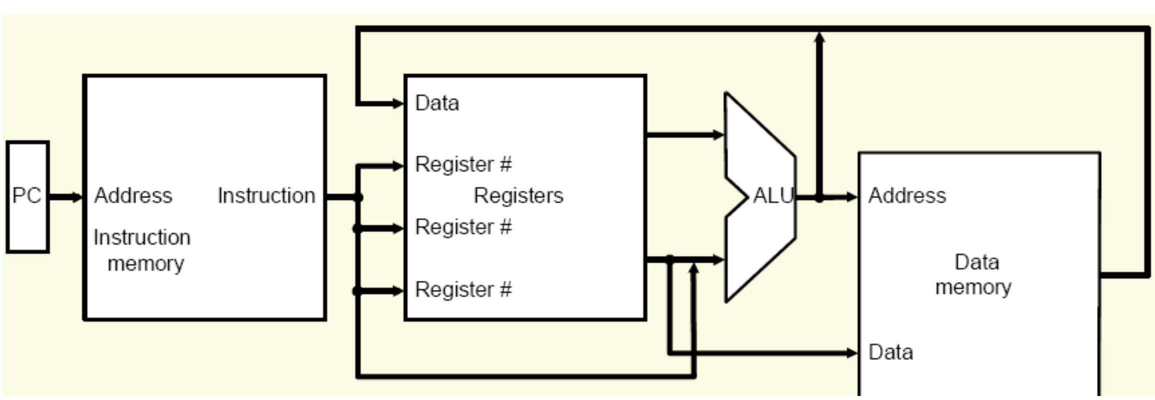
Ch5: Processor Design (Datapath and Control)

Introduction

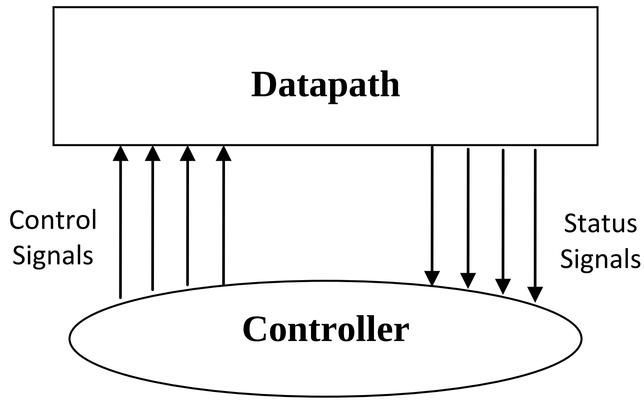
- The performance of a computer is determined by:
    - Instruction count
    - Clock cycle time
    - Clock cycles per instruction (CPI)
- The clock cycle time and the CPI are determined by the implementation of the processor.
- MIPS subset for implementation
    - Arithmetic-logic instructions (add, sub, and, or, slt)
    - Memory reference instructions (lw, sw)
    - Control flow instructions (beq, j)
- Generic Implementation
    - Send the program counter (PC) to the memory location that contains the code and fetch the instruction from that memory location.
    - Read one or two registers, using fields of the instruction to select the registers to read. For the load word and store word instructions we need to read only one register, but most other instructions require that we read two registers.
    - Perform the operation required by the instruction using the ALU.
        - All instructions use the ALU after reading the registers. Why?
            - Memory-reference instructions use the ALU for address calculation;
            - Arithmetic-logical instructions for operation execution; and
            - Branches for comparison.
    - Store the result in registers or memory locations, and change the value of the program counter in case of a branch instruction.

Design Overview



- The processor design consists of two parts

- Datapath
- Control



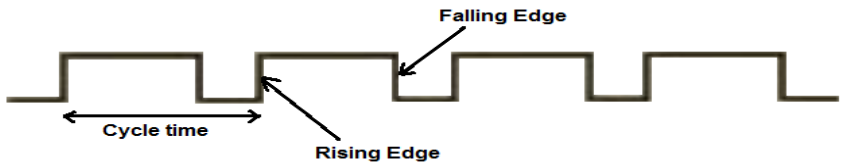
Building block types

- Two types of functional units:
  - Elements that operate on data values (combinational)
    - Output is function of current input
    - No memory
  - Elements that contain state (sequential)
    - Output is function of current and previous inputs
    - State = memory
- Combinational circuit examples
  - Gates: and, or, nand, nor, xor, inverter
  - Multiplexer
  - Decoder
  - Adder, subtractor, comparator
  - ALU
  - Array Multipliers
- Sequential circuit examples
  - Flip-flops
  - Counters (extends from one dimensional flip-flops)
  - Registers (extends from one dimensional flip-flops)
  - Registers files (extends from two dimensional flip-flops)
  - Memories (extends from two dimensional flip-flops)

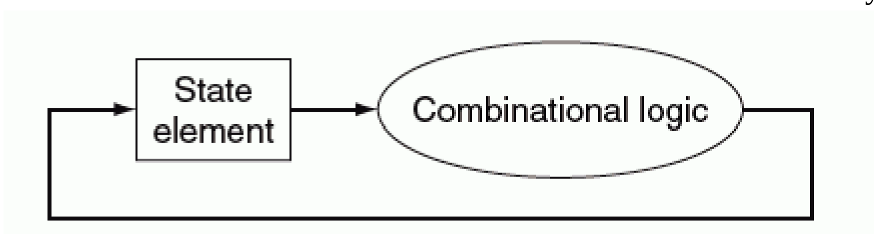
Clocked vs. Unclocked Circuit

- A **clock** is a free-running signal with a fixed **cycle time** (or called **clock period**) or, equivalently, a fixed **clock frequency** (i.e., inverse of the cycle time). Clocks are needed in sequential logic to decide when an element that contains state should be updated.

- Clocked state element
  - State changes only with clock edge
  - Example: Flip-flop
  - Edge-triggered clocking: rising edge vs. falling edge.
  - Clocked systems are also called synchronous systems.
- Unclocked state element
  - State changes can occur with changes in other inputs
  - Example: Latch



- **Note: Refresh your knowledge in latches, clocks, DFFs, registers, decoders, …**
- An edge triggered methodology: means that any values stored in a sequential logic element are updated only on a clock edge
- Typical execution:
  - read contents of some state elements (Registers) at the beginning of the clock cycle,
  - send values through some combinational logic,
  - write results to one or more state elements at the end of the clock cycle.



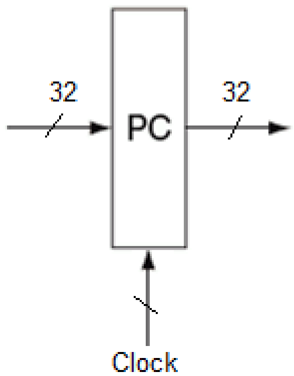
- An edge triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could to indeterminate data.

Single Cycle Design

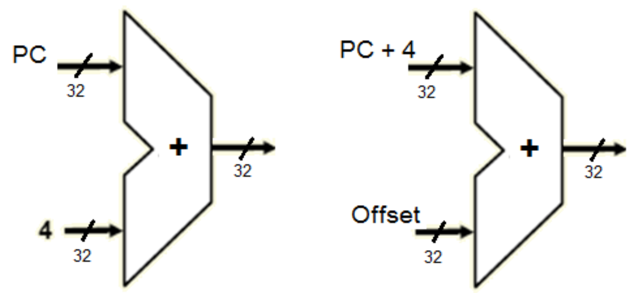
- We shall first design a simpler processor that executes each instruction in only one clock cycle time.
- This is not efficient from performance point of view, since:
  - a clock cycle time (i.e. clock rate) must be chosen such that the longest instruction can be executed in one clock cycle
  - makes shorter instructions execute in one unnecessary long cycle.
- Additionally, no resource in the design may be used more than once per instruction, thus some resources will be duplicated.
- Because of that, the singe cycle design will require:
  - two memories (instruction and data),
  - two additional adders.

Components for MIPS subset

- Register
- Adder
- ALU
- Multiplexer
- Register file
- Program memory
- Data memory
- Bit manipulation components



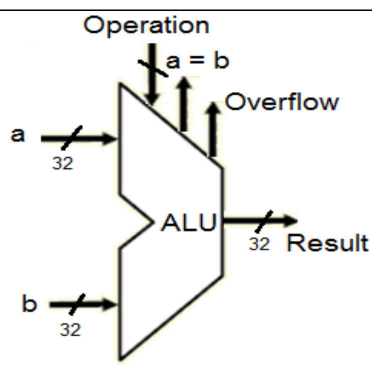
Register



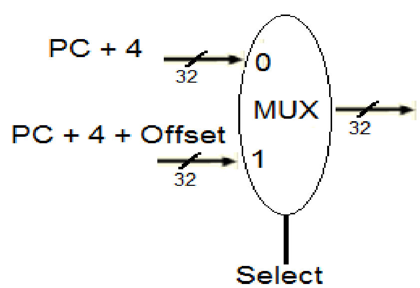
Adder

PC: a register that stores the address of the instruction being executed.

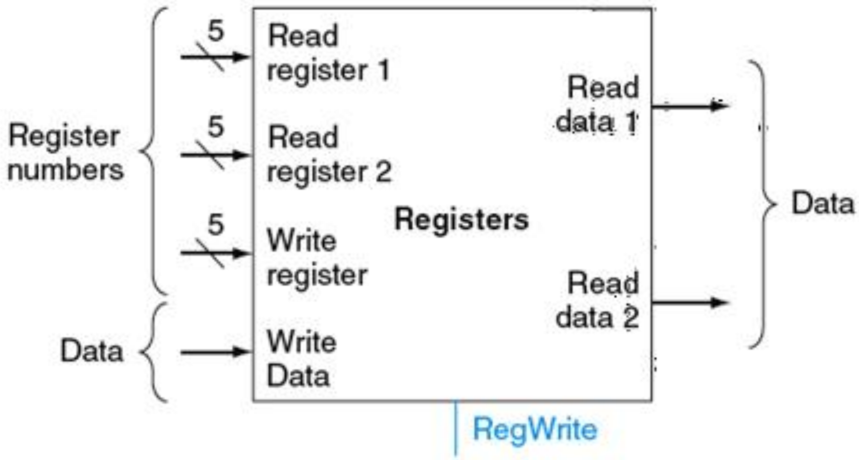
Adder: a unit that increments the program counter to the address of the next instruction.



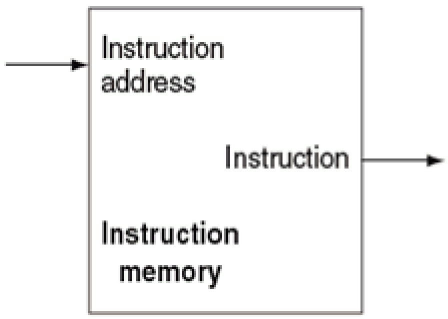
ALU



Multiplexer

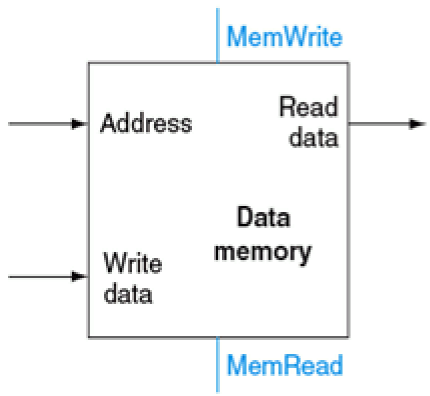


Register File

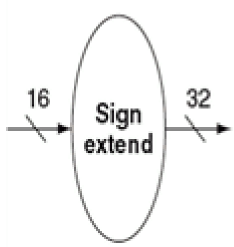


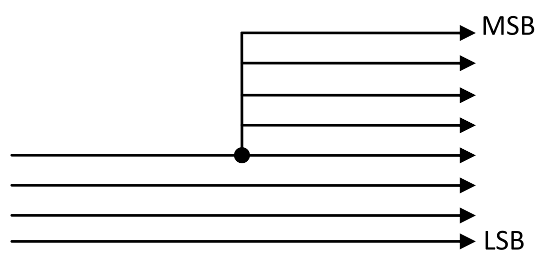
Program Memory

Instruction memory: a memory unit that stores the instructions of a program and supplies an instruction given its address.

