Designing
MIPS Processor
(Multi-Cycle)

Dr. Arjan Durresi
Louisiana State University
Baton Rouge, LA 70810
Durresi@Csc.LSU.Edu

These slides are available at:

http://www.csc.lsu.edu/~durresi/CSC3501_07/

Overview

- ❑ Datapath
- ❑ Control Unit

Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined directly by instruction
  - e.g., what should the ALU do for a "subtract" instruction?
- We'll use a finite state machine for control

MultiCycle Design Principles

- Break up execution of each instruction into steps.
- A number of steps and tasks in each step are instruction dependent.
- Each step takes one clock cycle.
- Balance the amount of work to be done in each clock cycle.
- 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.
- Major units are memory, register file and ALU, since we assume that they introduce the most significant delays during execution of instructions.
- We assume all other delays in the datapath negligible.

MultiCycle Design Principles

- During execution of any instruction, we may be reusing functional units, e.g.
  - Memory will be used for instruction and data,
  - 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.
- Control signals will not be determined solely by the instruction in execution but also by the particular clock cycle the instruction is being executed in.
- At the end of each cycle during instruction execution store intermediate values for use in later cycles.
- For that purpose, introduce additional "internal" registers (easiest thing to do).

Elaboration on Work Balance in Each Step

- During any step it is not allowed to have any serial combination of usage of the major functional units; for example:
  - It is not allowed that in one step contents of registers are read from the register file and then those contents are used as operands for ALU in the same step, or
  - It is not allowed that in one step ALU performs a function on some operands and its result is used as an address for memory read or write in the same step.
- This principle is introduced to avoid that any step requires unnecessary long duration, implying that clock cycles have to be of that unnecessary length.
- Notice that two of major functional units are allowed to be used in parallel, e.g. reading contents from a register file and that ALU performs a function on unrelated data at the same step is allowed.

Instructions from ISA perspective

- Consider each instruction from perspective of ISA.
- Example:
  - The add instruction changes a register.
  - Register specified by bits 15:11 of instruction.
  - Instruction specified by the PC.
  - New value is the sum ("op") of two registers.
  - Registers specified by bits 25:21 and 20:16 of the instruction

    ```
    Reg[Memory[PC][15:11]] <=  Reg[Memory[PC][25:21]]
    op
    Reg[Memory[PC][20:16]]
    ```

  - In order to accomplish this we must break up the instruction.
    (kind of like introducing variables when programming)

Breaking down an instruction

- ISA definition of arithmetic:

  ```
  Reg[Memory[PC][15:11]] <= Reg[Memory[PC][25:21]]  op
                               Reg[Memory[PC][20:16]]
  ```

- Could break down to:
  - `IR <= Memory[PC]`
  - `A <= Reg[IR[25:21]]`
  - `B <= Reg[IR[20:16]]`
  - `ALUOut <= A op B`
  - `Reg[IR[20:16]] <= ALUOut`

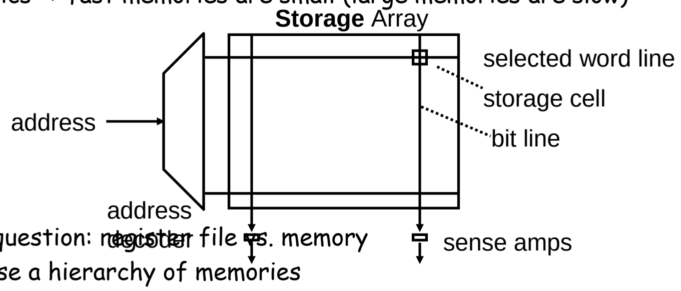
- We forgot an important part of the definition of arithmetic!
  - `PC <= PC + 4`

Idea behind multicycle approach

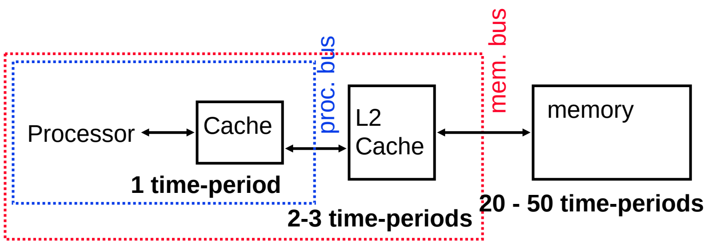
- We define each instruction from the ISA perspective (do this!)
- Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)
- Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)
- Finally try and pack as much work into each step (avoid unnecessary cycles) while also trying to share steps where possible (minimizes control, helps to simplify solution)
- Result: Our book's multicycle Implementation!

Memory Access Time

- Physics => fast memories are small (large memories are slow)

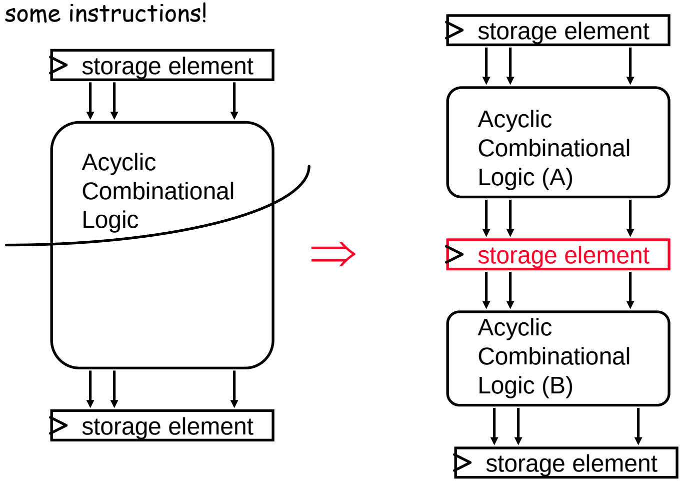


  - question: register file vs. memory
- => Use a hierarchy of memories



Reducing Cycle Time

- Cut combinational dependency graph and insert register / latch
- Do same work in two fast cycles, rather than one slow one
- May be able to short-circuit path and remove some components for some instructions!



storage element

Acyclic Combinational Logic

storage element

⇒

storage element

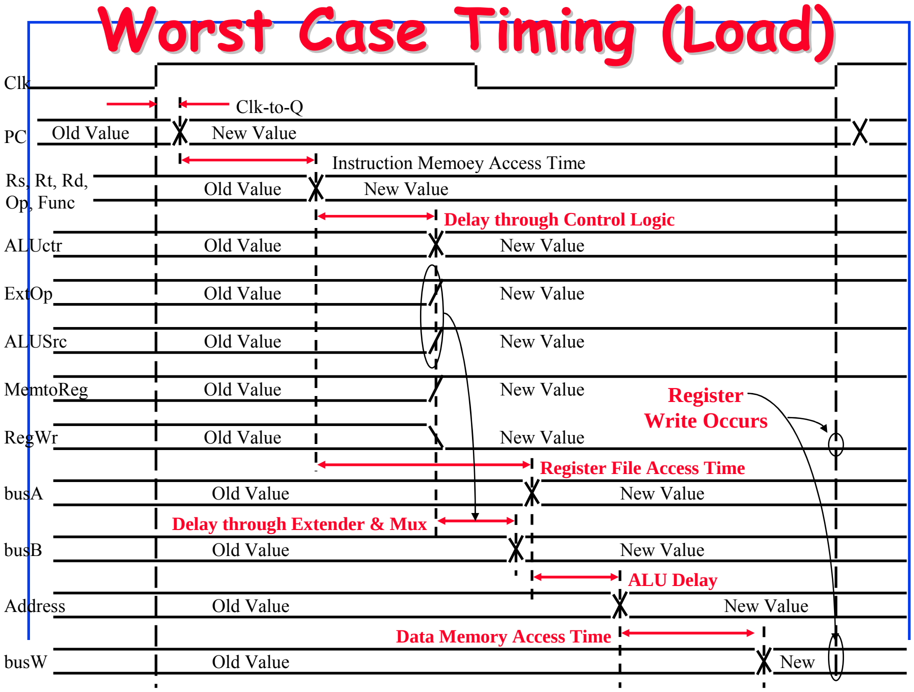
Acyclic Combinational Logic (A)

storage element

Acyclic Combinational Logic (B)

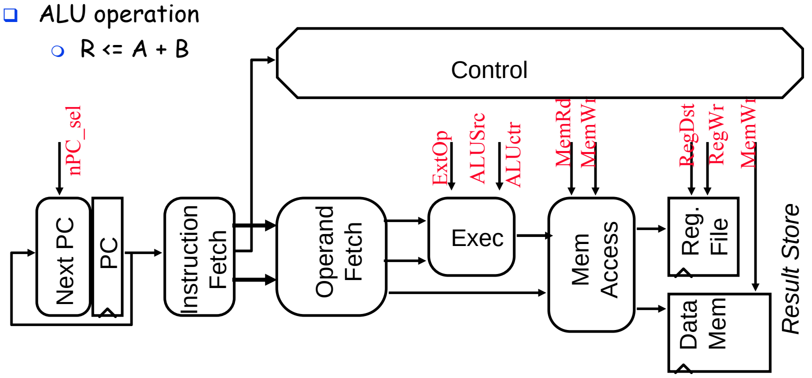
storage element

Worst Case Timing (Load)



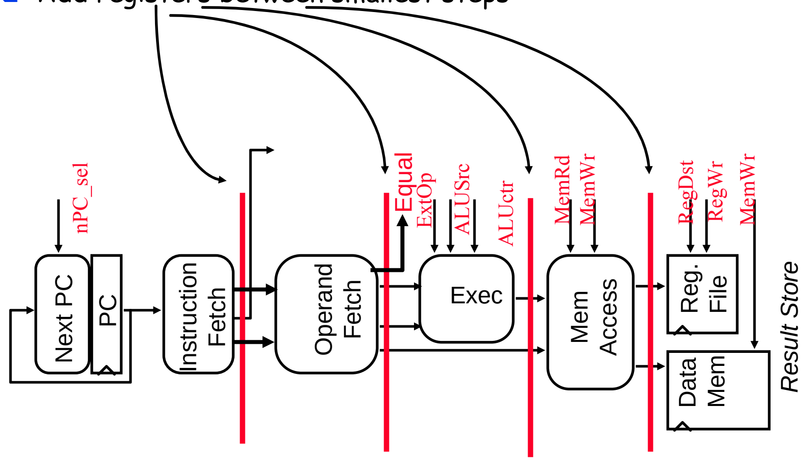
Basic Limits on Cycle Time

- Next address logic
  - PC <= branch ? PC + offset : PC + 4
- Instruction Fetch
  - InstructionReg <= Mem[PC]
- Register Access
  - A <= R[rs]
- ALU operation
  - R <= A + B



