Chapter 5

The Processor: Datapath and Control

MIPS Control Part - II

Control

- **Selecting the operations to perform (ALU, read/write, etc.)**
- **Controlling the flow of data (multiplexor inputs)**
- **Information comes from the 32 bits of the instruction**
- **Example:**

  add $8, $17, $18   Instruction Format:

000000	10001	10010	01000	00000	100000

op	rs	rt	rd	shamt	funct

- **ALU's operation based on instruction type and function code**

The ALU Control

- **ALU has 3 control inputs ($2^3 = 8$).**
- **Only 5 of the possible 8 input combinations are used.**
- **ALU control input — Function**

ALU control input	Function
`000`	`AND`
`001`	`OR`
`010`	`add`
`110`	`subtract`
`111`	`set-on-less-than`

- **Depending on the instruction class, ALU will need to perform one of these five functions.**
- **For load word and store word instructions, we use ALU to compute memory address by addition.**
- **For R-type instructions, ALU needs to perform 1 of 5 actions (AND, OR, SUB, ADD, or Set on less than), depending on the value of 6-bit funct field in low-order bits of instruction.**
- **For branch equal, ALU must perform a subtraction.**

The ALU Control

- **e.g., what should the ALU do with this instruction:**
  Example: lw $1, 100($2)

35	2	1	100
op	rs	rt	16 bit offset

- **Why is the code for subtract 110 and not 011?**
- **We can generate 3-bit ALU control unit input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field called ALUOp.**
- **ALUOp indicates the following:**
  - **Add for loads and stores → 00**
  - **Subtract for beq → 01**
  - **Determined by operation encoded in funct field → 10**

The ALU Control

ALU control bits are set depends on ALUOp control bits and different function codes for R-type instruction

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	Add	010
SW	00	store word	XXXXXX	Add	010
Branch equal	01	Branch equal	XXXXXX	Subtract	110
R-type	10	Add	100000	Add	010
R-type	10	Subtract	100010	Subtract	110
R-type	10	AND	100100	And	000
R-type	10	OR	100101	Or	001
R-type	10	Set on less than	101010	Set on less than	111

The ALU Control

- Notice from table in previous slide that when ALUOp code is 00 or 01, the output fields do not depend on the function code field.
- Also, when ALUOp value is 10, then the function code is used to set the ALU control input.
- Multiple levels of decoding: Main control unit generates ALUOp bits, which then are used as input to ALU control that generates actual signals to control the ALU unit.
- Using multiple levels of control can reduce the size of the main control unit.
- Using several smaller control units may also potentially increase the speed of the control unit.

The ALU Control

- Must describe hardware to compute 3-bit ALU control input
  - given instruction type
    - 00 = lw, sw
    - 01 = beq,
    - 10 = arithmetic

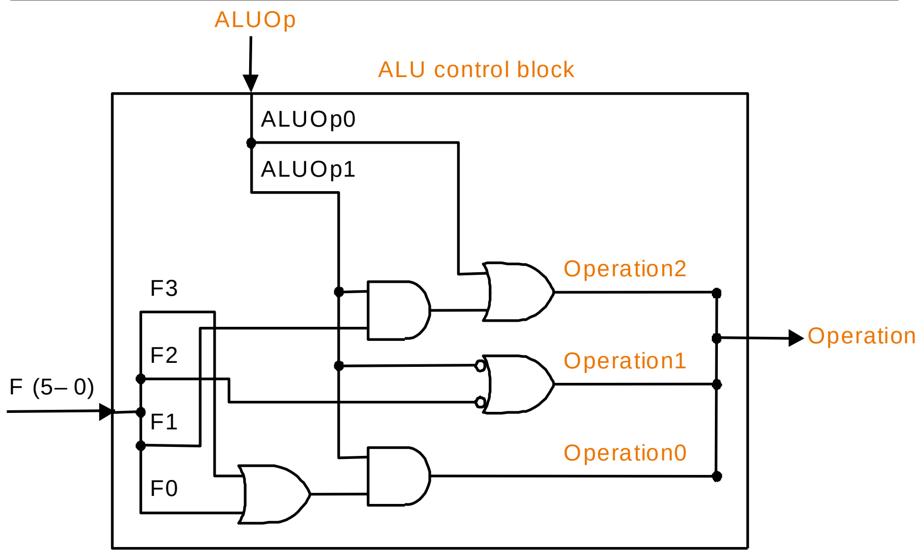
    ALUOp
    computed from instruction type
  - function code for arithmetic
- Describe it using a truth table (can turn into gates):