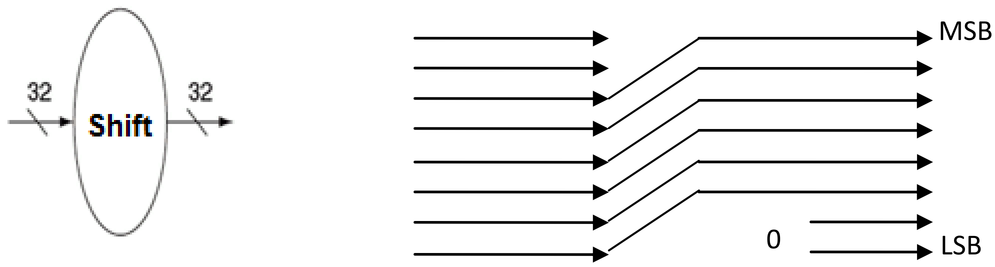


Data Memory





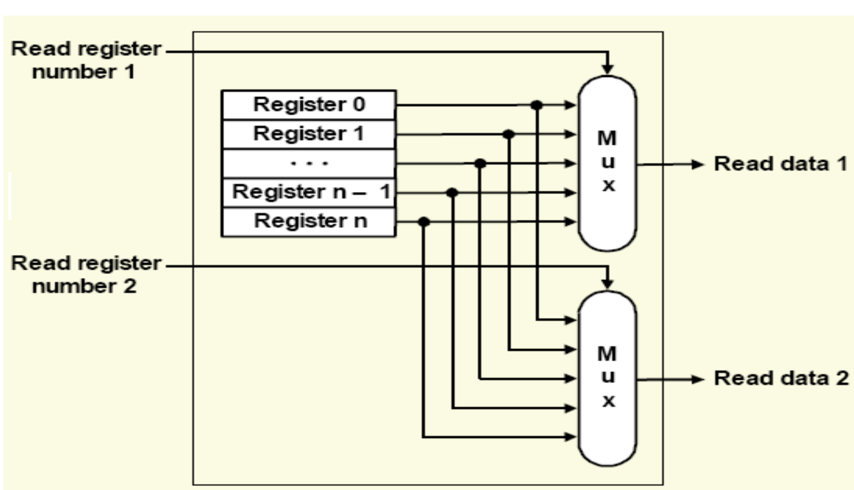
Bit Manipulation Component



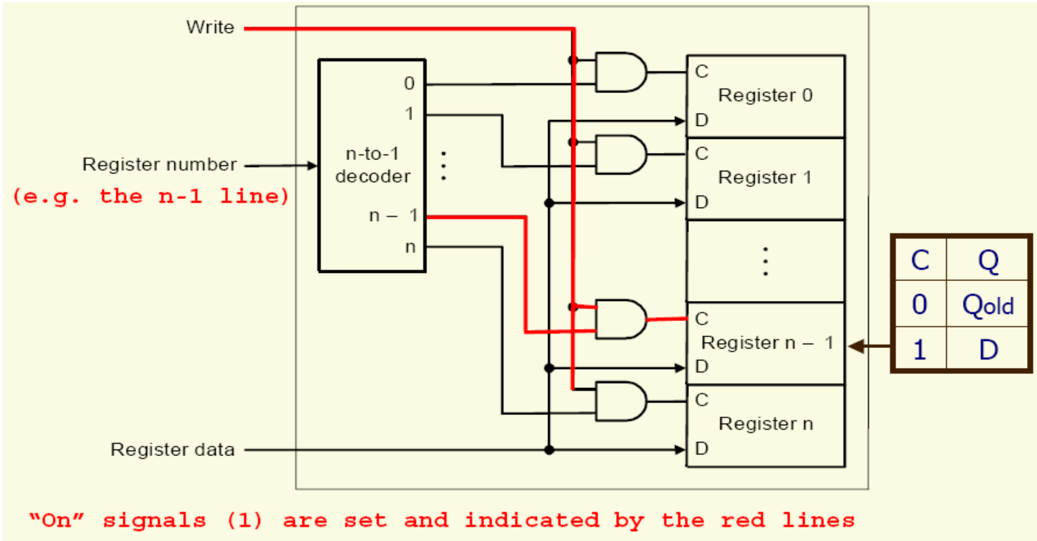
Bit Manipulation Component

Register File

- A register file is a structure in the datapath consisting of a set of registers that can be read and written by supplying a register number to be accessed.
- A register file can be implemented with a decoder, multiplexer and an array of registers built from D flipflops.
- Reading a register:
    - Input: a register number
    - Output: data contained in the specified register
- Writing a register:
    - Inputs: a register number, the data to write, and a clock that controls the writing into the register
- A Register File with Two Read Ports and One Write Port
- There are five inputs and two outputs.
- The read ports can be implemented with a pair of multiplexors, each of which is as wide as the number of bits in the register file.
    - To implement two read ports for a register file with n registers, we use two **n-to-1 multiplexors**, each of which is 32 bits wide.
    - The read register number signal is used as the **multiplexor selector signal**.

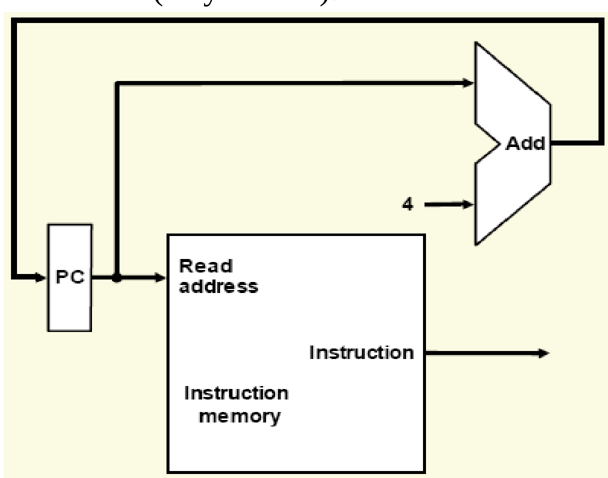


- To implement the write port for a register file with n registers, we use an **n-to-1 decoder** to generate a signal that can be used to determine which register to write.



Building a Datapath

- To execute any instruction, we first fetch the instruction from memory.
- To prepare for executing the next instruction, we increment the program counter so that it points at the next instruction (4 bytes later).

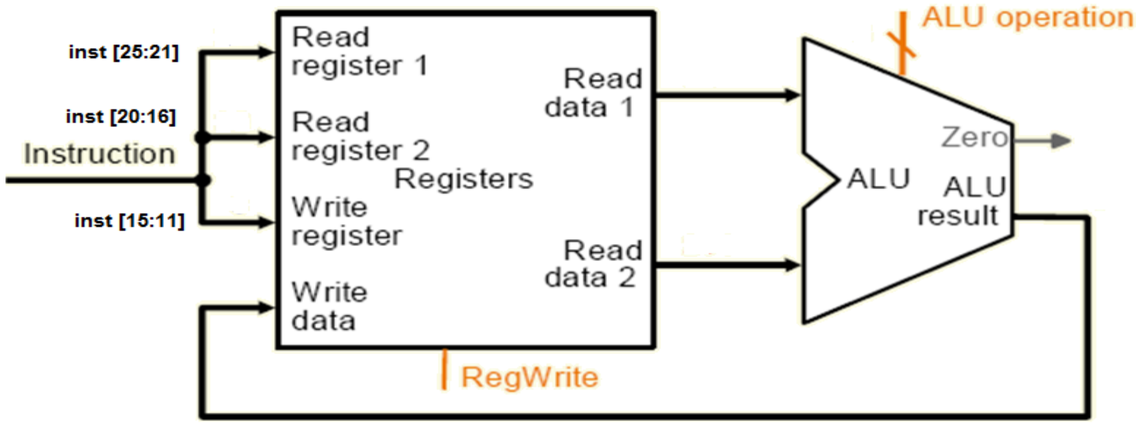


Datapath for arithmetic-logic Instruction (add, sub, and, or, slt)

- Actions required:
  - Fetch instruction
  - Address the register file
  - Pass operands to ALU
  - Pass result to register file
  - Increment PC

- R-Format

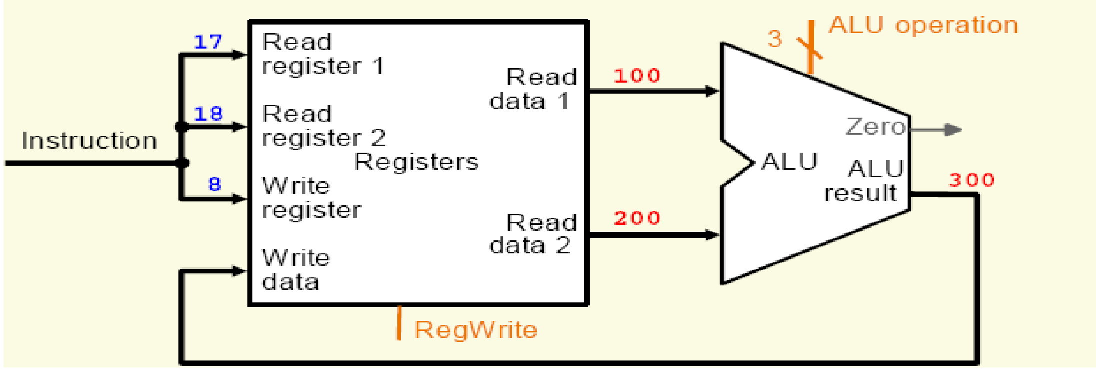
0	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0



- E.g.: add $t0, $s0, $s1

0	17	18	8	0	32
000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Register	Value
s0	100
s1	200



Datapath for Load and Store Instruction (lw, sw)

- Simple Single-Cycle Implementation:
    - Combine datapathes for each instruction into a single datapath, by sharing some of the resources among the different instructions instead of duplicating them.
- Implementation is based on:
    - An assumption that all instructions take ONLY one clock cycle each to complete.
    - No datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated.
    - As a consequence, we need a memory for instructions separate from data's.

- I-Format

35 or 43	rs	rt	address
31:26	25:21	20:16	15:0

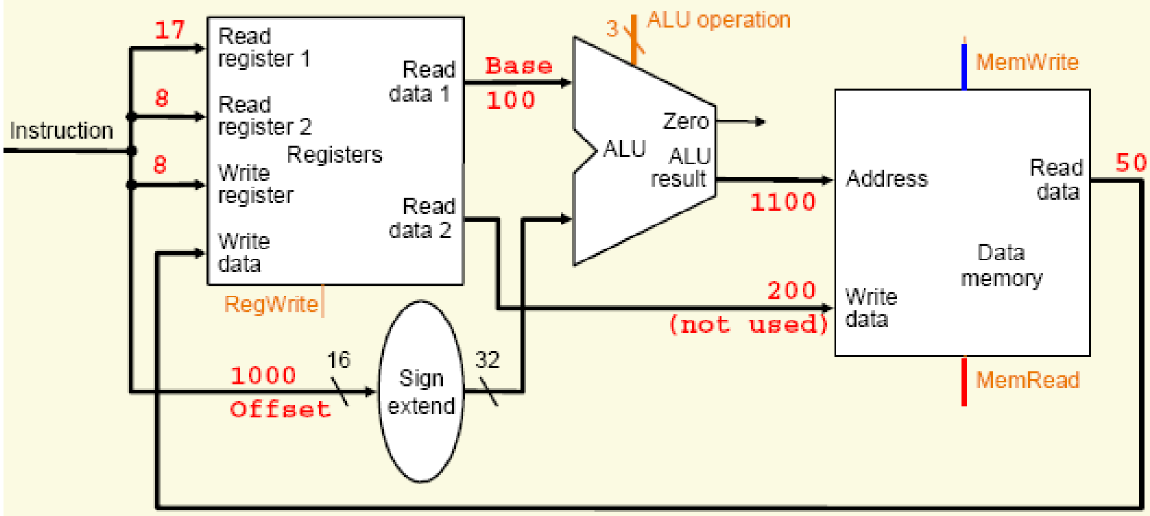
- Example: lw $t0, 1000 ($s1)

op	rs	Rt	address
6 bits	5 bits	5 bits	16 bits

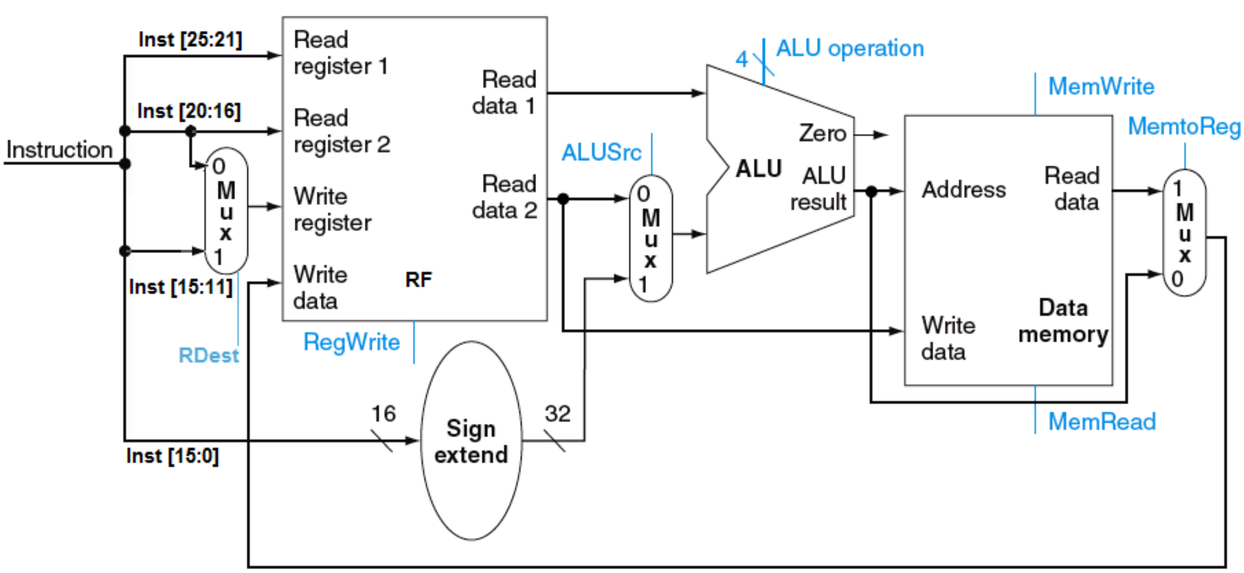
t0 = 200	
S1 = 100	
Mem[1100]=50	

lw $t0, 1000($s1)

