

Module 6

Computer Architecture

PROCESSOR: DATA PATH AND CONTROL

The implementation of the processor is studied with respect to a subset of the full instruction set: the core set (most common).

- Memory reference (lw and sw)
- Arithmetic-logical instructions (add, sub, and, or and slt).
- Branch and jump instructions (beq, j).

Guidelines: *Make the common case fast*
Simplicity and regularity

All instructions have this in common:

Use the content of the program counter as an address to the memory, which returns the next instruction;

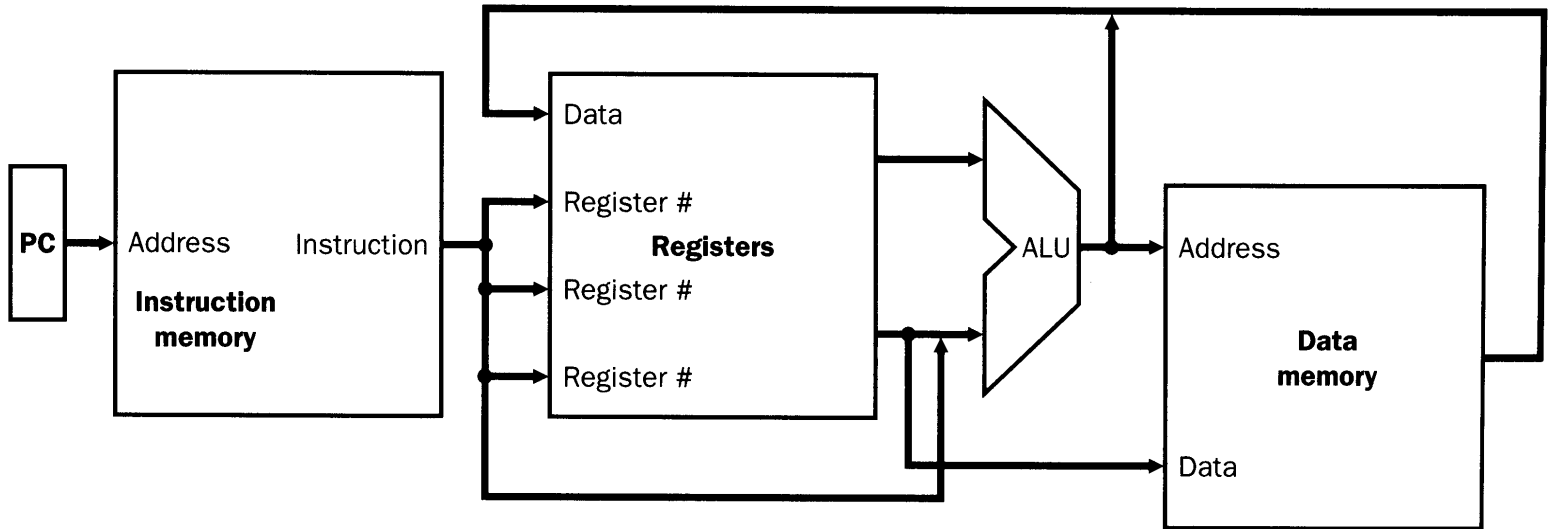
Read one or two registers (except j) using fields to select the registers to be read.

Then, depending on the instruction type, actions can differ, but they will all use the ALU (arithmetic-logicals, memory reference) to calculate effective addresses.

Finally, the actions needed to complete the instruction differ.

- *memory reference*: loads read memory and write back register file; stores write memory;
- *arithmetic-logicals*: write back to register file;
- *branches*: may change the content of the program counter, depending on a comparison.

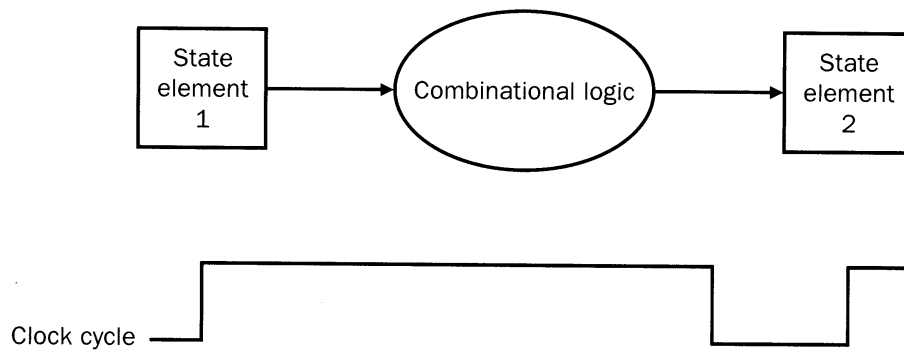
THE FLOW OF INFORMATION (DATA PATH)



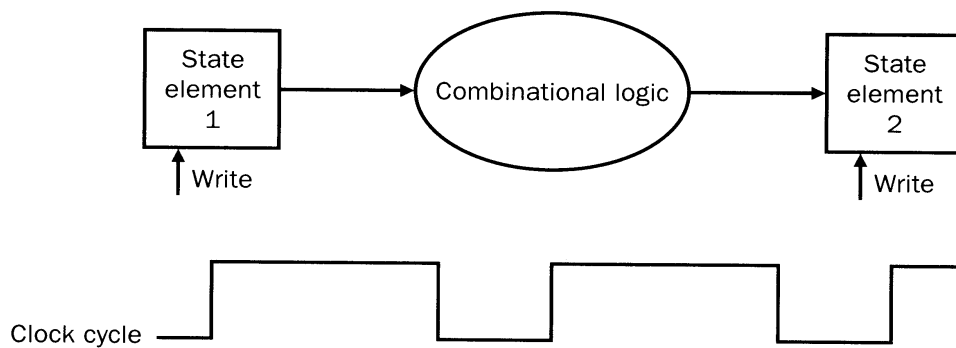
How information flows between functional units?: trace all possible paths needed to implement any instructions (5.1)

RECALL ON LOGIC CIRCUITS

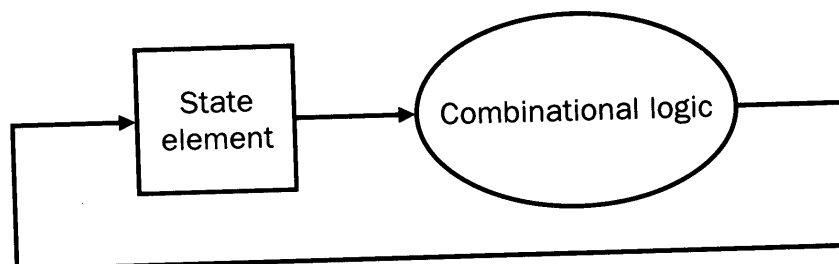
- Combinatorial: $Outputs = F(Inputs)$
- Sequential: $State_i = S(Inputs_{i+1}, State_i)$;
 $Outputs_i = G(Inputs_i, State_i)$.
- Edge triggered clock



(5.2)



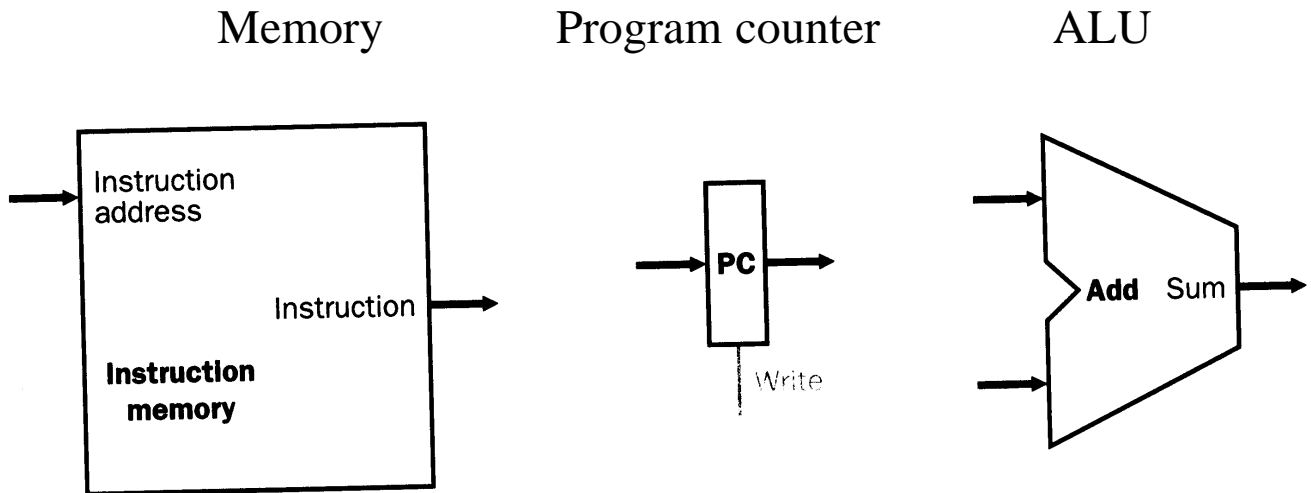
(5.3)



(5.4)

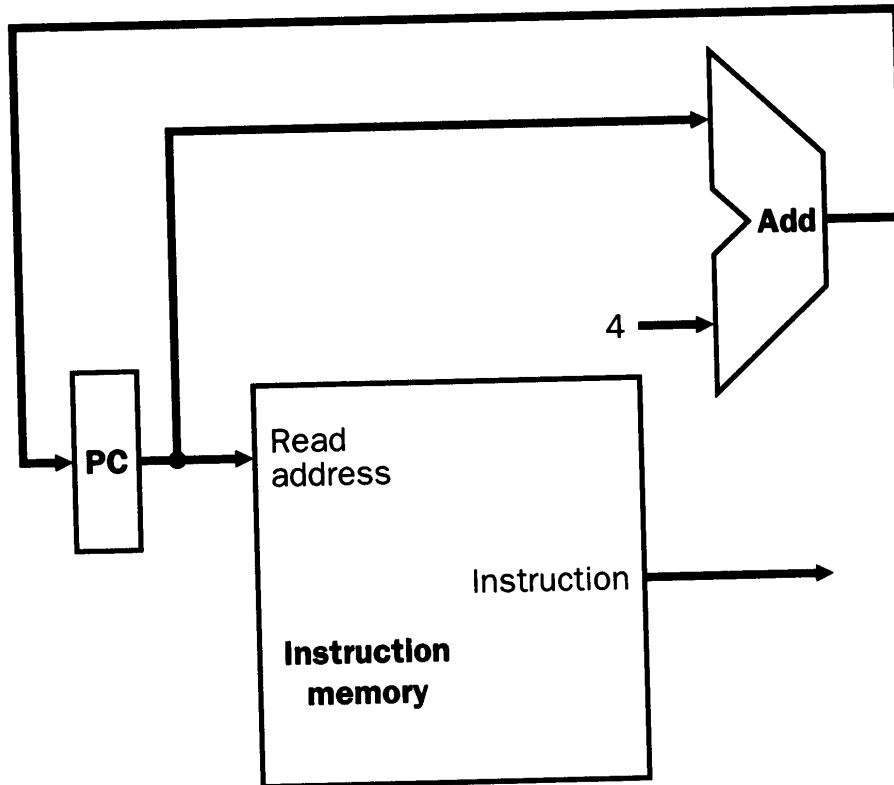
- Buses

BUILDING BLOCKS

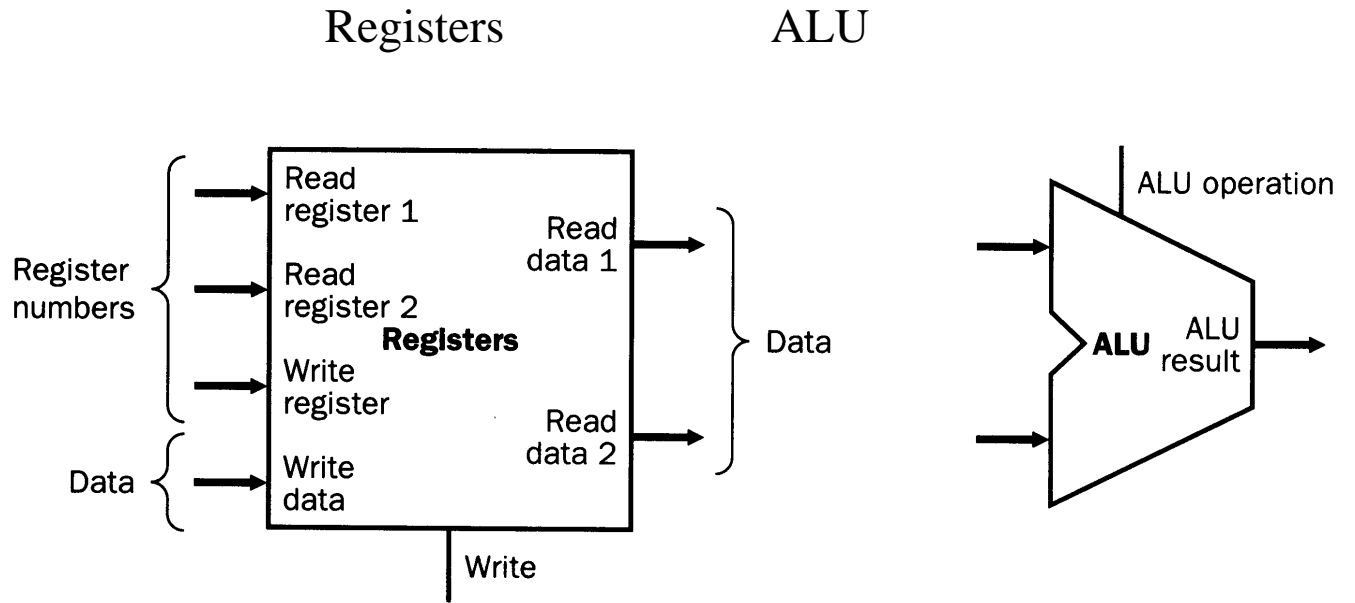


(5.5)

Put this together to implement the *instruction fetch*. (which includes the automatic PC increment circuit).

