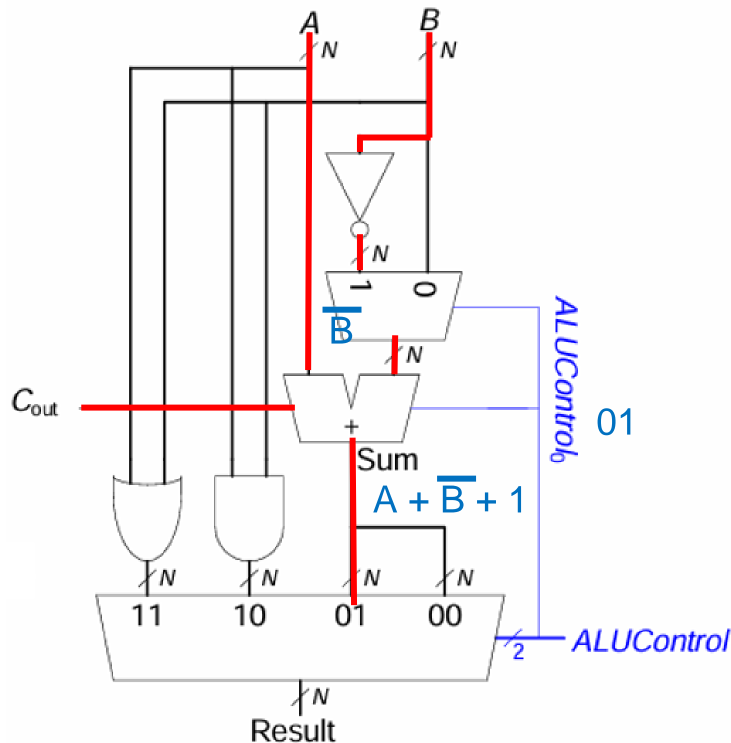
$ALUControl_0$



Simple ALU Implementation

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

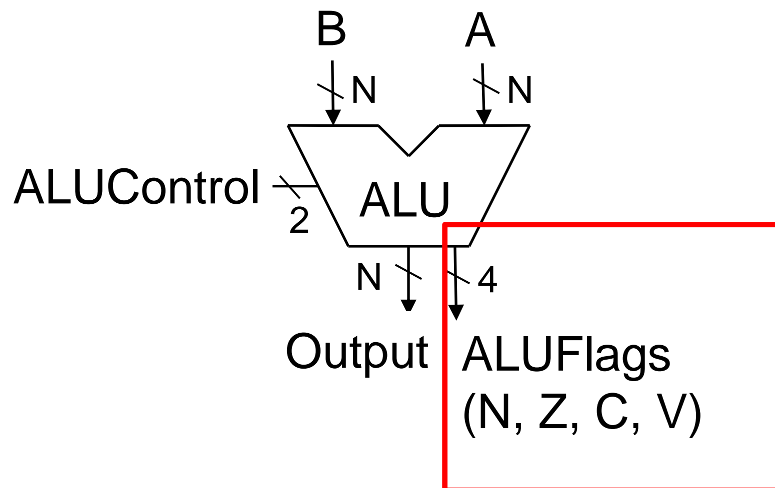
- Simple ALU
  - Contains an N-bit adder and N 2-input AND and OR gates
  - An inverter and a MUX to invert input B when $ALUControl_0$ is asserted
  - If ALUControl = 01 => A – B
    - $\overline{B}$ + 1 = -B in two's complement
    - The adder receives A and $\overline{B}$ and assert carry in
    - Perform A + $\overline{B}$ + 1 = A - B



Simple ALU Implementation

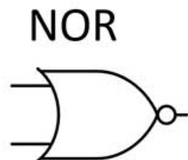
- ALU with status flags
  - Some ALUs produce extra outputs, call flags that indicate information about the ALU output

Flag	Description
N	Negative Result
Z	Zero Result
C	Adder produce Carry out
V	Adder oVerflowed

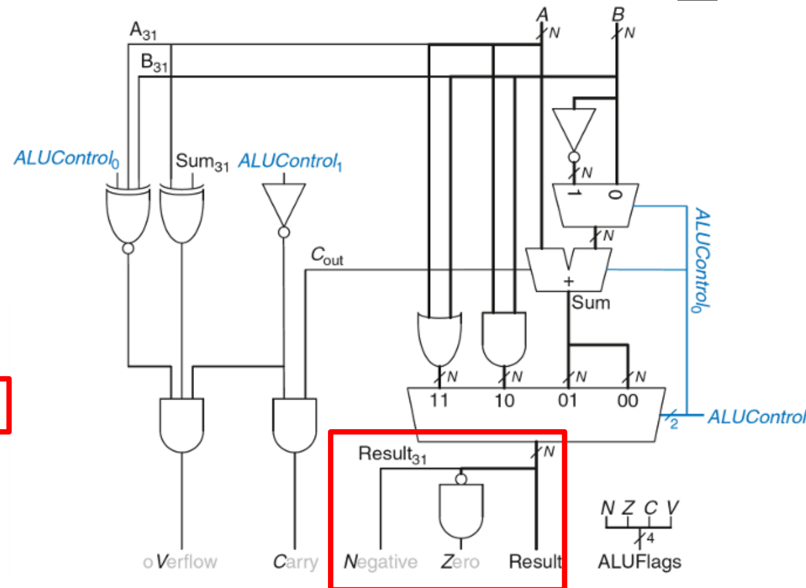


Simple ALU Implementation

- ALU with the N flag
  - MSB bit of two's complement number is 1 if it is negative
  - <u>The N flag is connected to the MSB bit of ALU output ($Result_{31}$)</u>
- ALU with the Z flag
  - The Z flag is asserted when all of the bits of Result is 0
  - Detect by the N-bit NOR gate



A	B	Output
0	0	1
1	0	0
0	1	0
1	1	0



Simple ALU Implementation

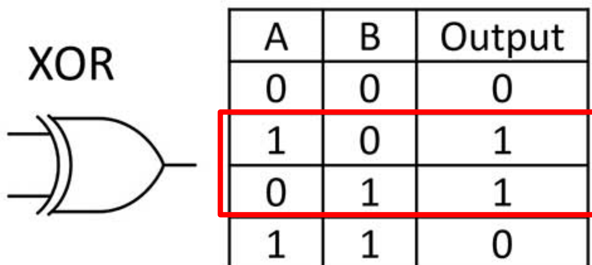
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

- ALU with the C flag
  - The C flag is asserted when the adder produces a carry out
  - The ALU performs ADD/SUB ($ALUControl_1 = 0$)
- ALU with the V flag
  - Overflow occurs when the addition of two same signed numbers produces a result with the opposite sign

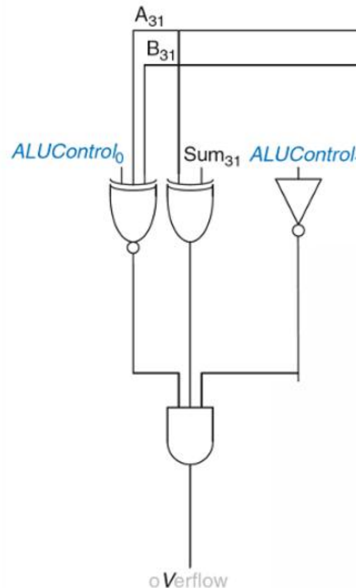
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Simple ALU Implementation

- ALU with the V flag
  - The V flag is asserted when all three conditions are true
    - 1) ALU performs ADD/SUB ($ALUControl_1 = 0$)
    - 2) A and Sum have opposite signs as detected by the XOR gate



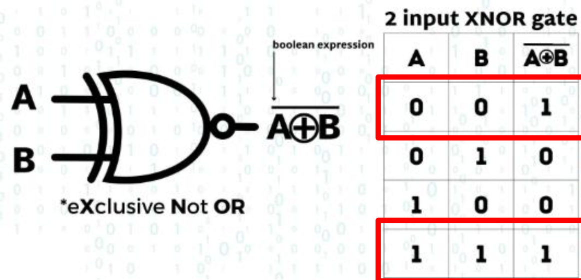
A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0



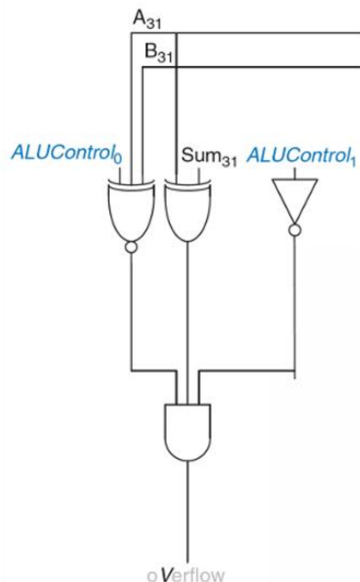
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Simple ALU Implementation

- ALU with the V flag
  - The V flag is asserted when all three conditions are true
    - 3) Overflow is possible as detected by XNOR gate
      - Either A and B have the same sign and $ALUControl_0$ = 0 or (ADD)
      - A and B have the opposite sign and $ALUControl_0$ = 1 (SUB)



A	B	A⊕B
0	0	1
0	1	0
1	0	0
1	1	1



Simple ALU Implementation

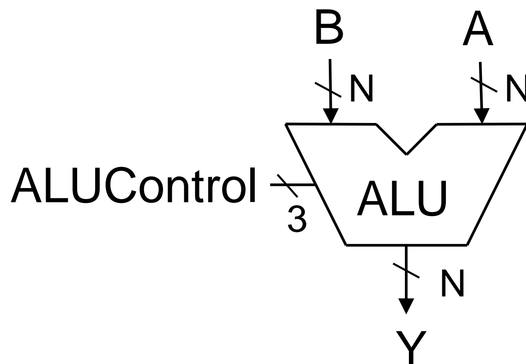
- ALU flags can also be used for comparison
  - To know if A = B
    - ALU computes A – B and looks at the flag Z
    - If Z is asserted, the result = 0, so A = B

Comparison	Signed	Unsigned
==	Z	Z
!=	~Z	~Z
<	N ^ V	~C
<=	Z | (N ^ V)	Z | ~C
>	~Z & ~(N ^ V)	~Z & C
>=	~(N ^ V)	C

Simple ALU Implementation

- ALU also can implement SLT (set if less than) instruction
  - When A < B, Result = 1. Otherwise, Result = 0
  - SLT treats inputs as signed. SLTU treats the inputs as unsigned
  - To add another function to the ALU
    - Update ALUControl signal for handling SLT

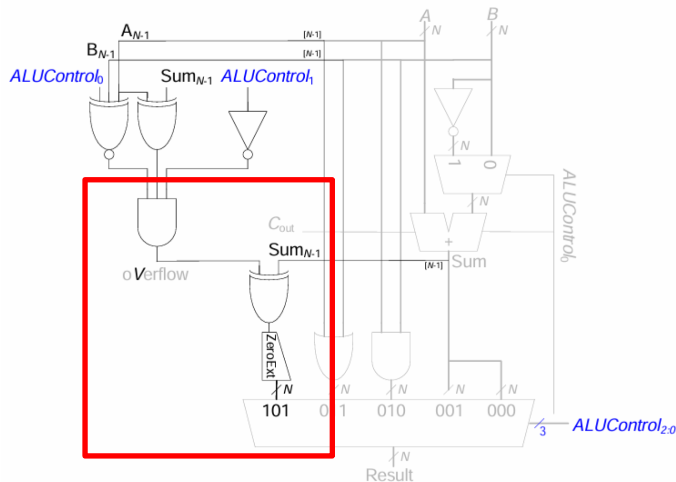
$ALUControl_{2:0}$	Function
000	Add
001	Subtract
010	AND
011	OR
101	SLT



Simple ALU Implementation

- ALU also can implement SLT (set if less than) instruction
  - We determine if A is less than B by performing A – B
    - $ALUControl_0$ = 1 causes the adder to perform A – B
  - If the result is negative, A is less than B
    - When Sum_{N-1} = 1, the result of A – B is negative
    - We zero-extend Sum_{N-1}
    - Feed it into 101 MUX input