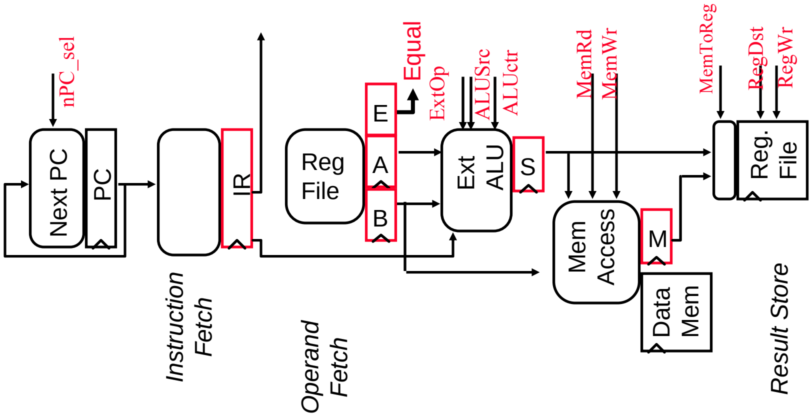
Partitioning the CPI=1 Datapath

- Add registers between smallest steps



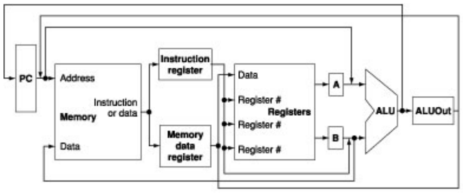
- Place enables on all registers

Example Multicycle Datapath

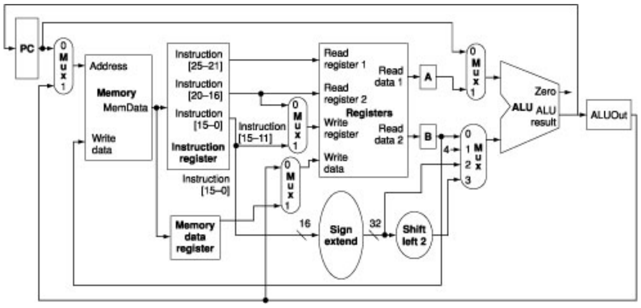


- Critical Path ?

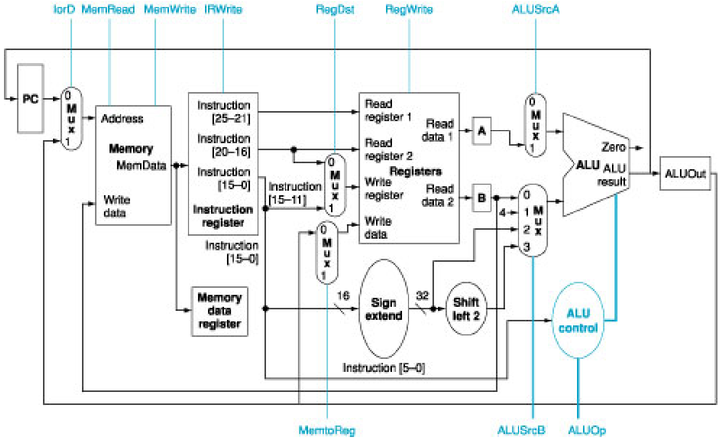
Multi-Cycle Datapath High Level View



- The use of shared functional units requires new temporary registers that
- hold data between clock cycles of the same instruction.
- The additional registers are:
  - Instruction register (IR),
  - Memory data register (MDR),
  - A and B registers,
  - ALUout register.



Multi-Cycle Datapath Detailed View



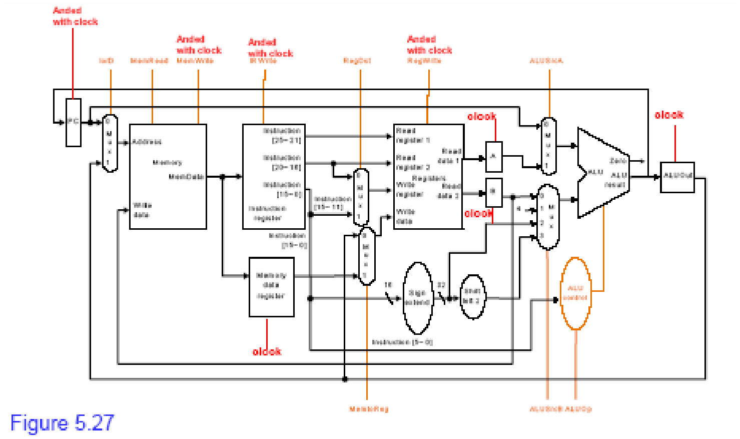
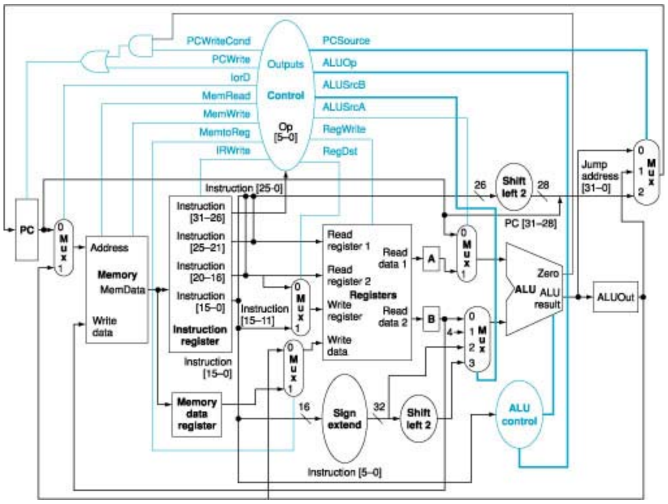


Figure 5.27
with additions in red

Multi-Cycle Datapath and Control



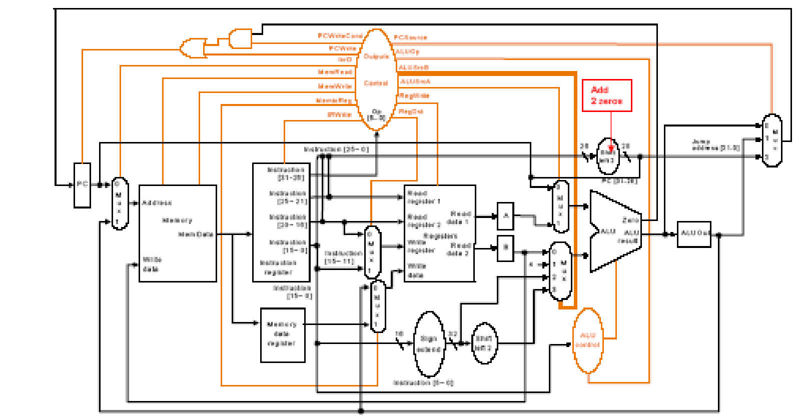


Figure 5.28
with a correction in red

Five Steps In Instruction Execution

- Major steps in execution of an instruction are:
  - Instruction Fetch
  - Instruction Decode and Register Fetch
  - Execution, Memory Address Computation, or Branch Completion
  - Memory Access or R-type instruction completion
  - Write-back step
- Not every instruction will have all those steps
- Instructions will take 3-5 steps, i.e. 3-5 clock cycles.
- The first two steps are common to all instructions.

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
    IR <= Memory[PC];
    PC <= PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Step 1: Instruction Fetch Control Signals

- Assert MemRead and IRWrite
- Set IorD to 0 to select the PC as the source of the address
- Set ALUSrcA to 0 (Sending the PC to ALU)
- Set ALUSrcB signal to 01 (sending 4 to ALU)
- Set ALUop to 00 (to make ALU to add)
- Set PCsource to 00 and set the PCWrite

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
```

- We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)
- Control
  - Set ALUSrcA to 0 (PC is sent to ALU)
  - Set ALUSrcB to 11 (so that the sign-extended and shifted offset field is sent to ALU)
  - Set ALUOp to 00 (so the ALU adds)

Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:

```
ALUOut <= A + sign-extend(IR[15:0]);
```

  - ALUSrcA to 1 (so the first ALU input is register A)
  - ALUSrcB to 10 (so the output of the sigh extension unit is used for the second ALU input), ALUOp to 00
- R-type:

```
ALUOut <= A op B;
```

  - ALUSrcA=1, ALUSrcB=00, ALUOp = 10 (the funct field determine the ALU control signal settings)
- Branch:

```
if (A==B) PC <= ALUOut;
```

  - ALUSrc = 1, ALUSrcB = 00, ALUOp = 01 (substract), PCWriteCond asserted, PCSource = 01 (PC value will come from ALUOut)
  - ALUOut was computed in the previous cycle.

Step 4 (R-type or memory-access)

- Loads and stores access memory

```
MDR <= Memory[ALUOut];
       or
Memory[ALUOut] <= B;
```

  - MemRead (for load) or MemWrite (for store) will need to be asserted
- R-type instructions finish