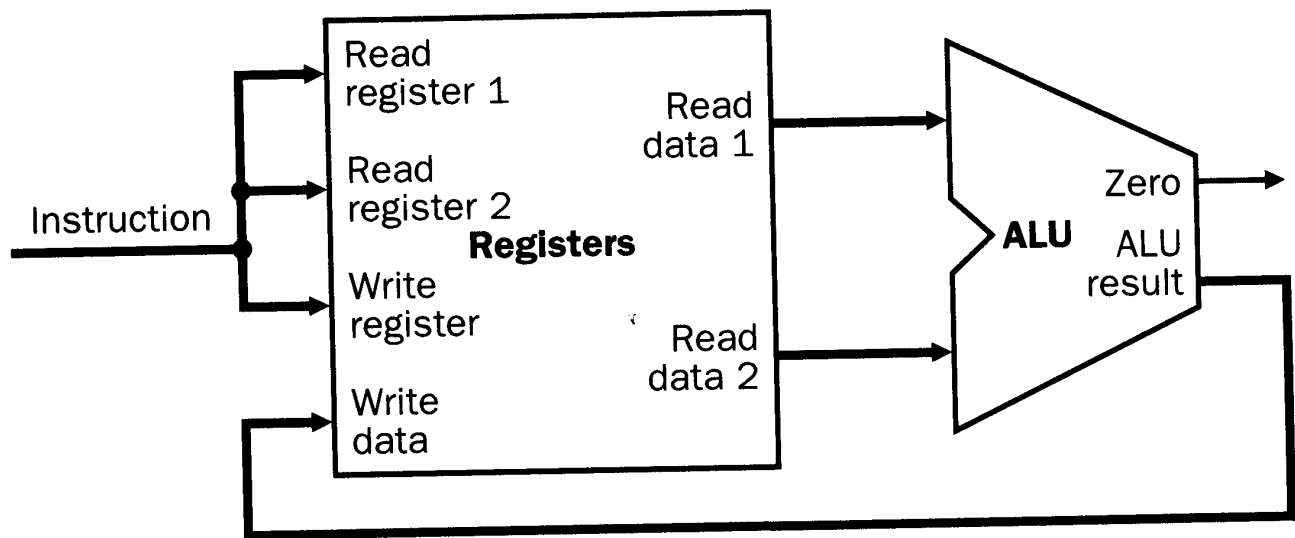


(5.6)



(5.7)

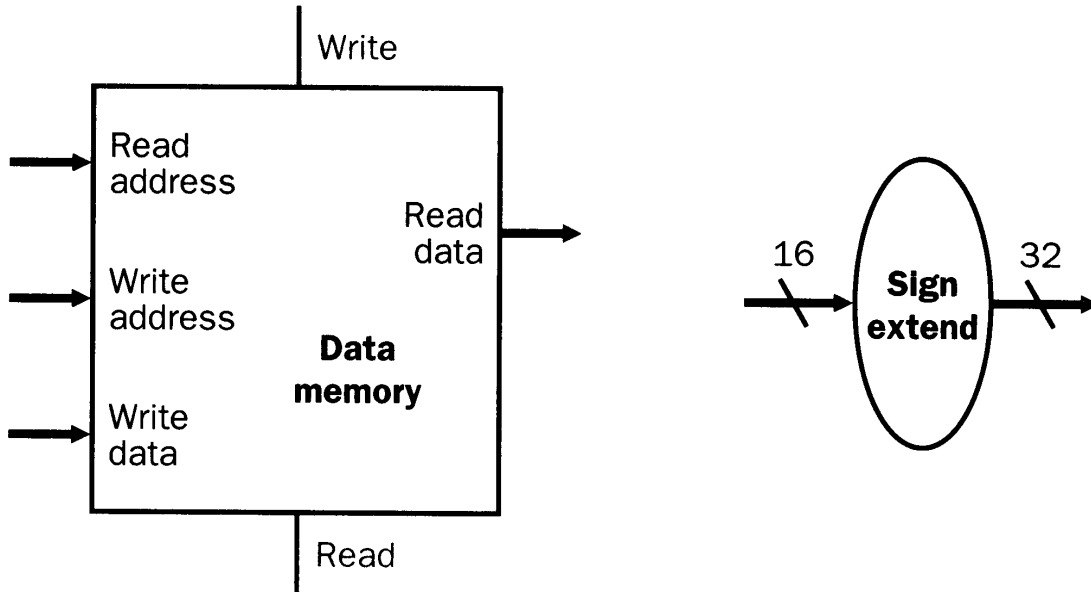
Data path for R-Type instructions



(5.8)

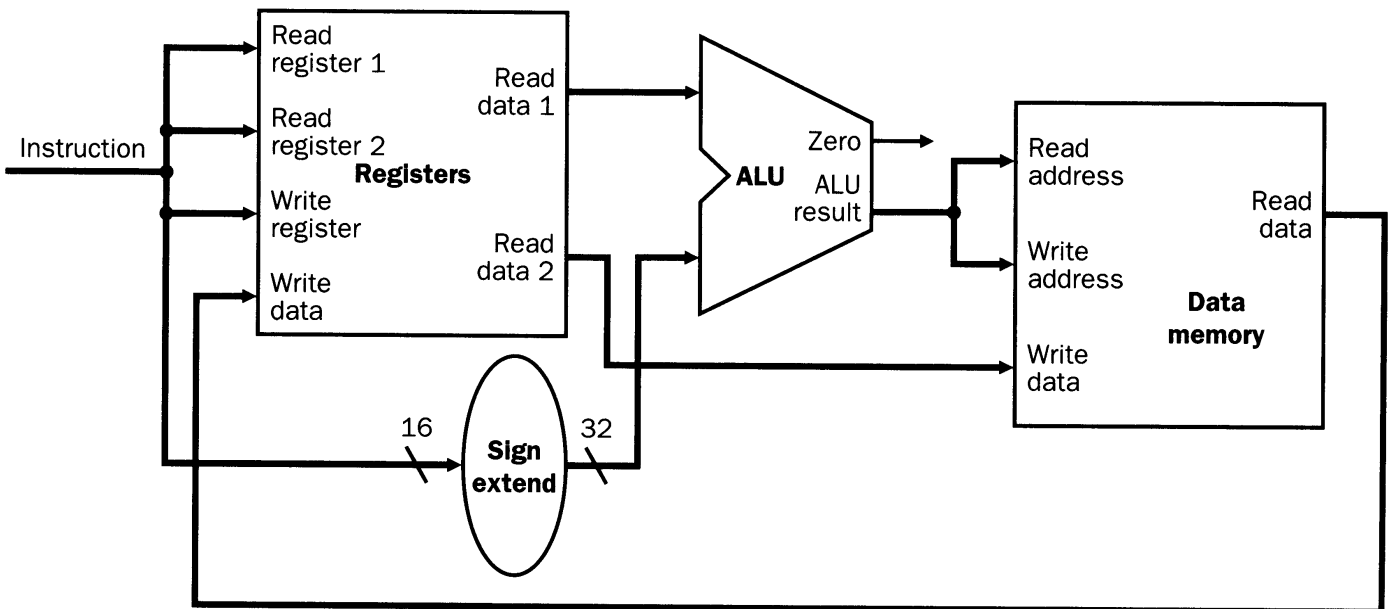
Memory

sign-extension unit

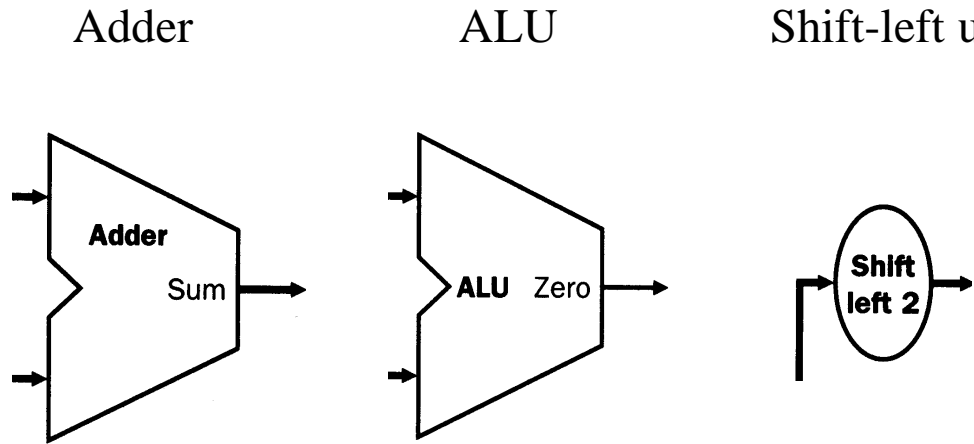


(5.9)

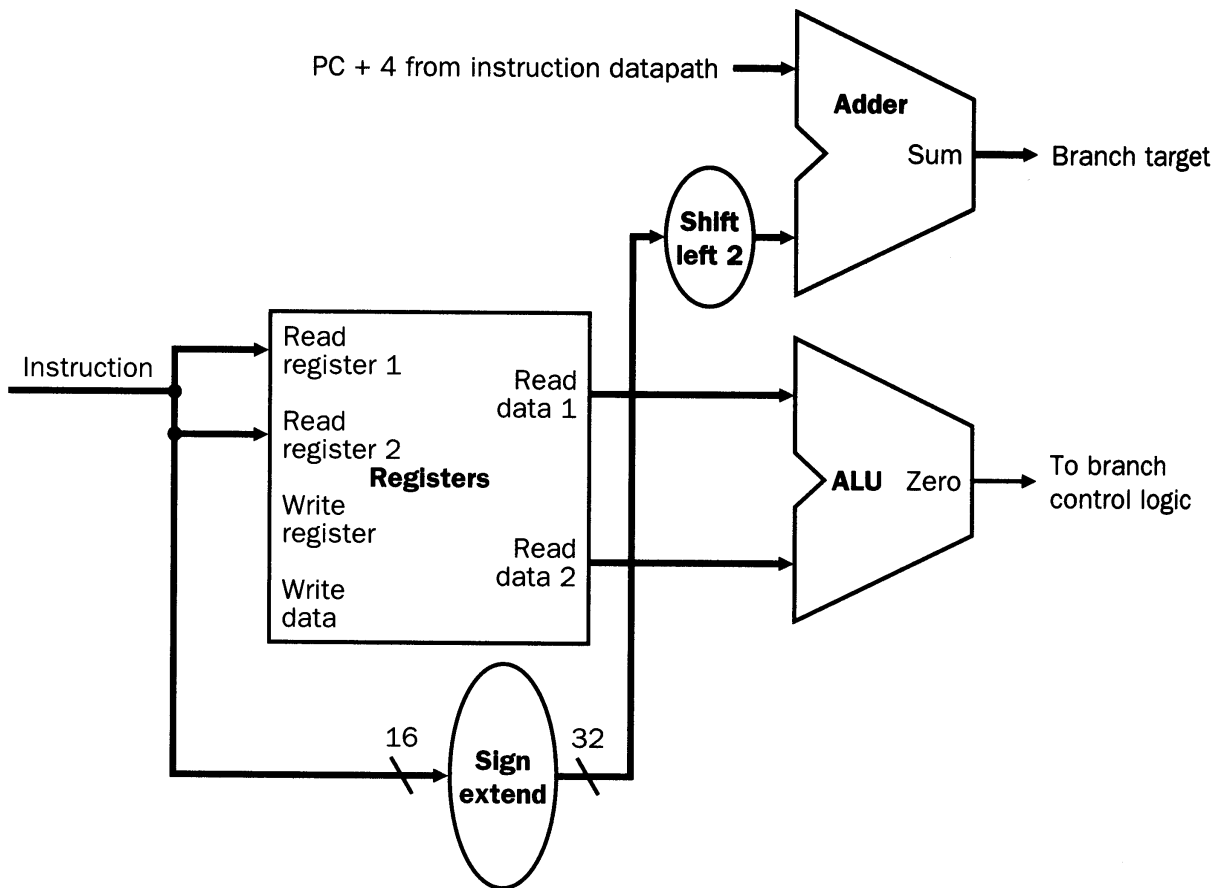
Data path for load or store.



(5.10)

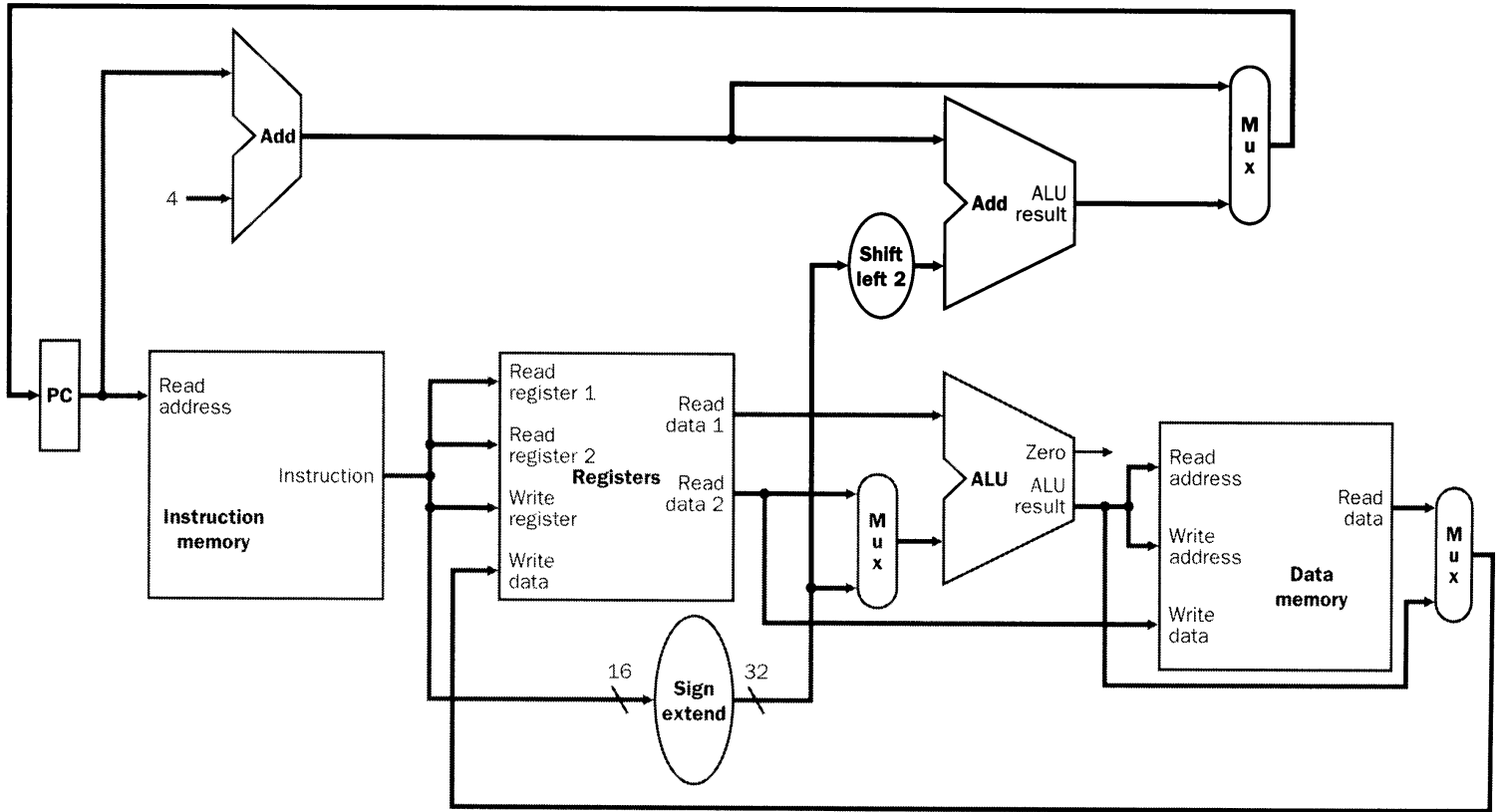


Data path for a branch conditional.



(5.11)

COMPLETE DATA PATH



(5.14)

CONTROL

ALU control

ALU control input	Function
000	And
001	Or
010	Add
110	Subtract
111	Set-less-than

ALU control as a function of Opcode and Function code.
 (This must be viewed as an example, there is nothing general, except the principle).