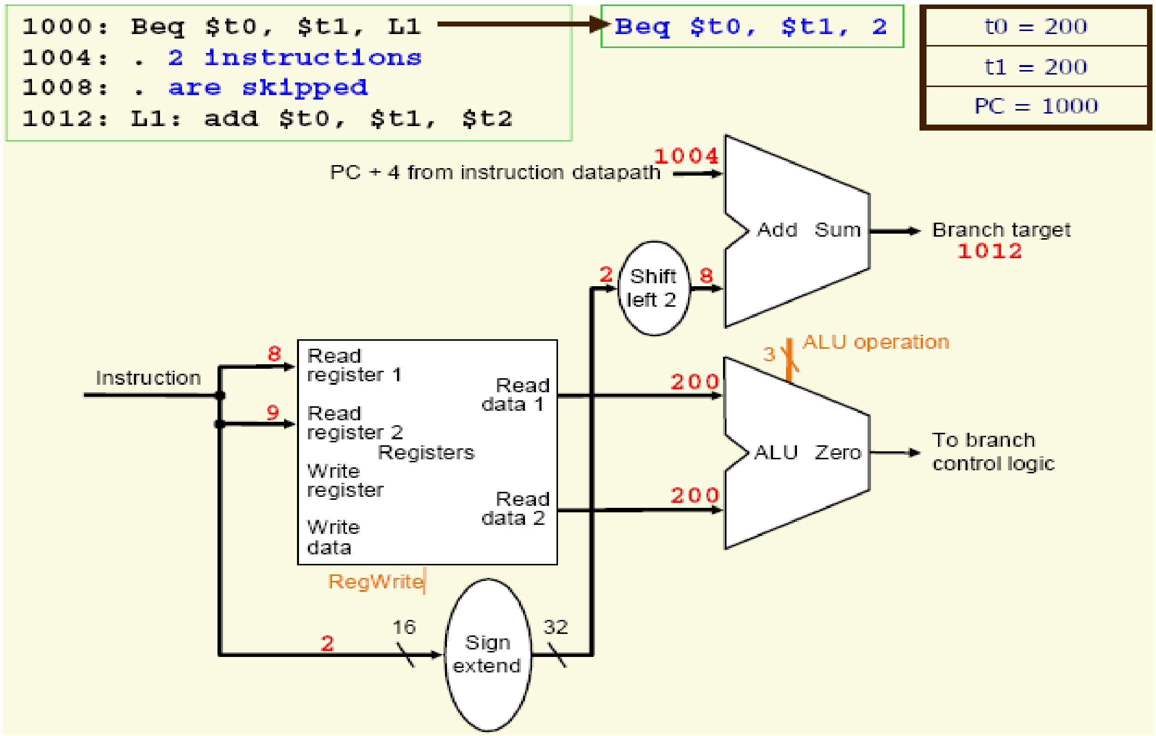
35	17 (s1)	8 (t0)	1000



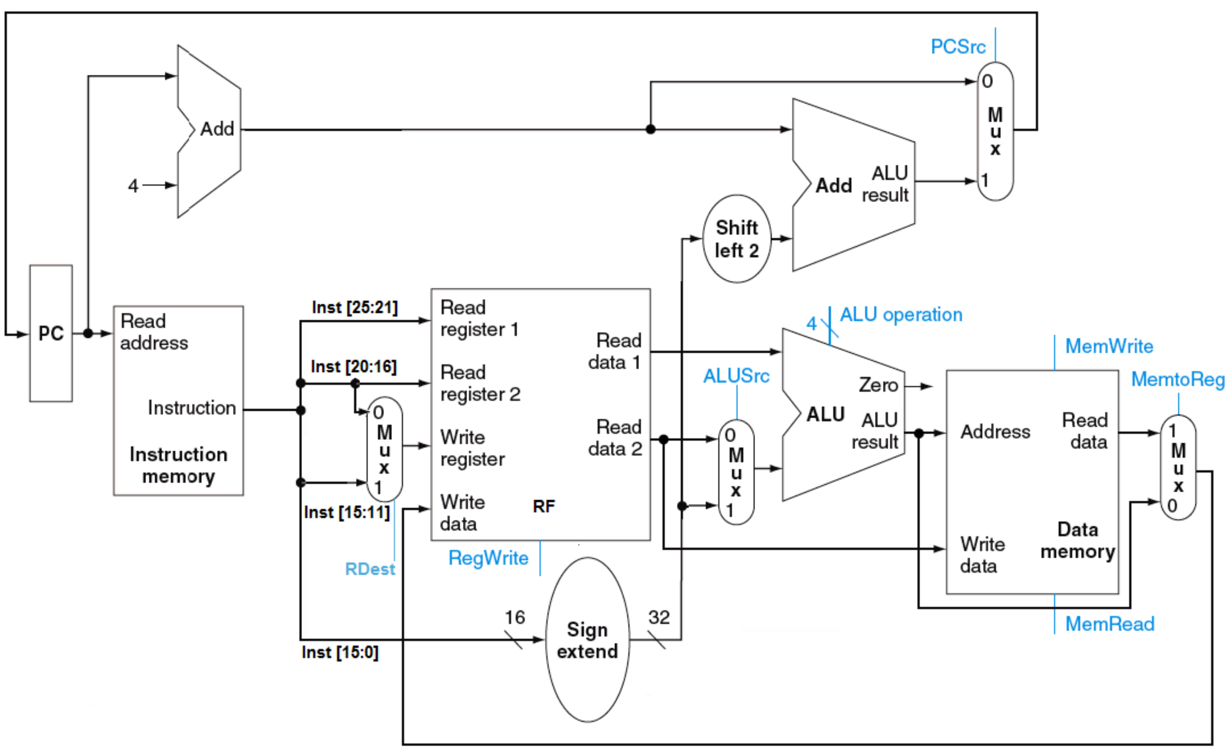
Combining the Datapath for R-Format and Memory Instruction (lw, sw)



Datapath for Branch (I-Format) Instruction



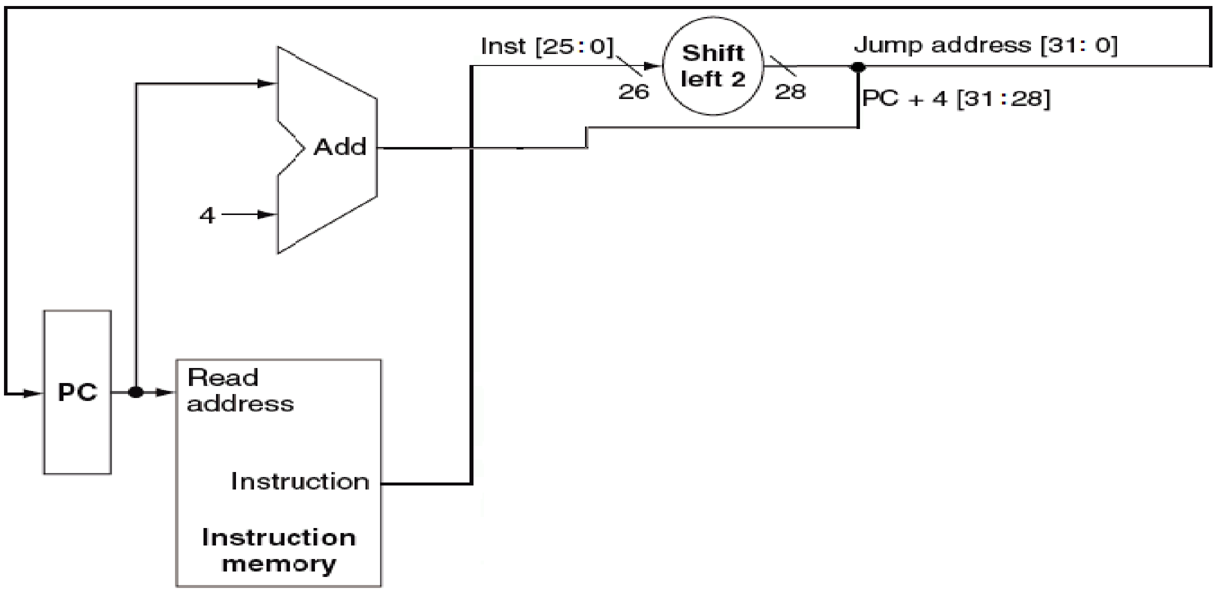
Combining the Datapath for R-Format and I-Format



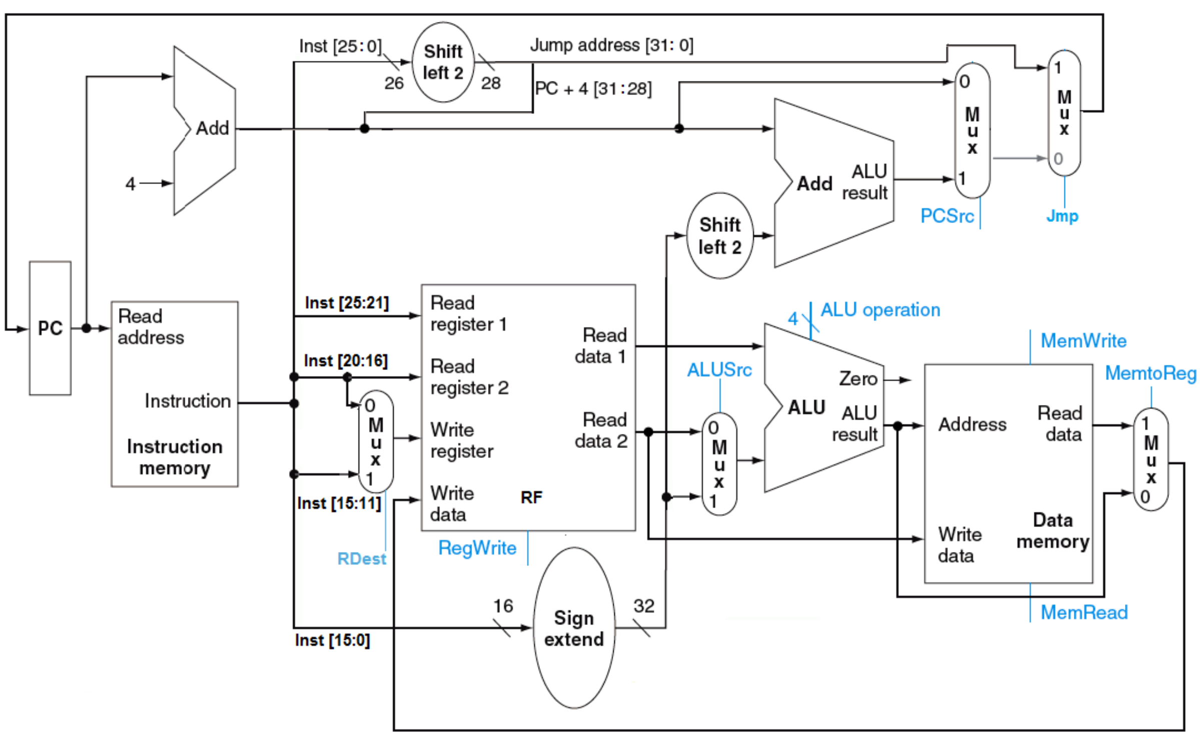
Datapath for Jump (J-Format) Instruction

- J-Format

2	address
31:26	25:0

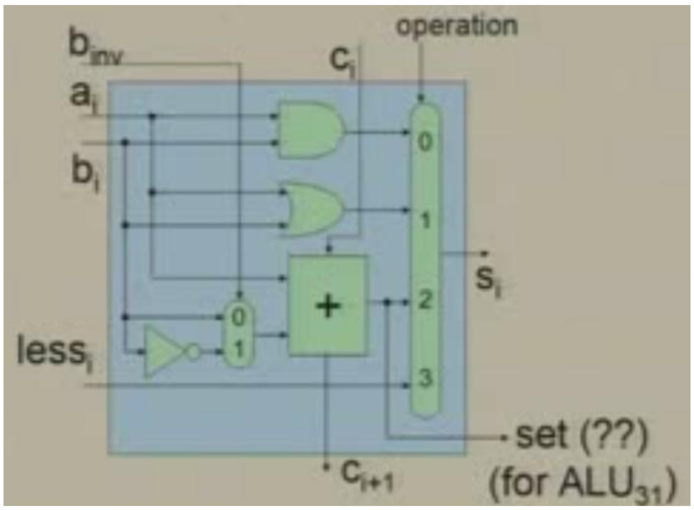


Combining the Datapath for the three format (R, I and J)



ALU Control

- The **control unit** controls the whole operation of the datapath by generating appropriate **control signals** (e.g., write signals for state elements, selector inputs for multiplexors, ALU control inputs) for the proper operation of the datapath.
- The **ALU control** is part of the **main control unit**.
- Control input bits for ALU:
  - We require three control input, one for adder (b_{inv}) and two for multiplexer (Operation)



ALU Control Input	Function
000	and
001	or
010	add
110	subtract
111	set on less than

b_{inv} Operation

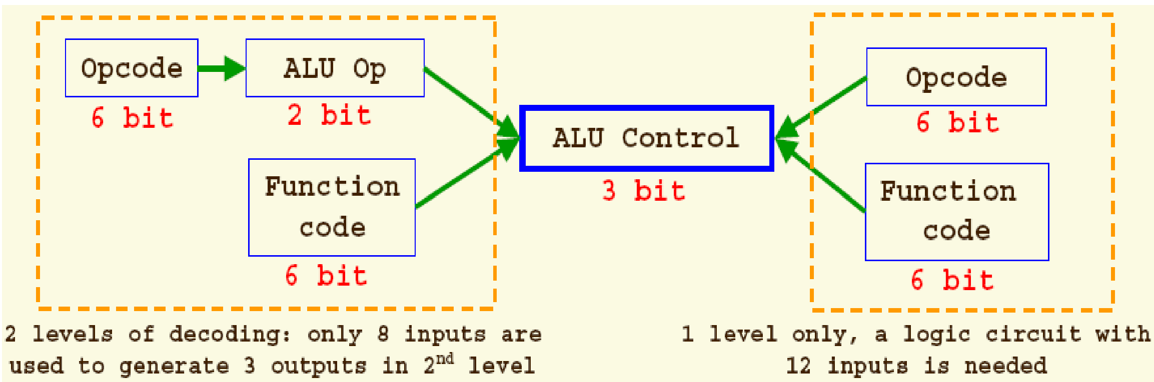
ALU Control Input Bits

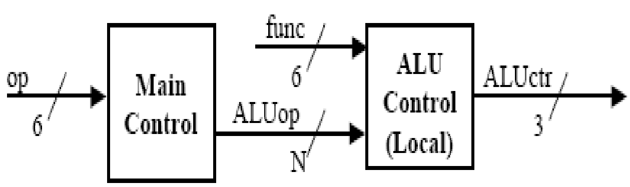
- Inputs used by control unit to generate ALU control input bits:
  - **ALUOp** (2 bits)
  - **Function code** of instruction (6 bits)

Instruction Type	ALUOp	Instruction Operation	Function Code	ALU Action	ALU Control Input
R-Type	10	add	100000	add	010
R-Type	10	subtract	100010	sub	110
R-Type	10	and	100100	and	000
R-Type	10	or	100101	or	001
R-Type	10	set on less than	101010	slt	111
Load word	00	load word	xxxxxx	add	010
Store word	00	store word	xxxxxx	add	010
Branch equal	01	branch equal	xxxxxx	sub	110
Jump	xx	jump	xxxxxx	xxx	xxx

Multiple Levels of Decoding

- **Level 1**: Generation of ALUOp bits by main control unit.
- **Level 2**: Generation of ALU control input bits from ALUOp bits and function code of instruction
- **Why multiple levels**?
  - Using multiple levels of control can reduce the size (complexity) of the logic circuit of the main control unit, and may also potentially increase the speed of the control unit.