```
Reg[IR[15:11]] <= ALUOut;
```

  - ALUOut corresponds to the ALU operation in the previous cycle
  - RegDst = 1 to force the rd field (bits 15:11) to be used to select the register file entry to write
  - RegWrite asserted, MemtoReg = 0, so the output of ALU is written
  - *The write actually takes place at the end of the cycle on the edge*

Write-back step

- `Reg[IR[20:16]] <= MDR;`

Which instruction needs this?

Write-back step

- Write the load data, which was stored into MDR in the previous cycle, to register file
- MemtoReg = 1 (to write the result from the memory),
- Assert RegWrite (to cause a write)
- RegDst = 0 to choose rt (bits 20:16) field as the register number

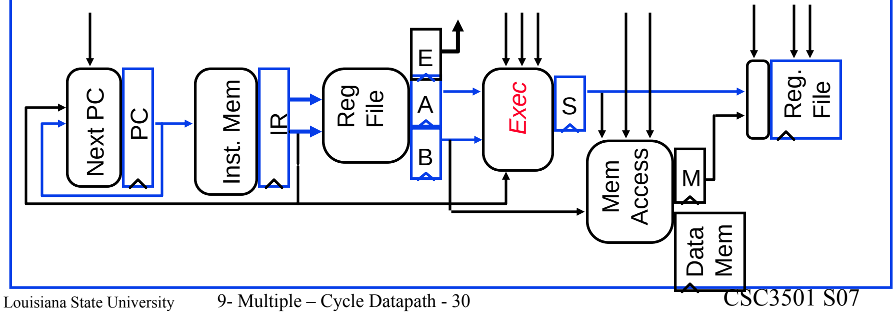
R-rtype (add, sub, . . .)

- Logical Register Transfer
- Physical Register Transfers

inst	Logical Register Transfers
ADDU	R[rd] <– R[rs] + R[rt]; PC <– PC + 4

Time

inst	Physical Register Transfers	
	IR <– MEM[pc]	
ADDU	A<– R[rs]; B <– R[rt]	
	S <– A + B	
	R[rd] <– S;	PC <– PC + 4



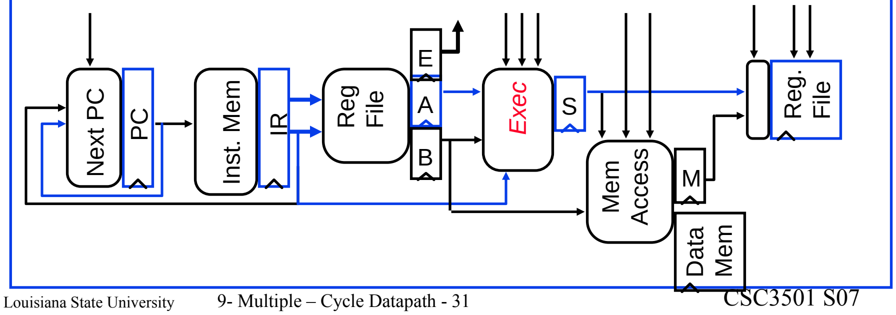
Logical immed

- Logical Register Transfer
- Physical Register Transfers

inst	Logical Register Transfers
ORI	R[rt] <– R[rs] OR ZExt(Im16); PC <– PC + 4

Time

inst	Physical Register Transfers	
	IR <– MEM[pc]	
ORI	A<– R[rs]; B <– R[rt]	
	S <– A *or* ZExt(Im16)	
	R[rt] <– S;	PC <– PC + 4



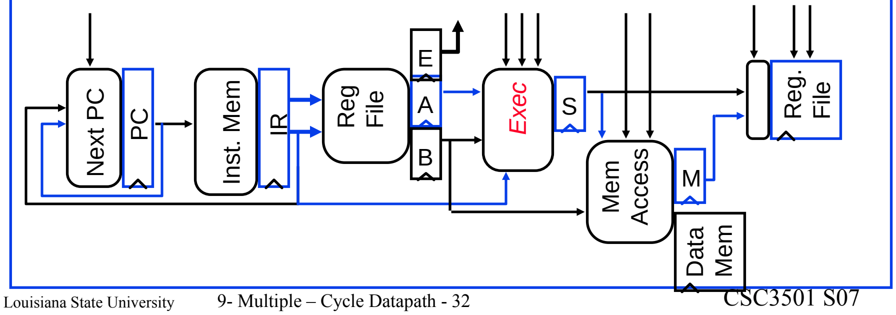
Load

- Logical Register Transfer
- Physical Register Transfers

inst	Logical Register Transfers
LW	R[rt] <– MEM[R[rs] + SExt(Im16)];
	PC <– PC + 4

Time

inst	Physical Register Transfers	
	IR <– MEM[pc]	
LW	A<– R[rs]; B <– R[rt]	
	S <– A + SExt(Im16)	
	M <– MEM[S]	
	R[rd] <– M;	PC <– PC + 4



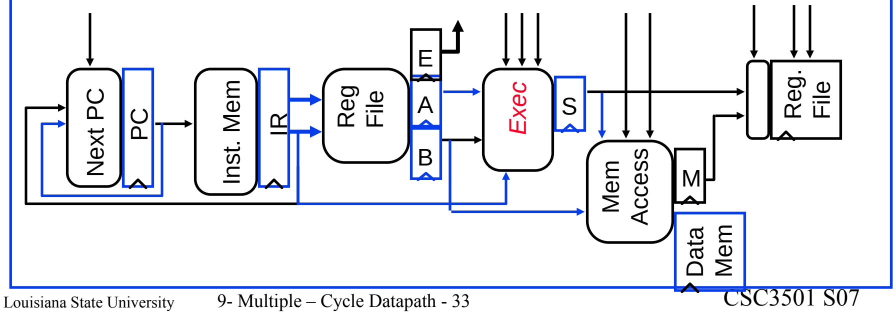
Store

- Logical Register Transfer
- Physical Register Transfers

inst	Logical Register Transfers
SW	MEM[R[rs] + SExt(Im16)] <– R[rt];
	PC <– PC + 4

Time

inst	Physical Register Transfers
	IR <– MEM[pc]
SW	A<– R[rs]; B <– R[rt]
	S <– A + SExt(Im16);
	MEM[S] <– B PC <– PC + 4



Branch

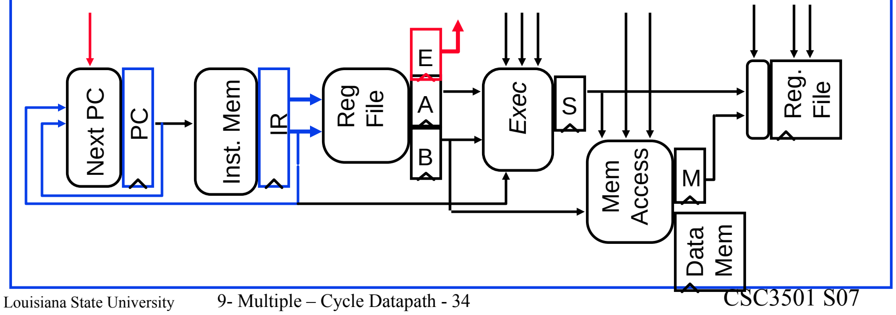
- Logical Register Transfer
- Physical Register Transfers

inst	Logical Register Transfers
BEQ	if R[rs] == R[rt]
	then PC <= PC + 4+SExt(Im16) || 00
	else PC <= PC + 4

Time

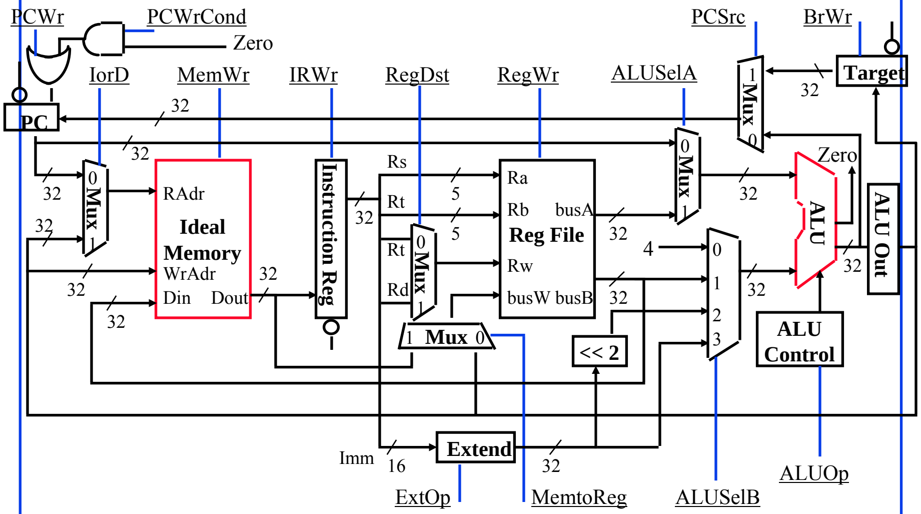
inst	Physical Register Transfers
	IR <– MEM[pc]
BEQ	E<– (R[rs] = R[rt])

	if !E then PC <– PC + 4
	else PC <–PC+4+SExt(Im16)||00

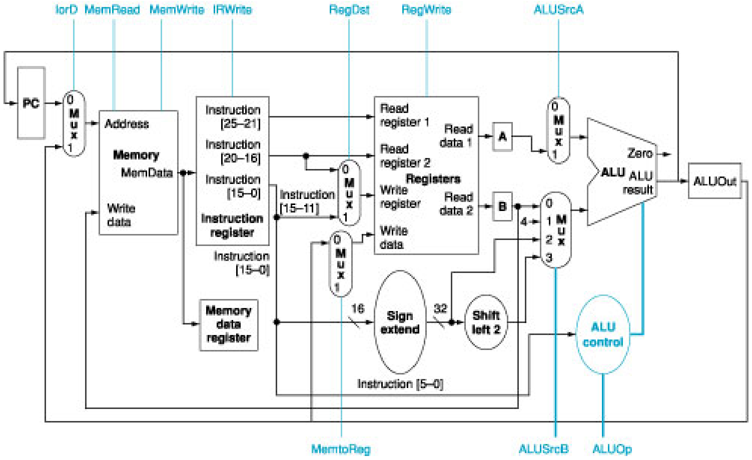


Multiple Cycle Datapath

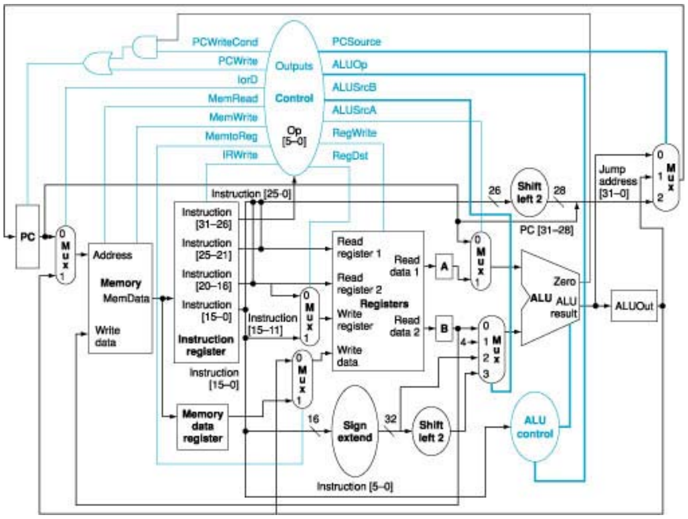
- Miminizes Hardware: 1 memory, 1 adder



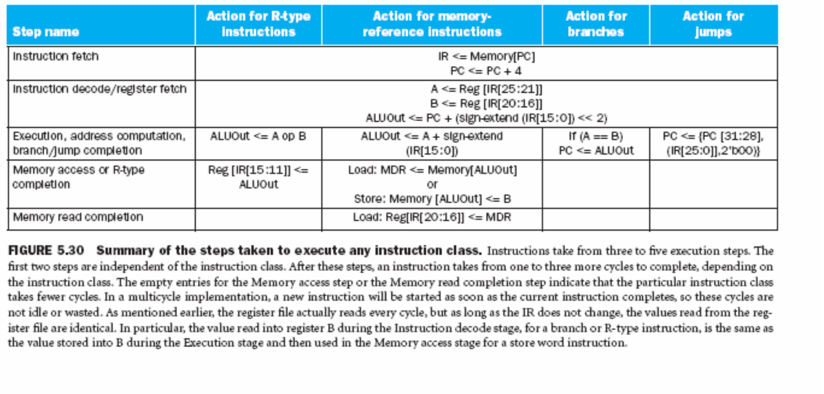
Multiple Cycle Datapath



Multiple Cycle Datapath

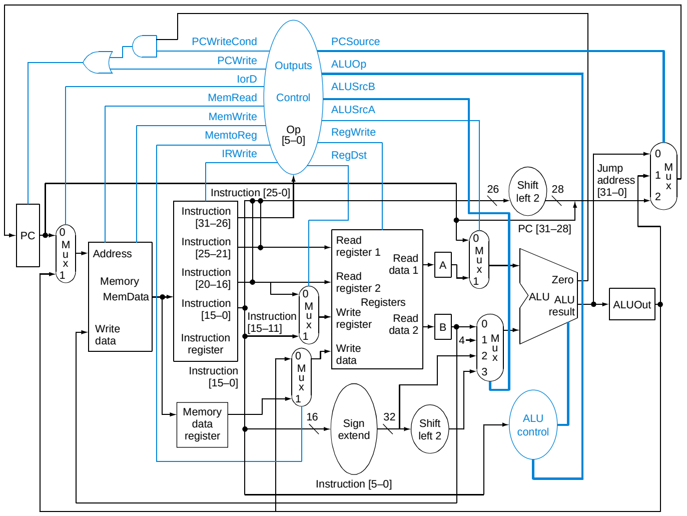


Summary:



Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.



CPI in a Multicycle CPU

- 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, 105 stores, 11% branches, 2% jumps, 52% ALU
- Number of cycle: Loads: 5, Stores 4, ALU 4, Branches 3, Jumps 3

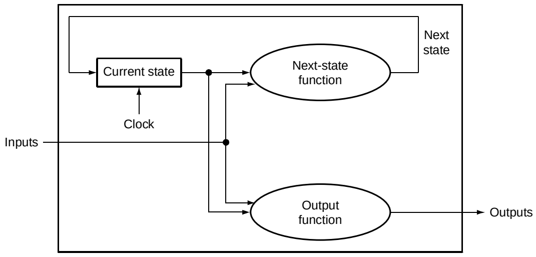
$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction counts}} = \frac{\sum \text{Instruction count}_i \text{ x } \text{CPI}_i}{\text{Instruction counts}}$$

CPI = 0.25x5 + 0.1x4 + 0.52x4 + 0.11 x3 = 4.12

Better than the worst case 5

Review: finite state machines

- Finite state machines:
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)



  - We'll use a *Moore machine* (output based only on current state)
  - If the output function can depend on both the current state and the current input, the machine is called a *Mealy machine*.

- These two machines are equivalent in their capabilities,
- One can be turned into the other mechanically.
- The basic advantage of a Moore machine is that it can be faster,
- while a Mealy machine may be smaller, since it may need fewer states than a Moore machine.

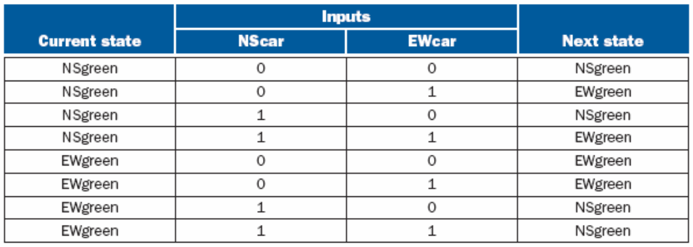
Example

- Our example concerns the control of a traffic light at an intersection of a north-south route and an east-west route.
- For simplicity, we will consider only the green and red lights.
- We want the lights to cycle no faster than 30 seconds in each direction, so we will use a 0.033 Hz clock so that the machine cycles between states at no faster than once every 30 seconds.
- There are two output signals:
- ■ *NSlite:* When this signal is asserted, the light on the north-south road is green; when this signal is deasserted the light on the north-south road is red.
- ■ *EWlite:* When this signal is asserted, the light on the east-west road is green; when this signal is deasserted the light on the east-west road is red.

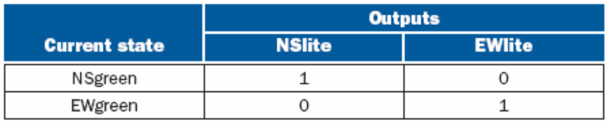
Example

- In addition, there are two inputs: NScar and EWcar.
- ■ *NScar:* Indicates that a car is over the detector placed in the roadbed in front of the light on the north-south road (going north or south).
- ■ *EWcar:* Indicates that a car is over the detector placed in the roadbed in front of the light on the east-west road (going east or west).
- The traffic light should change from one direction to the other only if a car is waiting to go in the other direction; otherwise, the light should continue to show green in the same direction as the last car that crossed the intersection.