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
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

The ALU Control

- Notice from table in previous slide, only entries for which ALU control is asserted are shown.
- ALUOp does not use encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01.
- Also, when the function field is used, the first two bits (F5 and F4) of these instructions are always 10, so they are don’t care terms and replaced with XX in the truth table.
- Finally, when the ALUOp bits are 00, as in the first line of the table, we always set the ALU control to 010, independent of the function code.

The ALU Control – (In gates level from table in slide 7)



Designing the Main Control Unit

- First, review the format of the three instruction classes:
  - R-type instruction:

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

  - Load or store instruction:

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

  - Branch instruction:

4	rs	rt	address
31-26	25-21	20-16	15-0

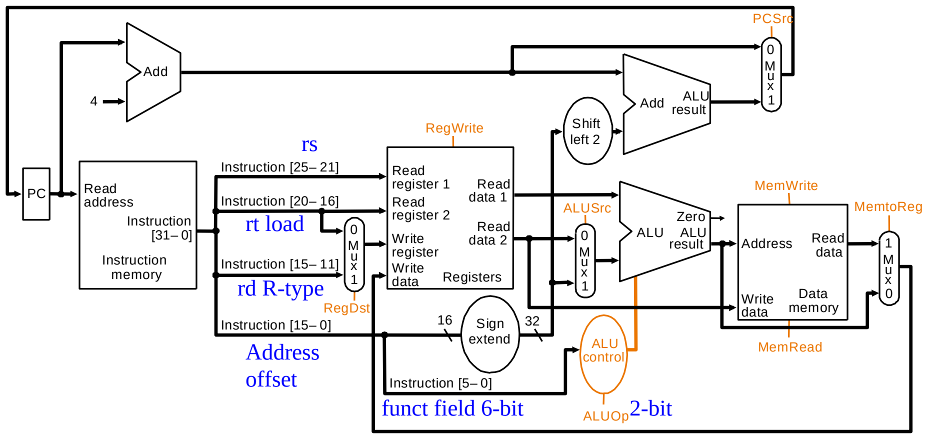
Three Instruction Classes Format

- Instruction format for R-type all have an opcode of 0
- R-type instructions have three operands: fields rs and rt are sources, and rd is the destination.
- Instruction format for load (opcode =35) and store (opcode =43) instructions.
  - Register rs is base register that is added to 16-bit address field to form memory address.
  - For loads, rt is destination register for loaded value.
  - For stores, rt is source register whose value should be stored into memory.
- Instruction format for branch equal (opcode = 4)
  - Registers rs and rt are the source registers that are compared for equality.
  - 16-bit address field is sign-extended, shifted, and added to PC to compute branch target address

Major Observations About Instruction Format

- The opcode is always contained in bits 31-26.
- The two registers to be read are always specified by rs and rt fields, at positions 25-21 and 20-16. This is true for R-type instructions, branch equal, and store.
- The base register for load and store instructions is always in positions 25-21 (rs).
- The 16-bit offset for branch equal, load, and store is always in positions 15-0.
- The destination register is in one of two places:
  - For a load it is in bit positions 20-16 (rt)
  - For R-type instruction it is in bit positions 15-11 (rd)
  - Thus, we will need to add a multiplexor to select which field of instruction is used to indicate the register number to be written.