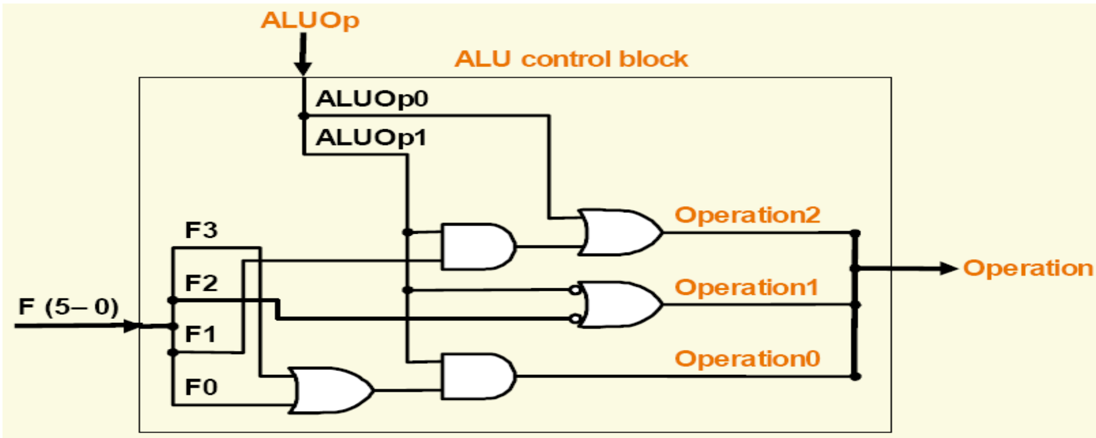


Truth Table for ALU Control Block

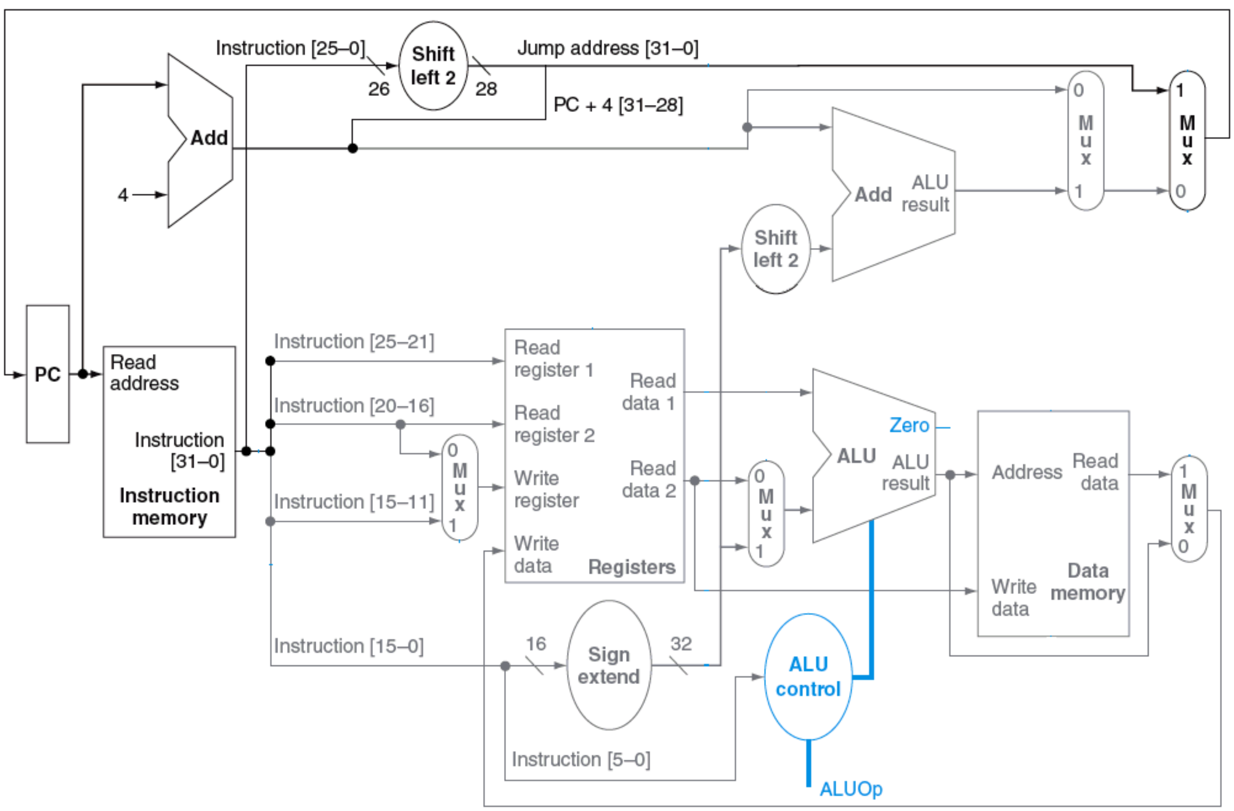
ALUOp		Function code						Operation	
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
0	0	X	X	X	X	X	X	010	lw, sw
X	1	X	X	X	X	X	X	110	beq
1	X	X	X	0	0	0	0	010	R-Type instruction
1	X	X	X	0	0	1	0	110	
1	X	X	X	0	1	0	0	000	
1	X	X	X	0	1	0	1	001	
1	X	X	X	1	0	1	0	111	

- Note that the truth table contains many don't-care terms, which can lead to simplified hardware implementation.

Hardware Implementation of ALU Control Block

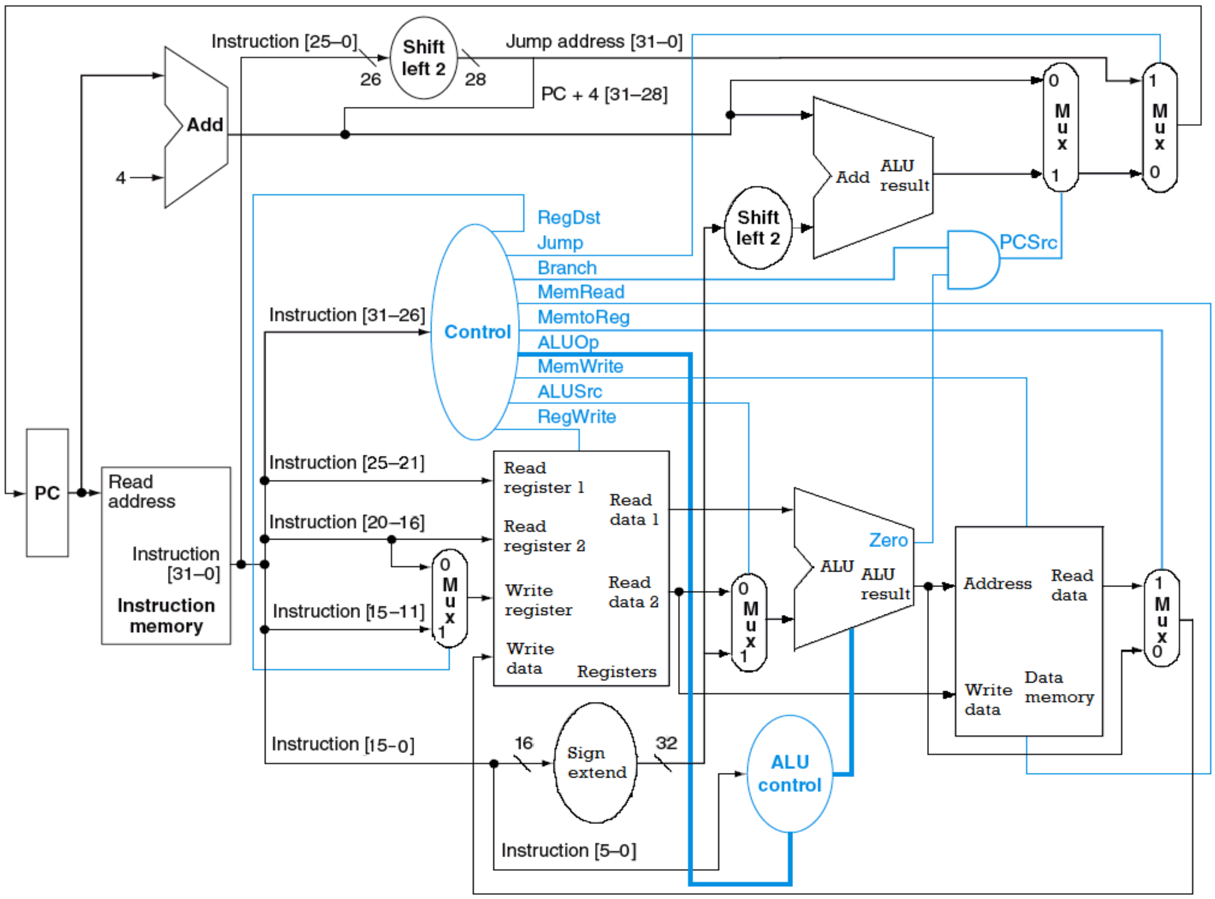


Datapath with ALU Control



Prepared By: Eng. Randa Al_Dallah

Datapath with Control Unit



Effects of Control Signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.
Jmp	The PC replaced by the value selected by the first multiplexer (PCScr)	The PC replaced by the output that computes the jump target

Setting of Control Signals

- The 10 control signals (8 from the previous table + 2 from ALUOp) can be set based entirely on the 6-bit opcode, with the exception of PCSrc.
- PCSrc control line:
  - Set if both conditions hold simultaneously:
    - Instruction is **beq**.
    - Zero output of ALU is true (i.e., the two source operands are equal).

Setting of Control Signals

- Setting of control lines is completely determined by opcode:

Instruction	RegDst	ALUScr	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	Jmp	ALUOp1	ALUOp0
R-Format	1	0	0	1	0	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0	0
sw	x	1	x	0	0	1	0	0	0	0
beq	x	0	x	0	0	0	1	0	0	1
j	x	x	x	0	0	0	x	1	x	X

- Note: sw, beq and j will not modify any register, it is ensured by making RegWrite to 0. So, we don't care what write register and write data.

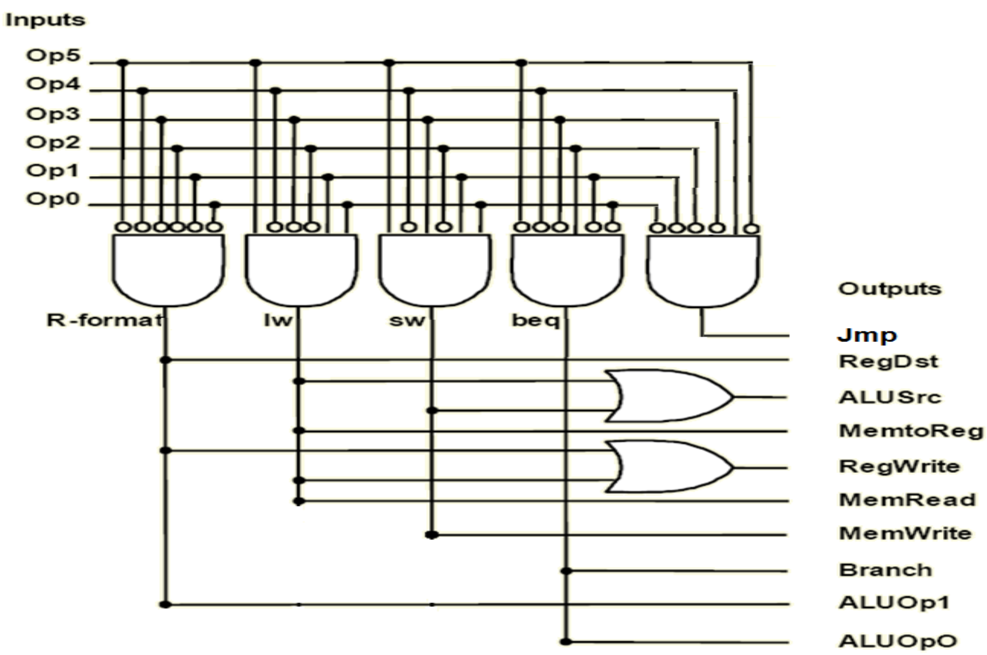
- **Input to datapath control unit:**

Instruction	Opcode in decimal	Opcode in binary					
		Op5	Op4	Op3	Op2	Op1	Op0
R-Format	0	0	0	0	0	0	0
lw	35	1	0	0	0	1	1
sw	43	1	0	1	0	1	1
beq	4	0	0	0	1	0	0
j	2	0	0	0	0	1	0

Truth Table for Datapath Control Unit

Inputs or Outputs	Signal Name	R-Format	lw	sw	beq	j
Inputs	Op5	0	1	1	0	0
	Op4	0	0	0	0	0
	Op3	0	0	1	0	0
	Op2	0	0	0	1	0
	Op1	0	1	1	0	1
	Op0	0	1	1	0	0
Outputs	RegDst	1	0	x	x	x
	ALUScr	0	1	1	0	x
	MemtoReg	0	1	X	x	x
	RegWrite	1	1	0	0	0
	MemRead	0	1	0	0	0
	MemWrite	0	0	1	0	0
	Branch	0	0	0	1	x
	Jmp	0	0	0	0	1
	ALUOp1	1	0	0	0	x
	ALUOp0	0	0	0	1	x

HW Implementation of Datapath Control Unit (using PLA)



Problems with Single-Cycle Datapath Implementation

- Every instruction takes one clock cycle (CPI = 1). The clock cycle is determined by the longest possible path in the machine.
- The longest path is for a load instruction which involves five functional units in series: **instruction memory**, **register file**, **ALU**, **data memory**, **register file**.
- Even though each instruction takes just one clock cycle, the clock cycle is expected to be large and hence the overall performance is poor because many instructions cannot fully utilize the unnecessarily long clock cycle.
- No sharing of hardware functional units is possible.

Cycle Time Calculation

- Different Instruction Classes

Instruction class	Functional units used by the instruction class				
R-format	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

- Example:
  - Calculate cycle time assuming negligible delays except:
    - Memory access (200ps),
    - ALU to perform function (100ps),
    - Register file access (50ps)
  - Instructions mix: 25% loads, 10% stores, 45% ALU, 15% branches, 5% jumps.
  - Which of the following implementations would be faster?
    - Case1: Every instruction operates in a 1 clock cycle of a fixed length
    - Case2: Every instruction operates in a 1 clock cycle of a variable length

  - **Solution:**

    CPU execution time = Instruction count x CPI x Clock cycle time

    Since CPI = 1

    CPU execution time = Instruction count x Clock cycle time

Using the critical paths we can compute the required length for each class:

	Instr. fetch	Reg. Read	ALU Opr.	Data Memory	Reg. Write	Total
R-Type	200	50	100		50	**400 ps**
lw	200	50	100	200	50	**600 ps**
sw	200	50	100	200		**550 ps**
branch	200	50	100			**350 ps**
Jump	200					**200 ps**

In case 1 the clock has to be 600ps depending on the longest instruction. (Clock rate = 1/ (600ps) HZ)

In case 2 a machine with a variable clock will have a clock cycle that varies between 200ps and 600ps.