Example



Current state	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen



Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

Example

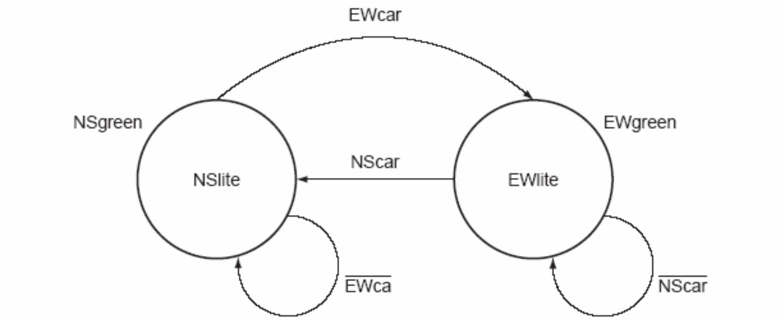


FIGURE B.10.2 The graphical representation of the two-state traffic light controller. We simplified the logic functions on the state transitions. For example, the transition from NSgreen to EWgreen in the next-state table is $(\overline{NScar} \cdot EWcar) + (NScar \cdot EWcar)$, which is equivalent to EWcar.

Example

- A finite state machine can be implemented with a register to hold the current state and a block of combinational logic that computes the next-state function and the output function.
- Next Figure shows how a finite state machine with 4 bits of state, and thus up to 16 states, might look.
- To implement the finite state machine in this way, we must first assign state numbers to the states. This process is called state assignment.
- For example, we could assign NSgreen to state 0 andEWgreen to state 1.
- The next-state function and the output function would be given as

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$

Example

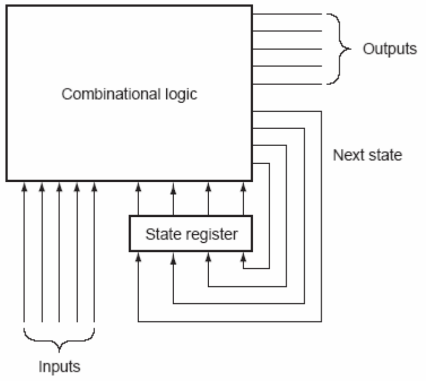
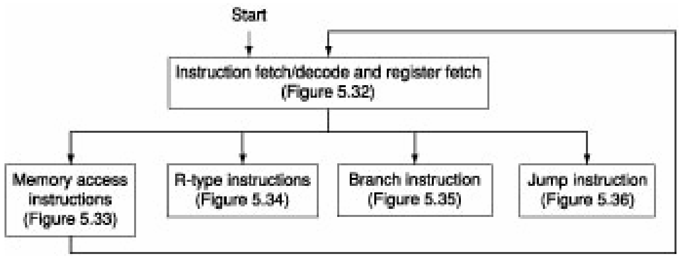


FIGURE B.10.3 A finite state machine is implemented with a state register that holds the current state and a combinational logic block to compute the next state and output functions. The latter two functions are often split apart and implemented with two separate blocks of logic, which may require fewer gates.

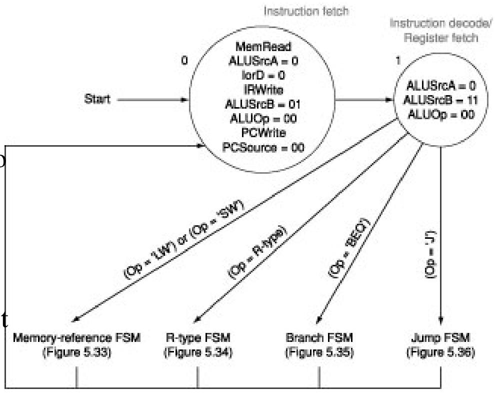
Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

The high-level view


Start
Instruction fetch/decode and register fetch (Figure 5.32)
Memory access instructions (Figure 5.33)
R-type instructions (Figure 5.34)
Branch instruction (Figure 5.35)
Jump instruction (Figure 5.36)

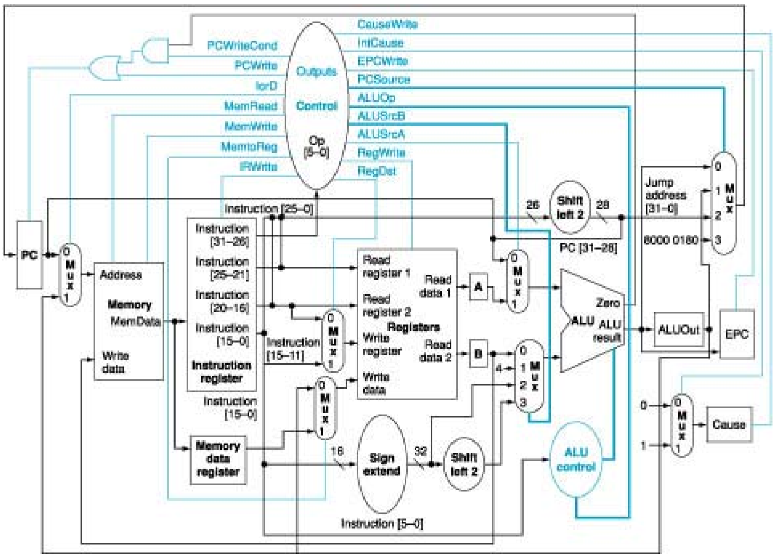
Fetch and Decode



Assert: MemRead to read the memory and IRWrite to write it into IR Set IoD to 0 to chose PC as the address source.

The signals: ALUSrcA, ALUSrcB, ALUOp, PCWrite and PCSource are set to compute PC+4 and store it in PC

In the next state we compute the branch target address by setting ALUSrcB to 11 (causing the shifted and sign-extended lower 16 bits of IR to be sent to ALU); ALUSrcA to0, ALUOp to 00

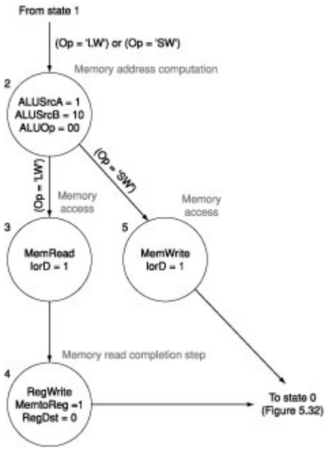


Memory reference

After performing a memory address calculation, a separate sequence is needed for load and store.

The setting of ALUSrcA, ALUSrcB, ALUOp is used to cause the memory address computation in state 2.

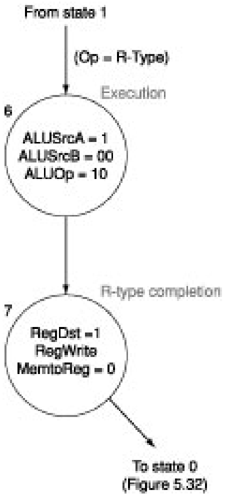
Loads require an extra state to write The result from MDR into register file



R-type instructions

The first state causes the ALU operation To occur, while the second state causes The ALU result to be written in the register File.

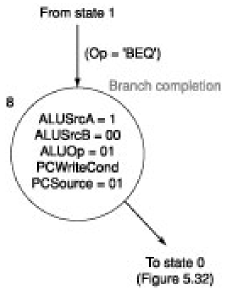
The three signals asserted during state 7 cause the content of ALUOut to be written Into the register file in the entry specified By rd field of the IR



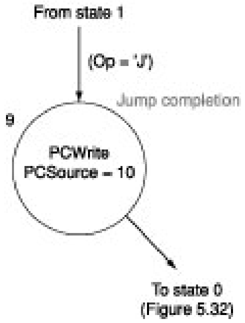
Branch instruction

The first three signals that are asserted cause the ALU to compare the registers ALUSrcA, ALUSrcB, and ALUOp, while the signals PCSource and PCWriteCond perform the Conditional write if the branch condition is true.

The branch target address is read from ALUOut Where it was saved at the end of state 1.



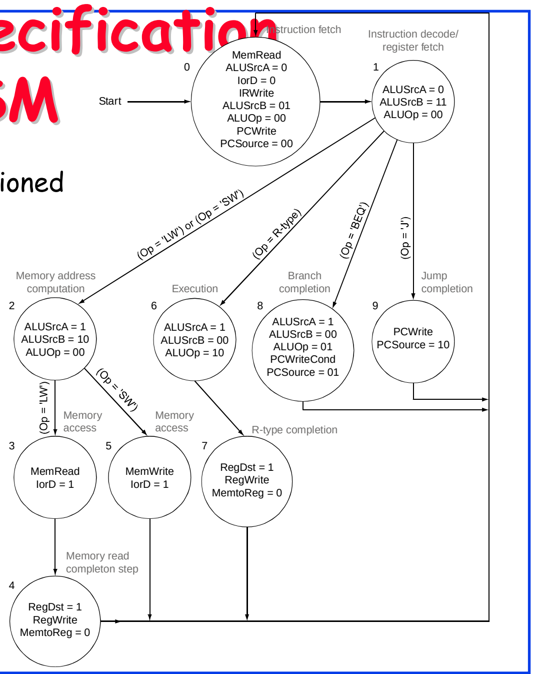
Jump instruction

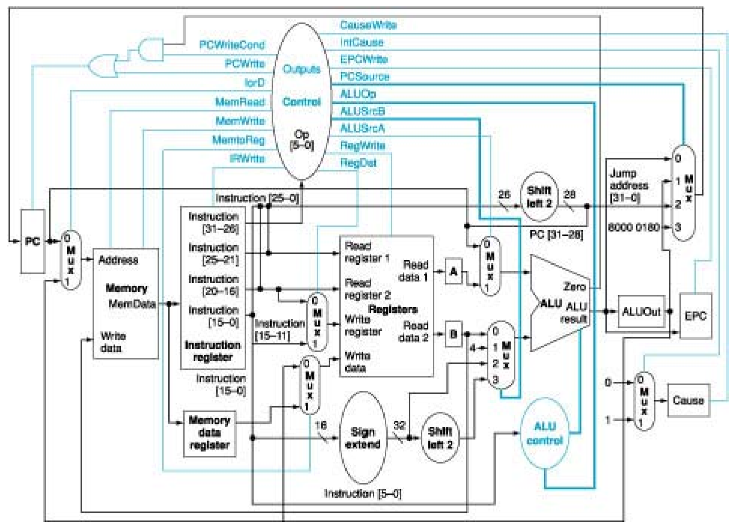


The jump instruction requires a single state That asserts two control signals to write The PC with the lower 26 bits of the Instruction Register shifted left 2 bits and concatenated to the upper 4 bits of the PC of this instruction

Graphical Specification of FSM

- Note:
  - don't care if not mentioned
  - asserted if name only
  - otherwise exact value
- How many state bits will we need?





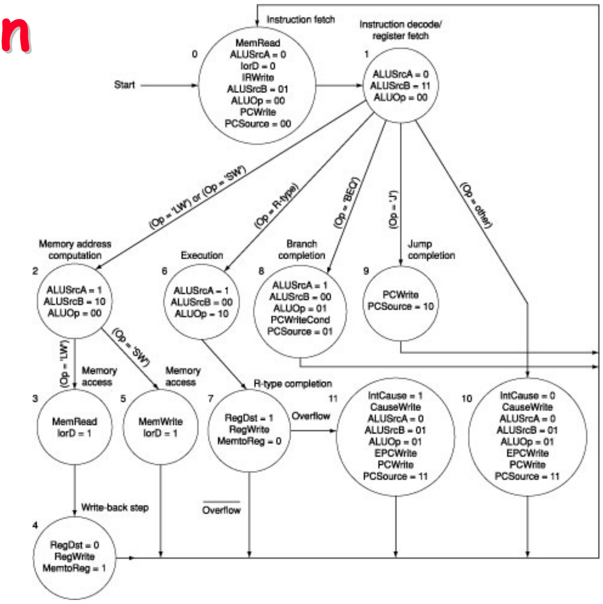
Exceptions

- An exception is an unexpected event from within the processor
  - arithmetic overflow
  - Invoke the operating system from user program
  - Using an undefined instruction
- An interrupt is an event that also causes an unexpected change in control flow but comes from outside of the processor
  - I/O device communication with processor
  - Hardware malfunction (could be both)

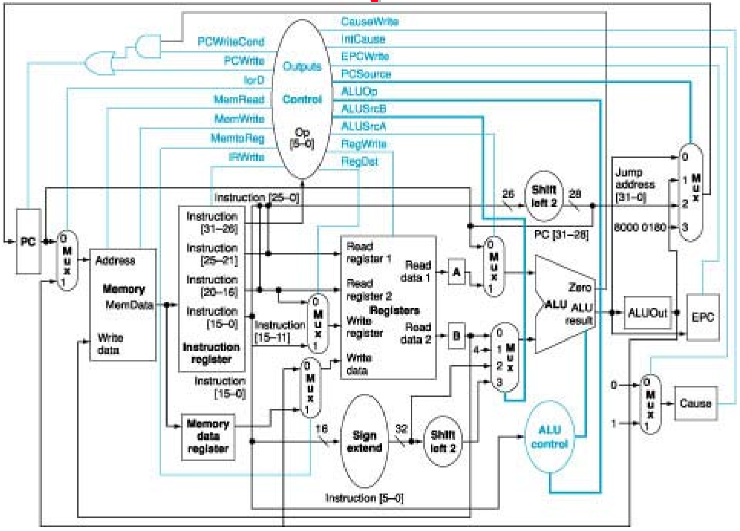
Exceptions

- When an exception happens:
  - Save the address of the offending instruction in the exception program counter (EPC) and then transfer control to the operating system at some specific address.
  - The reason is stored in the *Cause Register*.
  - The operating system takes the needed actions, then it can terminate the program or continue using the EPC to determine where to start.

+ Exception

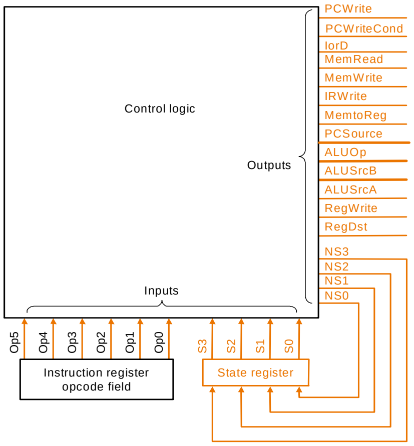


+ Exceptions



Finite State Machine for Control

- Implementation:
- With 10 states we will need 4 bits to encode the state number, and we call these state bits: S3, S2, S1, S0.