Using the observations, we can add instruction labels and extra multiplexor to simple datapath.



- PC does not require a write control, since it is written once at the end of every clock cycle. Also, since all the multiplexors have two inputs, they each require a single control line.
- There are seven 1-bit control lines + 2-bit ALUOp control signal.

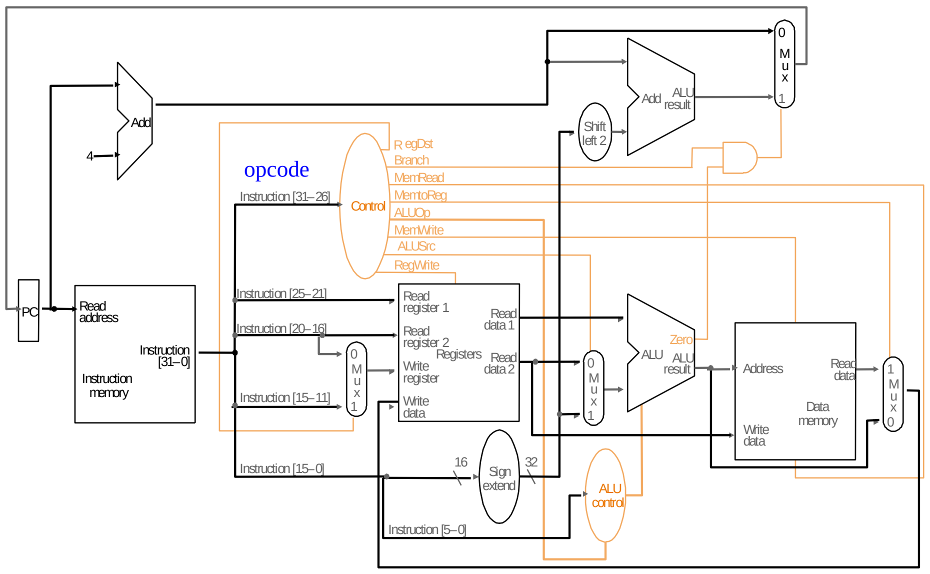
The effect of each of the seven control signals

Signal name	Effect when deasserted (MUX selects the 0 input)	Effect when asserted (MUX selects the 1 input)
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, lower16 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.

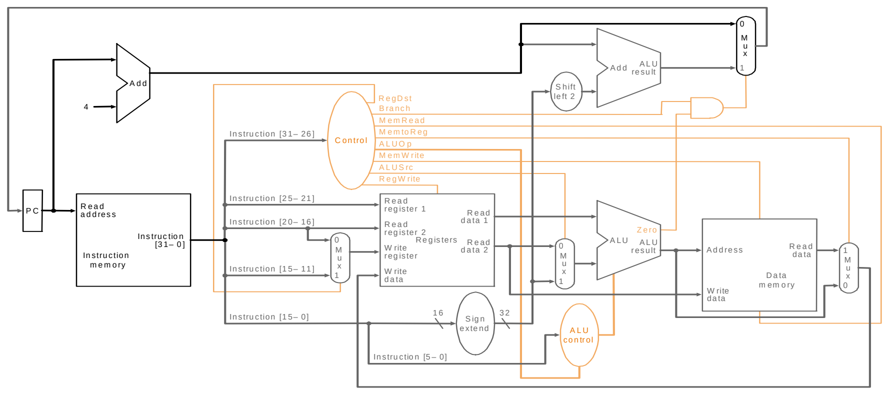
The Control Unit

- The input to the control unit is the 6-bit opcode field from the instruction; except the PCSrc control line.
- The outputs of the control unit consist of:
  - Three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg).
  - Three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite).
  - A 1-bit signal used in determining whether to possibly branch (Branch).
  - A 2-bit control signal for the ALU (ALUOp).
- For PCSrc control line: an AND gate is used to combine the branch control signal from the control unit and the Zero output from the ALU.
- The AND gate output controls the selection of the next PC.