The average CPU clock cycle= 600x25% + 550x10% + 400x45% + 350x15%+200x5% = 447.5ps

So the variable clock machine is faster 1.34 times

- A **variable clock implementation** would be faster, but it is very difficult to implement a variable-speed clock.

- As a result, the best solution is to consider a **shorter clock cycle** (derived from the basic functional unit delays) and allow different instructions to require **different numbers of clock cycles (Multicycle Datapath)**.

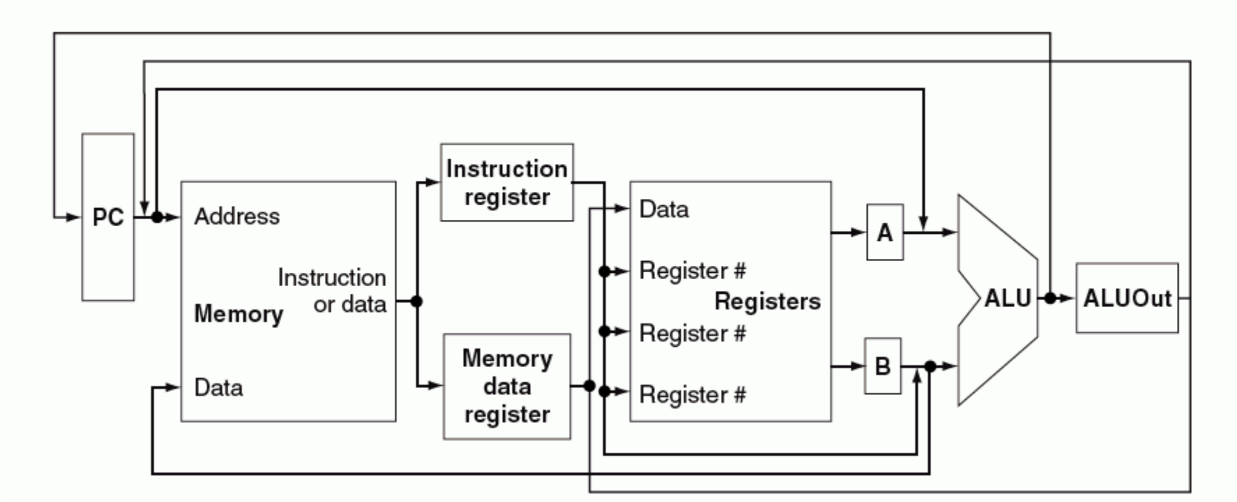
A Multicycle Implementation

- The execution of each instruction is broken into a series of steps that correspond to the **functional unit operations**. Each step takes one clock cycle to complete.
- Restrict each cycle to use only one major functional unit in the data path, or if more than one major functional unit used they should be used only in parallel.
- A single functional unit can be used more than once per instruction, as long as it is used on different clock cycles. This sharing can help to reduce the amount of hardware required. In particular,
    - A **single memory unit** is used for both instructions and data.
    - There is a **single ALU**, rather than an ALU and two adders.

- ALU will be used to compute not only tasks it performed in the single-cycle design (e.g. lw & sw addresses and R-type instruction calculations), but it will be used to increment PC (by 4) and to calculate branch target address.

- **One or more registers are added** after every major functional unit to hold the output of that unit until the value is used in a subsequent clock cycle.

High-Level View of the Multicycle Datapath



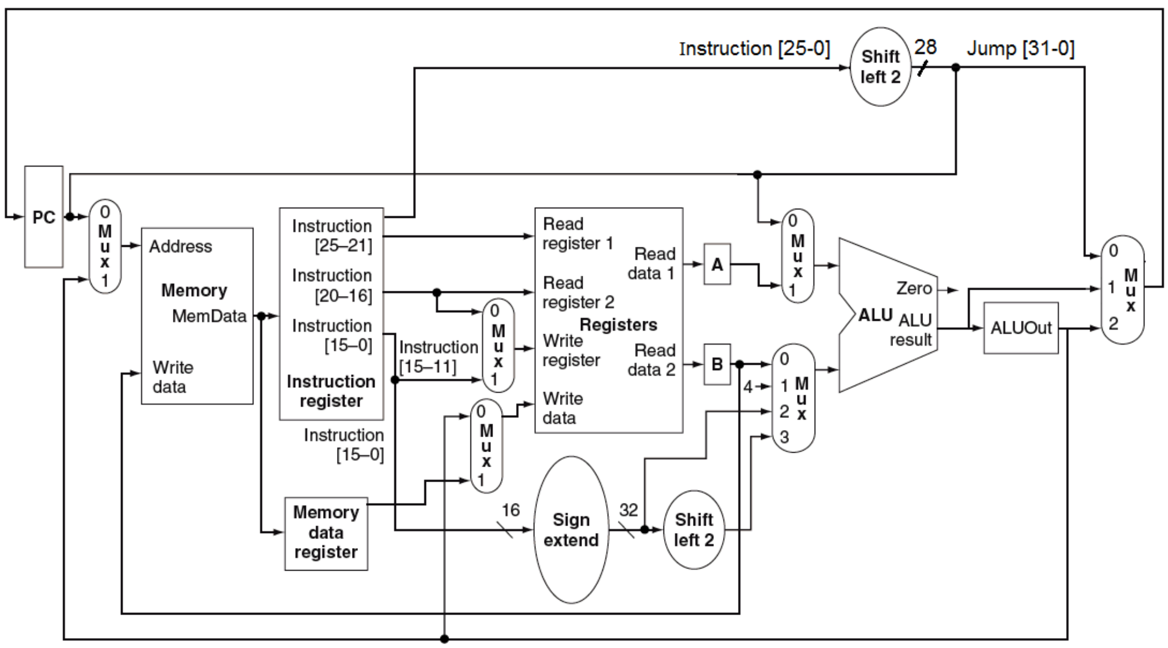
Constraints for Multicycle datapath

- The design of multicycle datapath is based on the constraints below:
  1. The memory and register file cannot be accessed (read / write) in the same cycle.
  2. On each cycle, One ALU operation can be performed.

     i. Provided that the inputs (operands) of the ALU are available at the beginning of that cycle.
     - E.g. for add instruction, finding the operands from register file & sum calculation has to be done in a different cycle.
     - However, PC + 4 can be done in any cycle.

     ii. The result evaluated by the ALU operation cannot be used by memory and register file immediately.
     - For add instruction, the sum cannot be written to register file in the same cycle.
  3. PC can be updated in any cycle, even the value comes from the ALU result.
- Based on constraint 1 & 2ii)
  - **Instruction register** (IR) and **memory data register** (MDR):
    - Hold output of memory for an instruction read and a data read, respectively.

- Based on constraint 2
    - **A** and **B registers**:
        - Hold register operand values from register file.
    - **ALUOut register**:
        - Holds output of ALU.
- The IR needs to hold the instruction until the end of execution of that instruction. Thus it requires a **write control signal**.
- Since several functional units are shared for different purposes, some **multiplexors** have to be added or expanded. Thus **additional control signals** are needed.



Multiple Execution Steps per Instruction

- Typical execution steps:
    - Instruction fetch
    - Instruction decode and register fetch
    - Execution, memory address computation, or branch completion
    - Memory access or R-type instruction completion
    - Memory read completion
- Each instruction takes a few (3-5) steps.

Break Instruction Execution into Cycles

R-Type Instructions

Cycle 1
IR = Mem [PC]
PC = PC + 4

Cycle 2
A = RF [IR [25-21]]
B = RF [IR [20-16]]

Cycle 3
ALUOut = A op B

Op depends upon IR [5-0]

Cycle 4
RF [IR [15-11]] = ALUOut

sw Instruction

Cycle 1
IR = Mem [PC]
PC = PC + 4

Cycle 2
A = RF [IR [25-21]]
B = RF [IR [20-16]]

Cycle 3
ALUOut = A + sx(IR[15-0])

Cycle 4
Mem [ALUOut] = B

lw Instruction

Cycle 1

IR = Mem [PC]

PC = PC + 4

↓

Cycle 2

A = RF [IR [25-21]]

↓

Cycle 3

ALUOut = A + sx(IR[15-0])

↓

Cycle 4

DR =Mem [ALUOut]

↓

Cycle 5

RF [IR [20-16]] = DR

beq Instruction

Cycle 1

IR = Mem [PC]

PC = PC + 4

↓

Cycle 2

A = RF [IR [25-21]]

B = RF [IR [20-16]]

ALUOut = PC + s2(sx(IR[15-0]))

↓

Cycle 3

If (A == B) PC = ALUOut

j Instruction

Cycle 1

IR = Mem [PC]

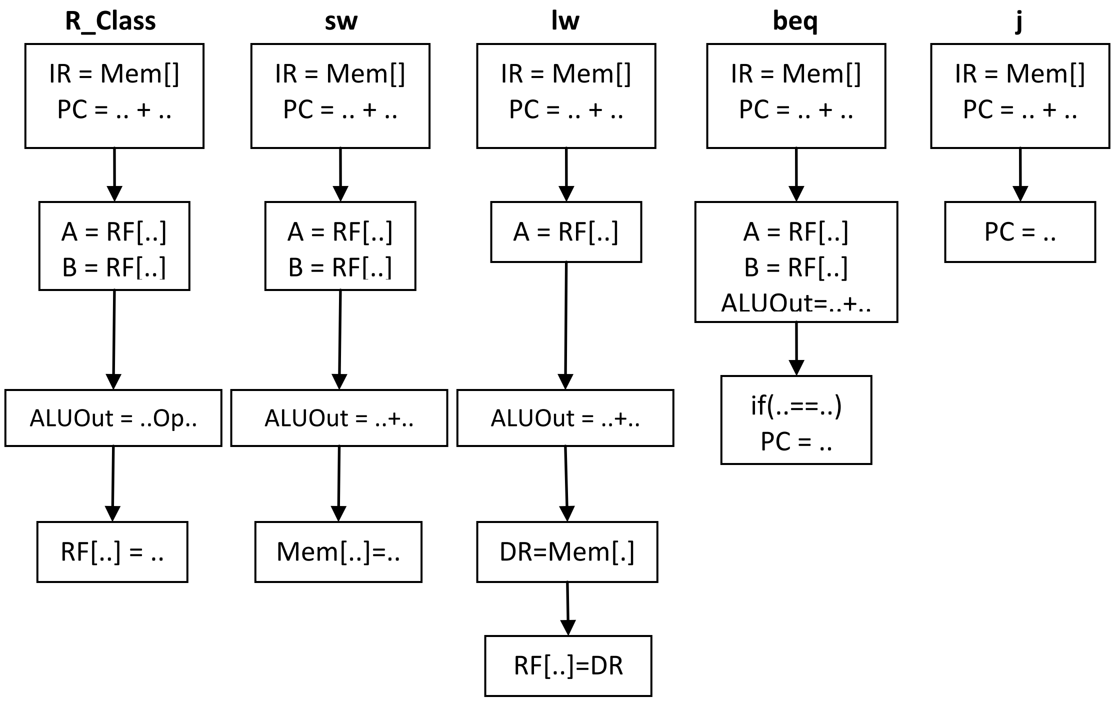
PC = PC + 4

↓

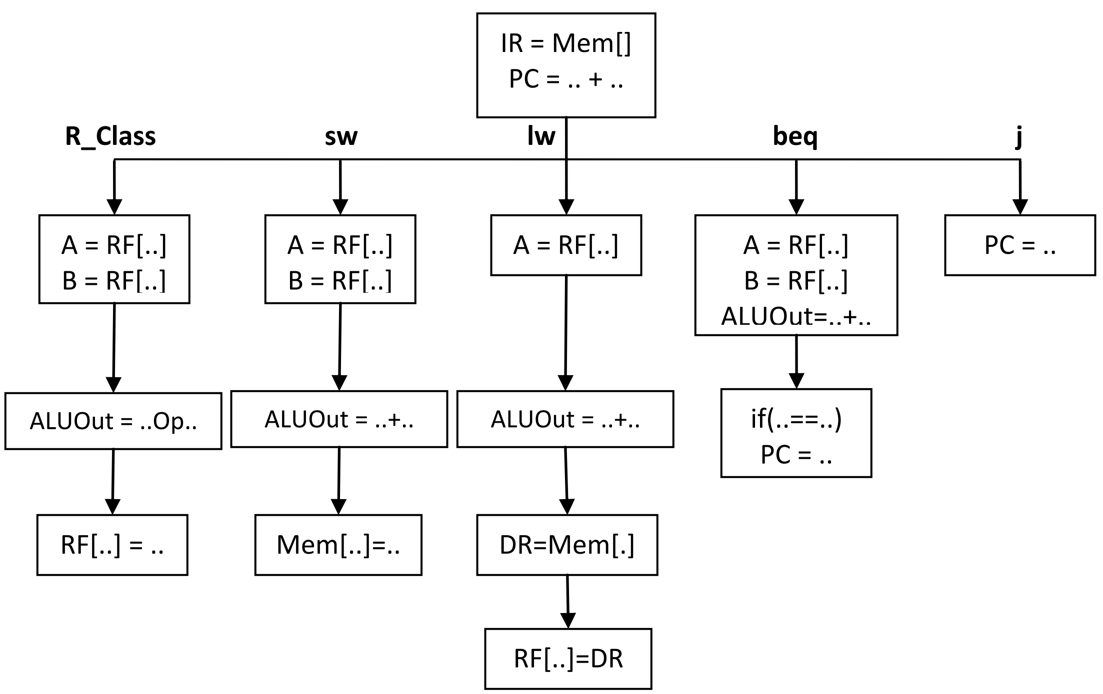
Cycle 2

PC = PC[31-28] || s2(IR[25-0])