$ALUControl_{2:0}$	Function
000	Add
001	Subtract
010	AND
011	OR
101	SLT

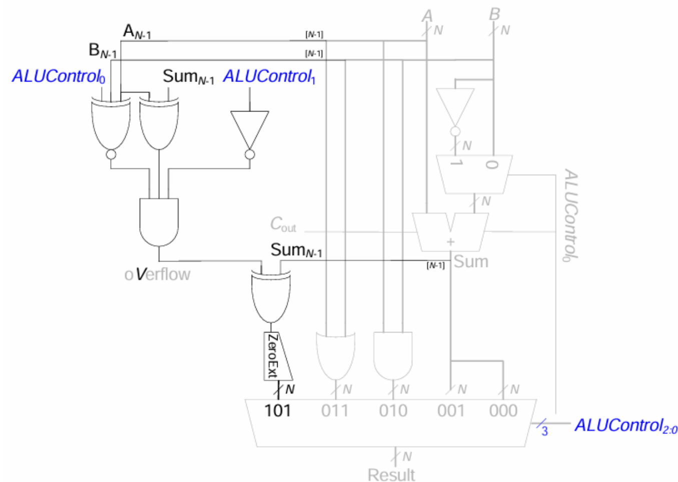


Simple ALU Implementation

- ALU also can implement SLT (set if less than) instruction
  - When overflow occurs
    - Sum will have incorrect sign
    - We XOR sign bit of SUM with V (overflow signal) to correctly indicate a negative Sum

$ALUControl_{2:0}$	Function
000	Add
001	Subtract
010	AND
011	OR
101	SLT

A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0

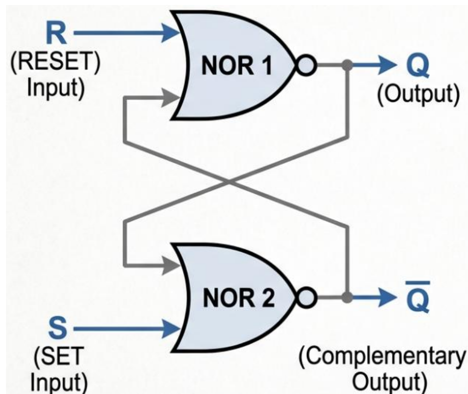


State Elements

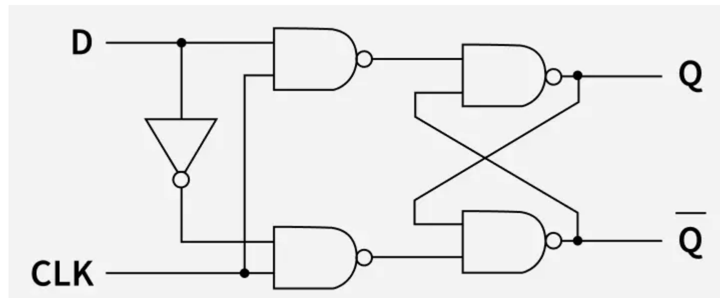
- Outputs of sequential logic depend on current and prior input values – it has **memory**
- **Sequential circuits**
  - Give sequence to events
  - Have memory (short-term)
  - Use feedback from output to input to store information

State Elements

- **State**
  - <u>Everything about the prior inputs to the circuit necessary to predict its future behavior</u>
  - State elements store state
    - SR Latch
    - D Latch
    - D Flip-flop



SR Latch

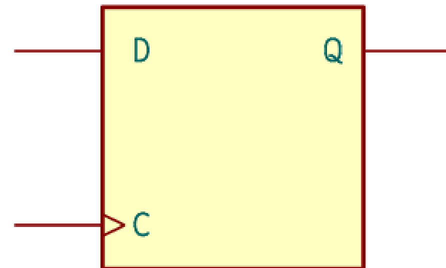


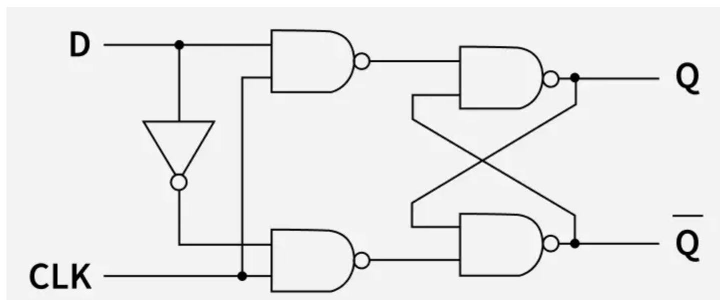
D Flip-Flop

State Elements

- **State**
  - D Flip-flop
    - Two inputs, data and clock input which controls the flip flop.
    - When clock input is high, the data is transferred to the output of the flip flop.
    - When the clock input is low, the output of the flip flop is held in its previous state.

Clock Edge	D (Input)	Q (Next State)
Rising	0	0
Rising	1	1
No Edge	X	Q (No Change)

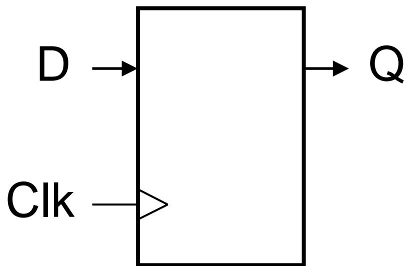


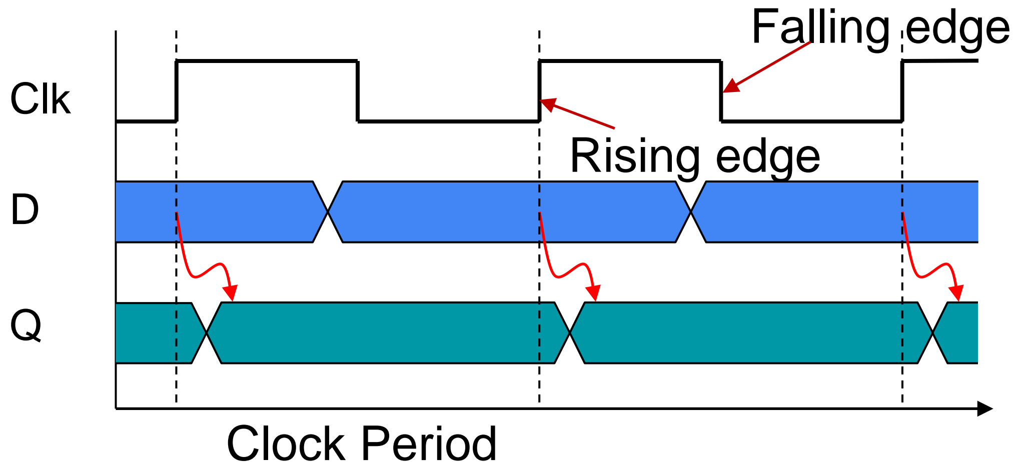


D Flip-Flop

State Elements

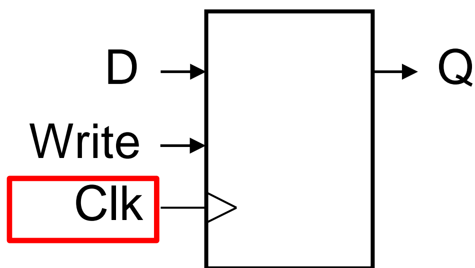
- Register: store data in a circuit
  - Use a clock signal to determine when to update the store value
  - The clock period is the time for one full cycle
  - Edge-triggered: update when Clk changes from 0 to 1 (Positive-edge triggered)

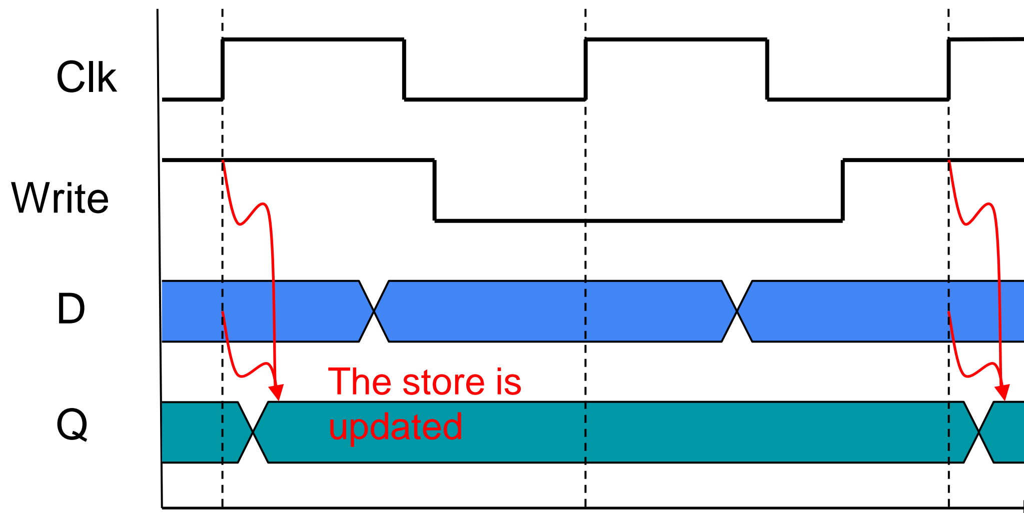




State Elements

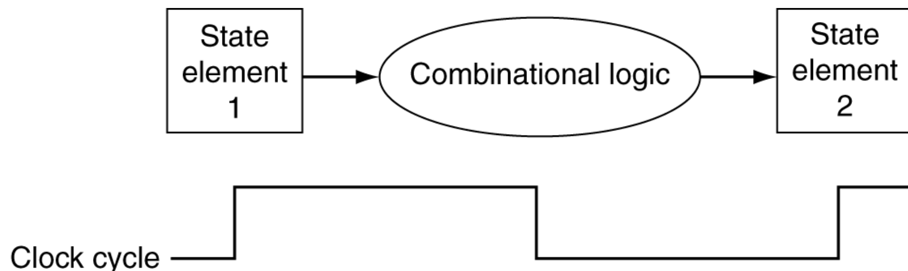
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later





State Elements

- An edge-triggered clock methodology
  - The combinational circuits cannot have feedback
  - State elements provide valid inputs and store valid outputs to/from the combinational logic block
  - <u>The state element outputs changes only after the clock edge</u>
  - To ensure that values written into the state elements on the active clock edge are valid

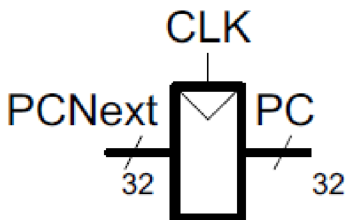


Clocking Methodology

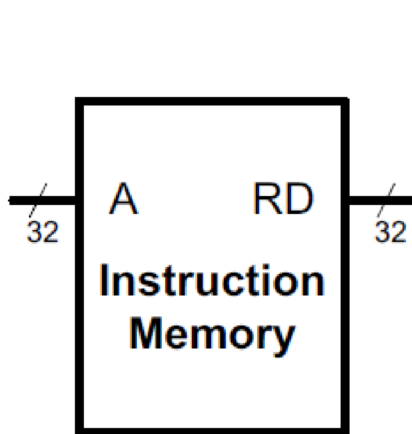
- An edge-triggered clock methodology
  - The clock must have a long enough period so that the input values are stable when the active clock edge occurs
  - Allows a state element to be read and written in the same clock cycle

State Elements on RISC-V

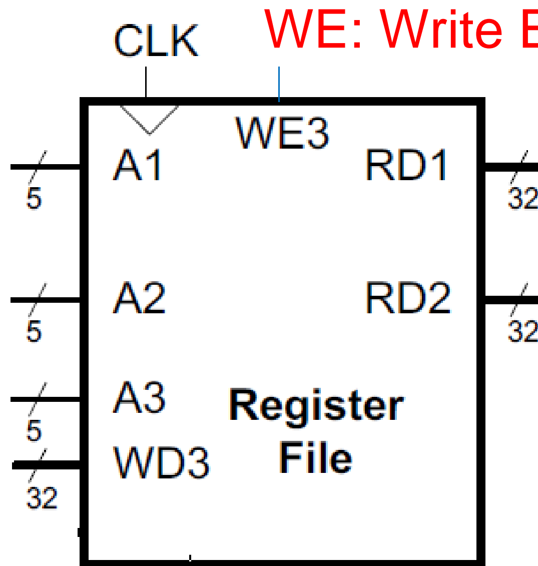
WE: Write Enable



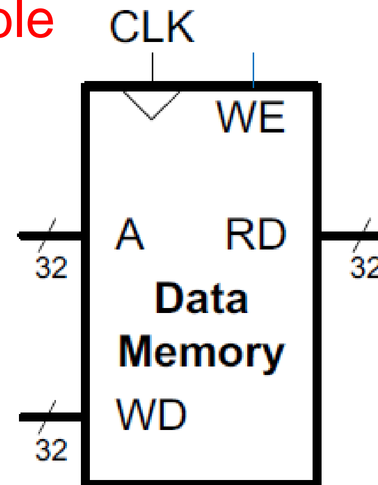
Program Counter



Instruction Memory (IMEM)