The simple datapath with the control unit



Defines how the control signals should be set for each opcode



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

The setting of control lines is determined by opcode fields of instruction

- **First row of table in previous slide corresponds to R-format instructions**
  - **For all R-format instructions, the source register fields are rs and rt and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set.**
  - **An R-type instruction writes a register (RegWrite=1), but neither reads nor writes data memory.**
  - **When Branch control signal is 0, PC is unconditionally replaced with PC + 4; otherwise, PC is replaced by branch target if Zero output of ALU is also high.**
  - **ALUOp field for R-type instructions is set to 10 to indicate that ALU control should be generated from funct field.**

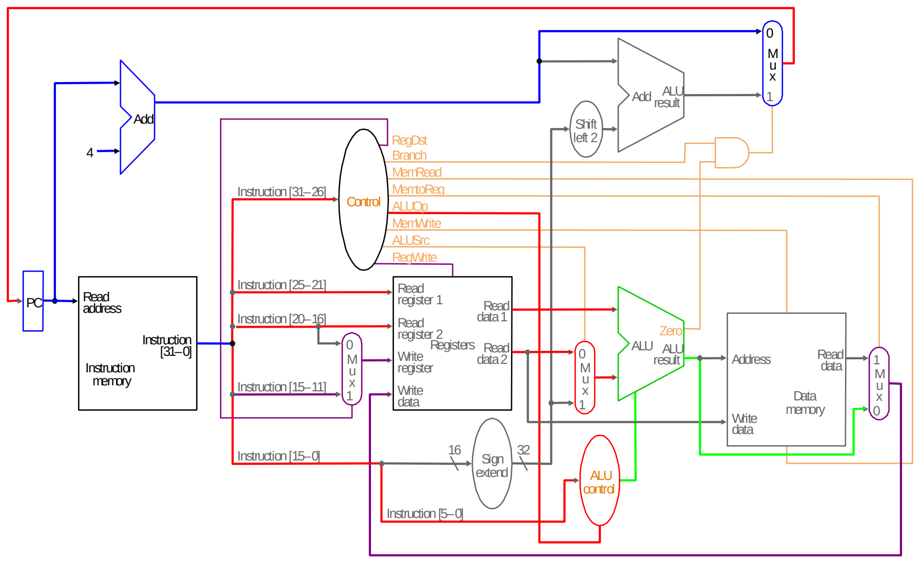
The setting of control lines is determined by opcode fields of instruction

- The second and third rows of table in previous slide give the control signal settings for lw and sw.
  - ALUSrc and ALUOp fields are set to perform the address calculation.
  - MemRead and MemWrite are set to perform the memory access.
  - RegDst and RegWrite are set for a load to cause the result to be stored into the rt register.
- The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU.
  - ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality.
  - Note: MemtoReg field is irrelevant when RegWrite signal is 0 – since register is not being written, the value of data on register data write port is not used. So, entry of MemtoReg in last two rows of table is replaced with X (Don’t care).
  - Don’t care can also be added to RegDst when RegWrite is 0.

Operation of the Datapath

- We show the flow of three different instruction classes through the datapath.
- Start with R-type instruction, such as `add $t1, $t2, $t3`
- There are four steps to execute an R-type instruction:
  1. An instruction is fetched from the instruction memory and PC is incremented (next slide highlighted in blue).
  2. Two registers, $t2 and $t3, are read from the register file. The main control unit computes the setting of the control lines during this step also (next slide highlighted in red).
  3. The ALU operates on the data read from the register file, using the function code (bits 5-0, which is the funct field, of the instruction) to generate the ALU function (next slide highlighted in green).
  4. The result from the ALU is written into the register file using bits 15-11 of the instruction to select the destination register $t1 (next slide highlighted in purple).
- The datapath operates in a single clock cycle.