Two steps to take advantage of the structure:

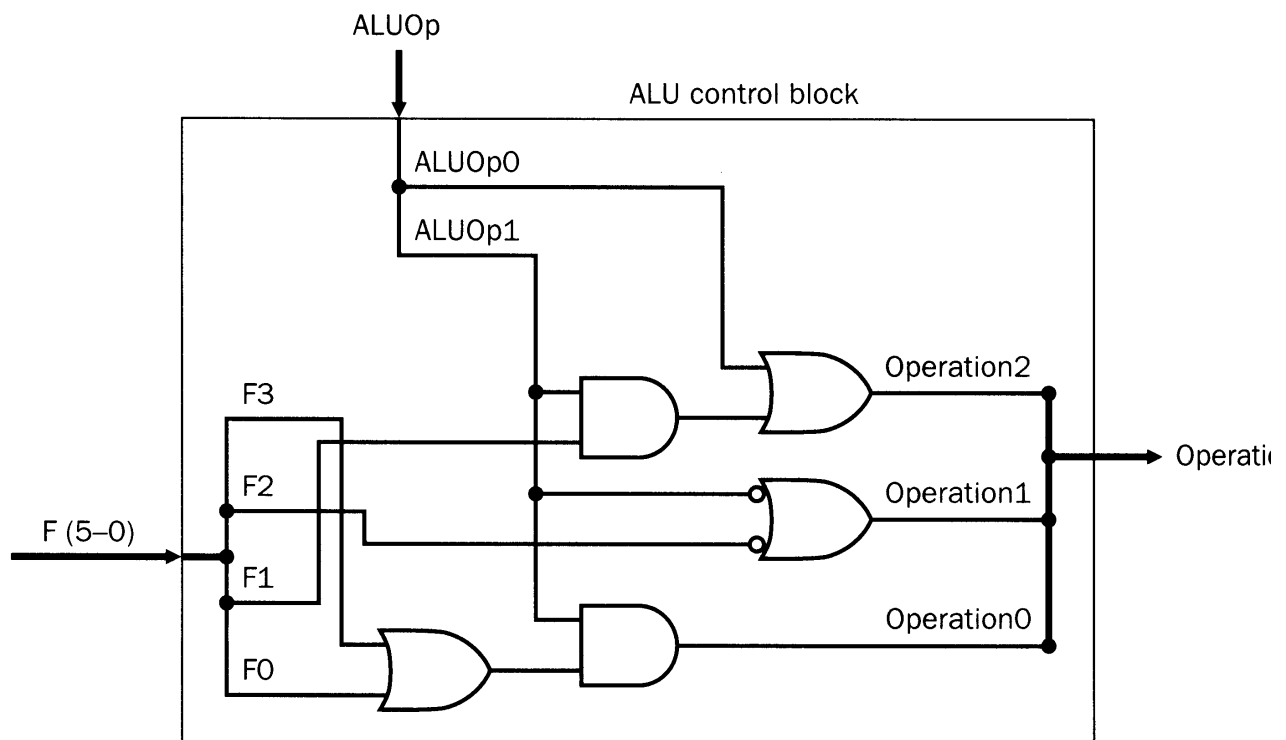
- Create a two bit control signal call ALUop to distinguish whether the ALU action is to be found in the Opcode, in the function field or if it is a branch.

Instr.		ALUop	Instr.	Function	ALU action	ALU
LW	35	00	load word	XXXXXX	add	010
SW	43	00	store word	XXXXXX	add	010
Br. equal	4	01	branch-equal	XXXXXX	subtract	110
R-Type	0	10	add	100000	add	010
R-Type	0	10	subtract	100010	subtract	110
R-Type	0	10	AND	100100	and	000
R-Type	0	10	OR	100101	or	001
R-Type	0	10	set-less-than	101010	set-less-than	111

Create truth tables:

ALUop	Function code	ALU control
00	XXXXXX	010
X1	XXXXXX	110
1X	XX0000	010
1X	XX0010	110
1X	XX0100	000
1X	XX0101	001
1X	XX1010	111

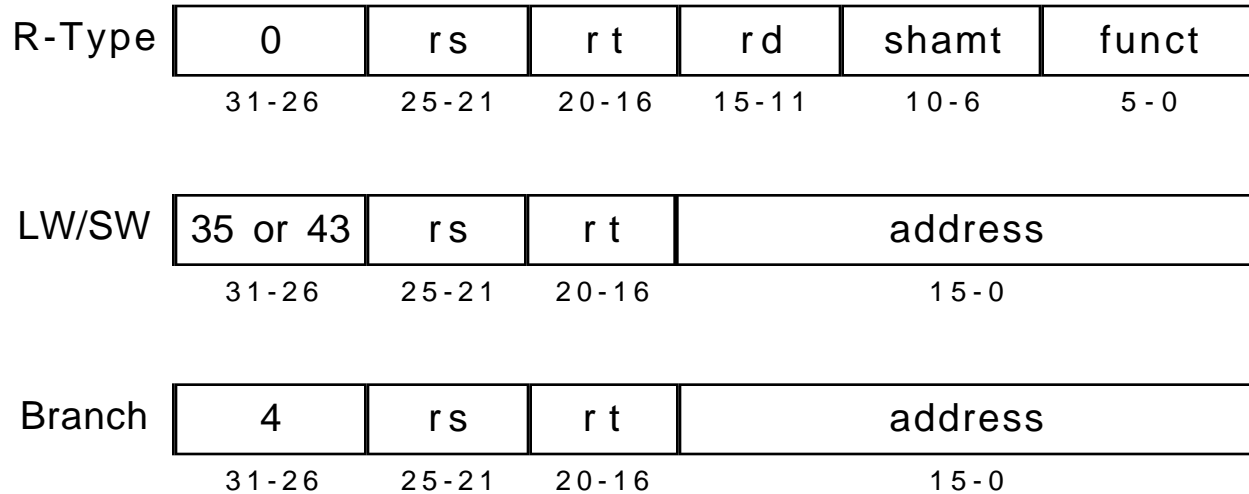
Derive a circuit:



(5.18)

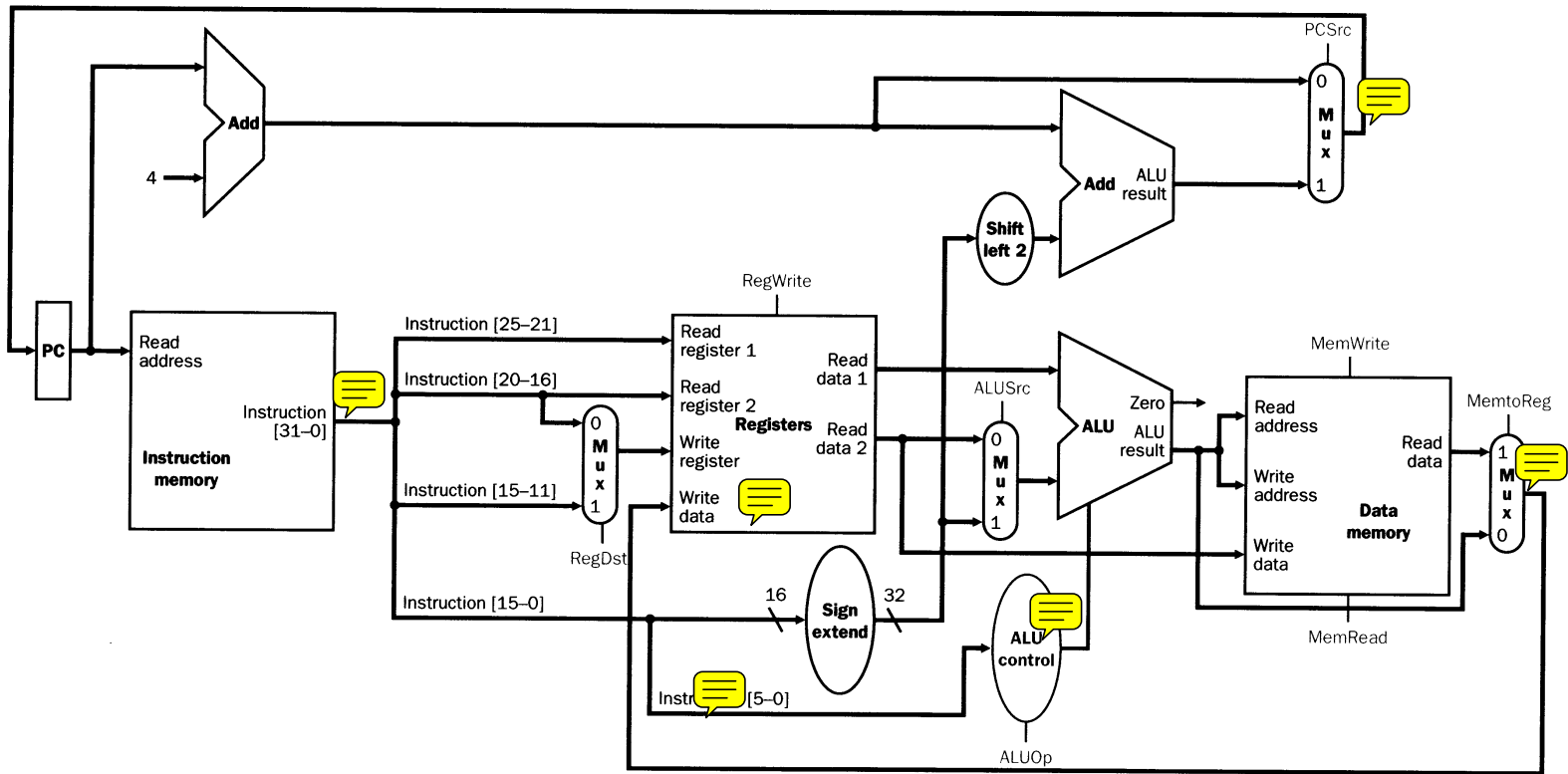
REST OF THE CONTROL

Look at the instruction formats again:



All the fields are more or less at same place:

- Opcode, bits 31-26, call them Opcode.
- The read registers are in rs and rt for all cases except loads.
- The base register is always in 25-21
- The 16-bit offset (found in the address field) is always in 15-0.
- The destination register is in rt for loads but in rd for R-Types. A multiplexer is needed here to select rt or rd.






The data path with multiplexors, and nine control lines. (5.20)

NAMING AND FUNCTIONS OF CONTROL LINES

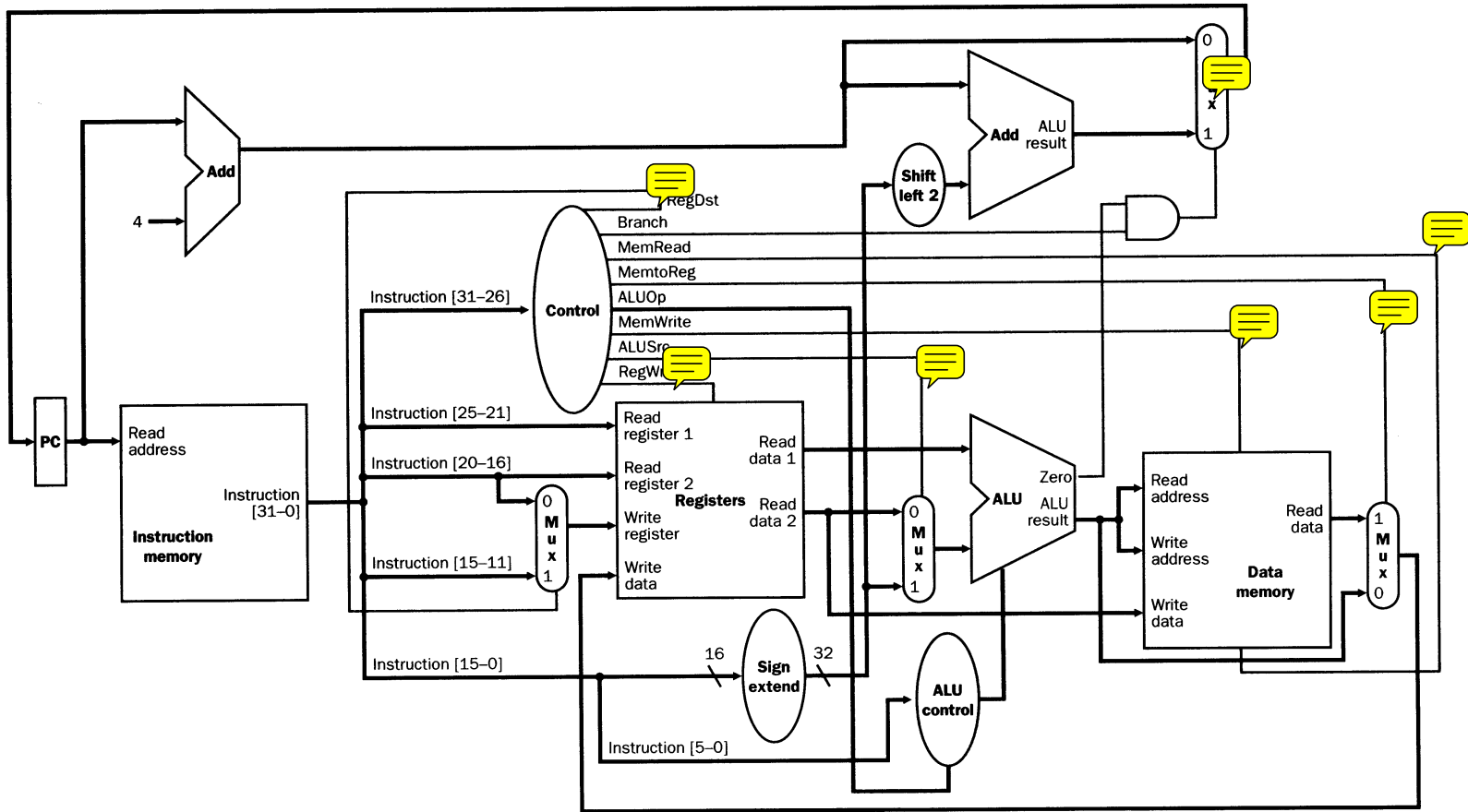
Name	Effect when deasserted	Effect when asserted
MemRead	None	read data at specified address
MemWrite	None	write data at specified address
ALUSrc	ALU input2 comes from read port2 of register file	ALU input2 is sign-extended offset
RegDst	destination register number found in rt	destination register number found in rd
PCSrc	$PC = PC + 4$	$PC = PC + 4 + \text{Offset} * 4$
MemtoReg	write port of register file from ALU output	write port of the register file from the memory

SETTINGS

Instr.	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read 	Mem Write	Branch	ALUop 
R-Type	1	0	0	1	0	0	0	10
lw	0	1	1	1	1	0	0	00
sw 	x	1	x	0	0	1	0	00
beq	x	0	x	0	0	0	1	01

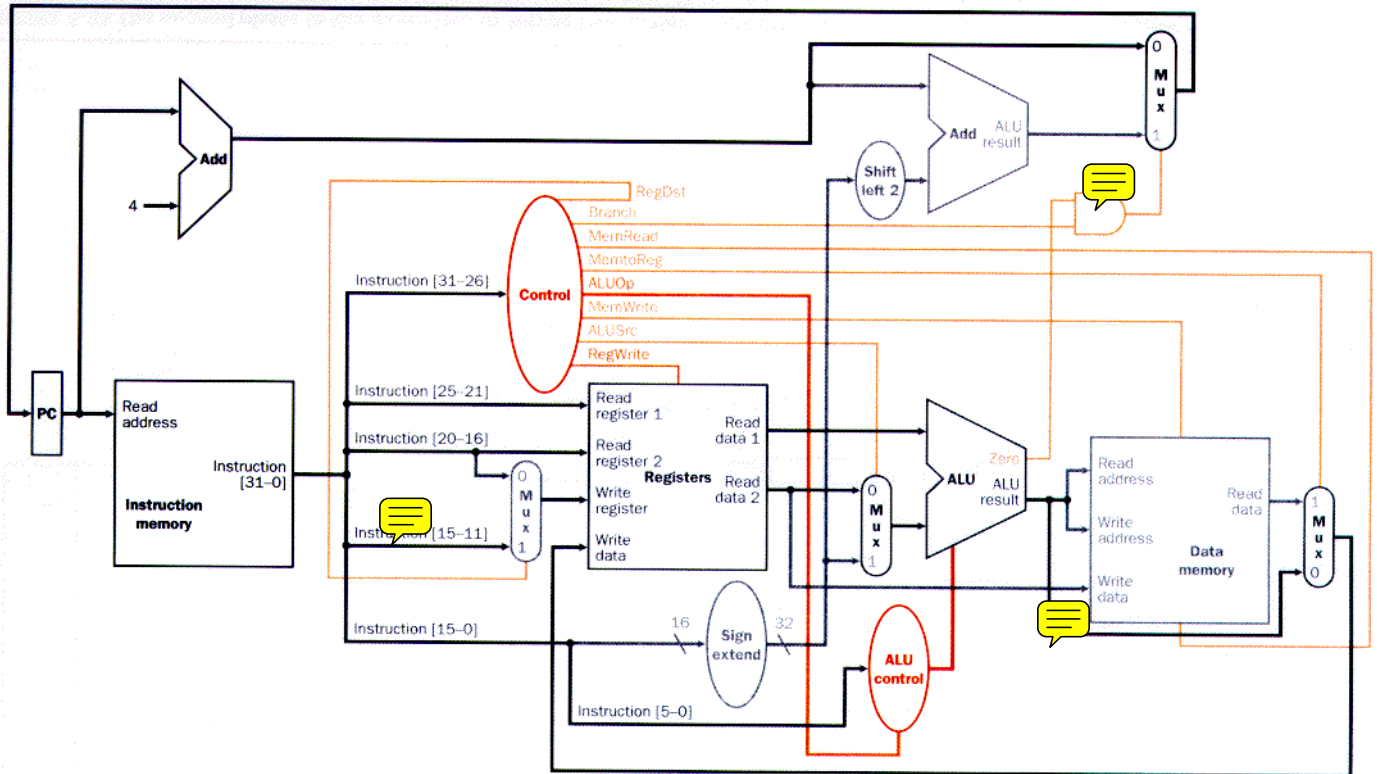
The settings are completely determined by the opcode.