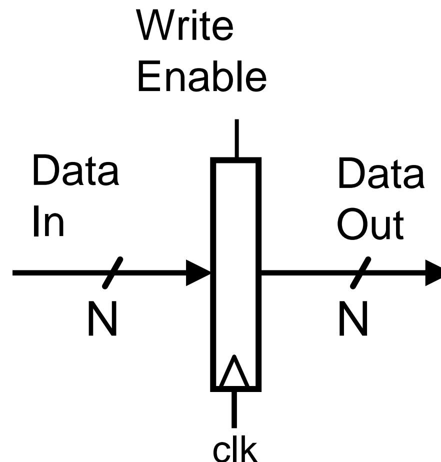


Register File



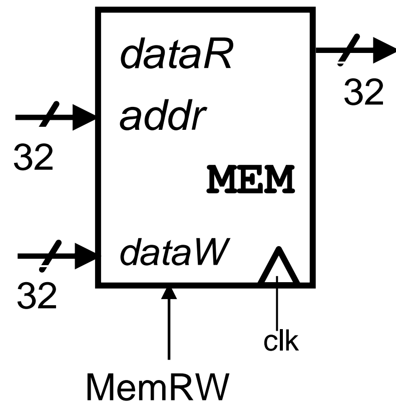
Data Memory (DMEM)

State Elements



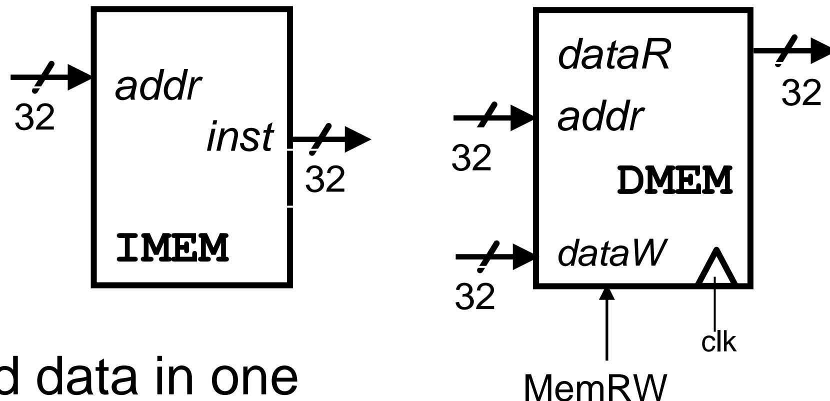
- **Program Counter**
  - The program counter is a 32-bit register
    - Hold the address of the current instruction
  - **Input**
    - Write Enable: “Control” bit (1: asserted/high, 0: de-asserted/0)
  - **Output**
    - N-bit data output bus
  - **Behavior**
    - If write enable is 1 on the rising clock edge, set Data Out = Data in
    - PC is written at the end of every clock cycle
    - No need a write control signal

State Elements



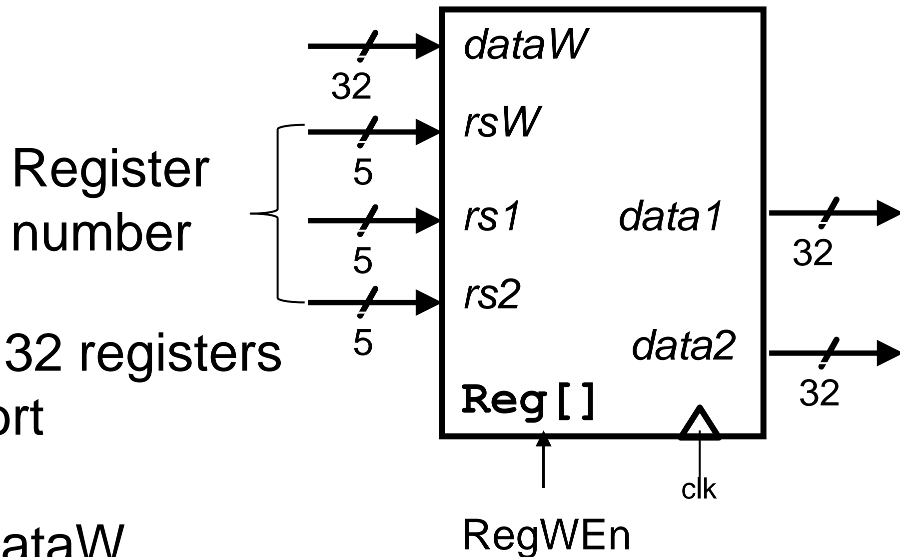
- **Memory**
  - 32-bit byte-addressed memory space
  - Memory access with 32-bit words
  - Memory words are accessed as follows
    - **Read**: Address addr selects word to put on dataR bus
    - **Write**: Set MemRW = 1
      Address (addr) selects word to be written with dataW bus
  - Like RegFile, clock input is only a factor on write
    - If MemRW = 1, write occurs on rising clock edge
    - If MemRW = 0 and address valid, then dataR valid after access time

State Elements



- **Two Memories (IMEM, DMEM)**
  - Memory holds both instructions and data in one contiguous 32-bit memory space
  - The processor will use two “separate” memories
    - **IMEM**: A read-only memory for fetching instruction
    - **DMEM**: A memory for loading (read) and storing (write) data words
  - Because IMEM is read-only, it behaves like a combinational block
    - If address valid, then instruction valid after access time
    - <u>No read control signal is needed</u>

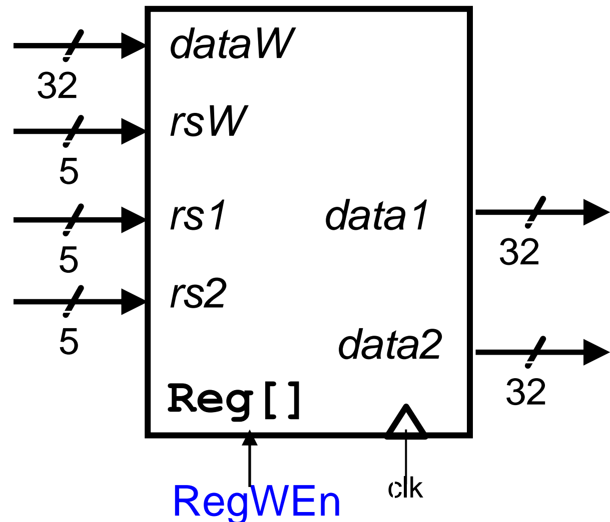
State Elements



- **Register File**
  - The Register File (RegFile) has 32 registers
  - Two read ports and one write port
  - **Input**
    - One 32-bit input data bus, dataW
    - Three 5-bit select busses, rs1, rs2, and rsW
  - **Output**
    - Two 32-bit output data busses, data1 and data2

National Yang Ming Chiao Tung University
Computer Architecture & System Lab

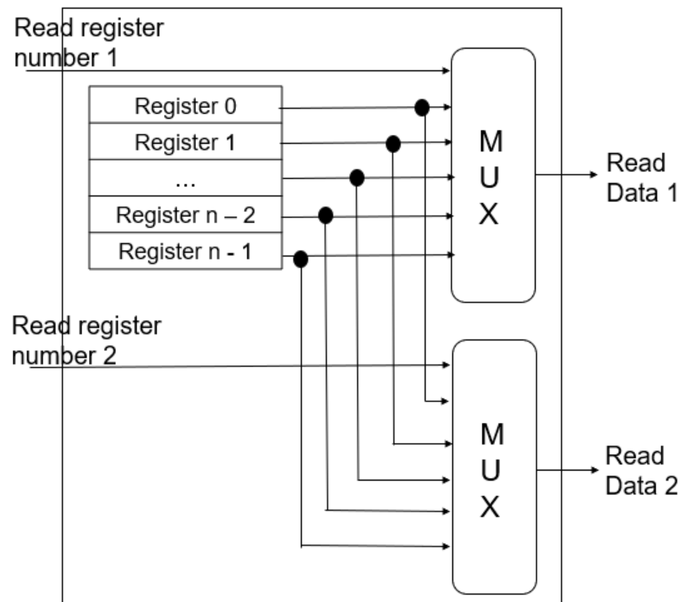
State Elements



- **Register File**
  - The inputs carrying the register number to the register file are all 5 bits wide
  - The lines carry data values are 32 bits wide
  - Clock behavior: Write operation occurs on rising clock edge
    - Legally read and write the same register within a clock cycle
    - The value written will be available to a read in a subsequent clock cycle
    - The read will get the value written in an earlier clock cycle
    - The register write must explicitly indicated by asserting the write control signal

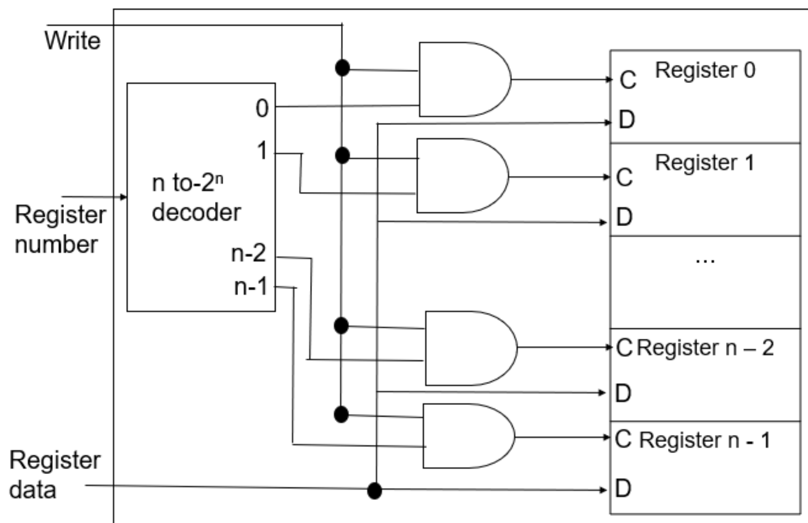
State Elements

- **Register File**
  - The implementation of two read ports for a register file with n registers can be done with a pair of n-to-1 multiplexors
  - Each 32 bits wide
  - The register read number signal
    - Used as the MUX selector signal



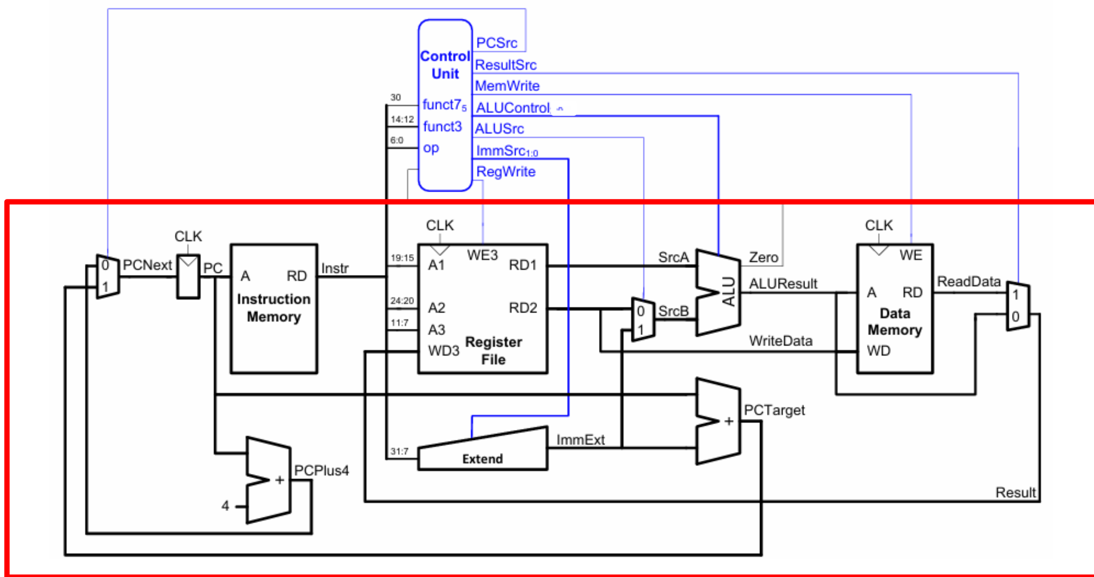
State Elements

- **Register File**
  - The write port for a register file is implemented with
    - A decoder with write signal to generate C input to registers