Operation of the datapath for ***R –type*** instruction



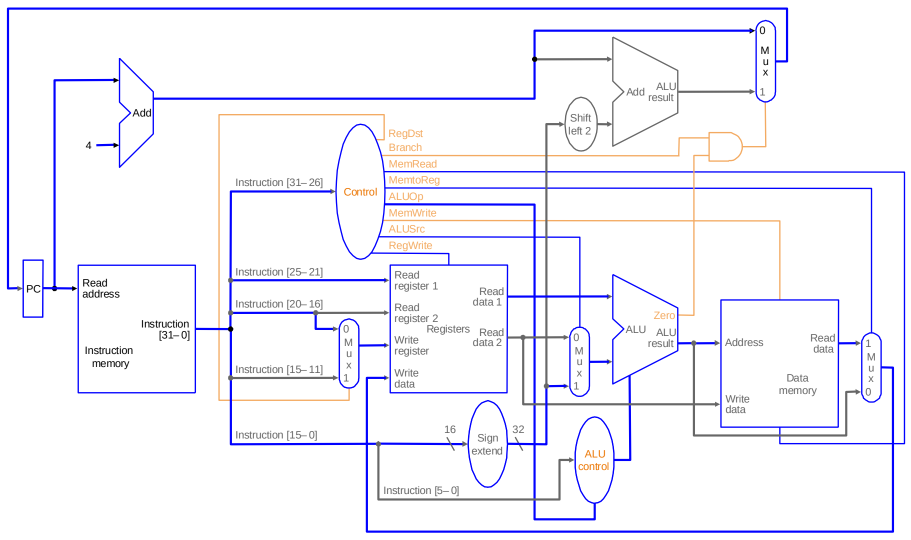
Operation of the Datapath for a ***load*** Instruction

- **Example:** `lw $t1, offset ($t2)`

We can think of a load instruction as operating in 5 steps (next slide shows these operations highlighted in blue):

1. **An instruction is fetched from instruction memory and PC is incremented**
2. **A register ($t2) value is read from register file.**
3. **The ALU computes the sum of the value read from register file and sign-extended, lower 16 bits of instruction (offset).**
4. **The sum from ALU is used as the address for data memory.**
5. **The data from memory unit is written into register file; register destination is given by bits 20-16 of the instruction ($t1).**

Operation of the Datapath for a ***load*** Instruction



Operation of the Datapath for a ***store*** Instruction

- **A store instruction would operate very similar to a load instruction.**
- **The main differences would be that:**
  - **The memory control would indicate a write rather than a read.**
  - **The second register value read would be used for the data to store.**
  - **The operation of writing the data memory value to the register file would not occur.**