DATA PATH WITH CONTROL LINES



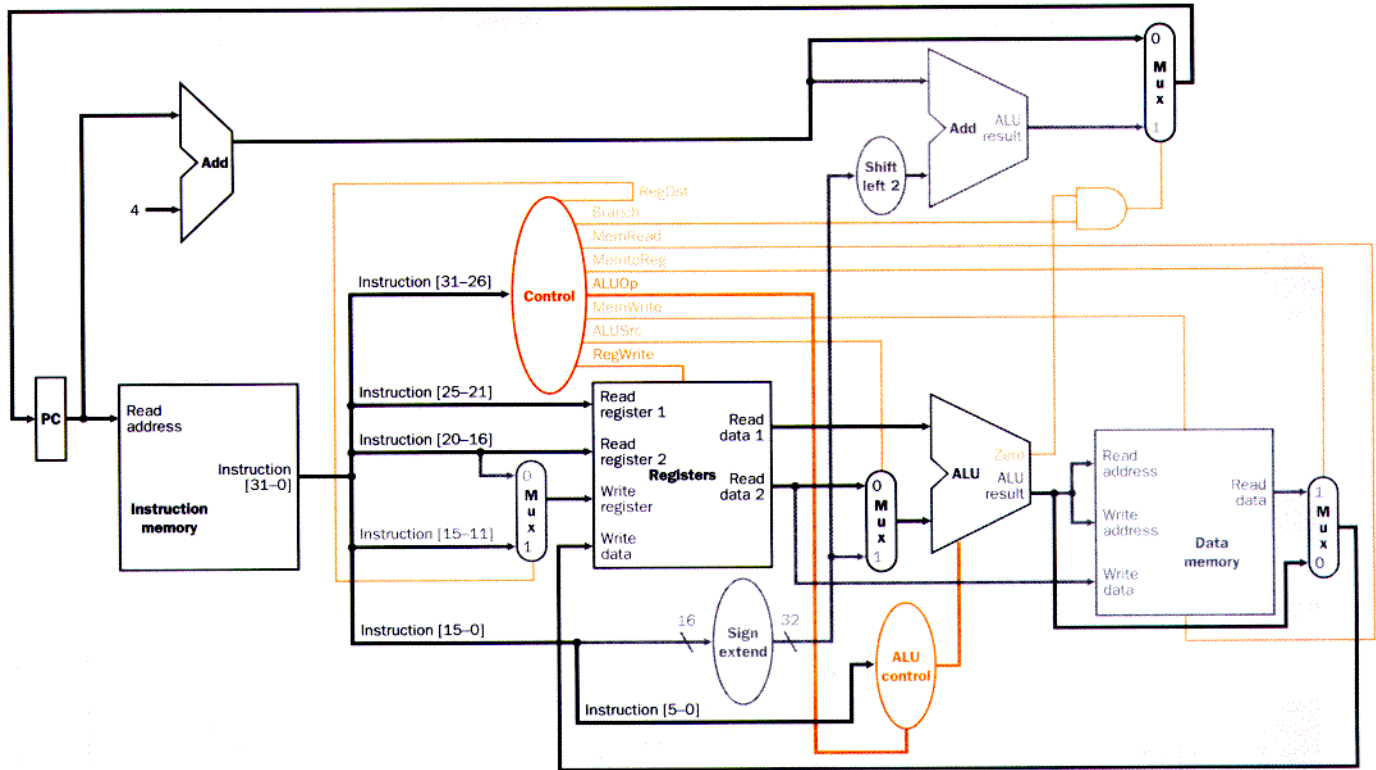
(5.22)

EXAMPLE: AN R-TYPE BROKEN DOWN IN TO 4 STEPS



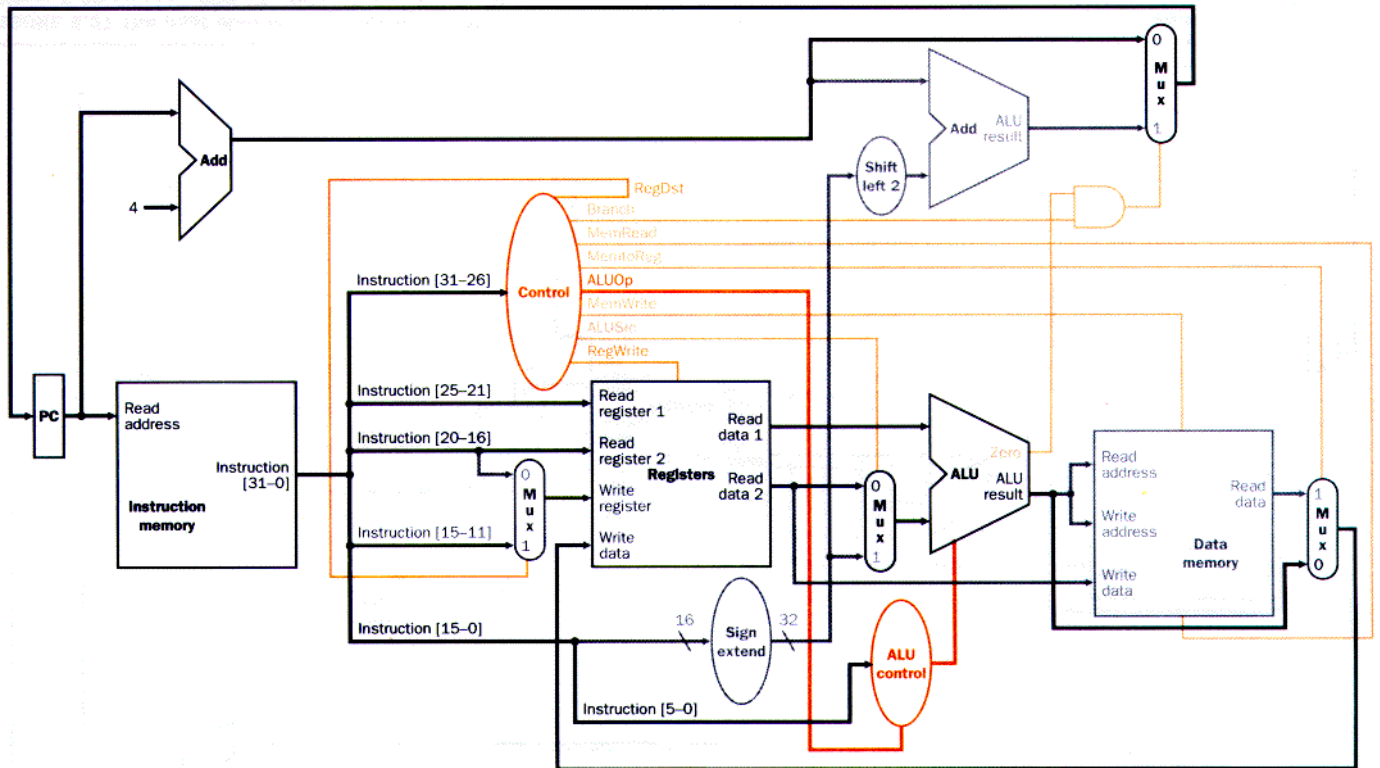
1. FETCH (Units involved shaded) (5.24)

EXAMPLE: AN R-TYPE BROKEN DOWN IN TO 4 STEPS (Cont'd)



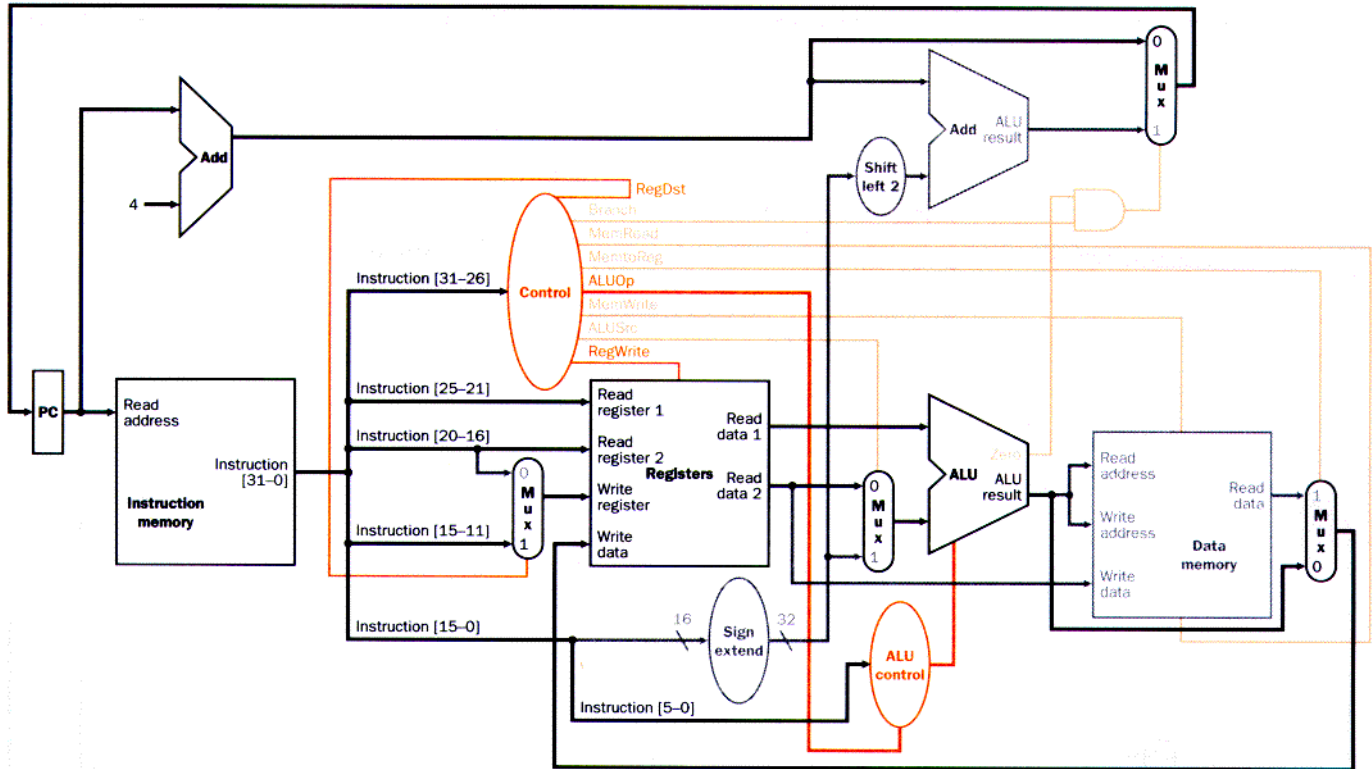
2. REGISTER READ (Units involved shaded) (5.25)

EXAMPLE: AN R-TYPE BROKEN DOWN IN TO 4 STEPS (Cont'd)



3. ALU OPERATION (Units involved shaded) (5.26)

EXAMPLE: AN R-TYPE BROKEN DOWN IN TO 4 STEPS (Cont'd)



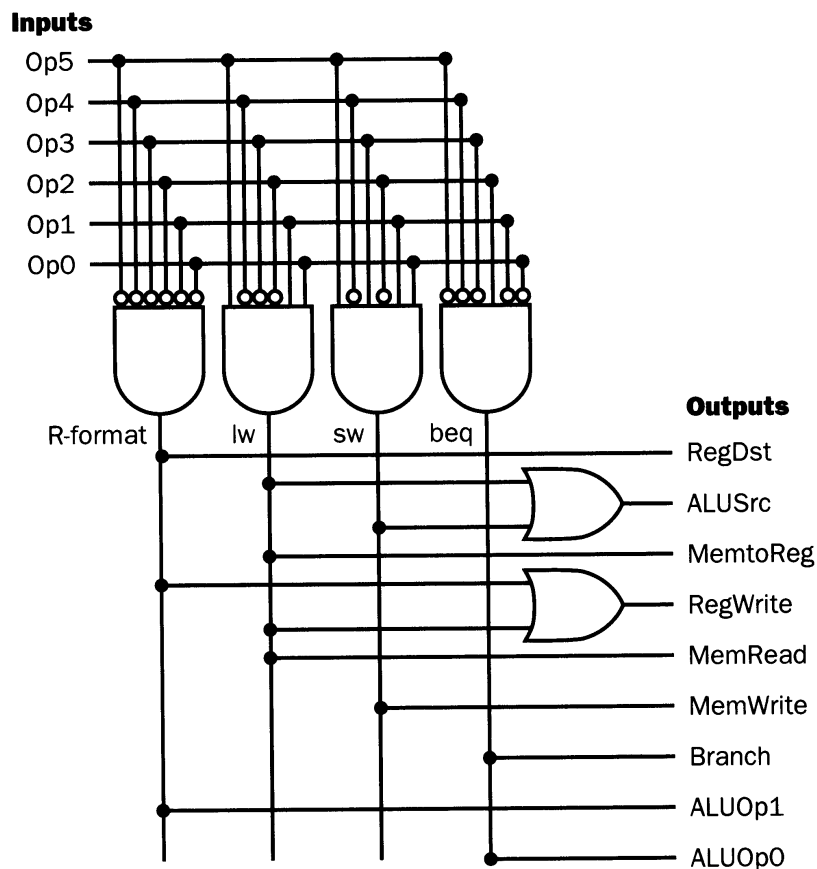
4. RESULT WRITE (Units involved shaded) (5.27)

CONTROL FUNCTION

Returning to the synthesis of the control, we summarize the control function with its truth table.