Build a Datapath

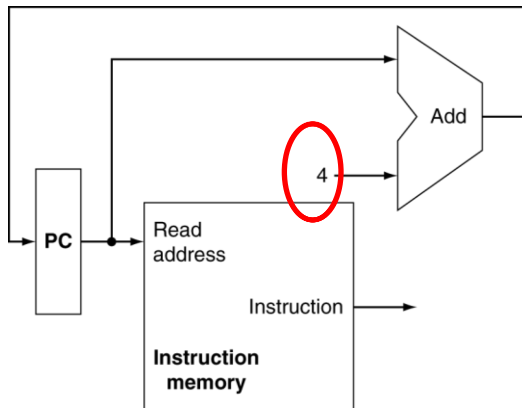
- Datapath elements on a CPU
  - A unit used to operate on or hold data within a processor
    - Registers, ALUs, Adder, instruction/data memories



Datapath

Instruction Fetch

- The first step is to
  - Read the instruction from instruction memory
  - Fetch instructions and increment the PC to obtain the address of the next sequential instruction
  - The program counter (PC) contains the address of the instruction to execute



Increment by 4 for next instruction

RISC-V Instructions

- R-type
  - add, sub, and, or, slt
- I-Type
  - lw
- S-Type
  - sw
- B-Type
  - beq

RISC-V Sample Program

- In a 32-bit processor
  - The PC begins at 0x1000, in fact, the PC is 0x00001000
  - Assume x5 initially contains the value 6 and x9 contains 0x2004
  - Memory location 0x2000 contains the value 10

Address	Instruction	Type	Fields	Machine Language
0x1000	L7: lw x6, -4(x9)	I	**$Imm_{11:0}$ rs1 f3 rd op** 1111 1111 1100 01001 010 00110 0000011	FFC4A303
0x1004	sw x6, 8(x9)	S	**$Imm_{11:5}$ rs2 rs1 f3 $imm_{4:0}$ op** 0000000 001100 01001 010 01000 0100011	0064A423
0x1008	or x4, x5, x6	R	**funct7 rs2 rs1 f3 rd op** 0000000 0010 00101 110 00100 0110011	0062E233
0x100c	beq x4, x4, L7	B	**$Imm_{12:10:5}$ rs2 rs1 f3 $imm_{4:1,11}$ op** 1111111 00100 00100 000 10101 1100011	FE430AE3

RISC-V Sample Program

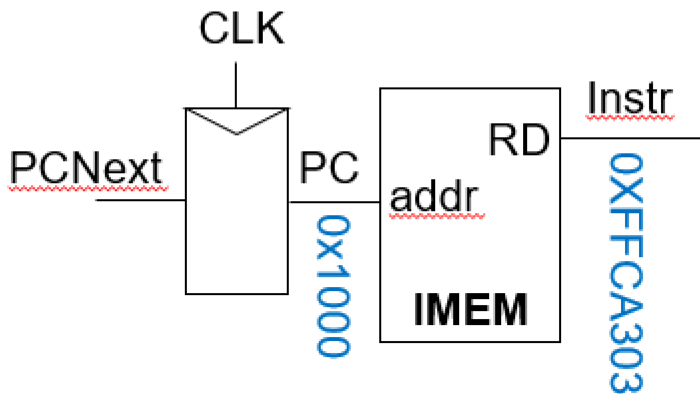
- In a 32-bit processor
  - The lw reads 10 from address (0x2004 - 4) and put it in x6
  - The sw writes 10 to address (0x2004 + 8 = 0x200C)
  - The or computes 6 | 10 = 0110_2 | 1010_2 = 1110_2 = 14

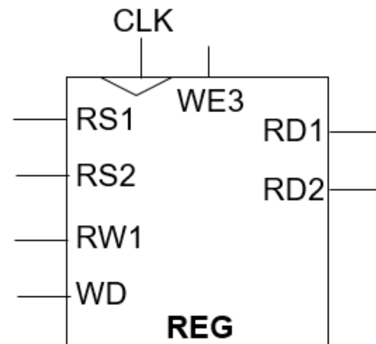
Address	Instruction	Type	Fields	Machine Language
0x1000	L7: lw x6, -4(x9)	I	**$Imm_{11:0}$ rs1 f3 rd op** 1111 1111 1100 01001 010 00110 0000011	FFC4A303
0x1004	sw x6, 8(x9)	S	**$Imm_{11:5}$ rs2 rs1 f3 $imm_{4:0}$ op** 0000000 001100 01001 010 01000 0100011	0064A423
0x1008	or x4, x5, x6	R	**funct7 rs2 rs1 f3 rd op** 0000000 0010 00101 110 00100 0110011	0062E233
0x100c	beq x4, x4, L7	B	**$Imm_{12:10:5}$ rs2 rs1 f3 $imm_{4:1,11}$ op** 1111111 00100 00100 000 10101 1100011	FE430AE3

Address	Instruction	Type	Fields					Machine Language
0x1000	L7: lw x6, -4(x9)	I	$Imm_{11:0}$	rs1	f3	rd	op	FFC4A303
			1111 1111 1100	01001	010	00110	0000011	

Load/Store Instructions : lw

- Steps for the lw instruction (I-Type)
  - 1. Fetch instruction from IMEM
  - 2. Read the source register containing the base address

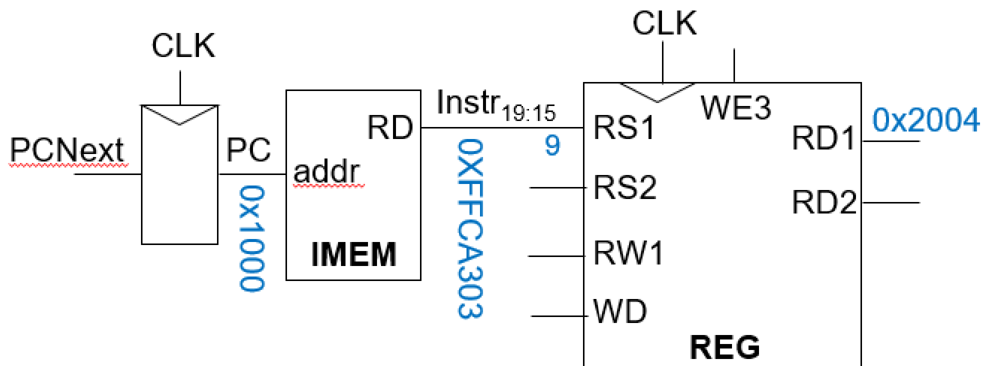




Address	Instruction	Type	Fields					Machine Language
			$Imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	1111 1111 1100	01001	010	00110	0000011	FFC4A303

Load/Store Instructions : lw

- Steps for the lw instruction (I-Type)
  - 2. Read the source register containing the base address
    - The base register is specified in the rs1 field ($Instr_{19:15}$)
    - These bits connect to the RS1 (x9) address of register file
    - The register file reads the register value (0x2004) onto RD1



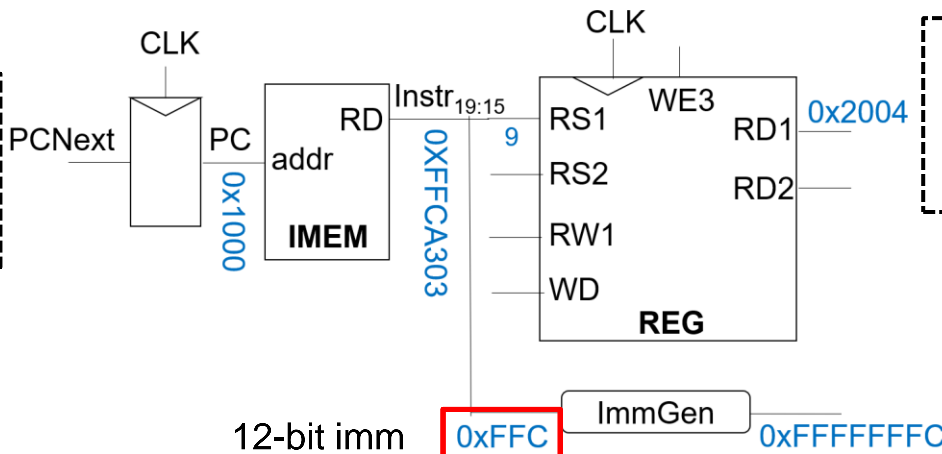
Address	Instruction	Type	Fields					Machine Language
			$Imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	1111 1111 1100	01001	010	00110	0000011	FFC4A303

Load/Store Instructions : lw

- Steps for the lw instruction (I-Type)
  - 2. Read the source register containing the base address
    - The lw instruction requires an offset
    - The offset is stored in the 12-bit immediate field ($Instr_{31:20}$)
    - The offset is a signed value and must be signed-extended to 32-bit

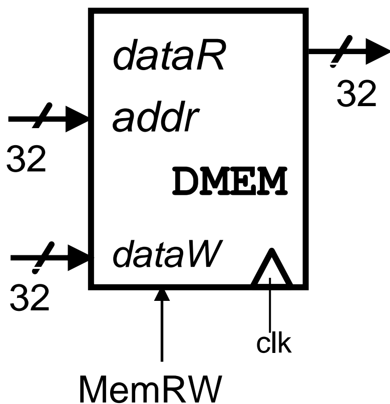
Offset: $4_{ten} = 0100_2$
1011_2 (1's complement of 4_{ten})

Offset: $-4_{ten} = 1100_2$ (2's complement) = $0xC_{hex}$

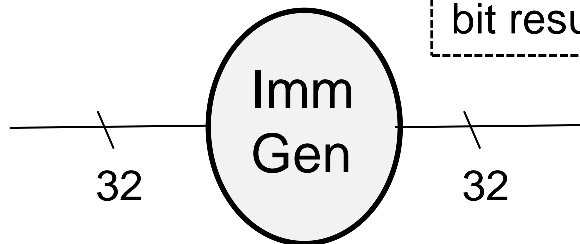


Load/Store Instructions : lw

- Read register operands
- Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



ImmGen has a 32-bit instruction as input that select 12-bit field for load, store, and beq that is sign-extended into a 32-bit result

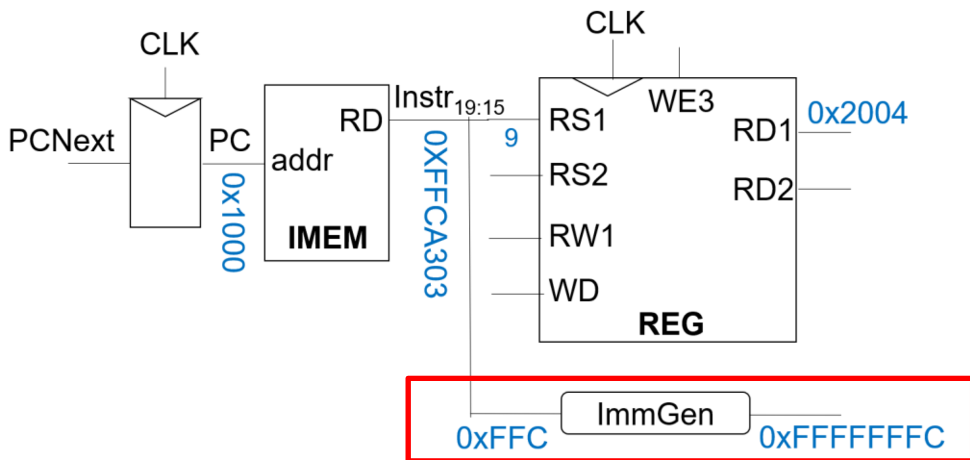


Immediate generation unit

Address	Instruction	Type	Fields					Machine Language
			$Imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	1111 1111 1100	01001	010	00110	0000011	FFC4A303

Load/Store Instructions : lw

- Steps for the lw instruction (I-Type)
  - 2. Read the source register containing the base address
    - Sign extension simply means copying the sign bit into MSB
    - Two's complement immediate -4 is 0xFFC in 12-bit