- The block labeled "control logic" in Figure is combinational logic.
- One part is the logic that determines the setting of the datapath control outputs, which depend only on the state bits.
- The other part of the control logic implements the next-state function; these equations determine the values of the nextstate bits based on the current-state bits and the other inputs (the 6-bit opcode).



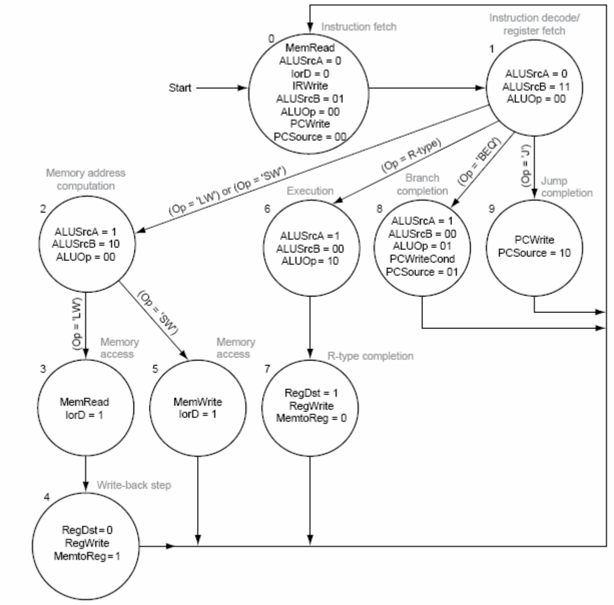
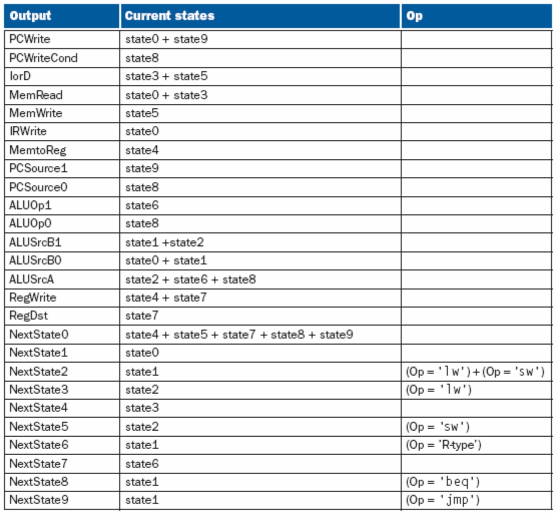


FIGURE C.3.1 The finite state diagram that was developed in Chapter 5; it is identical to Figure 5.37.



Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 +state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw')+(Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

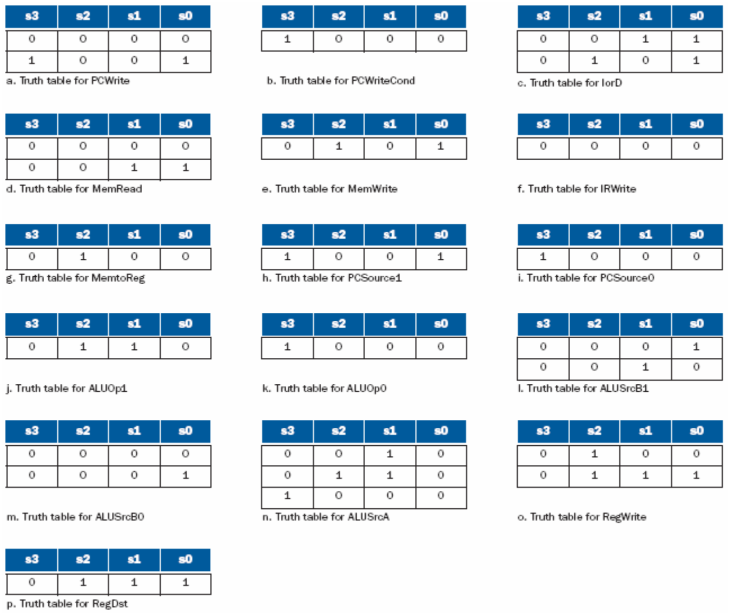
FIGURE C.3.3 The logic equations for the control unit shown in a shorthand form. Remember that "+" stands for OR in logic equations. The state inputs and NextState entries outputs must be expanded by using the state encoding. Any blank entry is a don't care.

Logic Equations for Next-State Outputs

- Give the logic equation for the low-order next-state bit, NS0.
- The next-state bit NS0 should be active whenever the next state has NS0 = 1 in the state encoding. This is true for NextState1, NextState3, NextState5, NextState7, and NextState9.

$$
\begin{aligned}
\text{NextState3} &= \text{State2} \cdot (\text{Op}[5\text{-}0] = \text{lw}) \\
&= \overline{\text{S3}} \cdot \overline{\text{S2}} \cdot \text{S1} \cdot \overline{\text{S0}} \cdot \text{Op5} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op3}} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \text{Op0} \\
\text{NextState5} &= \text{State 2} \cdot (\text{Op}[5\text{-}0] = \text{sw}) \\
&= \overline{\text{S3}} \cdot \overline{\text{S2}} \cdot \text{S1} \cdot \overline{\text{S0}} \cdot \text{Op5} \cdot \overline{\text{Op4}} \cdot \text{Op3} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \text{Op0} \\
\text{NextState7} &= \text{State6} = \overline{\text{S3}} \cdot \text{S2} \cdot \text{S1} \cdot \overline{\text{S0}} \\
\text{NextState9} &= \text{State1} \cdot (\text{Op}[5\text{-}0] = \text{jmp}) \\
&= \overline{\text{S3}} \cdot \overline{\text{S2}} \cdot \overline{\text{S1}} \cdot \text{S0} \cdot \overline{\text{Op5}} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op3}} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \overline{\text{Op0}}
\end{aligned}
$$

- NS0 is the logical sum of all these terms.



s3	s2	s1	s0
0	0	0	0
1	0	0	1

a. Truth table for PCWrite

s3	s2	s1	s0
1	0	0	0

b. Truth table for PCWriteCond

s3	s2	s1	s0
0	0	1	1
0	1	0	1

c. Truth table for IorD

s3	s2	s1	s0
0	0	0	0
0	0	1	1

d. Truth table for MemRead

s3	s2	s1	s0
0	1	0	1

e. Truth table for MemWrite

s3	s2	s1	s0
0	0	0	0

f. Truth table for IRWrite

s3	s2	s1	s0
0	1	0	0

g. Truth table for MemtoReg

s3	s2	s1	s0
1	0	0	1

h. Truth table for PCSource1

s3	s2	s1	s0
1	0	0	0

i. Truth table for PCSource0

s3	s2	s1	s0
0	1	1	0

j. Truth table for ALUOp1

s3	s2	s1	s0
1	0	0	0

k. Truth table for ALUOp0

s3	s2	s1	s0
0	0	0	1
0	0	1	0

l. Truth table for ALUSrcB1

s3	s2	s1	s0
0	0	0	0
0	0	0	1

m. Truth table for ALUSrcB0

s3	s2	s1	s0
0	0	1	0
0	1	1	0
1	0	0	0

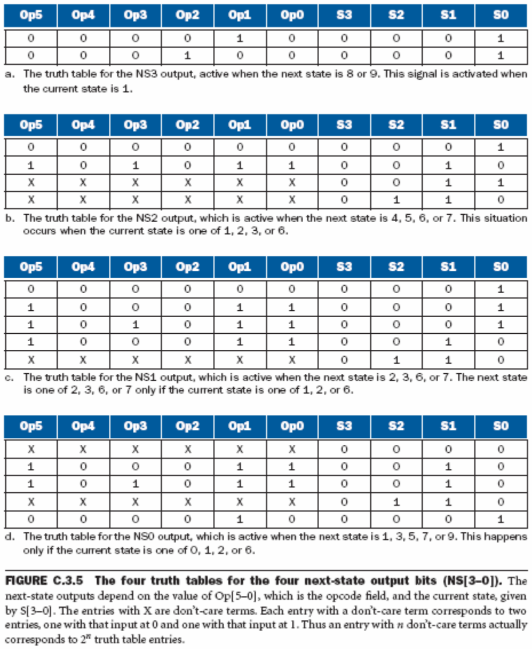
n. Truth table for ALUSrcA

s3	s2	s1	s0
0	1	0	0
0	1	1	1

o. Truth table for RegWrite

s3	s2	s1	s0
0	1	1	1

p. Truth table for RegDst



Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	0	1

a. The truth table for the NS3 output, active when the next state is 8 or 9. This signal is activated when the current state is 1.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	0	1	1
X	X	X	X	X	X	0	1	1	0

b. The truth table for the NS2 output, which is active when the next state is 4, 5, 6, or 7. This situation occurs when the current state is one of 1, 2, 3, or 6.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	0	1
1	0	1	0	1	1	0	0	0	1
1	0	0	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0

c. The truth table for the NS1 output, which is active when the next state is 2, 3, 6, or 7. The next state is one of 2, 3, 6, or 7 only if the current state is one of 1, 2, or 6.

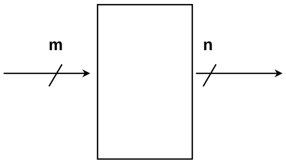
Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
X	X	X	X	X	X	0	0	0	0
1	0	0	0	1	1	0	0	1	0
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0
0	0	0	0	1	0	0	0	0	1

d. The truth table for the NS0 output, which is active when the next state is 1, 3, 5, 7, or 9. This happens only if the current state is one of 0, 1, 2, or 6.

FIGURE C.3.5 The four truth tables for the four next-state output bits (NS[3–0]). The next-state outputs depend on the value of Op[5–0], which is the opcode field, and the current state, given by S[3–0]. The entries with X are don't-care terms. Each entry with a don't-care term corresponds to two entries, one with that input at 0 and one with that input at 1. Thus an entry with n don't-care terms actually corresponds to 2^n truth table entries.

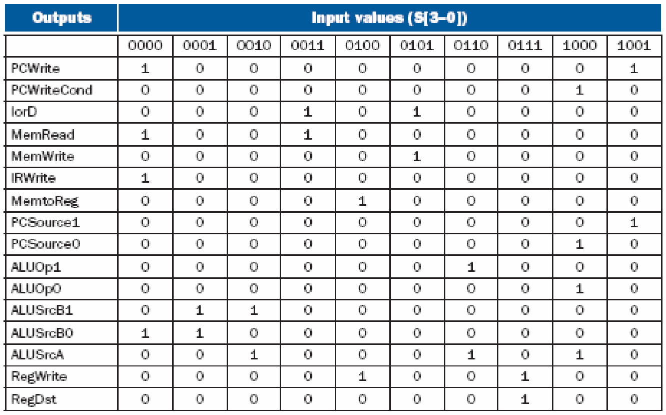
ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address 2^m entries in the ROM.