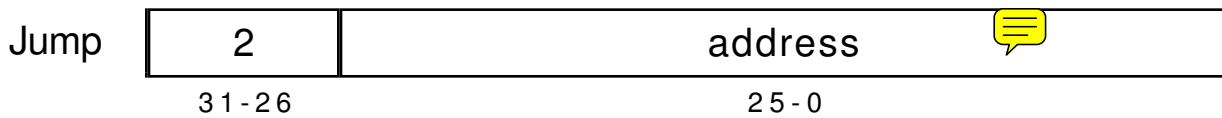
Inputs	Outputs								
Opcode	RegDst	ALUSrc	MemtoReg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
000000	1	0	0	1	0	0	0	1	0
100011	0	1	1	1	1	0	0	0	0
101011	x	1	x	0	0	1	0	0	0
000100	x	0	x	0	0	0	1	0	1

and turn this into a circuit, using a PLA for example:



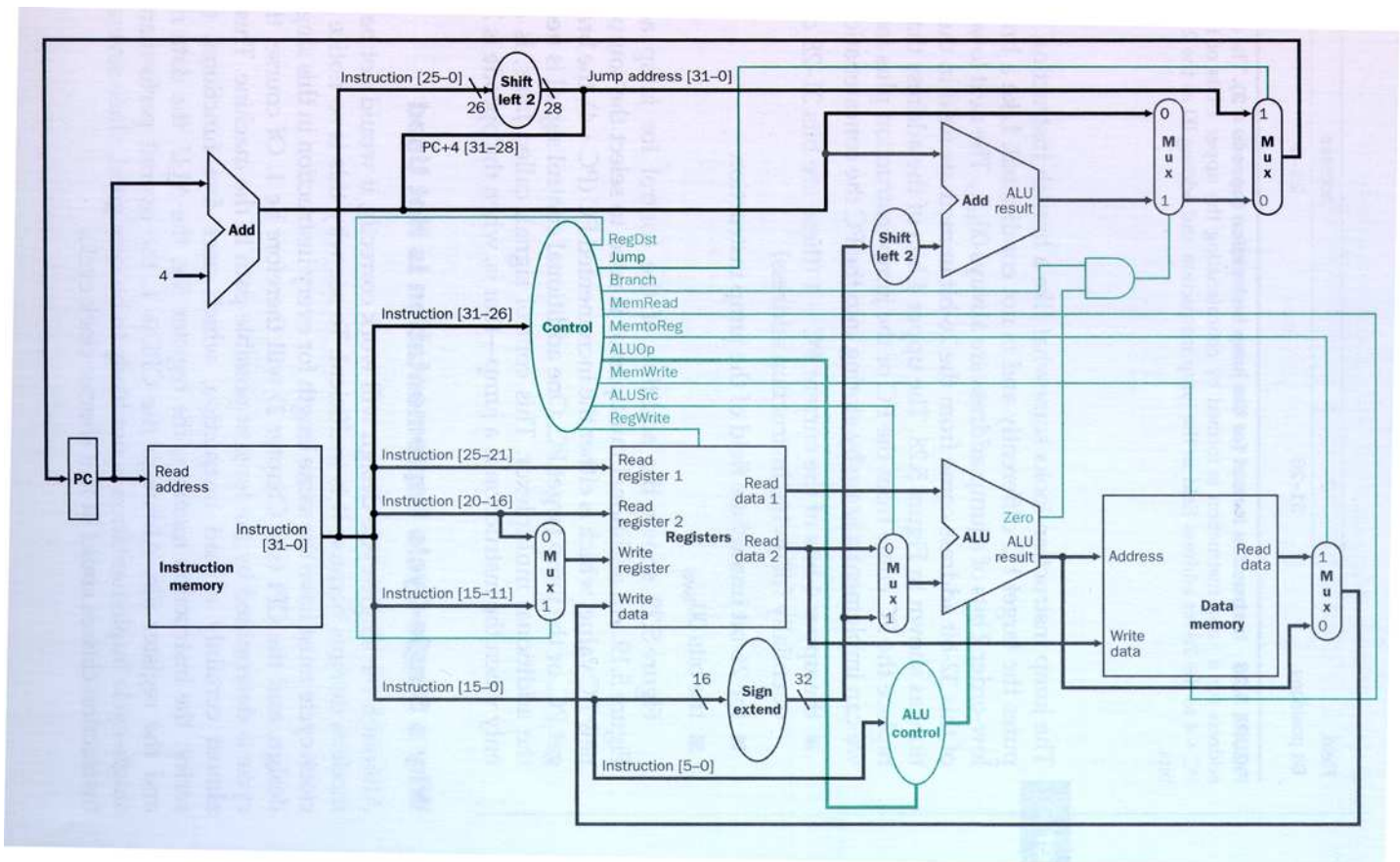
(5.31)

THE CASE OF THE JUMP



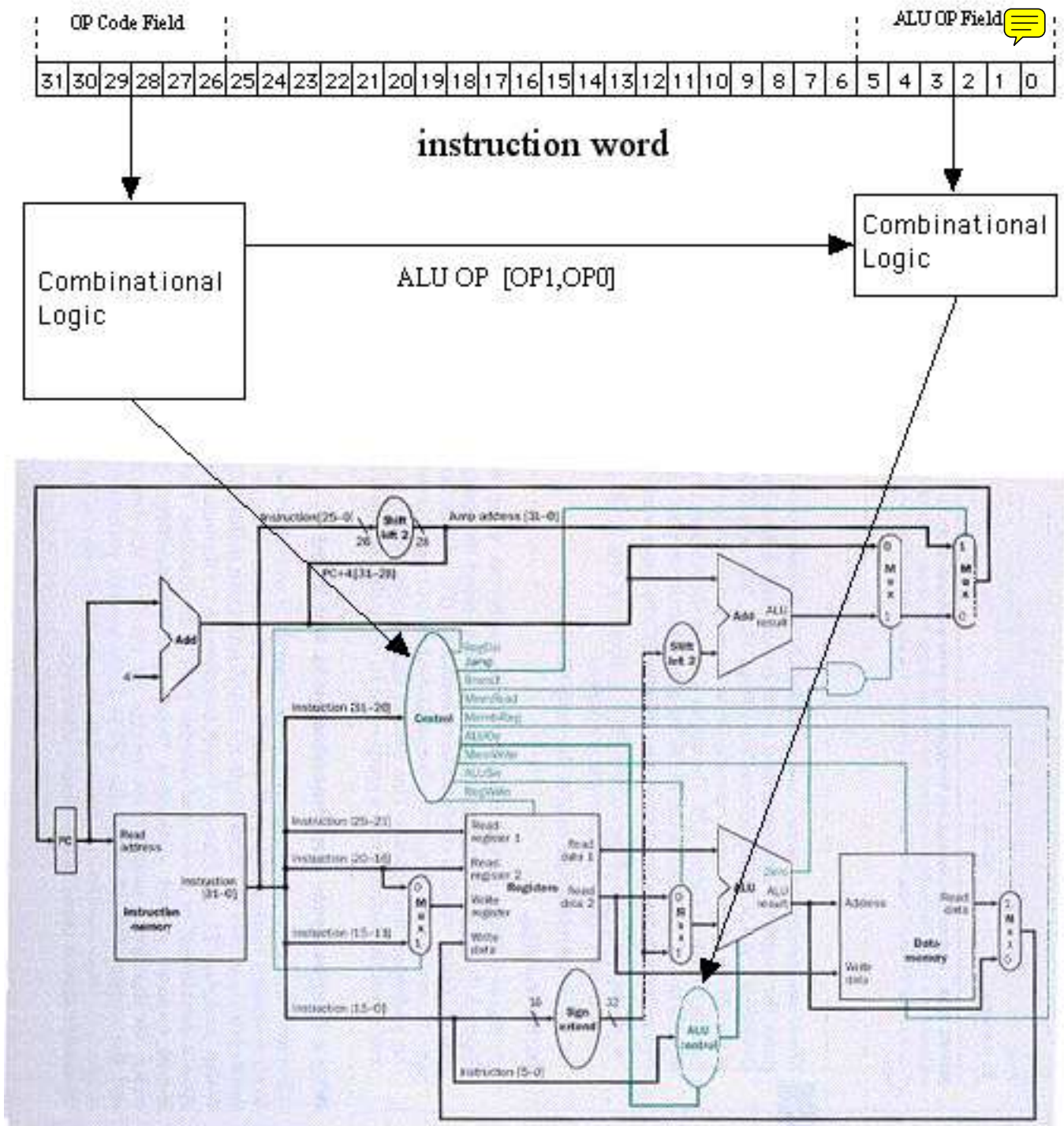
Like the branch, the lower 2 bits of the jumping address are always zero (because we jump always on word boundaries). The next 26 bits come from the instruction, and the upper 4 bits come from the current PC. Thus the jump does not really reach the entire 32-bit address space, but rather a sixteenth of it (which is still a huge space).

One additional multiplexor is needed to select the source of the new PC value and thus requires only one more control line. It is asserted only when the opcode is 2.



Data Path extended to include the jump (5.33)

Control scheme for the Single Cycle Datapath

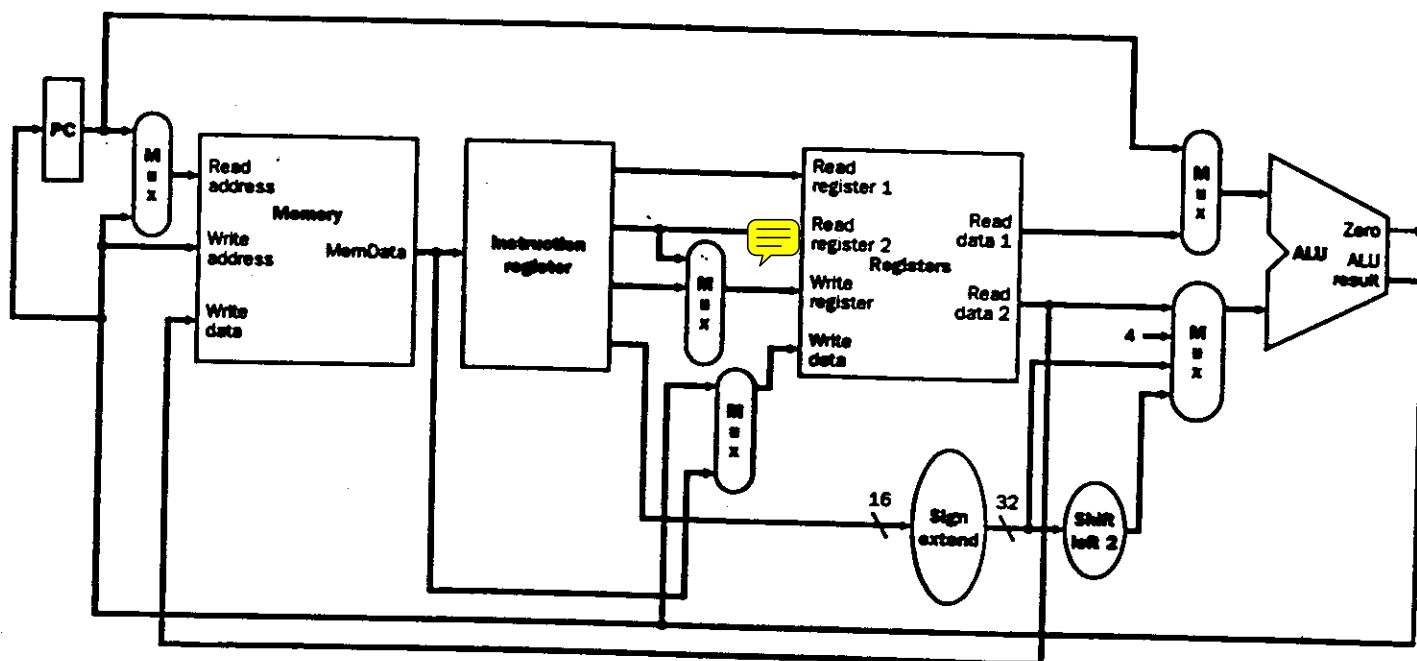


MULTIPLE CLOCK CYCLE IMPLEMENTATION

Single clock cycle implementation is not practical because the cycle time is limited by the time required by the longest of all instructions measured in terms of its critical path (the longest is sw, the shortest is j).

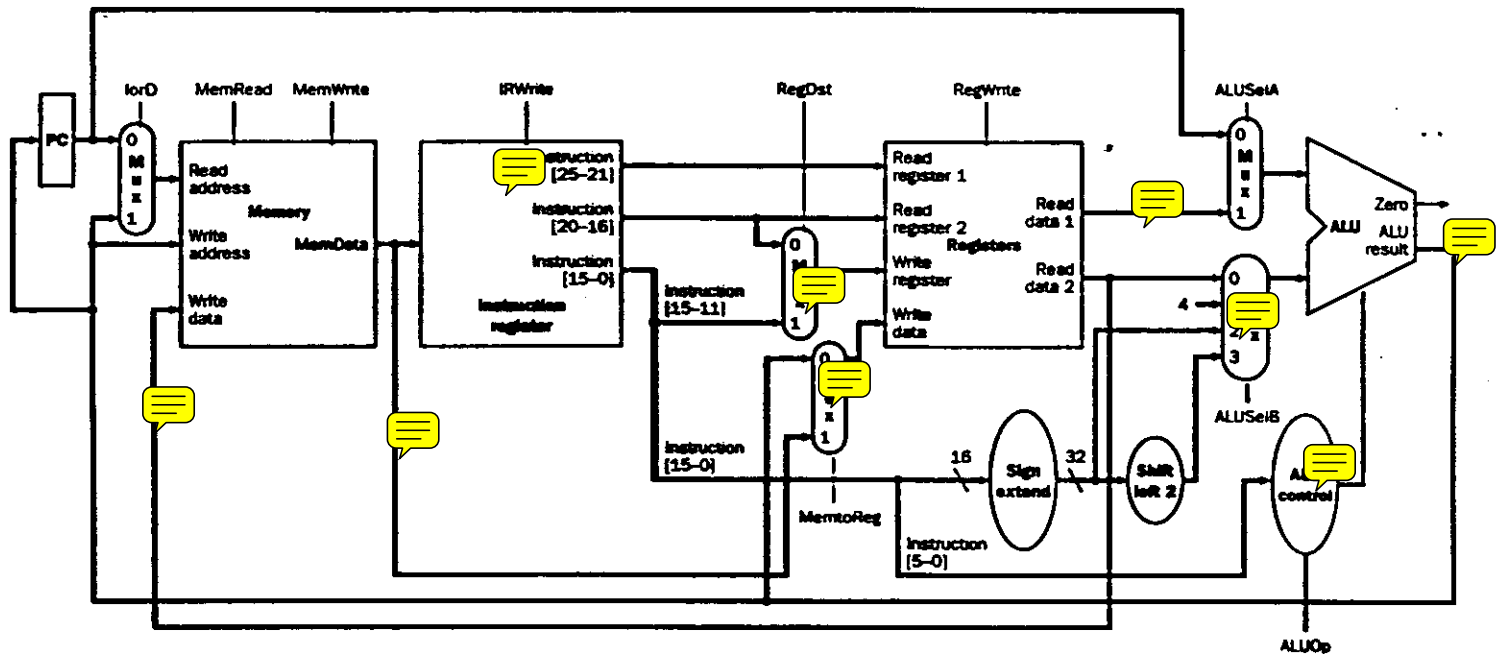
We break down each instruction into *steps* and assign equal clock cycle time to each step. To achieve this we

- use a single memory for instructions and data;
- Save the instruction once read into a register so the memory can be reused later to read or store data;
- Use a single ALU to perform all arithmetic operations so the same is reused at each stage;
- Multiplex the ALU inputs for all needed combinations.