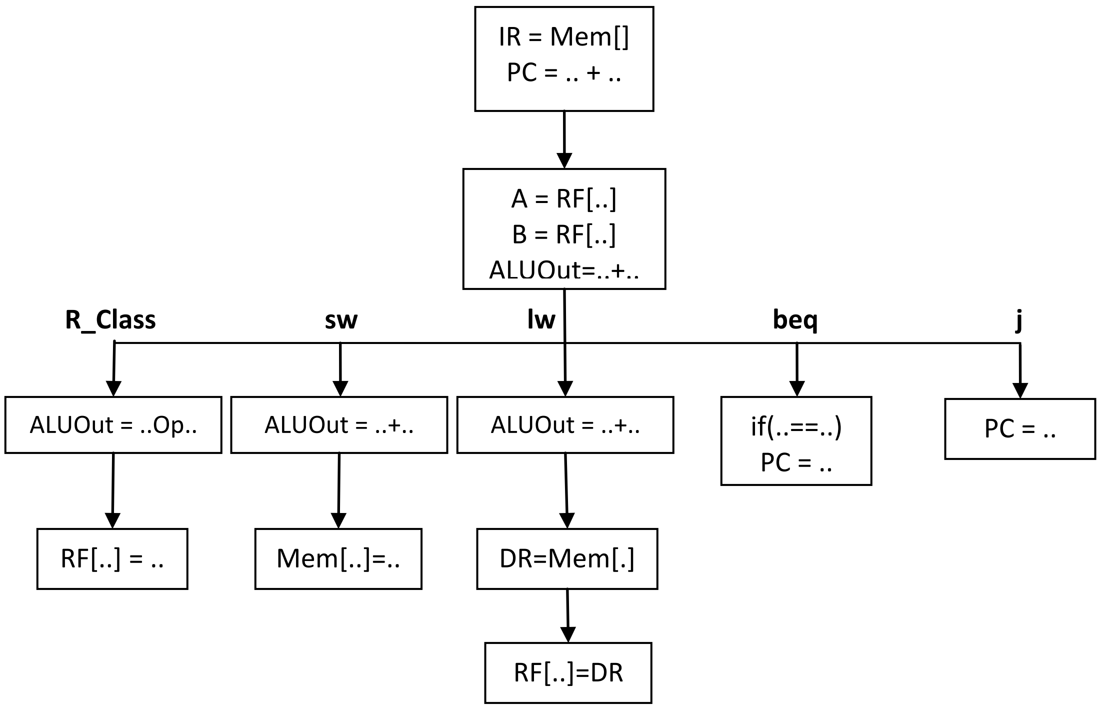
Put cycle sequences together



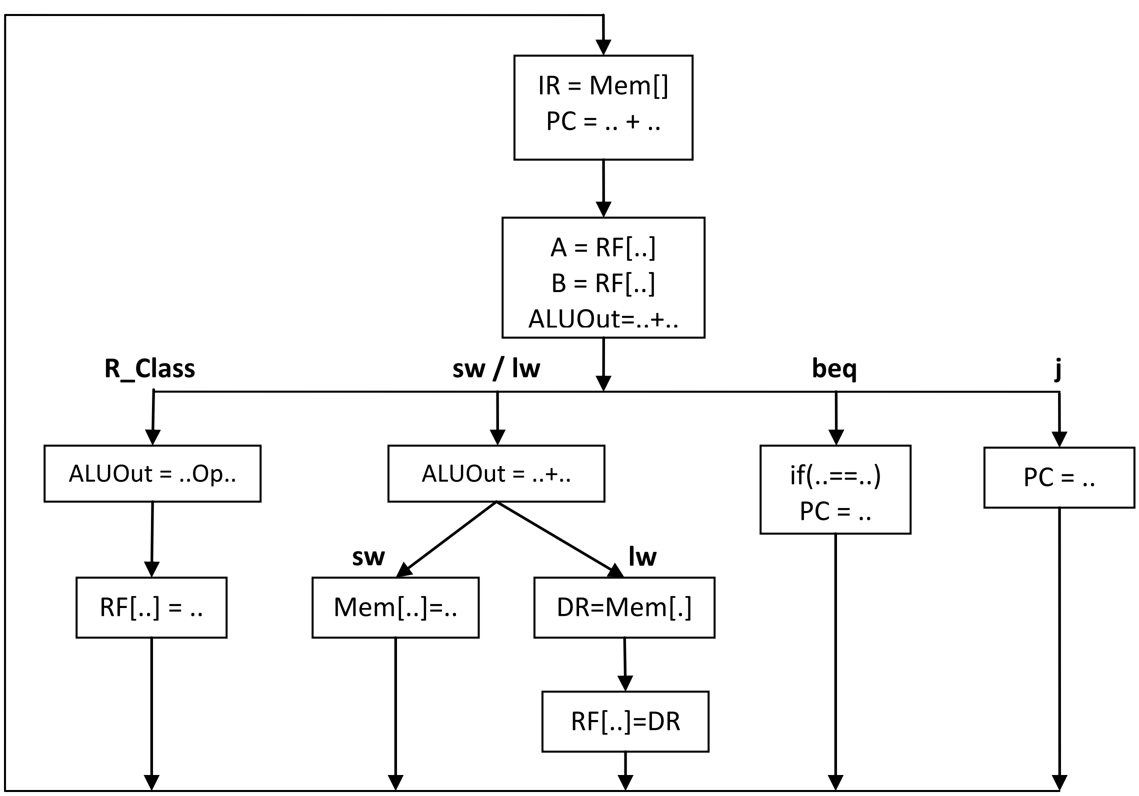
After merging fetch cycle



With a common decoding cycle



- Note: The second cycle of the jump instruction is postponed
- We can merge the third cycle of sw and lw, and then split it in the fourth cycle



State Transition Diagram

Summary of Execution Steps

I. Instruction Fetch Step

- Fetch the instruction from memory and compute the address of the next sequential instruction:

  IR = Mem [PC];
  PC = PC + 4;

- **Operations**:
  - Send the PC to the memory as address.
  - Read an instruction from memory.
  - Write the instruction into the IR.
  - Increment the PC by 4 without violating the constraints.
    - So 1 clock cycle can be reduced for most instructions.

II. Instruction Decode & Register Fetch Step

- Assuming the existence of two registers and an offset field (no harm to do the computation early even if they do not exist), fetch the two registers from the register file and compute the branch target address:

  A = RF[IR[25-21]];
  B = RF[IR[20-16]];
  ALUOut = PC + s2(sx(IR[15-0]));

- **Operations**:
- Access the register file to read rs and rt.
- Store the results into registers A and B.
- Compute the branch target address (sign extension and left shift).
- Store the address in ALUOut.
  - So for branch instruction, we don't need to spend an extra clock cycle to find the branch address after we know the branch condition is satisfied.

III. Execution, Memory Execution, Memory Address Computation, or Branch Completion Step

- In this step, the datapath operation is determined by the instruction class.
- **Memory reference instructions**:
  - Compute the memory address:

    ALUOut = A + sx(IR[15-0]);

- **Operations**:
  - Sign-extend the 16-bit offset to a 32-bit value.
  - Send both register A and the 32-bit offset to the ALU.
  - Add the two values.
  - Store the result in ALUOut.

- **Arithmetic-logical (R-type) instructions**:
    - Perform the ALU operation specified by the function code:

      ALUOut = A op B;
- **Operations**:
    - Send both registers A and B to the ALU.
    - Perform the specified operation on the two values.
    - Store the result in ALUOut.

- **Branch instructions**:
    - Compare registers A and B and set the PC to the branch target address if A and B are equal:

      if (A == B)

      PC = ALUOut;
- **Operations**:
    - Send both registers A and B to the ALU.
    - Compare A and B by performing subtraction in the ALU and set the Zero output signal to 1 if A and B are equal.
    - If Zero is equal to 1, then write ALUOut to the PC.

- **Jump instructions:**
    - Compute the jump address and set the PC to this address:

      PC = PC[31-28] || s2(IR[25-0])
- **Operations**:
    - Left-shift the 26-bit address field by 2 bits to give a 28-bit value.
    - Concatenate the four leftmost bits of the PC with the 28-bit value to form a 32-bit jump address.
    - Write the jump address to the PC.

IV. Memory Access or R-Type Instruction Completion Step

- **Memory reference instructions**:
    - Read from or write to memory:

      DR = Mem[ALUOut]; // for load instruction

      or

      Mem [ALUOut] = B; // for store instruction
- **Operations**:
    - Use the address computed during the previous step and stored in ALUOut.
    - For a load instruction, a data word is retrieved from memory with the specified address.
    - For a store instruction, a data word is written into memory with the specified address.

- **Arithmetic-logical (R-type) instructions**:
    - Write the result of the ALU operation into a destination register inside the register file:

      RF[IR[15-11]] = ALUOut;
- **Operations**:
    - Get from ALUOut the value which was the result of the ALU operation in the previous step.
    - Write the value into a register in the register file.

V. Memory Read Completion Step

- A load instruction completes by writing back the value from memory into a register in the register file:

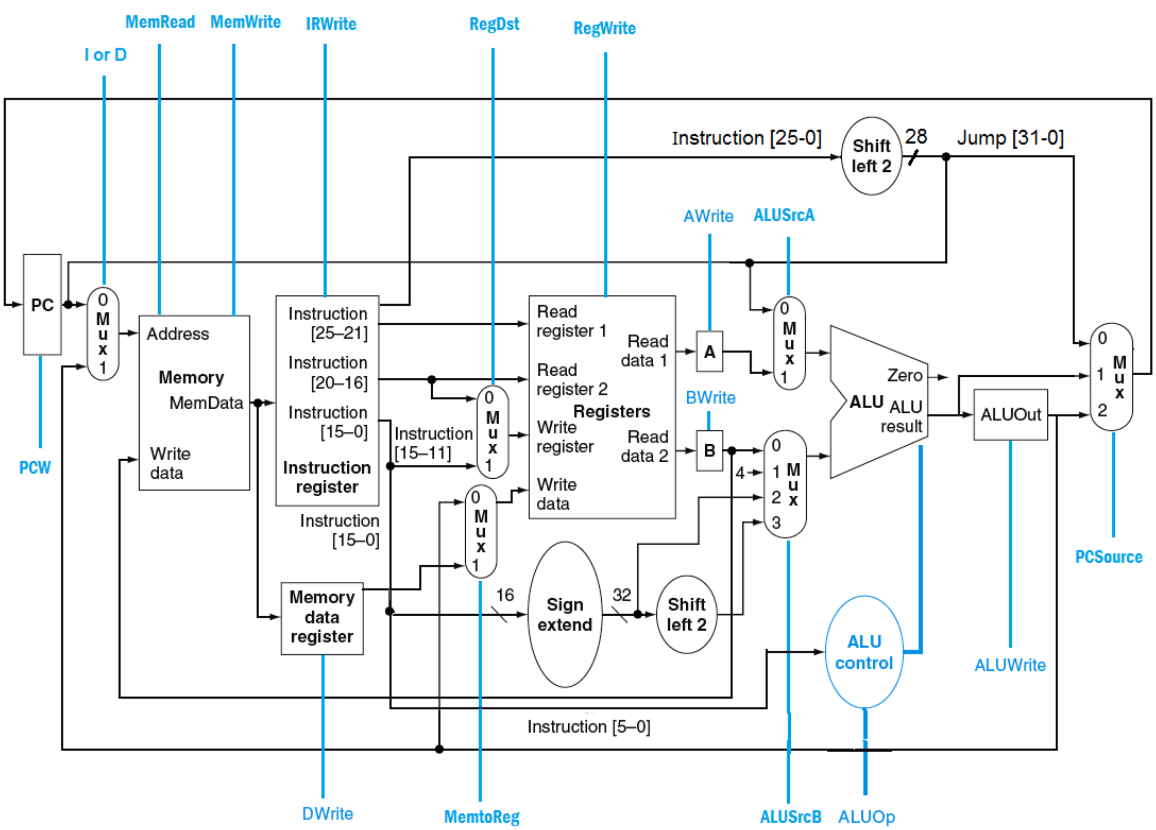
  RF[IR[20-16]] = DR;
- **Operations**:
    - Get from DR the value which was read from memory in the previous step.
    - Write the value into a register in the register file.

CPI in a Multicycle CPU

- Example: Using the SPECINT2000 instruction mix, what is the CPI, assuming that each state in a multicycle CPU requires 1 clock cycle?
    - The mix is: 25% loads, 10% stores, 11% branches, 2% jumps, 52% ALU

- Solution:
    - Number of cycle: Loads: 5, Stores 4, ALU 4, Branches 3, Jumps 3
    - CPI = 0.25x5 + 0.1x4 + 0.52x4 + 0.11 x3 + 0.02x3 = 4.12
        - Better than the worst case 5

Control Unit Design



Micro operations and control signals

PC Group

Micro Operation	PCWriteUncond	PCWriteCond	PCSource	Opr Name
PC = PC + 4	1	x	1 (01)	PCinc
if (A == B) PC = ALUOut	0	1	2 (10)	Branch
PC = PC[31-28] || s2(IR[25-0])	1	x	0 (00)	Jump
Default	0	0	X (xx)	NOP

PCWrite = PCWriteUncond + (Zero . PCWriteCond)

Memory Group

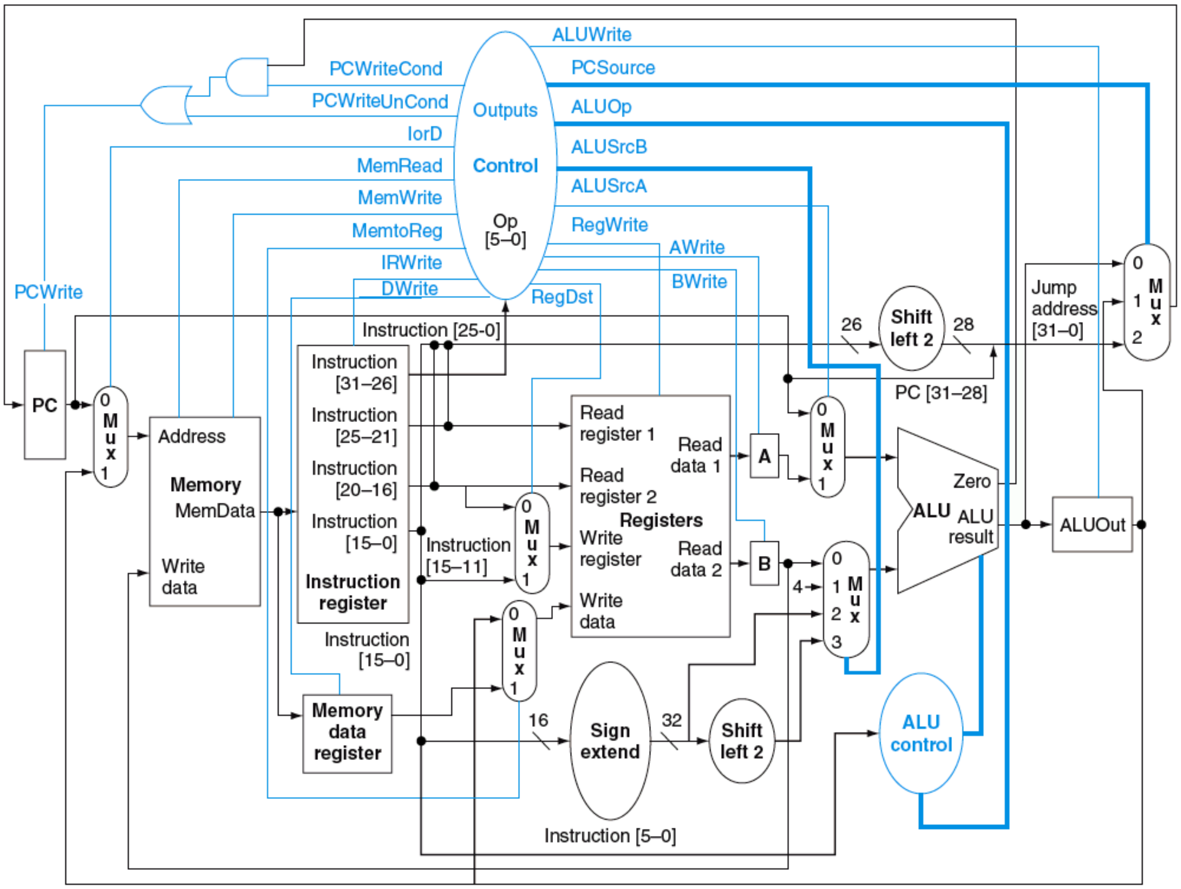
Micro Operation	MemWrite	MemRead	IorD	IRWrite	DWrite	Opr Name
IR = Mem[PC]	0	1	0	1	0	Fetch
DR = Mem[ALUOut]	0	1	1	0	1	M_Rd
Mem[ALUOut] = B	1	0	1	0	0	M_Wr
Default	0	0	x	0	0	NOP