  - our outputs are the bits of data that the address points to.
  - The number of entries in the memory for the truth tables of Figures C.3.4 and C.3.5 is equal to all possible values of the inputs (the 6 opcode bits plus the 4 state bits), which is 2 # inputs = 2^{10}= 1024.
  - The width of each entry (or word in the memory) is 20 bits since there are 16 datapath control outputs and 4 next-state bits. This means the total size of the ROM is 210×20 = 20 Kbits.
  -



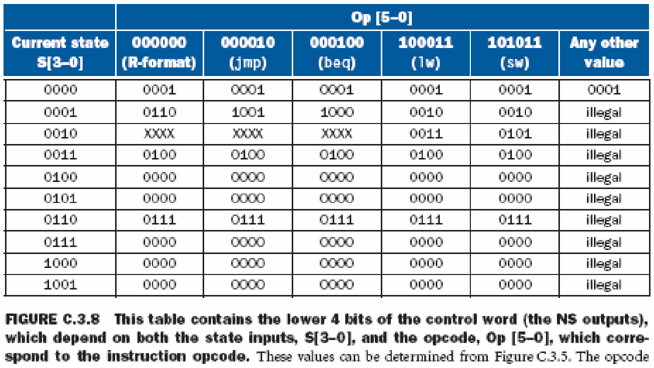
0	0	0	0	0	1	1
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	1	1	0
1	1	1	0	1	1	1

m is the "height", and n is the "width"



Outputs	Input values (S[3–0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	1	0	0	0	0	0
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

FIGURE C.3.6 The truth table for the 16 datapath control outputs, which depend only on the state inputs. The values are determined from Figure C.3.4. Although there are 16 possible values for the 4-bit state field, only 10 of these are used and are shown here. The 10 possible values are shown at the top; each column shows the setting of the datapath control outputs for the state input value that appears at the top of the column. For example, when the state inputs are 0011 (state 3), the active datapath control outputs are IorD or MemRead.

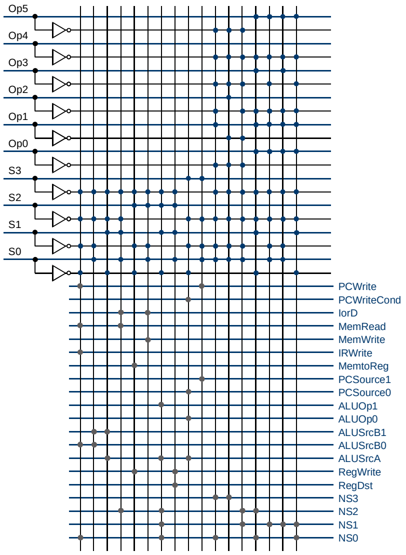


Current state S[3–0]	Op [5–0] 000000 (R-format)	000010 (jmp)	000100 (beq)	100011 (lw)	101011 (sw)	Any other value
0000	0001	0001	0001	0001	0001	0001
0001	0110	1001	1000	0010	0010	illegal
0010	XXXX	XXXX	XXXX	0011	0101	illegal
0011	0100	0100	0100	0100	0100	illegal
0100	0000	0000	0000	0000	0000	illegal
0101	0000	0000	0000	0000	0000	illegal
0110	0111	0111	0111	0111	0111	illegal
0111	0000	0000	0000	0000	0000	illegal
1000	0000	0000	0000	0000	0000	illegal
1001	0000	0000	0000	0000	0000	illegal

FIGURE C.3.8 This table contains the lower 4 bits of the control word (the NS outputs), which depend on both the state inputs, S[3–0], and the opcode, Op [5–0], which correspond to the instruction opcode. These values can be determined from Figure C.3.5. The opcode

PLA (programmable logic array) Implementation

- If I picked a horizontal or vertical line could you explain it?



ROM Implementation

- How many inputs are there?
  6 bits for opcode, 4 bits for state = 10 address lines
  (i.e., 2^{10} = 1024 different addresses)
- How many outputs are there?
  16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is 2^{10} x 20 = 20K bits (and a rather unusual size)
- Rather wasteful, since for lots of the entries, the outputs are the same
  — i.e., opcode is often ignored

ROM vs PLA

- Break up the table into two parts
  — 4 state bits tell you the 16 outputs, 2^4 x 16 bits of ROM
  — 10 bits tell you the 4 next state bits, 2^{10} x 4 bits of ROM
  — Total: 4.3K bits of ROM
- PLA is much smaller
  — can share product terms
  — only need entries that produce an active output
  — can take into account don't cares
- Size is (#inputs × #product-terms) + (#outputs × #product-terms)
  For this example = (10x17)+(20x17) = 510 PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)

- Much of the logic is used to specify the next-state function. In fact, for the implementation using two separate ROMs, 4096 out of the 4368 bits (94%) correspond to the next-state function!
- To encode these more complex control functions efficiently, we can use a controlunit that has a counter to supply the sequential next state.
- This counter often eliminates the need to encode the next-state function explicitly in the control unit.

Another Implementation Style

- Complex instructions: the "next state" is often current state + 1

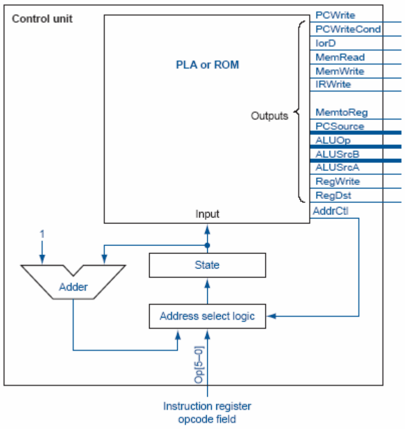
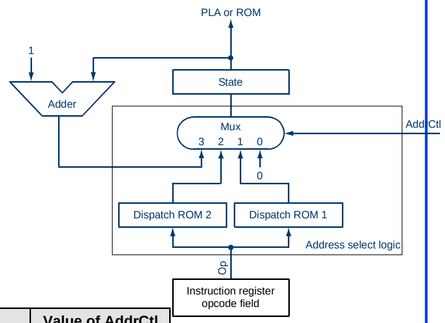


FIGURE C.4.1 The control unit using an explicit counter to compute the next state. In this control unit, the next state is computed using a counter (at least in some states). By comparison, Figure C.3.2 on page C-10 encodes the next state in the control logic for *every* state. In this control unit, the signals labeled *AddrCtl* control how the next state is determined.

Details

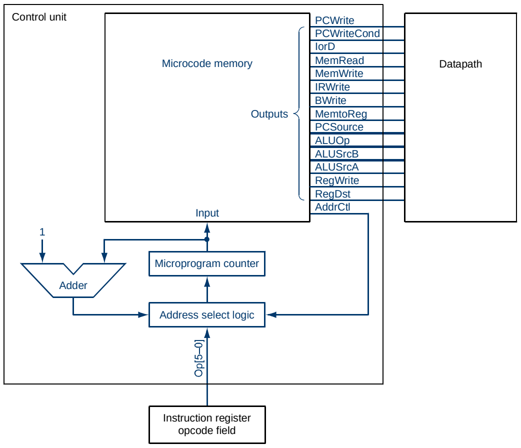
Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	**Opcode name**	**Value**
100011	lw	0011
101011	sw	0101



State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

Microprogramming



- What are the "microinstructions" ?

Microprogramming

- A specification methodology
  - appropriate if hundreds of opcodes, modes, cycles, etc.
  - signals specified symbolically using microinstructions

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

- *Will two implementations of the same architecture have the same microcode?*
- *What would a microassembler do?*

Microprogramming

- Consider instead an implementation of the full MIPS-32 instruction set, which contains over 100 instructions
- In one implementation, instructions take from 1 clock cycle to over 20 clock cycles
- Clearly, the control function will be much more complex.