Multistage Data Path (5.35)

CONTROL OVERVIEW



(5.36)

Name	Effect when deasserted	Effect when asserted
MemRead	None	read data at specified address
MemWrite	None	write data at specified address
ALUSelA	ALU input1 comes PC	ALU input1 is register rs
RegDst	destination register is rt	destination register is rd
RegWrite	None	Write register file
MemtoReg	ALU to reg. file write port	memory to reg. file write port
lorD	PC to memory address	ALU to memory address
IRWrite	None	write IR from memory
ALUSelB	00	ALU input2 from register number rt
	01	ALU input2 is constant 4
	10	ALU input2 is sign-extended offset
	11	ALU input2 is sign-extended offset * 4
ALUOp	00	ALU performs an add
	01	ALU performs a subtract
	10	function code determines the ALU operation

INSTRUCTION STEPS

We try to balance each step so each clock cycle is well used. All operation in series must occur in separate clock cycles. Operations in parallel can be done within one clock cycle.

1. Instruction fetch

$$\begin{aligned} \text{IR} &= \text{Memory}[\text{PC}] \\ \text{PC} &= \text{PC} + 4 \end{aligned}$$

Control signals: (IRWrite, lOrD, ALUSelB, ALUSelA, ALUOp)

Note - Need some more control for the PC.

2. Instruction Decode and register fetch

$$\begin{aligned} \text{A} &= \text{Register}[\text{IR}[\text{rs}]] \\ \text{B} &= \text{Register}[\text{IR}[\text{rt}]] \\ \text{Target} &= \text{PC} + (\text{sign-extend}(\text{IR}[\text{address}] \ll 2) \end{aligned}$$

This step is to prepare for later stages. Work is useful for the majority of instructions.

Control signals: (ALUSelB, ALUSelA)

Note - Need a new register: Target

3. Execution, memory address calculation, or branch completion:

memory reference: $ALUoutput = A + \text{sign-extend}(IR[\text{address}])$

Control signals: (ALUSelA, ALUSelB, ALUOp)

Arithmetic-logical: $ALUoutput = A \text{ op } B$

Control signals: (ALUSelA, ALUSelB, ALUOp)

Branch: if (A == B) then PC = Target

Control signals: (ALUSelA, ALUSelB, ALUOp)

Note - Need more control for the PC

4. Memory Access or R-Type completion

Load: $\text{memory-data} = \text{Memory}[ALUoutput]$

Control signals: (MemRead, IorD)

Store: $\text{Memory}[ALUoutput] = B$

Control signals: (MemWrite, IorD)

R-Type: $\text{Register}[IR[\text{rd}]] = ALUoutput$

Control signals: (RegDst, RegWrite, MemtoReg)

Note - Keep values of ALUSelA, ALUSelB, ALUOp stable

5. Write-back

$\text{Reg}[\text{IR}[\text{rt}]] = \text{memory-data}$

Control signals: (RegWrite, RegDst)

Note - this one is for loads only

Note - Keep values of ALUSelA, ALUSelB, ALUOp stable

SUMMARY

Steps	Action for R-Types	Action for load/stores	Action for Branches
Instruction Fetch	$\text{IR} = \text{Memory}[\text{PC}]$		
Instruction Decode Register Fetch	$A = \text{Register}[\text{IR}[\text{rs}]]$ $B = \text{Register}[\text{IR}[\text{rt}]]$ $\text{Target} = \text{PC} + (\text{sign-extend}(\text{IR}[\text{address}]) \ll 2)$		
Execution, address calculation or branch completion	$\text{ALUoutput} = A \text{ op } B$	$\text{ALUoutput} = A + \text{sign-extend}(\text{IR}[\text{address}])$	if (A == B) then PC = Target
Memory access or R-Type completion	$\text{Register}[\text{IR}[\text{rd}]] = \text{ALUoutput}$	$\text{memory-data} = \text{Memory}[\text{ALUoutput}]$	
Write-back		$\text{Register}[\text{IR}[\text{rt}]] = \text{memory data}$	

To complete the data path, control is needed for the PC.
Including the jump instruction, the PC can take its source from:



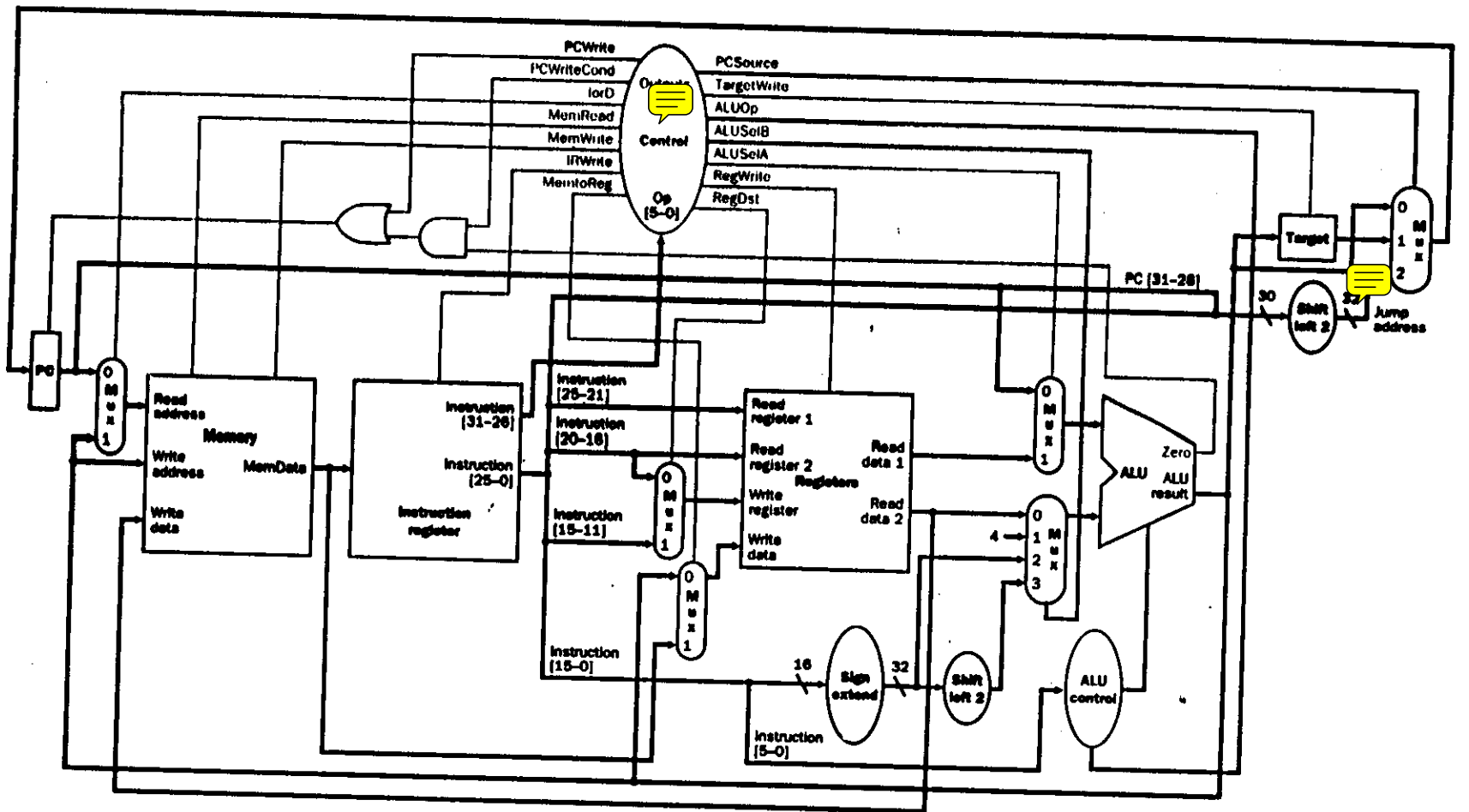
- ALUoutput for incrementation.
- Target register for a branch.
- The lower 26 bits of the IR shifted by two and combined with the upper 4 bits of the PC.

This gives rise to a two bit control signal PCSource and a 3 input multiplexor, and also a control signal TargetWrite.

To handle the branches we make the write to PC conditional to the zero output of the ALU if the signal PCWriteCond is asserted.

Name	Effect when deasserted	Effect when asserted
PCWrite	None	PC written
PCWriteCond	None	PC written if the zero output of the ALU
TargetWrite	None	Target is written from ALU output
		Effect of values
PCSource	00	ALU output is sent to PC for writing
	01	Target is sent to PC for writing
	10	The jump address is sent to PC for writing

(5.39)

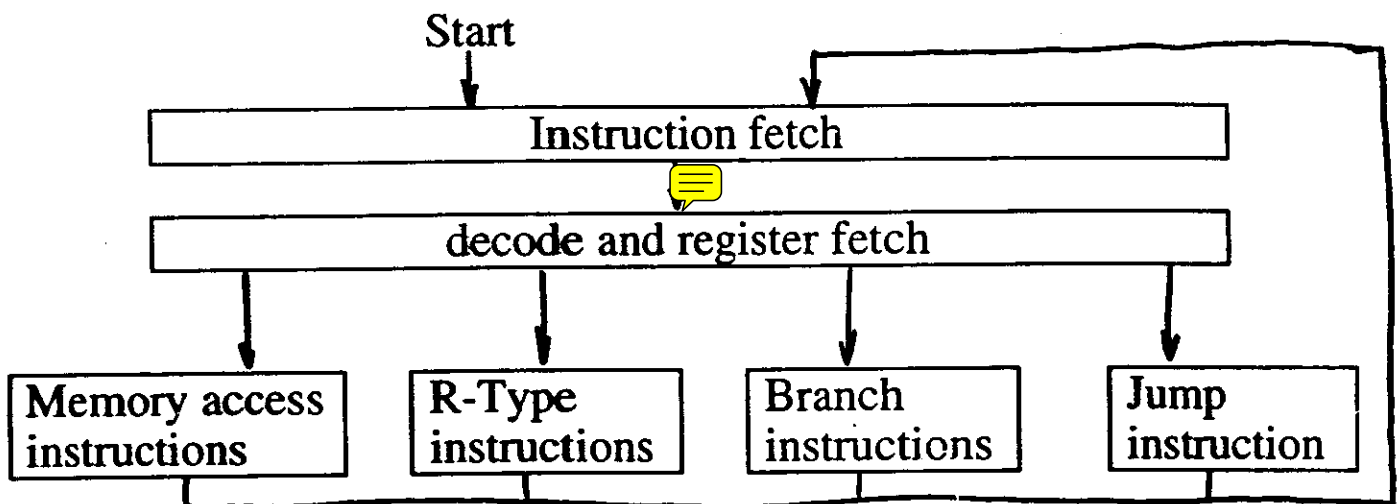


DEFINING THE CONTROL

The control must specify the signals in each step and the next step in sequence. It can be done two ways:

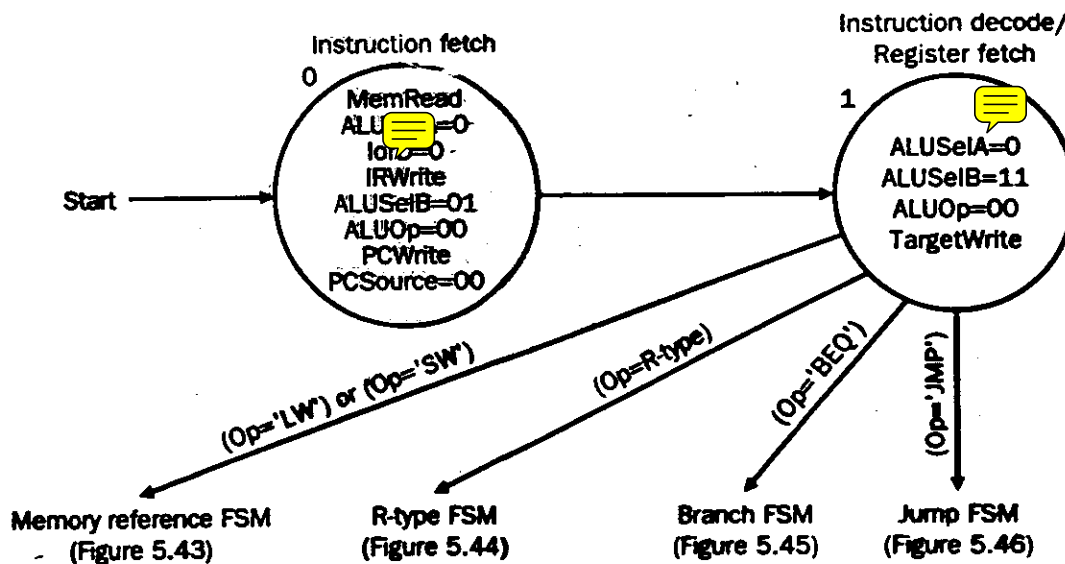
- • *Finite state machine* implemented with logic circuits;
- • *Microprogramming* (which also is a finite state machine).

A high level view is the finite state machine is:

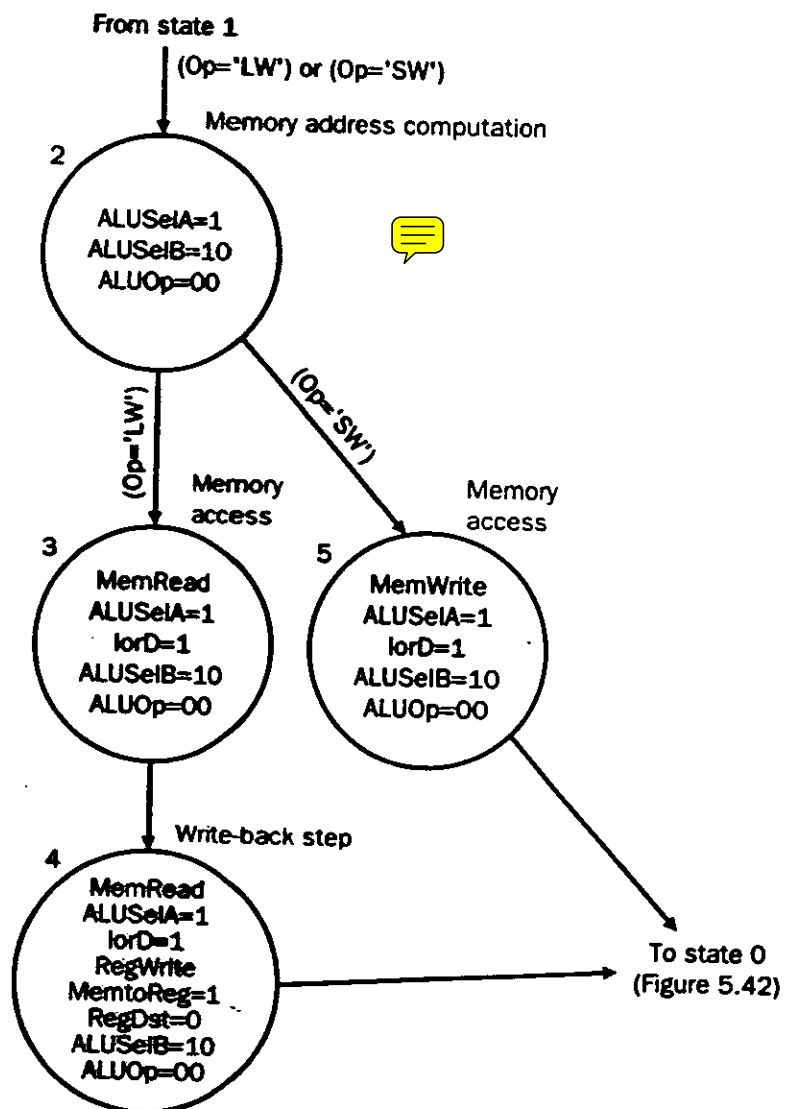


By default, control signals not specified are left to 0.