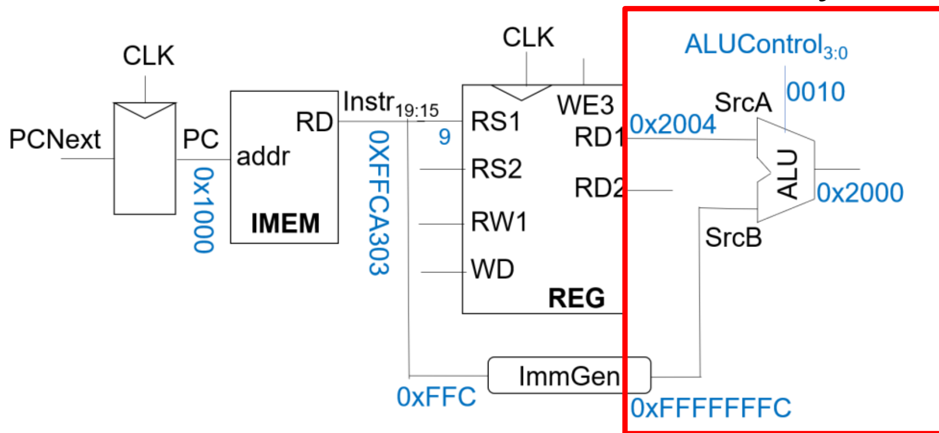


$4_{10} = 0100_2$
$-4_{10} = 1100_2 = 0xC$

Address	Instruction	Type	Fields					Machine Language
			$Imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	1111 1111 1100	01001	010	00110	0000011	FFC4A303

Load/Store Instructions : lw

- Steps for the lw instruction (I-Type)
  - 3. ALU adds the base address to the offset
    - The 4-bit ALUControl specifies the operation (ADD (0010))
    - ALU receives 32-bit operands and generates a 32-bit ALUResult
    - ALUResult is sent to data memory as the address to read



0x2004 + 0xFFFFFFFC
= 0x2000

Address	Instruction	Type	Fields					Machine Language
0x1000	L7: lw x6, -4(x9)	I	$Imm_{11:0}$	rs1	f3	rd	op	FFC4A303
			1111 1111 1100	01001	010	00110	0000011	

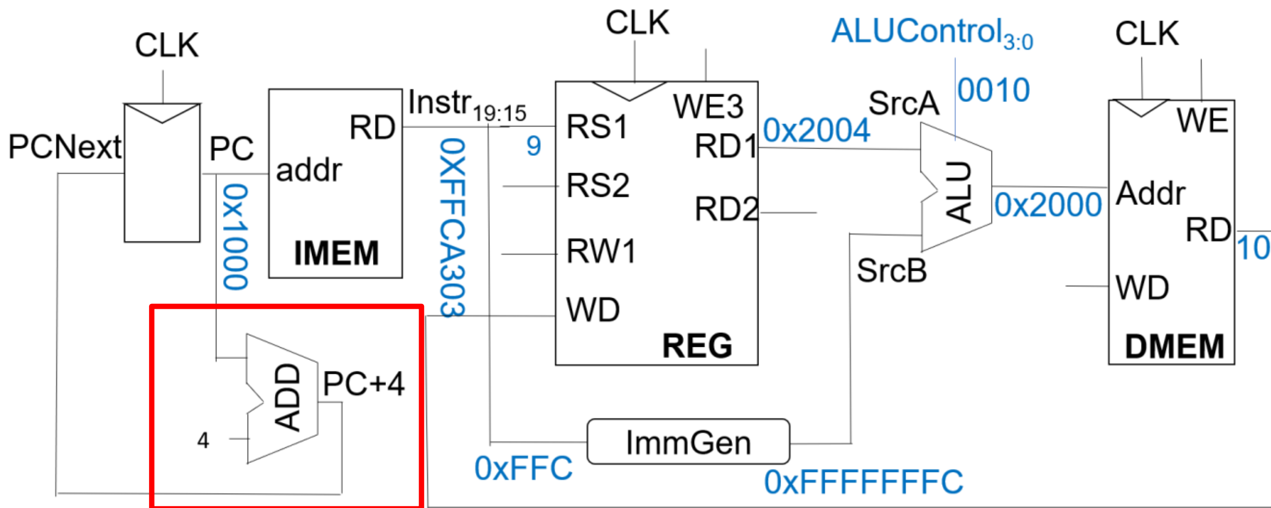
Load/Store Instructions : lw

- Steps for the lw instruction (I-Type)
  - 4. The memory address form ALU is provided to DMEM addr port
    - The data is read from the DMEM onto the ReadData bus
    - Write back to rd register through RD port in REG

Address	Instruction	Type	Fields					Machine Language
			$Imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	1111 1111 1100	01001	010	00110	0000011	FFC4A303

Load/Store Instructions : lw

- Steps for the lw instruction (I-Type)
  - 5. Compute the address of the next instruction (PCNext)
    - The next instruction is at PC + 4, because instruction is 4 bytes
    - PCNext = 0x1000 + 4 = 0x1004

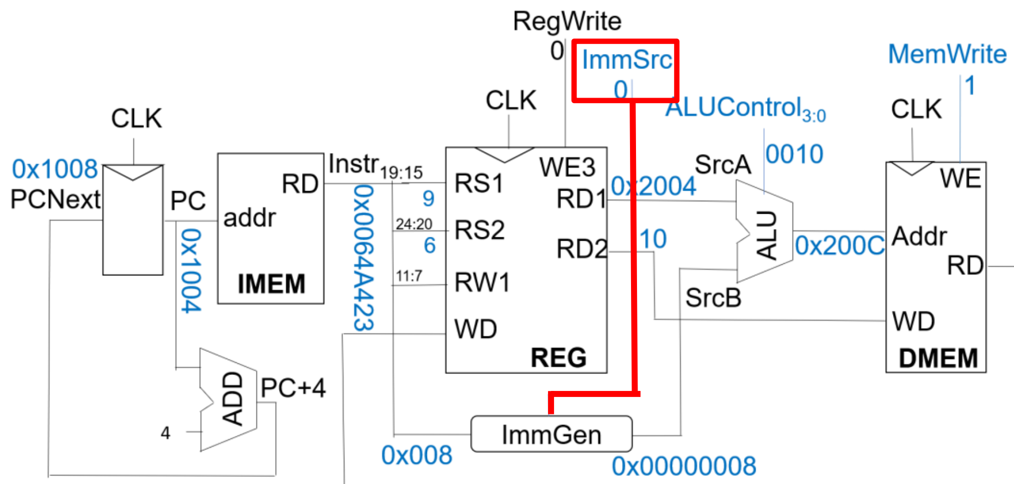


The new address is written into PC on the next rising edge of the clock

Address	Instruction	Type	Fields						Machine Language
			$Imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423

Load/Store Instructions : sw

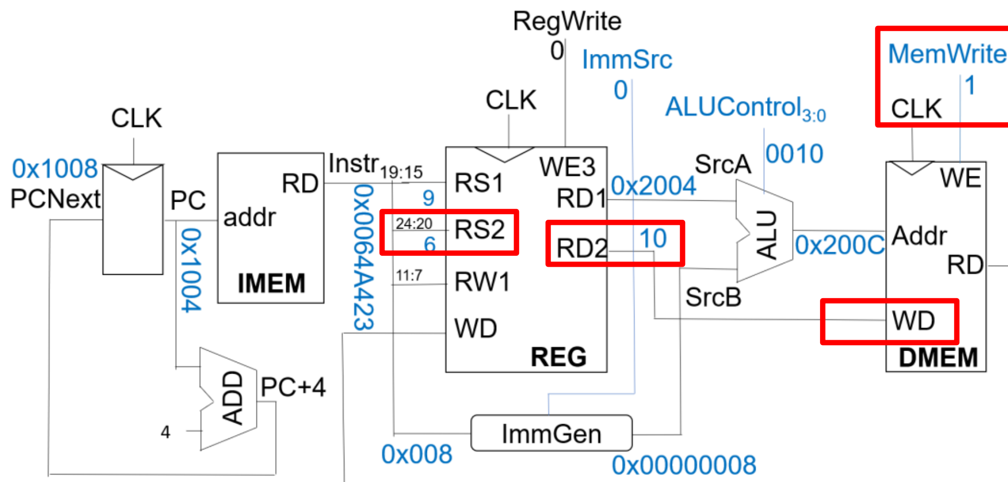
- Steps for the sw instruction (S-Type)
  - The 12-bit signed immediate is stored in $Instr_{31:25,11:7}$
  - A control signal (ImmSrc) decides which instr bits to use as the imm
  - ImmSrc = 0 is for lw. ImmSrc = 1 is for sw



Address	Instruction	Type	Fields						Machine Language
			$Imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	001100	01001	010	01000	0100011	0064A423

Load/Store Instructions : sw

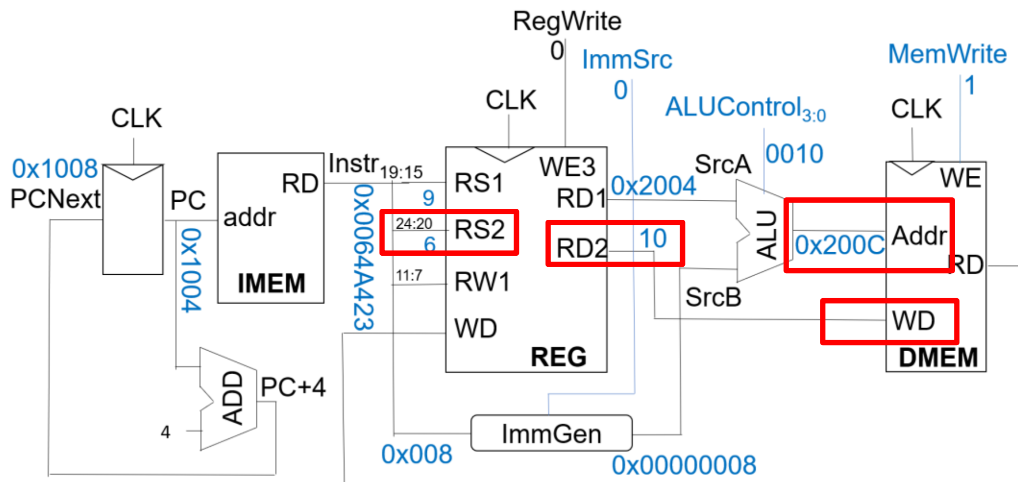
- Steps for the sw instruction (S-Type)
  - Read a second register form REG and write its contents to DMEM
  - The write enable port (WE) of the DMEM is controlled by MemWrite
  - RegWrite = 0, because nothing should be written to REG



Address	Instruction	Type	Fields	Machine Language
0x1004	sw x6, 8(x9)	S	$\text{Imm}_{11:5}$ rs2 rs1 f3 $\text{imm}_{4:0}$ op 0000000 001100 01001 010 01000 0100011	0064A423

Load/Store Instructions : sw

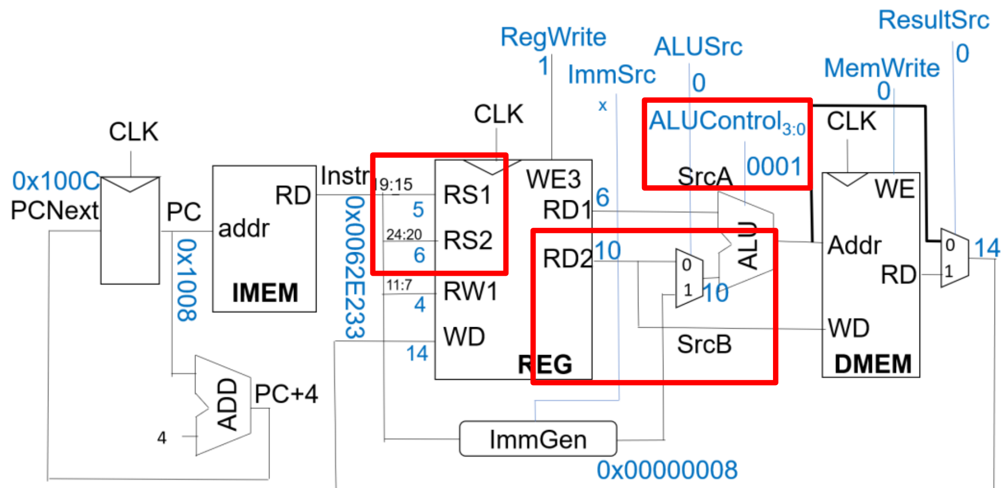
- Steps for the sw instruction (S-Type)
  - The register reads 0x2004 (base address) from x9 and 10 from x6
  - The ImmGen extends the offset 8 from 12 to 32 bits
  - The ALU computes 0x2004 + 8 = 0x200C



Address	Instruction	Type	Fields						Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	0010	00101	110	00100	0110011	0062E233