Register File (RF) Grouup

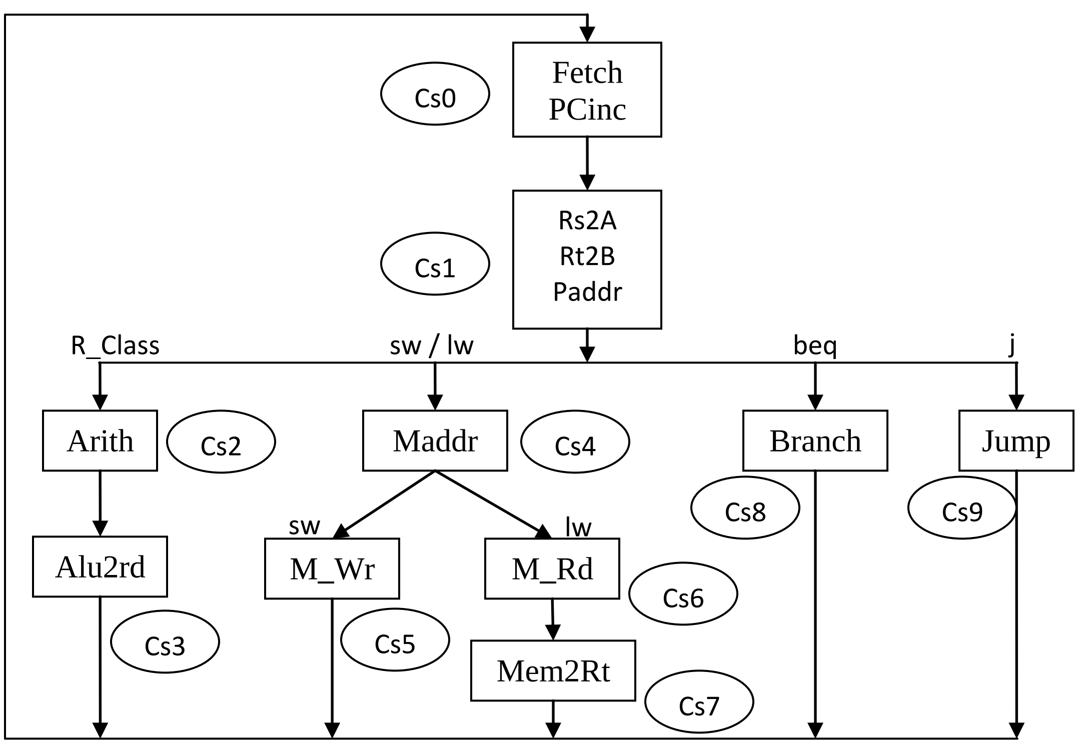
Micro Operation	RegWrite	RegDst	Memto Reg	AWrite	BWrite	Opr Name
A = RF[IR[25-21]]	0	x	x	1	0	Rs2A
B = RF[IR[20-16]]	0	x	x	0	1	Rt2B
RF[IR[15-11]] = ALUOut	1	1	0	0	0	Alu2rd
RF[IR[20-16]] = DR	1	0	1	0	0	Mem2rt
Default	0	x	x	0	0	NOP

ALU Group

Micro Operation	ALUOp	ALUSrc A	ALUSrc B	ALUWrite	Opr Name
PC = PC + 4	0 (00 : add)	0	1 (01)	0	PCinc
ALUOut = A op B	2 (10 : func)	1	0 (00)	1	Arith
ALUOut = A + sx(IR[15-0])	0 (00 : add)	1	2 (10)	1	Maddr
ALUOut = PC + s2(sx(IR[15-0]))	0 (00 : add)	0	3 (11)	1	Paddr
if(A == B) PC = ALUOut	1 (01 : sub)	1	0 (00)	0	Branch
Default	X (xx)	x	X (xx)	0	NOP



Control states and micro operations



Control States and signal values

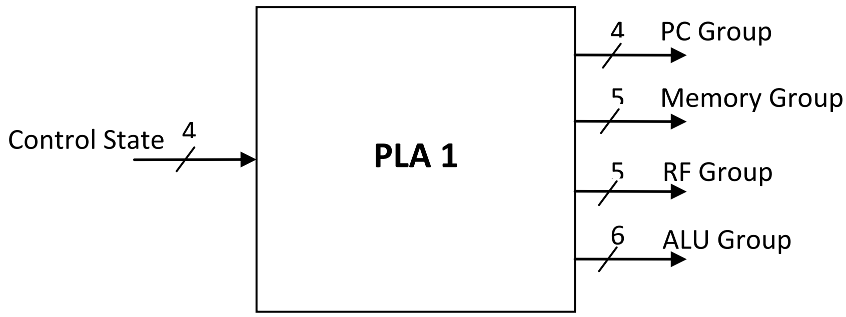
State	PC group	Mem group	RF group	ALU group
Cs0	PCinc	Fetch	NOP	PCinc
Cs1	NOP	NOP	Rs2A, Rt2B	Paddr
Cs2	NOP	NOP	NOP	arith
Cs3	NOP	NOP	Alu2rd	NOP
Cs4	NOP	NOP	NOP	Maddr
Cs5	NOP	M_wr	NOP	NOP
Cs6	NOP	M_rd	NOP	NOP
Cs7	NOP	NOP	Mem2rt	NOP
Cs8	branch	NOP	NOP	branch
Cs9	jump	NOP	NOP	NOP

- In the following table, each micro operation is replaced with bit vector (the bit vector defines the relative control signals), and also convert the control state to binary code.

Inputs	Outputs			
State	**PC group**	**Mem group**	**RF group**	**ALU group**
0000	1x01	01010	0xx00	000010
0001	00xx	00x00	0xx11	000111
0010	00xx	00x00	0xx00	101001
0011	00xx	00x00	11000	xxxxx0
0100	00xx	00x00	0xx00	001101
0101	00xx	10100	0xx00	xxxxx0
0110	00xx	01101	0xx00	xxxxx0
0111	00xx	00x00	10100	xxxxx0
1000	0110	00x00	0xx00	011000
1001	1x00	00x00	0xx00	xxxxx0

- This can be implemented by using PLA

PLA to generate control signals



Control state transitions

	R_Class	sw	lw	beq	j
Cs0	Cs1	Cs1	Cs1	Cs1	Cs1
Cs1	Cs2	Cs4	Cs4	Cs8	Cs9
Cs2	Cs3	x	x	x	x
Cs3	Cs0	x	x	x	x
Cs4	x	Cs5	Cs6	x	x
Cs5	x	Cs0	x	x	x
Cs6	x	x	Cs7	x	x
Cs7	x	x	Cs0	x	x
Cs8	x	x	x	Cs0	x
Cs9	x	x	x	x	Cs0

- The binary codes for state transition are:

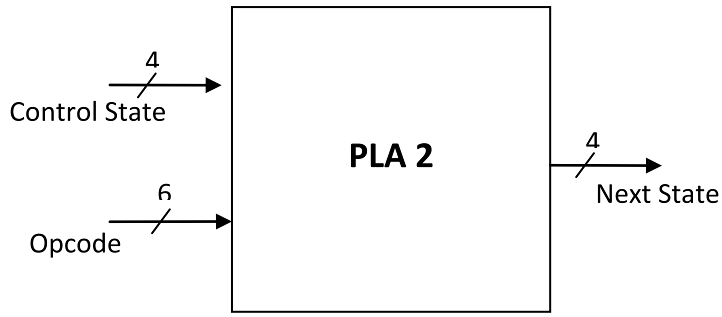
		Opcode				
		000000	**101011**	**100011**	**000100**	**000010**
Present state	**0000**	0001	0001	0001	0001	0001
	0001	0010	0100	0100	1000	1001
	0010	0011	xxxx	xxxx	xxxx	xxxx
	0011	0000	xxxx	xxxx	xxxx	xxxx
	0100	xxxx	0101	0110	xxxx	xxxx
	0101	xxxx	0000	xxxx	xxxx	xxxx
	0110	xxxx	xxxx	0111	xxxx	xxxx
	0111	xxxx	xxxx	0000	xxxx	xxxx
	1000	xxxx	xxxx	xxxx	0000	xxxx
	1001	xxxx	xxxx	xxxx	xxxx	0000

- Rearrange the control state transitions

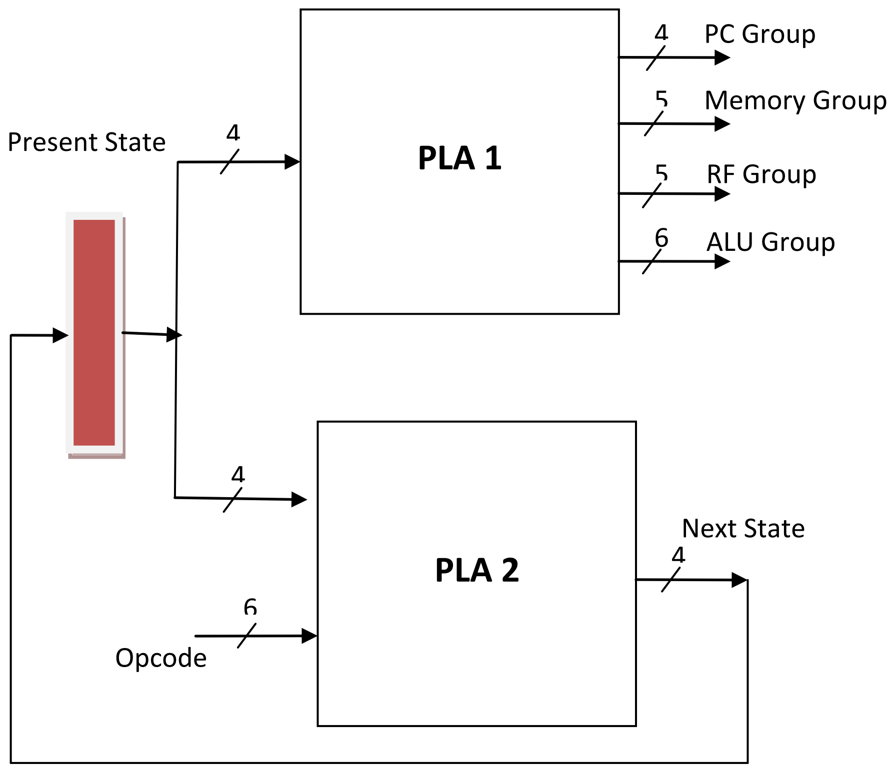
Present State	Opcode	Next State
Cs0	x	Cs1
Cs1	R_Class	Cs2
Cs1	sw / lw	Cs4
Cs1	beq	Cs8
Cs1	j	Cs9
Cs2	x	Cs3
Cs3	x	Cs0
Cs4	sw	Cs5
Cs4	lw	Cs6
Cs5	x	Cs0
Cs6	x	Cs7
Cs7	x	Cs0
Cs8	x	Cs0
Cs9	x	Cs0

Present State	Opcode	Next State
0000	xxxxxx	0001
0001	000000	0010
0001	10x011	0100
0001	000100	1000
0001	000010	1001
0010	xxxxxx	0011
0011	xxxxxx	0000
0100	101011	0101
0100	100011	0110
0101	xxxxxx	0000
0110	xxxxxx	0111
0111	xxxxxx	0000
1000	xxxxxx	0000
1001	xxxxxx	0000

PLA to determine next state



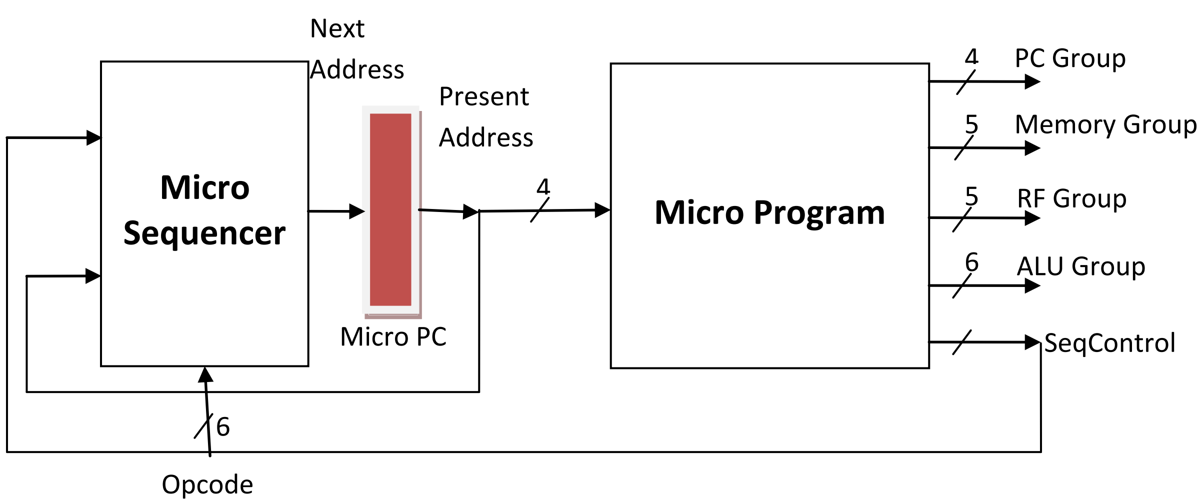
Controller Design with PLAs



Microprogrammed control

- In hardwired control, we saw how all the control signals required inside the CPU can be generated using a state counter and a PLA circuit
- The hardwired control unit lack flexibility in design. In addition, it is quite difficult to design, test and implement as in many computers the number of control lines is in hundreds.
- Is there any alternate approach of implementing control unit? What about a programming approach for implementing control unit? Can we somehow implement the sequence of execution of micro-operations through a program?
- Such a program will consist of instructions, with each of the instruction describing:
  - One or more micro-operations to be executed
  - and the information about the microinstruction to be executed next.
- Such an instruction is termed as **Microinstruction** and such a program is termed as a **Microprogram**.
- A Microprogrammed control is a midway between hardware and software
- A Microprogrammed control is a control unit with its binary control values stored as words in memory. Each word in the control memory contains a microinstruction that specifies one or more microoperations for the system. A sequence of microinstructions constitutes a microprogram.
- Using microprogramming, architects could build simple hardware and then microprogram that hardware to execute complex instructions

Microprogrammed control organization



Micro Program:

- Micro program stores in the memory
- Memory word consists of the following fields:
    - Control signals (20-bit)
        - 4-bit for PC Group
        - 5-bit for Memory Group
        - 5-bit for RF Group
        - 6-bit for ALU Group

    - Branch address (BA)
        - The number of instructions in the microprogram is 10, so we need 4-bit for addressing the instructions.
        - In this design, we ignore the branch address and using simple logic circuit (PLA) to generate the branch address depending on the Opcode.