INSTRUCTION FETCH AND DECODE

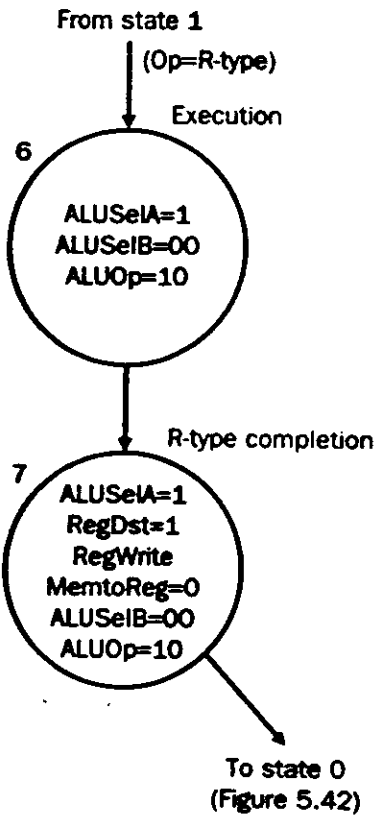


MEMORY REFERENCE

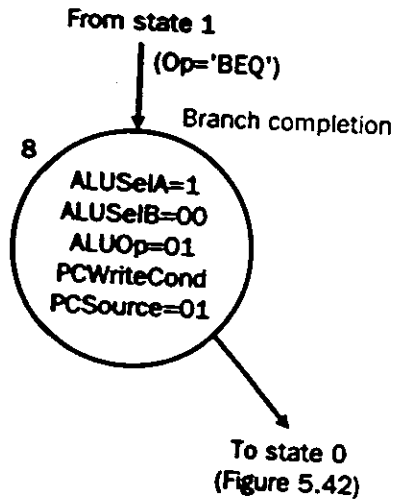


(5.43)

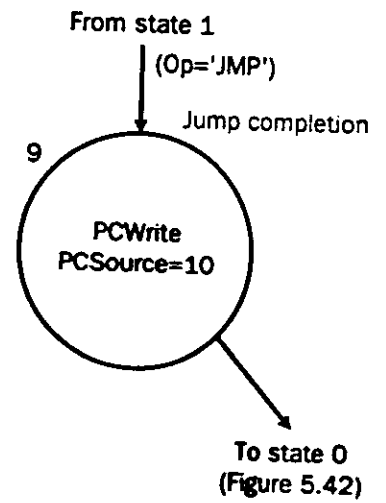
R-TYPE, BRANCHES AND JUMP



5.44



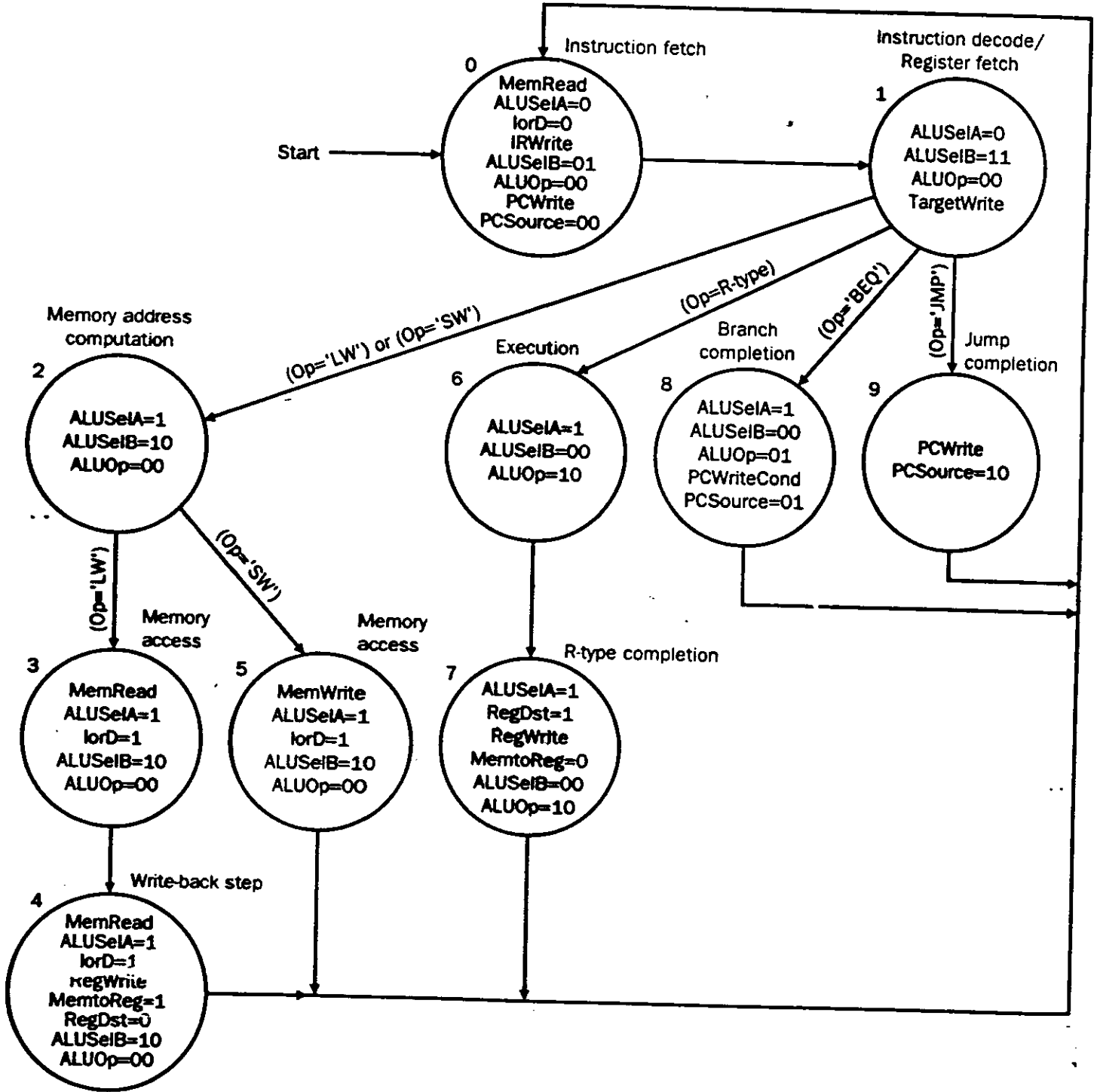
5.45



5.46

START

COMPLETE MACHINE



(5.47)

MICROPROGRAMMING

The complete control of a real-life machine may have hundreds of states. Microprogramming allows the implementation in an orderly fashion.

The idea is to define the control as a sequence of μ instructions assembled into a μ program running in a μ machine. Each μ instruction defines the control lines assertion values. They are specified symbolically and turned into binary values automatically by a μ assembler.

μ instructions are broken down into fields which specify the control lines by groups, plus one field which determines the sequencing according to three methods:

1. Increment the address of the current μ instruction to get the next, this is indicated by the symbol Seq.
2. Branch to a labeled μ instruction, for example Label.
3. Select the next μ instruction from an external input. The name of a *dispatch table* (ROM or PLA) is indicated.

We have 8 fields:

Field name	Function
ALU control	ALU operation during this clock
SRC1	source of the first ALU operand
SRC2	source of the second ALU operand
ALU destination	which register receives the ALU output
Memory	Read/Write and what address
Memory Register	register read or written to or from memory
PCWrite control	how is the PC written
Sequencing	next, branch or dispatch

FIELD VALUES

Field name	Values	Function specified
ALU Control	Add	Cause the ALU to add
	Func code	Use instruction's function code to determine ALU control
	Subt	Cause the ALU to subtract
SRC1	PC	Use the PC as the first ALU input
	rs	Register rs is the first ALU input
SRC2	4	Use constant 4 as second ALU input
	Extend	Use the output of the sign extension unit as second ALU input
	Extshft	Use the output of the shift by 2 unit as the second ALU input
	rt	Register rt is the second ALU input
ALU destination	Target	ALU output is written into the register Target
	rd	ALU output is written into register rd
Memory	Read PC	read memory using the PC as address
	Read ALU	Read memory using the ALU output as address
	Write ALU	Write memory using the ALU output as address
Memory register	IR	Data read from memory is written into the instruction register
	Write rt	Data read from memory is written into the register rt
	Read rt	Data written into memory comes from register rt
PCWrite control	ALU	Write the output of the ALU into the PC
	Target-cond	If the Zero output of the ALU is active, write PC with content of the register Target
	jump address	Write the PC with the jump address from the instruction
Sequencing	Seq	Choose the next microinstruction
	Fetch	Go to the first microinstruction to begin a new instruction
	Dispatch i	Dispatch using the ROM specified by i (1 or 2)

WRITING A MICROPROGRAM

Label μ instructions we need to jump to, i.e. the first of each instruction.

A blank means two possible things:

- If the control lines control a state element such as a register, a blank means that it remains unchanged;
- If the control lines control a combinatorial circuit, it means we don't care.

All instructions start with a fetch:

Label	ALU control	SRC1	SRC2	ALU destination	Memory	Memory register	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	IR	ALU	Seq
	Add	PC	Extsh	Target				Dispatch1

To understand its effect, look at the groups of fields. For the first:

Fields	Effect
ALU control, SRC1, SRC2	compute PC + 4
Memory and memory register	Fetch instruction into IR
PCWrite control	Causes the output of the ALU to be written into the PC
Sequencing	go to the next

For the second:

Fields	Effect
ALU control, SRC1, SRC2, ALU destination	Store PC + sign-extension(IR[offset]) << 2 into target
Sequencing	Use dispatch table 1 to select the next microinstruction

After the dispatch, the μ program jumps to the place corresponding to the instruction.

Loads and stores

Label	ALU control	SRC1	SRC2	ALU destination	Memory	Memory register	PCWrite control	Sequencing
LWSW1	Add	rs	Extend					Dispatch2
LW2	Add	rs	Extend		Read ALU			Seq
	Add	rs	Extend		Read ALU	Write rt		Fetch
SW2	Add	rs	Extend		Write ALU	Read rt		Fetch

Look at the group of fields in the first μ instruction

Fields	Effect
ALU control, SRC1, SRC2	Compute the memory address; Register(rs) + sign-extend(IR[offset])
Sequencing	Use the second dispatch table to jump to either LW2 or SW2

The μ instruction labeled LW2:

Fields	Effect
ALU control, SRC1, SRC2	The output of the memory is still the memory address
Memory	Read memory using the ALU output as the address
Sequencing	Go to the next

The one after:

Fields	Effect
ALU control, SRC1, SRC2	The output of the memory is still the memory address
Memory and memory register	Read memory using the ALU output as the address and write the result into the register selected by rt
Sequencing	Go to the microinstruction Fetch

Note that since the field of the two μ instructions do not conflict. They could be combined into one.

Label	ALU control	SRC1	SRC2	ALU destination	Memory	Memory register	PCWrite control	Sequencing
LW2	Add	rs	Extend		Read ALU	Write rt		Fetch

But the cycle time might have to be increase since both the memory access and the register would have to occur in the same μ instruction.

The store μ instruction labeled SW2 operates similarly:

Fields	Effect
ALU control, SRC1, SRC2	The output of the memory is still the memory address
Memory and memory register	Write memory using the ALU output as the address the register selected by rt has the value to write
Sequencing	Go to the microinstruction Fetch

The μ program sequence for R-Type instruction has two μ instructions:

Label	ALU control	SRC1	SRC2	ALU destination	Memory	Memory register	PCWrite control	Sequencing
Rformat1	Func code	rs	rt					Seq
	Func code	rs	rt	rt				Fetch

The two μ instructions perform these operations:

Fields	Effect
ALU control, SRC1, SRC2, ALU destination	The first μ instruction causes the ALU to operate on registers rs and rt according to the func field. The second does the same but writes the result.
Sequencing	go to the next and then go to Fetch.

For branches, one μ instruction:

Label	ALU control	SRC1	SRC2	ALU destination	Memory	Memory register	PCWrite control	Sequencing
BEQ1	Subt	rs	rt				Target-cond	Fetch

The asserted fields are:

Fields	Effect
ALU control, SRC1, SRC2	The ALU rt from rs and generates the Zero output
PCWrite Control	Causes the PC to be written using the value in Target, if the Zero output is true.
Sequencing	go to Fetch.

The jump μ code consists also of one μ instruction:

Label	ALU control	SRC1	SRC2	ALU destination	Memory	Memory register	PCWrite control	Sequencing
JUMP1							jump address	Fetch

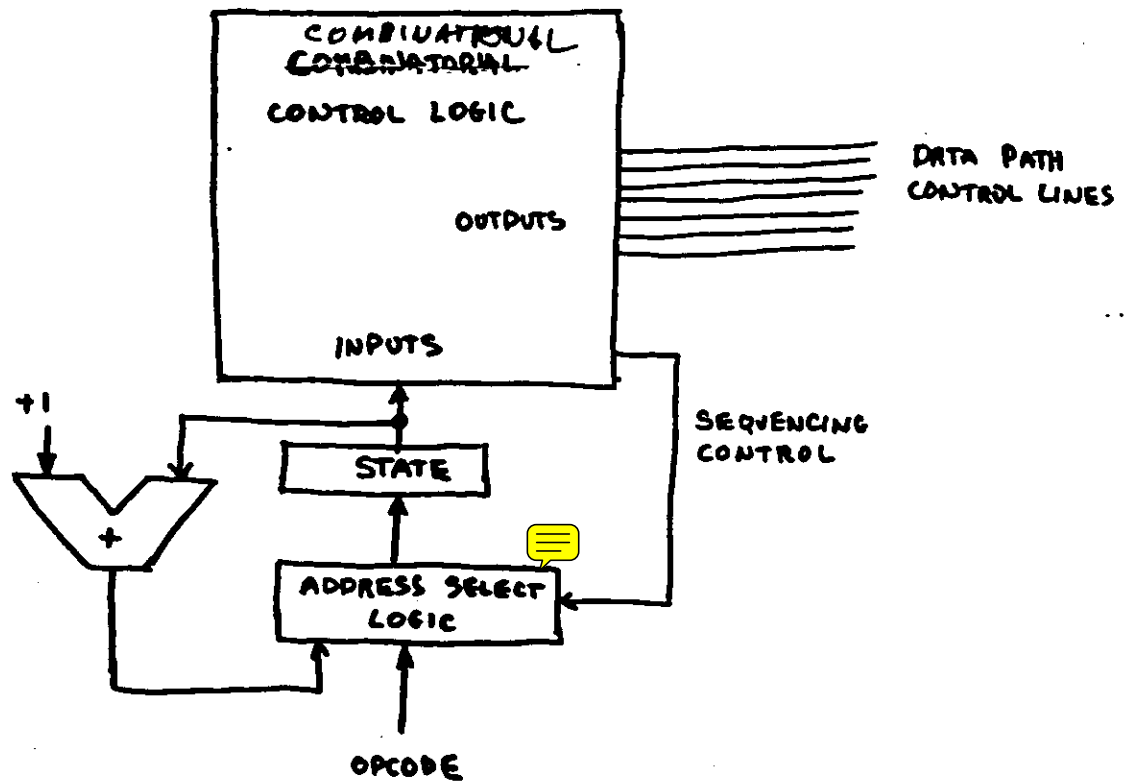
The asserted fields are:

Fields	Effect
PCWrite Control	Causes the PC to be written using the jump field.
Sequencing	go to Fetch.