R-Type Instructions : or

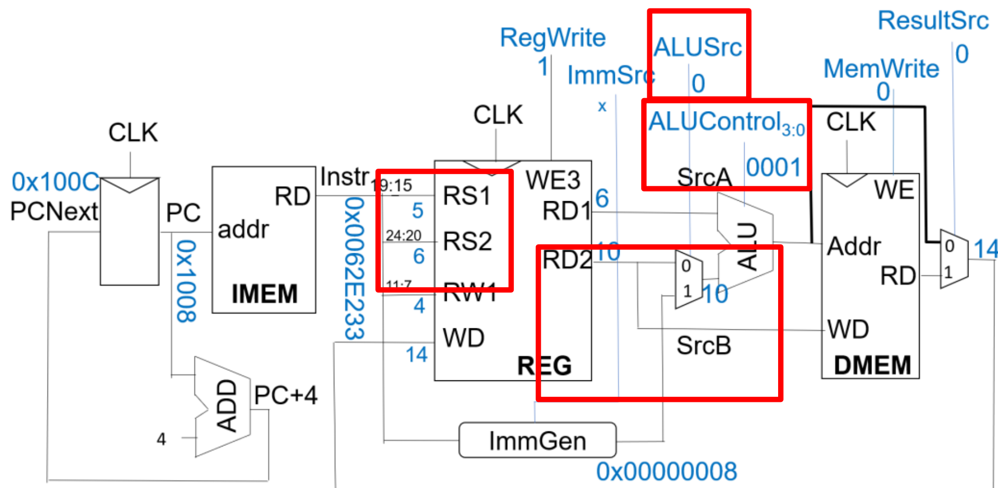
- Steps for the or instruction (R-Type)
  - ALUControl (0001) is for the “or” instruction
  - The datapath reads rs1 and rs2 from ports 1 and 2 of REG
  - ALUSrc selects between ImmExt and RD2 as the second source of ALU



Address	Instruction	Type	Fields						Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	0010	00101	110	00100	0110011	0062E233

R-Type Instructions : or

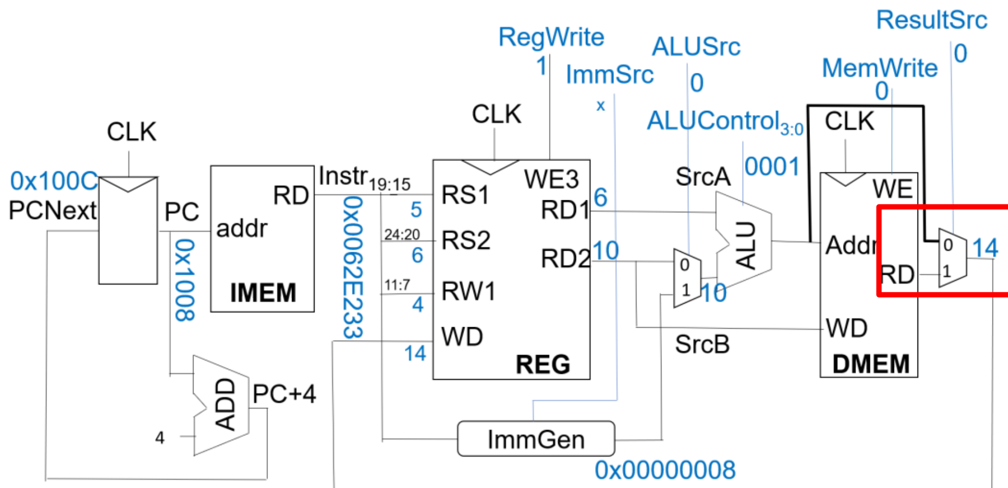
- Steps for the or instruction (R-Type)
  - ALUControl (0001) is for the “or” instruction
  - The datapath reads rs1 and rs2 from ports 1 and 2 of REG
  - ALUSrc selects between ImmExt and RD2 as the second source of ALU



Address	Instruction	Type	Fields						Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	0010	00101	110	00100	0110011	0062E233

R-Type Instructions : or

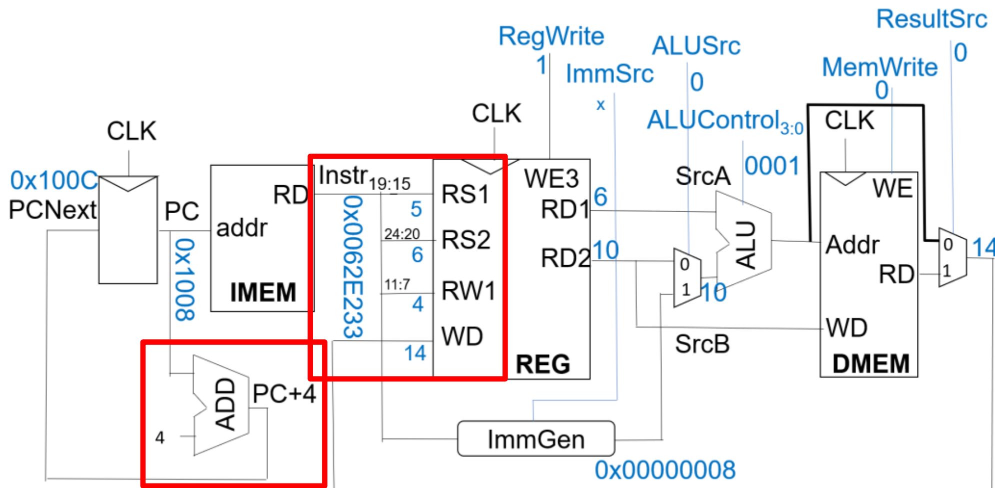
- Steps for the or instruction (R-Type)
  - Result comes from the ALUResult output of the ALU
  - Add the Result MUX to choose the proper Result based on the type of instruction (ResultSrc = 0 for R-type, = 1 for I-type (lw))



Address	Instruction	Type	Fields						Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	0010	00101	110	00100	0110011	0062E233

R-Type Instructions : or

- Steps for the or instruction (R-Type)
  - The register reads source operands 6 from x5 and 10 from x6
  - ALU computes 6 | 10 = 14
  - The result is written back to x4, the PCNext = 0x1008 + 4 = 0x100C



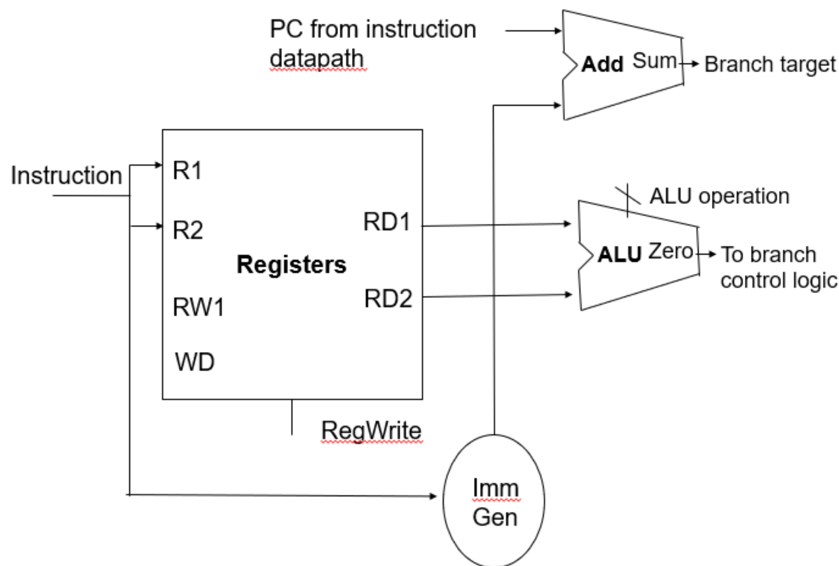
Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero flag
- Calculate target address
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value
- Branch taken
  - A branch where the branch condition is satisfied and PC becomes the branch target (jump to the target label)

```
  beq x5, x6, target
  addi x7, x0, 1
target:
   addi x7, x0, 2
```

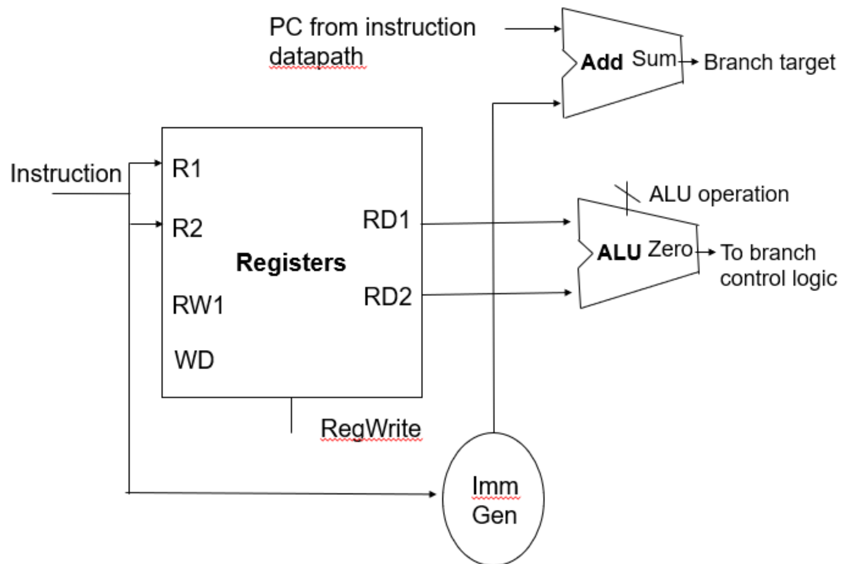
Branch Instructions

- The branch datapath must do two operations
  - Compute the branch target address (Adder)
  - Test the register contents (ALU)
    - If the Zero signal (flag) ofthe ALU is asserted
    - The register values are equal



Branch Instructions

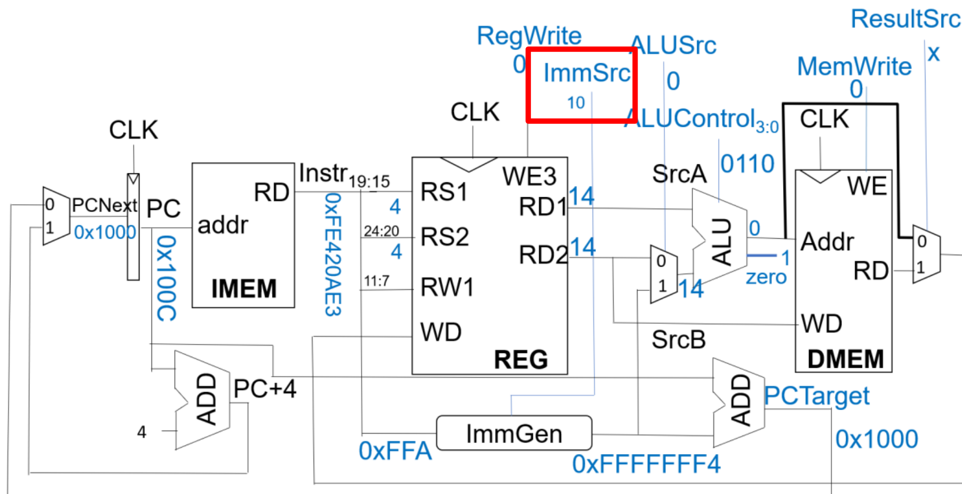
- The portion of a datapath for a branch
  - Uses the ALU to evaluate the branch condition (“==”, “<”, “!=” ..)
  - Uses the adder to compute the branch target as the sum of PC and imm
  - Control logic decides
    - increment PC or
    - branch target should replace the PC
    - Based on zero output of ALU



Address	Instruction	Type	Fields					Machine Language
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$ op	
0x100c	beq x4, x4, L7	B	1111111	00100	00100	000	10101 1100011	FE430AE3

B-Type Instructions : beq

- Steps for the beq instruction (B-Type)
  - The branch offset is a 13-bit signed immediate stored in the 12-bit immediate field of the B-type instruction
    - ImmSrc is increased to 2-bits