    - Condition select (CS)
        - The microprogram includes the following types of sequences:
            - No branch (sequence)
            - Dispatch1 (condition1)
            - Dispatch2 (condition2)
            - Unconditional branch (reset)
        - So we need 2-bit for sequence control

- The word length is: 2 + 4 + (6 + 5 + 5 + 4) = 26 bits
- The memory size is: (word length) x (# of instructions) = 26 x 10 = 260 bits

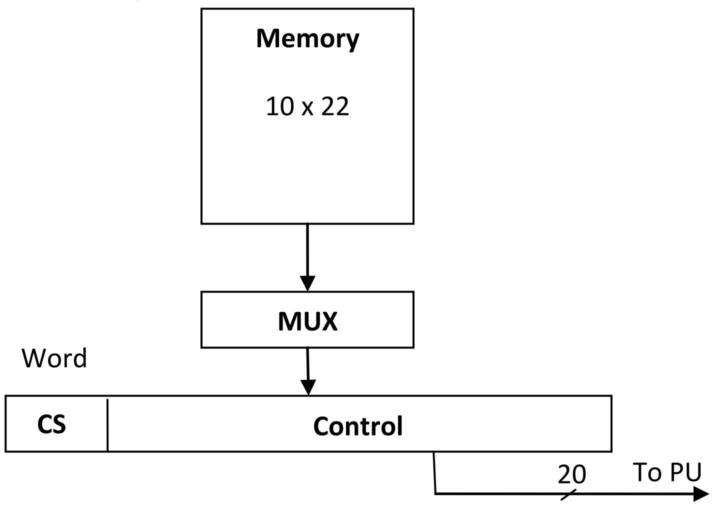
CS	BA	Control			
		ALU Group	RF Group	Mem Group	PC Group

- Note: in this design we ignore the branch address, so we need 22 bits for memory word. The size of the memory = 22 x 10 = 220 bits

- Microprogram:

First:	fetch, PCinc, seq
	Rs2A, rt2B, Paddr, dispatch1
1a:	arith, seq
	Alu2rd, reset
1b:	Maddr, dispatch2
2a:	M_Wr, reset
2b:	M_Rd, seq
	Mem2rt, reset
1c:	branch, reset
1d:	jump, reset

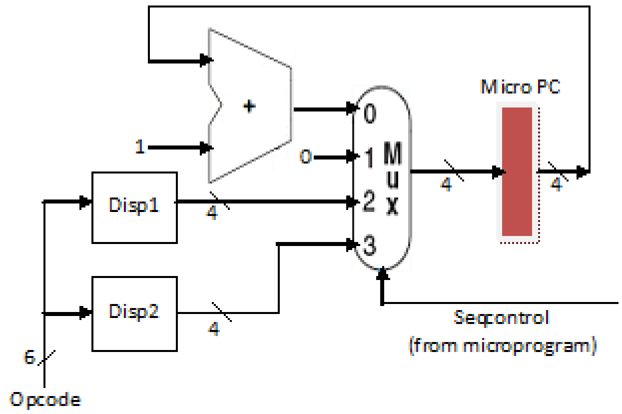
Microprogram design



MicroSequencer

- Micro sequencer is used to ensure that the right address stored in the micro PC
- Micro sequence control operations
  - Sequence
  - Branch to cs2 | cs4 | cs8 | cs9
    - Dispatch 1
  - Branch to cs5 | cs6
    - Dispatch 2
  - Goto cs0
    - Reset
- There are 4 types:
  - Sequence 00
  - Reset 01
  - Dispatch1 10
  - Dispatch2 11

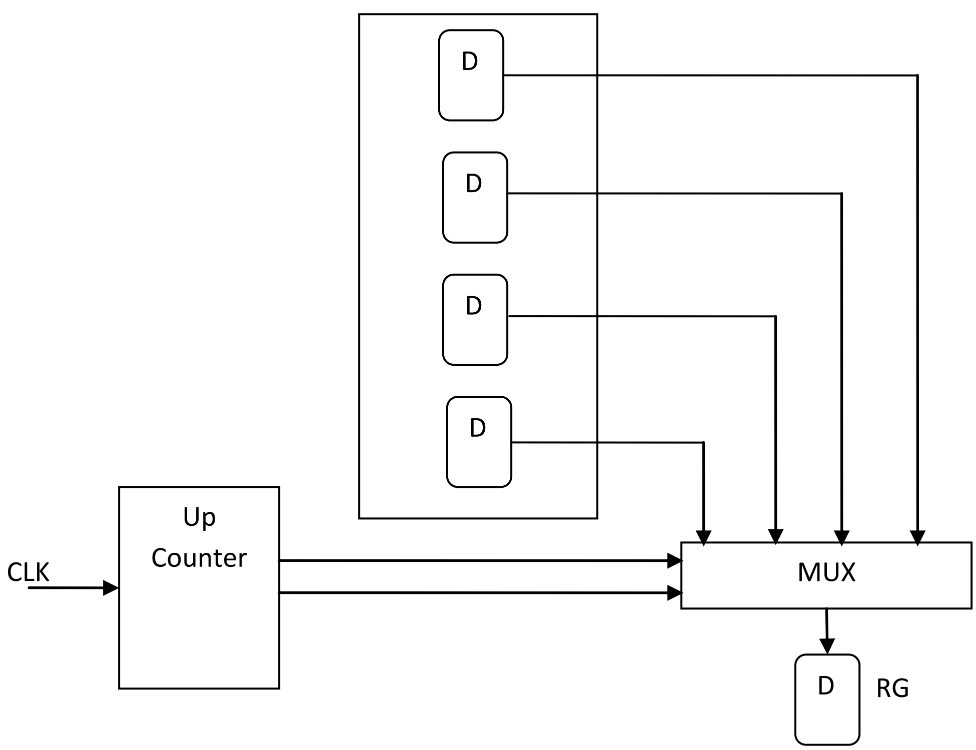
Microsequencer Design



- Depending on the opcode, dispatch1 generate a correct address between the 4 possible addresses and dispatch2 generate a correct address between the 2 possible addresses.
- We can implement dispatch1 and 2 using PLA

Example1:

Consider a memory consisting of 4 bits, each bit is form a word, show how to read the bits sequentially (bit$_0$, bit$_1$, …) into a register



Example2:

Design a control unit for the following multiplication algorithm

Pseudo code:

```
        Input A      //Multiplicand
        Input B      //Multiplier
        S <= 0       //Product
L2:     If B[0] = 0 goto L1
        S <= S + A
L1:     SHL A
        SHR B
        If B <> 0 goto L2
        Output S
L3:     goto L3
```

Solution:

Step1: Convert the pseudo code into a form of control signals

```
        C0
        C1
        C2
L2:     If cond1 goto L1
        C3
L1:     C4
        C5
        If cond2 goto L2
        C6
L3:     goto L3
```

Step2: Define the word structure (memory word)

- Memory word consists of the following fields:
  1. Control signals (7 bits (C0, C1, …. , C6))
  2. Branch address (BA): the number of instructions =10 so we need 4bits for addressing the instructions in the memory
  3. Condition select (CS): from the pseudo code we can see that it is include the following types of sequencing:
     - No branch
     - Branch if cond1
     - Branch if cond2
     - Unconditional branch
- We need 2 bits to code these types 00, 01, 10, 11

- The word structure will be as follows:

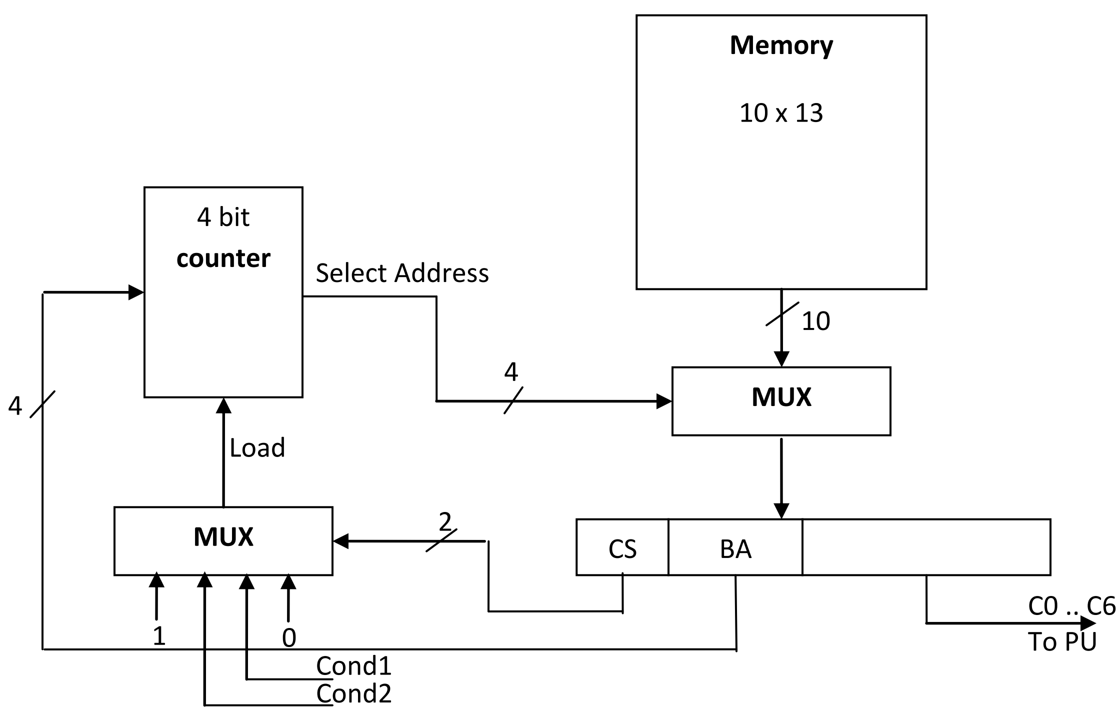
CS		BA				Control						
						C6	C5	C4	C3	C2	C1	C0

- Word length = 2 + 4 + 7 = 13 bits
- Memory size = number of words x word length
  = 10 x 13 = 130 bits (flip-flops)

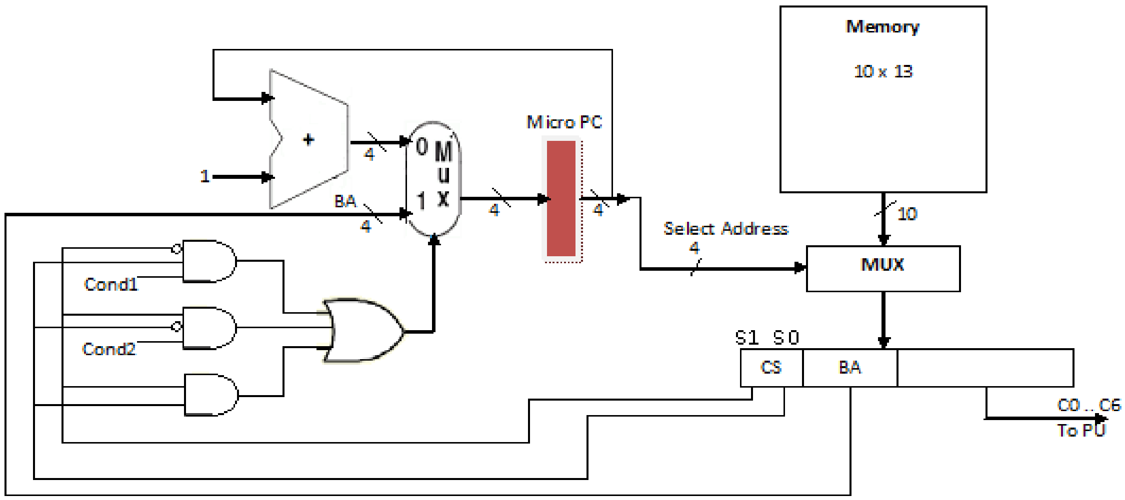
Step3: Convert the pseudo code to binary (the binary program to be loaded into memory)

Memory Address	CS	BA	Control
0000	00	0000	0000001
0001	00	0000	0000010
0010	00	0000	0000100
0011	01	0101	0000000
0100	00	0000	0001000
0101	00	0000	0010000
0110	00	0000	0100000
0111	10	0011	0000000
1000	00	0000	1000000
1001	11	1001	0000000

Step4: Build the controller



Another design (without counter)



Example: **Design a control unit for the following algorithm.**