COMPLETE μ PROGRAM

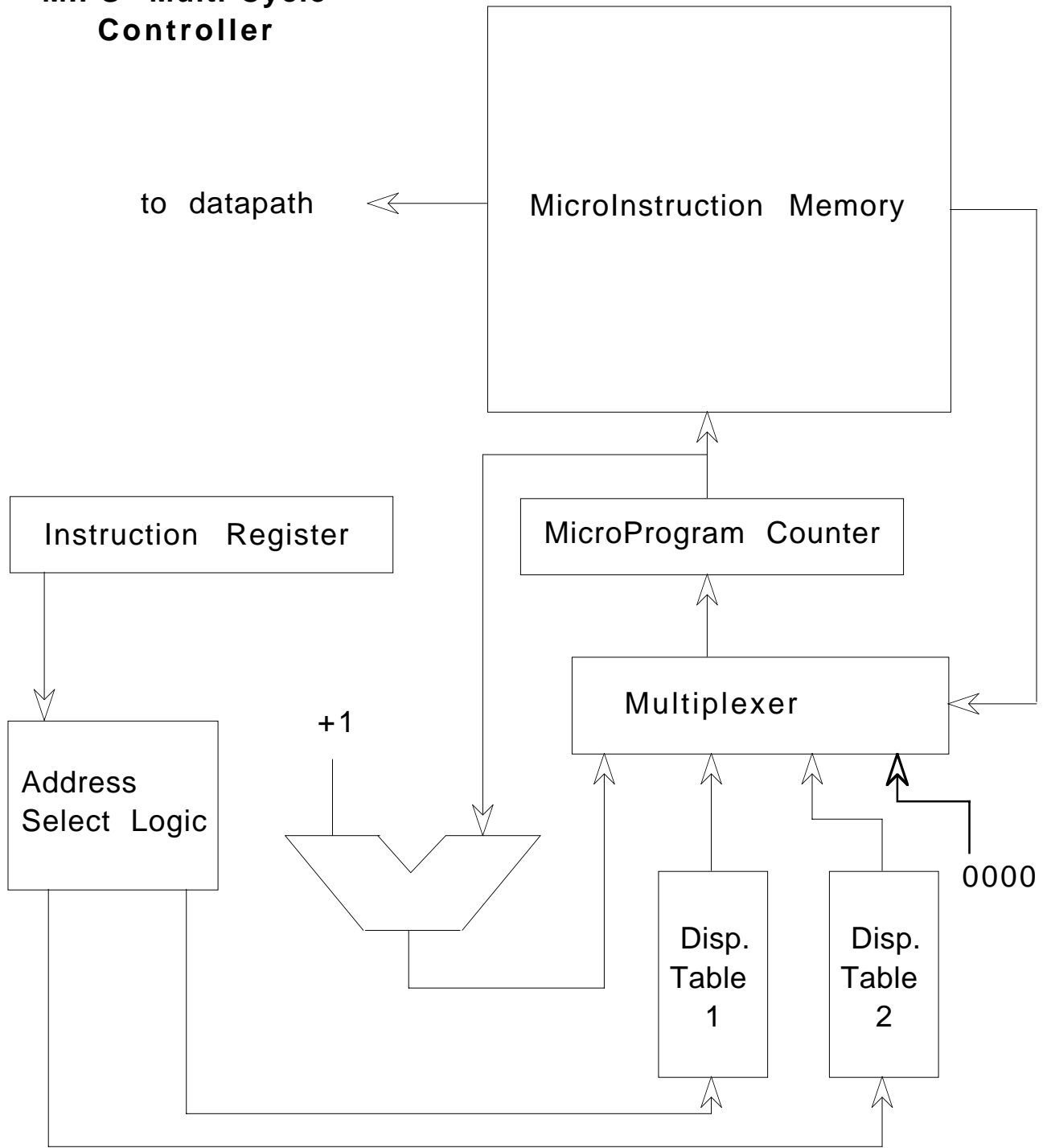
Label	ALU control	SRC1	SRC2	ALU destination	Memory	Memory register	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	IR	ALU	Seq
	Add	PC	Extsh	Target				Dispatch1
LWSW1	Add	rs	Extend					Dispatch2
LW2	Add	rs	Extend		Read ALU			Seq
	Add	rs	Extend		Read ALU	Write rt		Fetch
SW2	Add	rs	Extend		Write ALU	Read rt		Fetch
Rformat1	Func code	rs	rt					Seq
	Func code	rs	rt	rt				Fetch
JUMP1							jump address	Fetch

IMPLEMENTATION



(5.52)

MIPS Multi-Cycle Controller



EXCEPTIONS

An *exception* is an unexpected event from within the processor, e.g. arithmetic overflow. An *interrupt* is similar except it comes from an event outside the processor.

Some examples

I/O device request	External	Interrupt
Invoke the operating system from a user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunction	Either	Exception or Interrupt

To handle an exception, the basic actions are:

1. Save the address of the current (or offending) instruction in a register called EPC (Exception Program Counter);
2. Transfer control to some specified address in the OS (exception handler).

After handling the exception, execution may or may not restart. (example of upward compatibility, I/O).

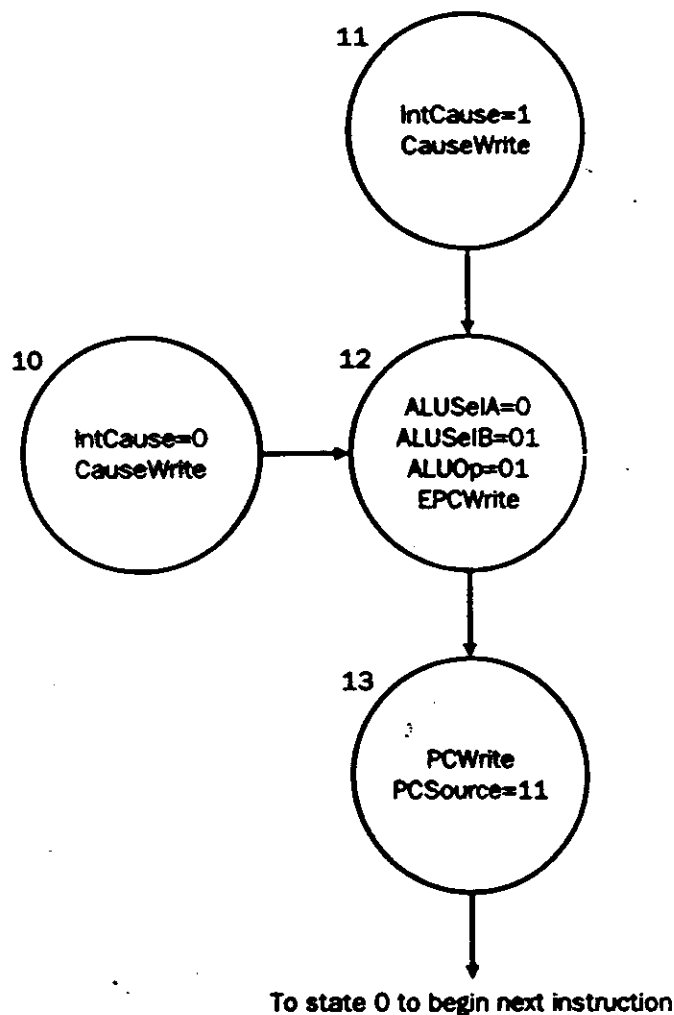
Two main methods to determined the cause of an exception:

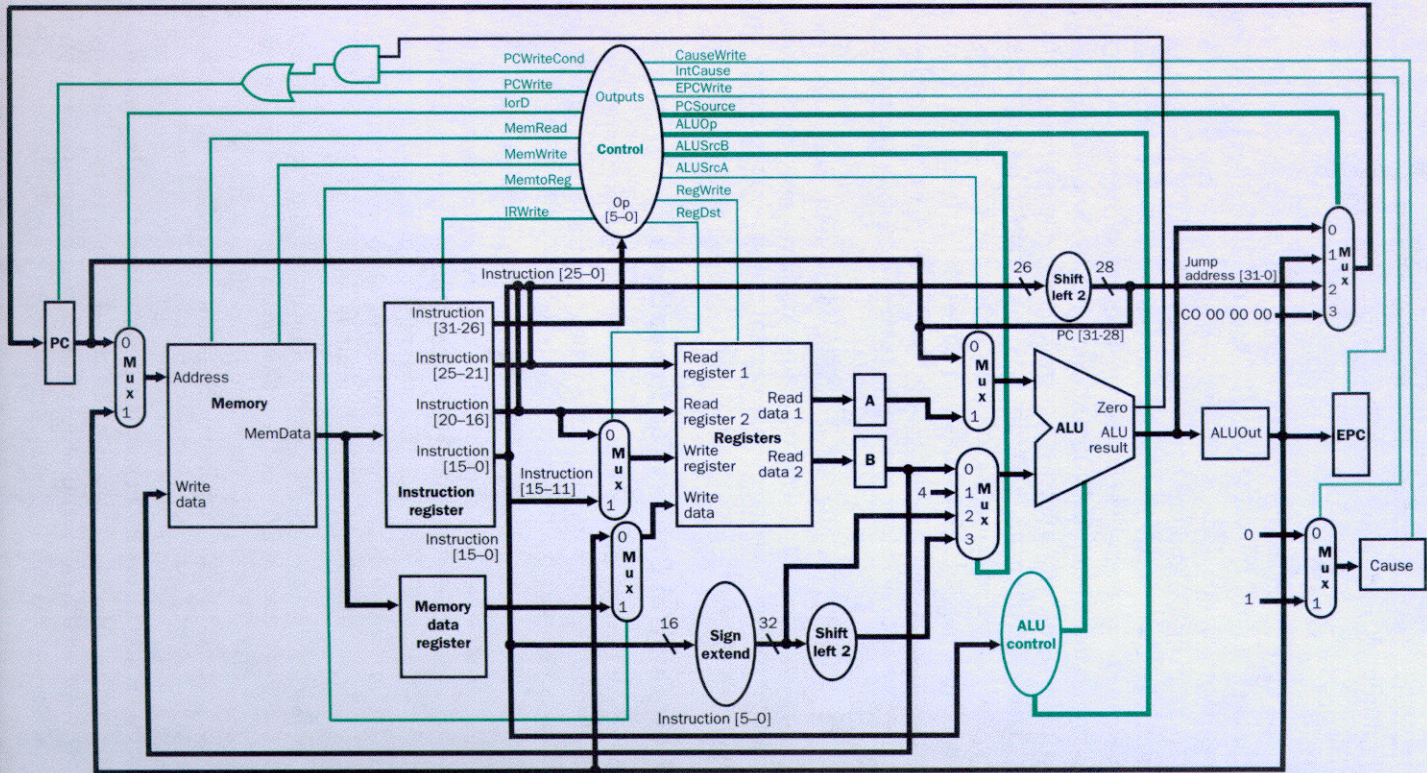
1. Include a status register which is loaded with a code indicating the cause of the exception, the Cause register in MIPS.
2. Vectored interrupts where the address of the handler is stored into a table which one entry per kind of exception.

MIPS APPROACH

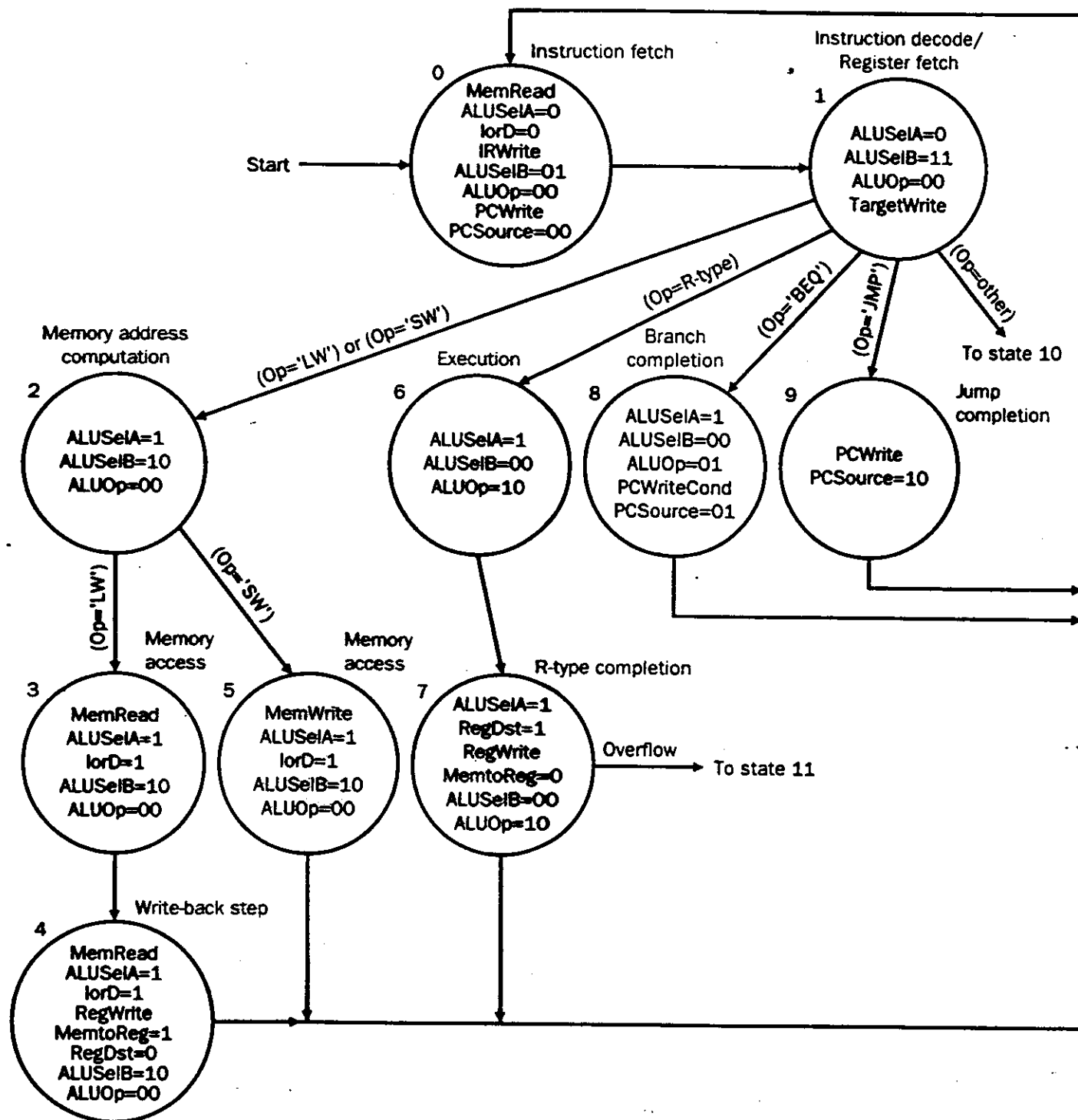
In MIPS, we have added two registers: EPC and Cause with two new control signals EPCWrite and CauseWrite. For simplicity, we assume that there is only two possible causes indicated by control signal IntCause (undefined: 0, Overflow: 1).

Currently, the PCSource signal indicates how to feed the PC and there are three possible cases. We add one more case to feed the PC the constant 8000 0000 (in hexadecimal). In case of exception, we make the machine jump to that address and augment the finite state machine with four states:





Then connect it to the original machine.



(5.54)