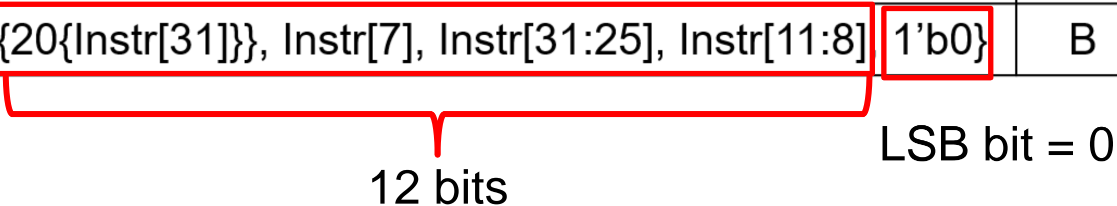


Address	Instruction	Type	Fields						Machine Language
0x100c	beq x4, x4, L7	B	**$Imm_{12:10:5}$**	**rs2**	**rs1**	**f3**	**$imm_{4:1,11}$**	**op**	FE430AE3
			1111111	00100	00100	000	10101	1100011	

B-Type Instructions : beq

- Steps for the beq instruction (B-Type)
  - ImmExt is either
    - The sign-extended immediate (ImmSrc = 00 or 01) or
    - The branch offset (ImmSrc = 10)

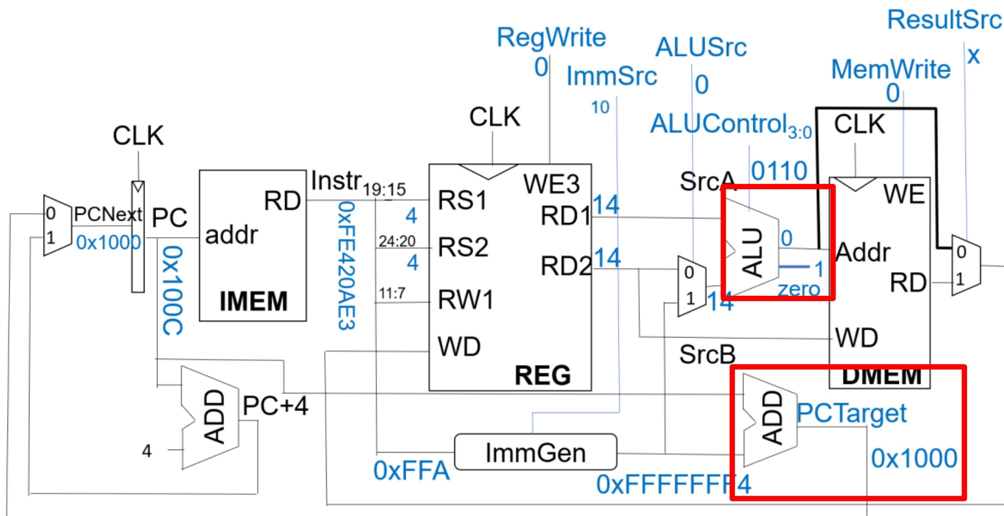
ImmSrc	ImmExt	Type	Description
00	{{20{Instr[31]}}, Instr[31:20]}	I	12-bit signed Immediate
01	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	S	12-bit signed Immediate
10	{{20{Instr[31]}}, Instr[7], Instr[31:25], Instr[11:8], 1'b0}	B	13-bit signed Immediate



Address	Instruction	Type	Fields						Machine Language
			$imm_{12:10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100c	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE430AE3

B-Type Instructions : beq

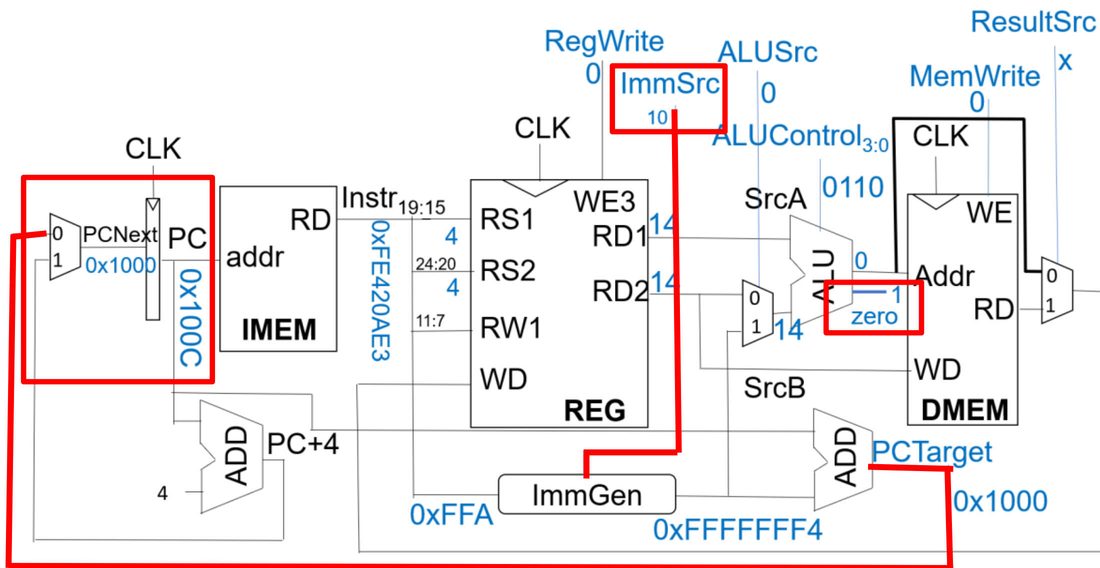
- Steps for the beq instruction (B-Type)
  - PCTarget = PC + ImmExt
  - The two source registers are compared by compute (SrcA - SrcB)
    - If ALUResult = 0 (Zero flag), the registers are equal



Address	Instruction	Type	Fields						Machine Language
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100c	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE430AE3

B-Type Instructions : beq

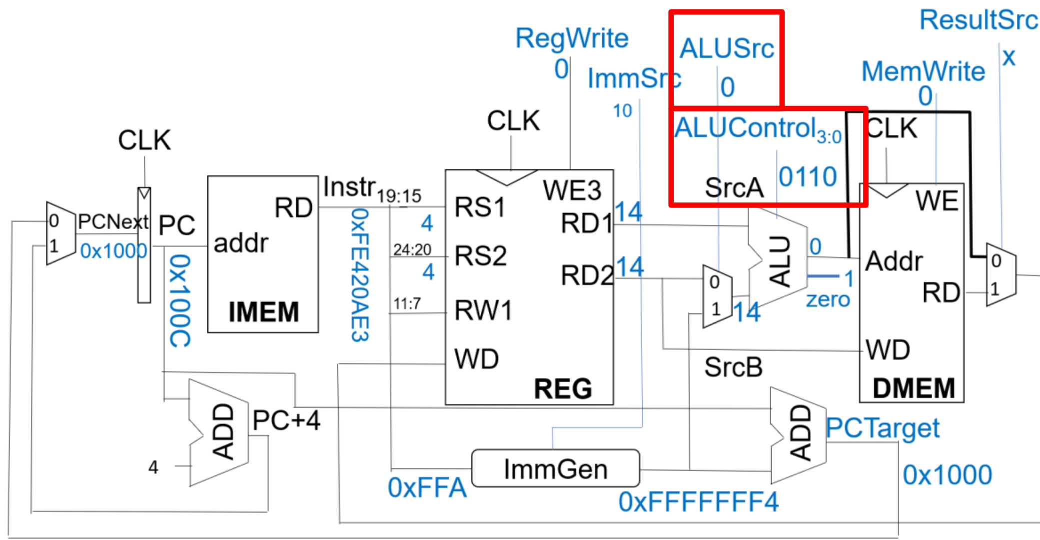
- Steps for the beq instruction (B-Type)
  - Add a MUX to choose PCNext from either PC+4 or PCTarget
    - PCTarget is selected if the instruction is branch and the Zero (Z) flag is asserted



Address	Instruction	Type	Fields						Machine Language
			$imm_{12:10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100c	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE430AE3

B-Type Instructions : beq

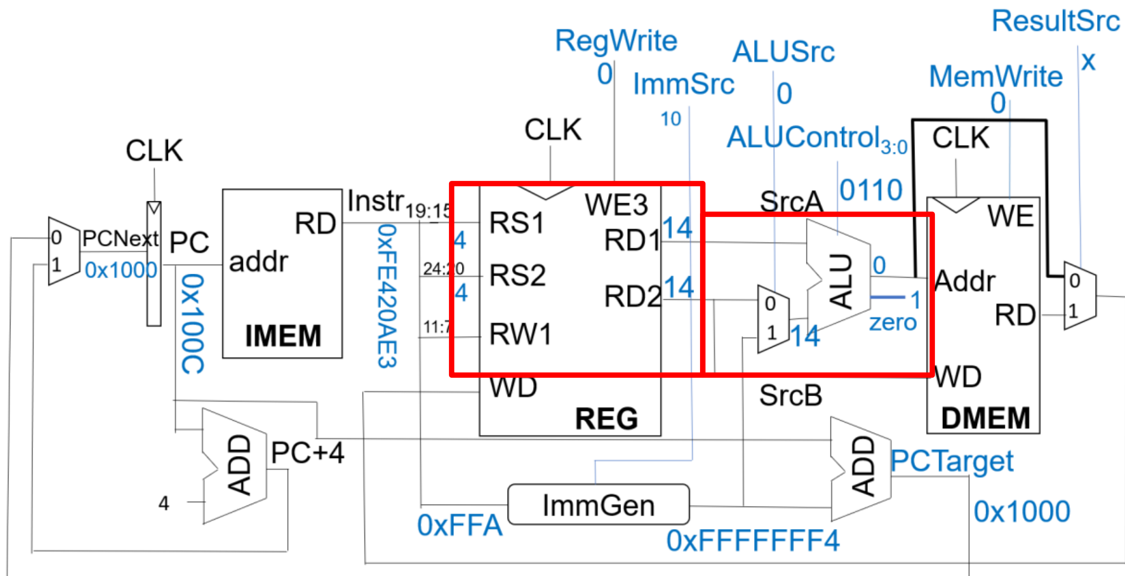
- Steps for the beq instruction (B-Type)
  - ALUControl = 0110, so that the ALU performs a subtraction
  - ALUSrc = 0 to choose SrcB from the register file
  - RegWrite and MemWrite = 0, because a branch doesn't write to REG or DMEM



Address	Instruction	Type	Fields						Machine Language
0x100c	beq x4, x4, L7	B	$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	FE430AE3
			1111111	00100	00100	000	10101	1100011	

B-Type Instructions : beq

- Steps for the beq instruction (B-Type)
  - Both source registers are x4, so the REG reads 14 on both ports
  - ALU computes 14 – 14 = 0, and Z flag is asserted

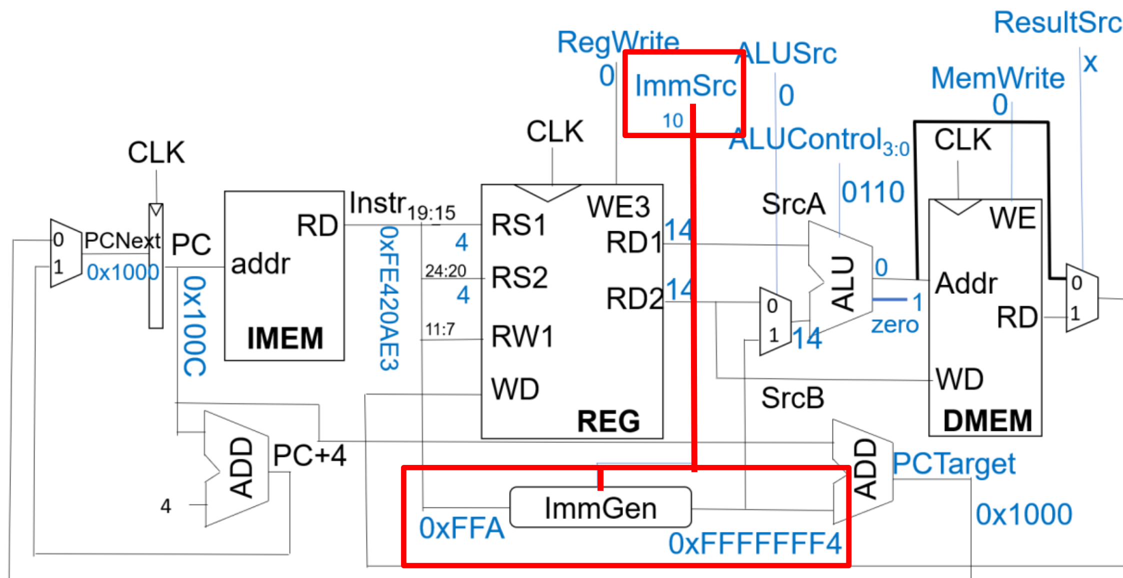


Address	Instruction	Type	Fields						Machine Language
			$imm_{12:10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100c	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE430AE3