Operation of the Datapath for a ***branch-on-equal*** Instruction

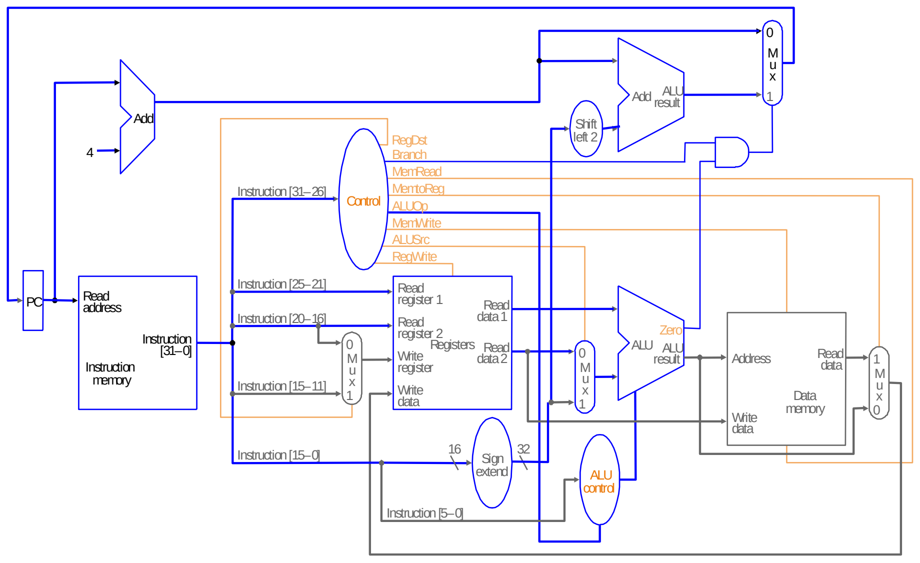
- **Example:** `beq $t1, $t2, offset`

  It operates like an R-format instruction, but ALU output is used to determine whether PC is written with PC + 4 or branch target address.

  So, we can think of a branch equal instruction as operating in 4 steps (next slide shows these operations highlighted in blue):

  1. **An instruction is fetched from the instruction memory and the PC is incremented.**
  2. **Two registers, $t1 and $t2, are read from the register file.**
  3. **The ALU performs a subtract on data values read from register file. The value of PC + 4 is added to sign-extended, lower 16 bits of the instruction (offset) shifted left by 2; the result is branch target address.**
  4. **The Zero result from ALU is used to decide which adder result to store into the PC.**

Operation of the Datapath for a ***branch-on-equal*** Instruction



Finalizing the Control

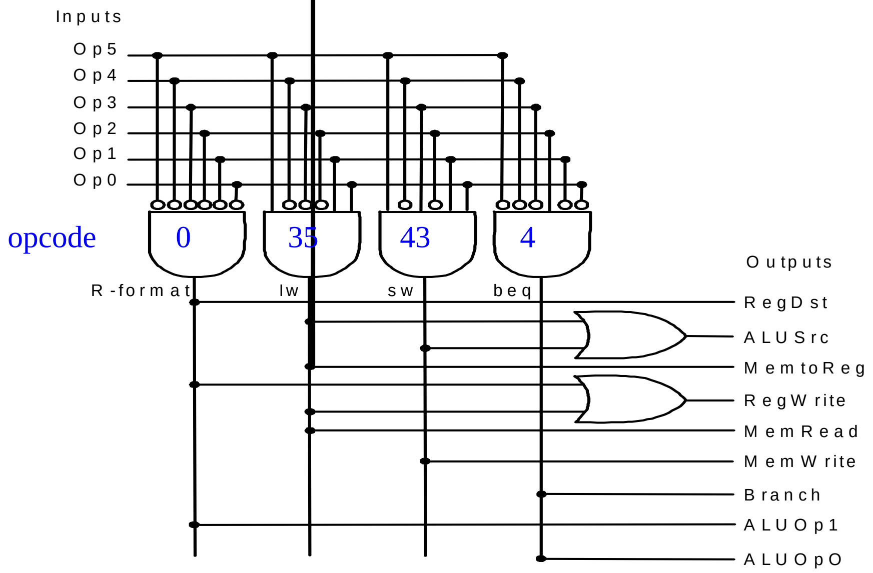
- The outputs of the control unit are the control lines.
- The inputs of the control unit are the 6-bit opcode field.
- The encoding for each of the opcodes, both as a decimal number and as a series of bits that are input to the control unit:

Name	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

The control function for single-cycle implementation is specified by truth table:

Input or Output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

The Control Unit – In Gates level



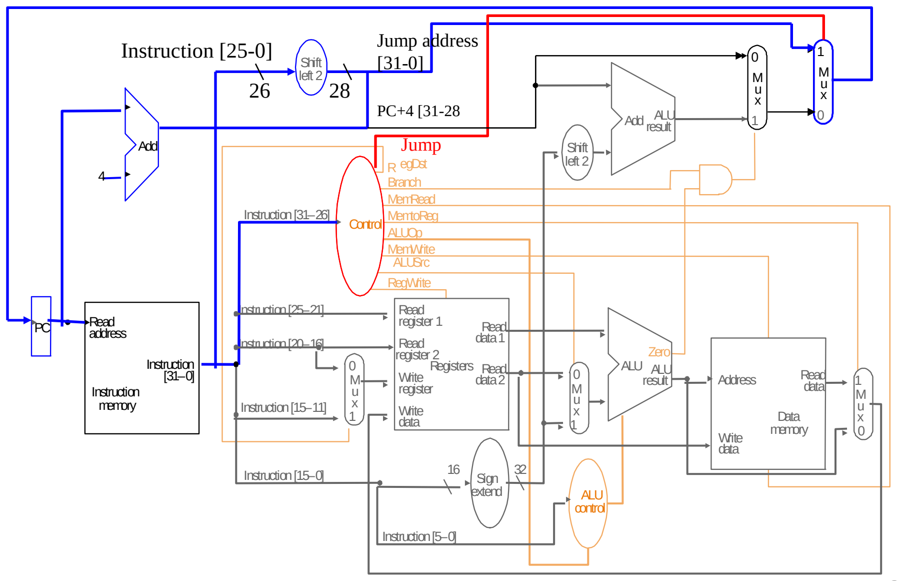
Implementing Jumps

- **The jump instruction looks like a branch instruction but computes the target PC differently and is not conditional.**
- **Instruction format for jump instruction (opcode = 2):**

Field	**2**	**Address**
Bit Position	**31-26**	**25-0**

- **We can implement a jump by storing into PC the concatenation of the following:**
  - **The upper 4 bits of the current PC + 4 (these bits 31-28 of the sequentially following instruction address)**
  - **The 26-bit immediate field of the jump instruction**
  - **Adding 00 as the 2 low-order bits.**

The control and datapath are extended to handle the jump instruction



The control and datapath are extended to handle the jump instruction

- **From the figure in the previous slide, an additional multiplexor is used to select the source for the new PC value, which is either the incremented PC (PC+4), the branch target PC, or the jump target PC.**
- **One additional control signal is needed for the additional multiplexor.**
- **This control signal, called `jump`, is asserted only when the instruction is a jump – that is, when the opcode is 2.**

Why a Single-Cycle Implementation Is not Used

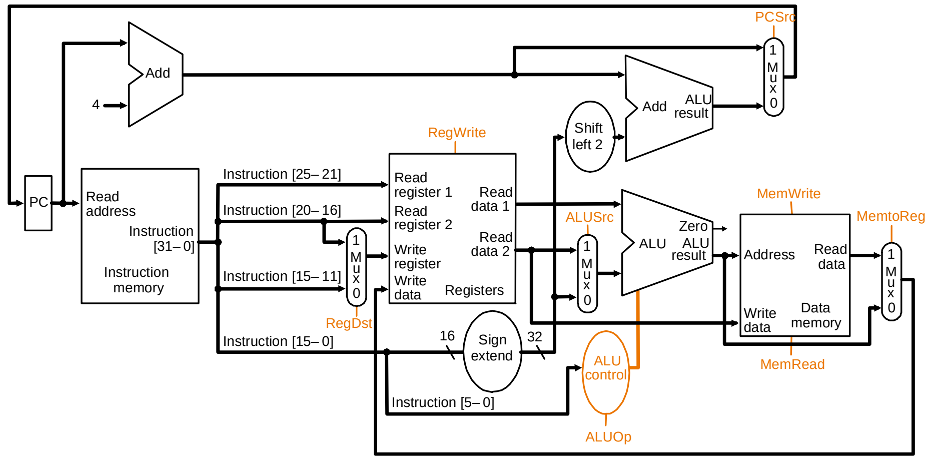
- It is inefficient, because the clock cycle must have the same length for every instruction in this single-cycle design, and the CPI will therefore be 1.
- The clock cycle is determined by the longest path in the machine.
- This path is the load instruction, which uses 5 functional units in series: instruction memory, register file, ALU, data memory, and register file.
- Although the CPI is 1, the overall performance of a single-cycle implementation is not likely to be very good, since several of the instruction classes could fit in a shorter clock cycle.