- For such a design we would likely use a hardware design language, such as Verilog and have the finite state control synthesized,

Microprogramming

- Consider, however, an instruction set with several hundred instructions of widely varying classes, such as the IA-32 architecture.
- The control unit could easily require thousands of states with hundreds of different sequences.
- In such a case, specifying the control unit with a graphical representation will be impossible.
- Even using a finite state abstraction where the next state must be explicitly specified is likely to be cumbersome.

Microprogramming

- Suppose we think of the set of control signals that must be asserted in a state as an microinstruction to be executed by the datapath.
- Each microinstruction defines the set of datapath control signals that must be asserted in a given state.
- Executing a microinstruction has the effect of asserting the control signals specified by the microinstruction
- Designing the control as a program that implements the machine instructions in terms of simpler microinstructions is called *microprogramming*
- The key idea is to represent the asserted values on the control lines symbolically, so that the microprogram is a representation of the microinstructions, just as assembly language is a representation of the machine instructions.

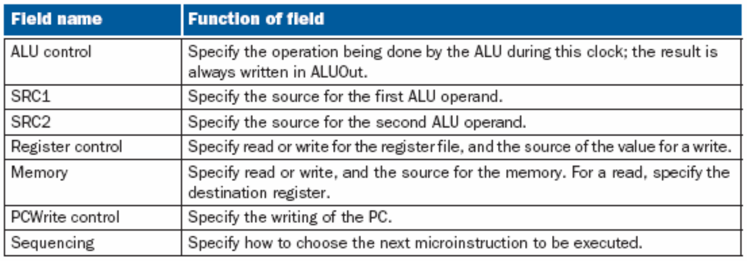
Microprogramming

- One other important idea from software is often incorporated into microprogrammed control: the *concept of subroutines*.
- In the implementation of a large instruction set with many complex instructions - it is likely that there are opportunities to reuse microcode sequences.
- Supporting subroutines in the microcode enables sharing of such microprogram sequences without having to duplicate the microinstructions.

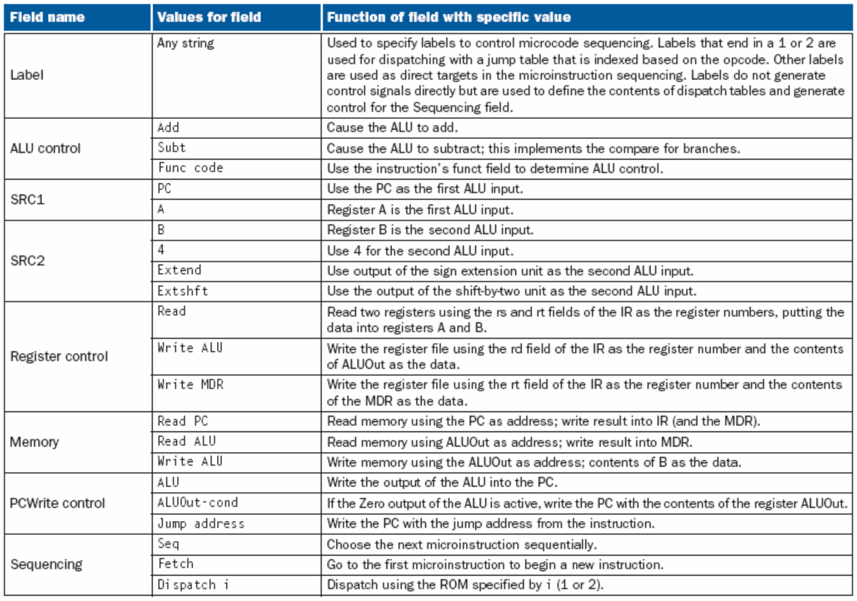
Defining a Microinstruction Format

- The microprogram is a symbolic representation of the control that will be translated by a program to control logic.
- The format of the microinstruction should be chosen so as to simplify the representation, making it easier to write and understand the microprogram.
- For example, it is useful to have one field that controls the ALU and a set of three fields that determine the two sources for the ALU operation as well as the destination of the ALU result

Microprogramming



Field name	Function of field
ALU control	Specify the operation being done by the ALU during this clock; the result is always written in ALUOut.
SRC1	Specify the source for the first ALU operand.
SRC2	Specify the source for the second ALU operand.
Register control	Specify read or write for the register file, and the source of the value for a write.
Memory	Specify read or write, and the source for the memory. For a read, specify the destination register.
PCWrite control	Specify the writing of the PC.
Sequencing	Specify how to choose the next microinstruction to be executed.

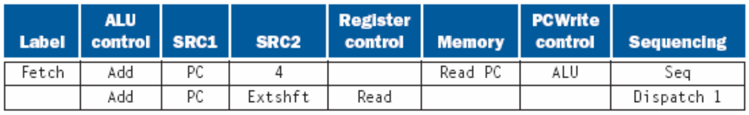


Field name	Values for field	Function of field with specific value
Label	Any string	Used to specify labels to control microcode sequencing. Labels that end in a 1 or 2 are used for dispatching with a jump table that is indexed based on the opcode. Other labels are used as direct targets in the microinstruction sequencing. Labels do not generate control signals directly but are used to define the contents of dispatch tables and generate control for the Sequencing field.
ALU control	Add	Cause the ALU to add.
	Subt	Cause the ALU to subtract; this implements the compare for branches.
	Func code	Use the instruction's funct field to determine ALU control.
SRC1	PC	Use the PC as the first ALU input.
	A	Register A is the first ALU input.
SRC2	B	Register B is the second ALU input.
	4	Use 4 for the second ALU input.
	Extend	Use output of the sign extension unit as the second ALU input.
	Extshft	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read	Read two registers using the rs and rt fields of the IR as the register numbers, putting the data into registers A and B.
	Write ALU	Write the register file using the rd field of the IR as the register number and the contents of ALUOut as the data.
	Write MDR	Write the register file using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	Read memory using ALUOut as address; write result into MDR.
	Write ALU	Write memory using the ALUOut as address; contents of B as the data.
PCWrite control	ALU	Write the output of the ALU into the PC.
	ALUOut-cond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	Jump address	Write the PC with the jump address from the instruction.
Sequencing	Seq	Choose the next microinstruction sequentially.
	Fetch	Go to the first microinstruction to begin a new instruction.
	Dispatch i	Dispatch using the ROM specified by i (1 or 2).

Each field of the microinstruction has a number of values that it can take on.

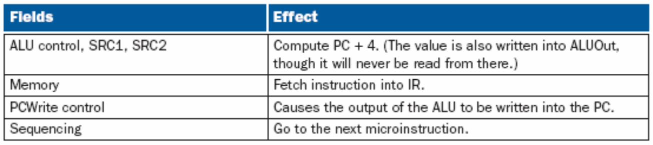
Microprogramming

- Fetch the instructions, decode them, and compute both the sequential PC and branch target PC.
- The two microinstructions needed for these first two steps are shown below:



Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1

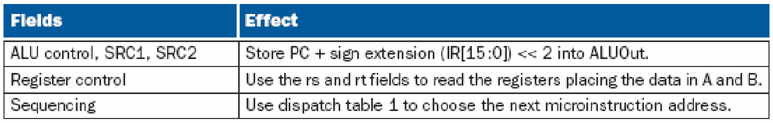
- In the first microinstruction, the fields asserted and their effects are the following:



Fields	Effect
ALU control, SRC1, SRC2	Compute PC + 4. (The value is also written into ALUOut, though it will never be read from there.)
Memory	Fetch instruction into IR.
PCWrite control	Causes the output of the ALU to be written into the PC.
Sequencing	Go to the next microinstruction.

Microprogramming

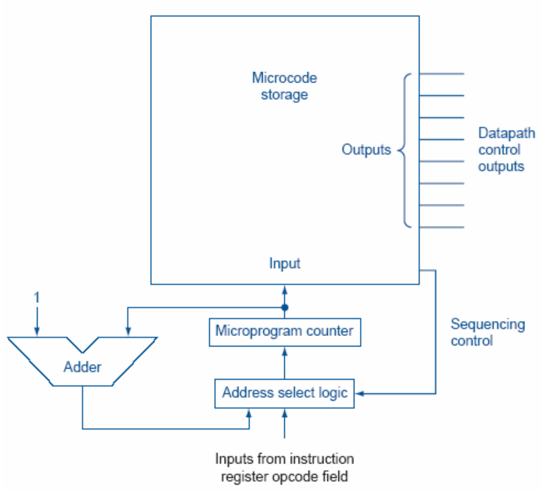
- For the second microinstruction, the operations controlled by the microinstruction are the following:



Fields	Effect
ALU control, SRC1, SRC2	Store PC + sign extension (IR[15:0]) << 2 into ALUOut.
Register control	Use the rs and rt fields to read the registers placing the data in A and B.
Sequencing	Use dispatch table 1 to choose the next microinstruction address.

Microprogramm Implementation

- Translating a microprogram into hardware involves two aspects:
  - deciding how to implement the sequencing function
  - choosing a method of storing the main control function.
- Using a PLA to encode both the sequencing function as well as the main control
- The alternative form of implementation involves storing the control function in a read-only memory (ROM) and implementing the sequencing function separately



Microinstruction format

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.

	Write ALU	RegWrite,
		RegDst = 1,
		MemtoReg = 0
			Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.

	Write MDR	RegWrite,
		RegDst = 0,
		MemtoReg = 1
			Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.

Memory	Read PC	MemRead, lorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Maximally vs. Minimally Encoded

- No encoding:
  - 1 bit for each datapath operation
  - faster, requires more memory (logic)
  - used for Vax 780 — an astonishing 400K of memory!
- Lots of encoding:
  - send the microinstructions through logic to get control signals
  - uses less memory, slower
- Historical context of CISC:
  - Too much logic to put on a single chip with everything else
  - Use a ROM (or even RAM) to hold the microcode
  - It's easy to add new instructions

Microcode: Trade-offs

- Distinction between specification and implementation is sometimes blurred

- Specification Advantages:
  - Easy to design and write
  - Design architecture and microcode in parallel
- Implementation (off-chip ROM) Advantages
  - Easy to change since values are in memory
  - Can emulate other architectures
  - Can make use of internal registers
- Implementation Disadvantages, SLOWER now that:
  - Control is implemented on same chip as processor
  - ROM is no longer faster than RAM
  - No need to go back and make changes

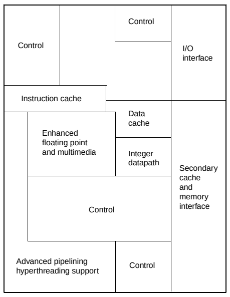
Historical Perspective

- In the '60s and '70s microprogramming was very important for implementing machines
- This led to more sophisticated ISAs and the VAX
- In the '80s RISC processors based on pipelining became popular
- Pipelining the microinstructions is also possible!
- Implementations of IA-32 architecture processors since 486 use:
  - "hardwired control" for simpler instructions
    (few cycles, FSM control implemented using PLA or random logic)
  - "microcoded control" for more complex instructions
    (large numbers of cycles, central control store)

- The IA-64 architecture uses a RISC-style ISA and can be implemented without a large central control store

Pentium 4

- Pipelining is important (last IA-32 without it was 80386 in 1985)

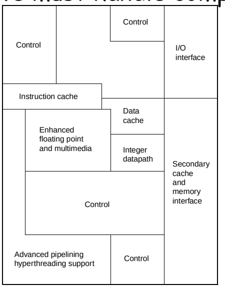

Control
Control
I/O interface
Instruction cache
Data cache
Enhanced floating point and multimedia
Integer datapath
Secondary cache and memory interface
Control
Advanced pipelining hyperthreading support
Control

- Pipelining is used for the simple instructions favored by compilers

"Simply put, a high performance implementation needs to ensure that the simple instructions execute quickly, and that the burden of the complexities of the instruction set penalize the complex, less frequently used, instructions"

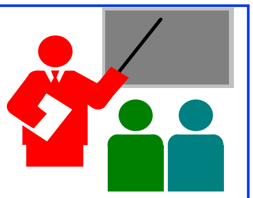
Pentium 4

- Somewhere in all that "control we must handle complex instructions


Control
Control
I/O interface
Instruction cache
Data cache
Enhanced floating point and multimedia
Integer datapath
Secondary cache and memory interface
Control
Advanced pipelining hyperthreading support
Control

- Processor executes simple microinstructions, 70 bits wide (hardwired)
- 120 control lines for integer datapath (400 for floating point)
- If an instruction requires more than 4 microinstructions to implement, control from microcode ROM (8000 microinstructions)
- Its complicated!

Summary



- If we understand the instructions…
  We can build a simple processor!
- If instructions take different amounts of time, multi-cycle is better
- Datapath implemented using:
  - Combinational logic for arithmetic
  - State holding elements to remember bits
- Control implemented using:
  - Combinational logic for single-cycle implementation
  - Finite state machine for multi-cycle implementation