B-Type Instructions : beq

ImmSrc	ImmExt	Type	Description
10	{{20{Instr[31]}}, Instr[7], Instr[31:25], Instr[11:8], 1'b0}	B	13-bit signed Immediate

- Steps for the beq instruction (B-Type)
  - The ImmGen produces 0xFFFFFFF4 (-12), which is added to PC to obtain PCTarget = 0x1000 (the address of L7 label)



Imm_B = 1111 1111 1010
F F A

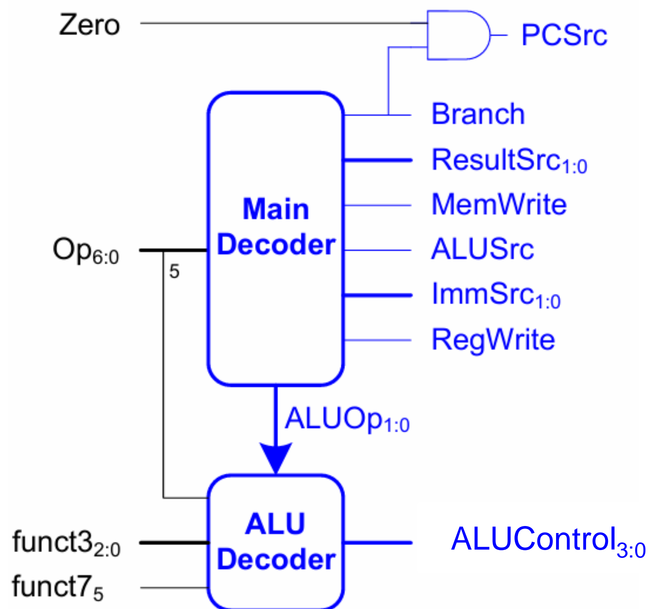
Imm_A = 1111 1111 0100
F F 4 1'b0

0xFFF4 + 0x100C=
1111 1111 1111 0100
0001 0000 0000 1100
0001 0000 0000 0000 (0x1000)

Designing the Main Control Unit

- Controller (Decoder)
  - Decodes what the instruction should do
  - Main decoder
    - Decide the instruction type from the opcode
    - Produce proper control signals for datapath

Instruction	Op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
lw	0000011	1	00	1	0	1	0	00
sw	0100011	0	01	1	1	X	0	00
R-Type	0110011	1	xx	0	0	0	0	10
beq	1100011	0	10	0	0	X	1	01



Designing the Main Control Unit

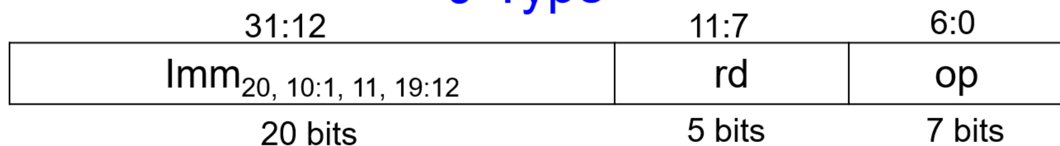
- Controller (Decoder)
  - ALU Decoder produces ALUControl based on ALUOp and $func_3$
  - In “sub” and “add” instruction
    - ALU Decoder also uses $funct7_5$ and op_5 to decide ALUControl

ALUOp	Funct3	$\{Op_5, Funct7_5\}$	ALUControl	Instruction
00	X	X	0010 (add)	lw, sw
01	X	X	0110 (subtract)	beq
10	000	00, 01, 10	0010 (add)	add
	000	11	0110 (subtract)	sub
	010	X	0111 (set less than)	slt
	110	X	0001 (or)	or
	111	X	0000 (and)	and

J-Type Instructions : jal

- Steps for the jal instruction (J-Type)
  - jal writes PC+4 to rd and changes PC to the jump target address, PC + imm
  - The LSB bit of immediate is always 0
  - The 21-bit immediate is sign-extended

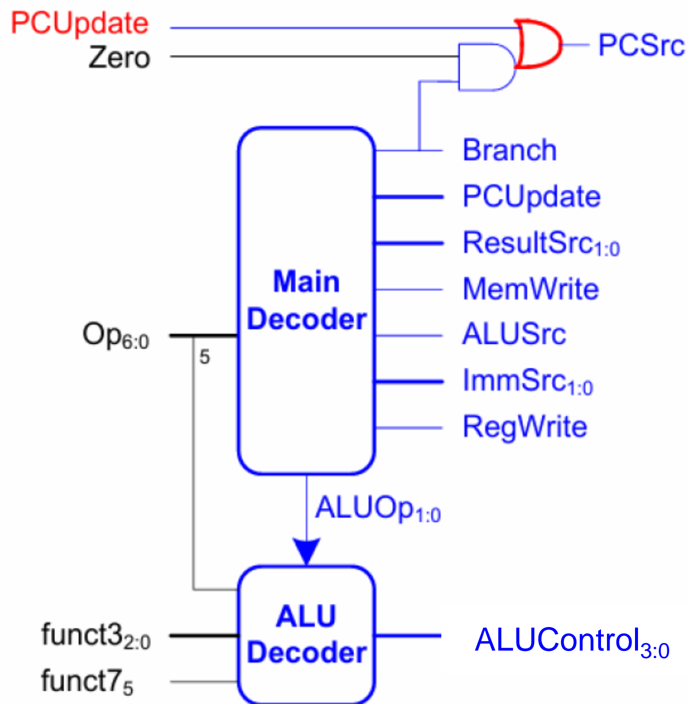
ImmSrc	ImmExt	Type	Description
11	{{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0}	J	21-bit signed Immediate



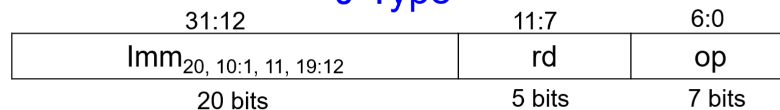
`jal rd, offset`

J-Type Instructions : jal

- Steps for the jal instruction (J-Type)
  - The control unit needs to set PCSrc = 1 for the jump
  - Add an OR gate and another control signal PCUpdate
  - When PCUpdate asserts, PCSrc = 1 and PCTarget is selected as the next PC



J-Type Instructions : jal



`jal rd, offset`

- Steps for the jal instruction (J-Type)
  - RegWrite = 1 and ResultSrc = 10 to write PC + 4 into rd
  - ImmSrc = 11 to select the 21-bit jump offset
  - ALUSrc and ALUOp don't matter, because the ALU is not used
  - MemWrite = 0, because instruction isn't a store

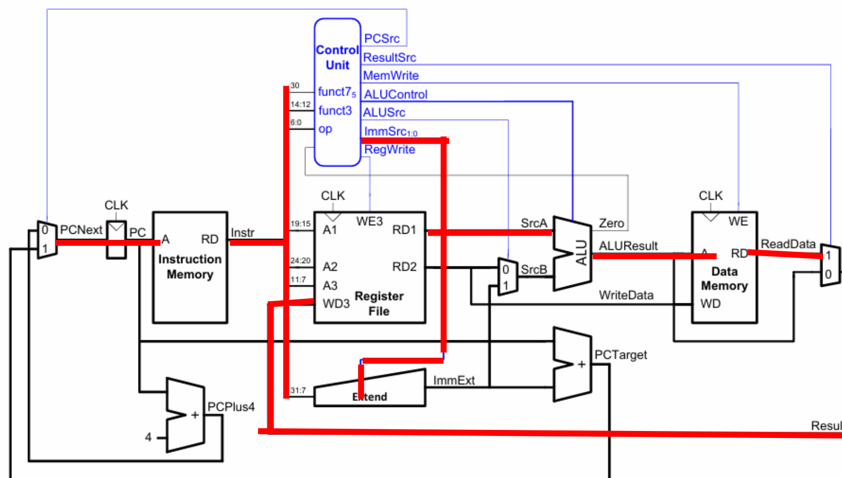
Instruction	Op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	PCUpdate
lw	0000011	1	00	1	0	01	0	00	0
sw	0100011	0	01	1	1	XX	0	00	0
R-Type	0110011	1	xx	0	0	00	0	10	0
beq	1100011	0	10	0	0	XX	1	01	0
jal	1101111	1	11	X	0	10	0	xx	1

Single-Cycle RISC-V Processor

Performance Analysis

- In the single-cycle processor
  - Each instruction takes one clock cycle
  - The clock cycles per instruction (CPI) is 1
  - The cycle time is set by the critical path
    - lw is the most time-consuming

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$



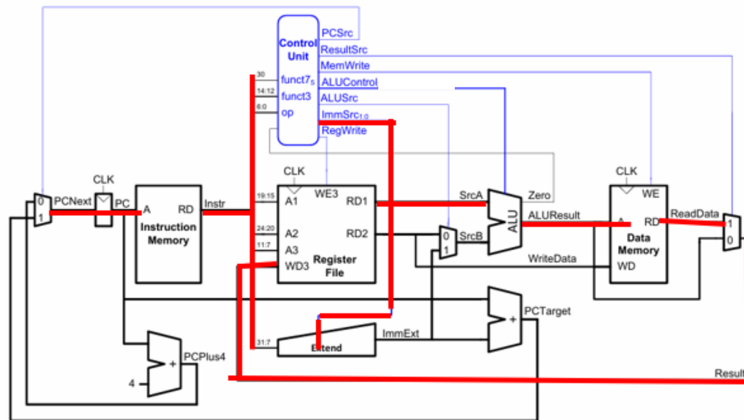
Performance Analysis

- In the single-cycle processor
  - The cycle time of the single-cycle processor

    $T_{c_single} = t_{pcq_PC} + t_{mem} + \max[t_{RFread}, t_{dec} + t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$

  - The critical is through the register t_{RFread}

    $T_{c_single} = t_{pcq_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$



Performance Analysis

- In the single-cycle processor
  - What is the execution time for a program with 100 billion instructions?

$T_{c_single} = t_{pcq_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$
$= 40 + 2(200) + 100 + 120 +30 + 60 = 750$ ps

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{AND\text{-}OR}$	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register File read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Performance Analysis

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- In the single-cycle processor
  - What is the execution time for a program with 100 billion instructions?

$$T_{single} = (100 \times 10^9 \text{ instructions}) \times (1 \text{ cycle/instruction}) \times (750 \times 10^{-12} \text{ s/cycle}) = 75 \text{ seconds}$$

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{AND\text{-}OR}$	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register File read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Multicycle Processor

- The single-cycle processor has three weakness
  - Separate memories for instructions and data
    - Most processors have only a single external memory holding both instructions and data
  - The cycle time is limited by the instructions with the longest cycles
  - Adders are relatively expensive circuit
    - Three adders (One in the ALU, two for the PC logic)

Multicycle Processor

- Multicycle processor
  - Breaking an instruction into multiple shorter steps
  - Keep the delay for each short step approximately equal
  - The processor can use only one of those units in each step
  - Use a single memory
    - The instruction is read in one step
    - Data is read or written in a later step
  - Only one adder
    - Reuse for different purposes one different steps

Performance Analysis

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- In the multi-cycle processor
  - The SPECINT2000 benchmark consists of 25% load, 10% stores, 11% branches, 2% jumps, and 52% R- or I-type instructions
  - What is the average CPI for this benchmark?
    - CPI for branch is 3, for loads is 5, for R/I/jumps/stores is 3
    - Average weighted CPI = (0.11)(3) + (0.1+0.02+0.52)(4) + (0.25)(5) = 4.14

Performance Analysis

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- In the multi-cycle processor
  - Shorten the critical path of the instruction execution by
    - Breaking instruction exec. path into multiple steps
    - Unified memory
    - Reduce # of adders

$$T_{c_single} = t_{pcq_PC} + t_{dec} + \max[t_{ALU} + t_{mem}] + 2t_{mux} + t_{setup}$$
$$= 40 + 25 + 2(30) + 200 + 50 = 350 \text{ ps}$$

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{AND\text{-}OR}$	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register File read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Performance Analysis

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- In the multi-cycle processor
  - What is the execution time for a program with 100 billion instructions?
  - Multi-cycle processor
    - Might slower than single-cycle processor
    - Reduces the critical path of instruction execution, but significantly increases CPI

T_{single} = (100 x 10^9 instructions) x (4.14 cycle/instruction) x (350 x 10^{-12} s/cycle)
= 155 seconds

Conclusion

- ALU
- State Element
- Design the Datapath
- Design the Control Unit