Example 1: Performance of Single-Cycle Machines

- Assume that the operation times for the major functional units in the single-cycle implementation are the following:
  - Memory units: 2 nanoseconds (ns)
  - ALU and adders: 2 ns
  - Register file (read or write): 1 ns
- Assuming that the multiplexors, control unit, PC accesses, sign extension unit, and wires have no delay, *which of the following implementations would be faster and by how much?*
  1. An implementation in which every instruction operates in 1 clock cycle of a fixed length.
  2. An implementation where every instruction executes in 1 clock cycle using a variable-length clock, which for each instruction is only as long as it needs to be.
- To compare the performance, assume the following instruction mix: 24% loads, 12% stores, 44% ALU instructions, 18% branches, and 2% jumps.

Single Cycle Implementation

- **Calculate cycle time assuming negligible delays except:**
  - **memory (2ns), ALU and adders (2ns), register file access (1ns)**



Answer: Performance of Single-Cycle Machines

- **CPU execution time =**

  Instruction count x CPI x Clock cycle time
  - **Since CPI is equal to 1 then:**
  - **CPU execution time =**

    Instruction count x Clock cycle time

- **We need only to find the clock cycle time for the two implementations, since the instruction count and CPI are the same for both implementations.**

Answer (Cont.)

- **The critical path for different instruction classes is as follows:**

Instruction class	Functional units used by the instruction class				
ALU type	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				

Answer (Cont.)

- **Using these critical paths from the previous slide, we can compute the required length for each instruction class:**

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
ALU type	2	1	2	0	1	6 ns
Load word	2	1	2	2	1	8 ns
Store word	2	1	2	2		7 ns
Branch	2	1	2			5 ns
Jump	2					2 ns

Answer (Cont.)

- The clock cycle for a machine with a single clock for all instructions will be determined by the longest instruction, which is 8 ns.
- A machine with a variable clock will have a clock cycle that varies between 2 ns and 8 ns.
- We can find the average clock cycle length for a machine with a variable-length clock using the information in previous tables and the instruction frequency distribution.
- The average time per instruction with a variable clock is:

$$\text{CPU clock cycle} = 8\times24\% + 7\times12\% + 6\times44\% + 5\times18\% + 2\times2\% = 6.3 \text{ ns}$$

Answer (Cont.)

- Since the variable clock implementation has a shorter average clock cycle, it is clearly faster.
- Performance ration:

$$\frac{\text{CPU Performance}_{\text{variable clock}}}{\text{CPU Performance}_{\text{single clock}}} = \frac{\text{CPU Execution Time}_{\text{single clock}}}{\text{CPU Execution Time}_{\text{variable clock}}}$$

$$= \frac{\text{IC x CPU clock cycle}_{\text{single clock}}}{\text{IC x CPU clock cycle}_{\text{variable clock}}}$$

$$= \frac{\text{CPU clock cycle}_{\text{single clock}}}{\text{CPU clock cycle}_{\text{variable clock}}}$$

$$= 8 / 6.3 = 1.27$$

- The variable clock implementation would be 1.27 times faster.

Remarks from Example 1

- **Unfortunately, implementing a variable-speed clock for each instruction class is extremely difficult.**
- **An alternative is to use a shorter clock cycle that does less work and then vary the number of clock cycles for different instruction classes.**

General Remarks

- **With the single-cycle implementation, each functional unit can be used only once per clock; therefore, some functional units must be duplicated, raising the cost of the implementation.**
- **A single-cycle design is inefficient both in its performance and in its hardware cost!**
- **In the next chapter, we will look at another implementation technique, called pipelining, that uses a datapath very similar to the single-cycle datapath, but is much more efficient.**
- **Pipelining gains efficiency by overlapping the execution of multiple instructions, increasing hardware utilization and improving performance.**