

# **EECE 417 Computer Systems Architecture**

Department of Electrical and Computer Engineering  
Howard University

Charles Kim

Spring 2007

# **Computer Organization and Design (3<sup>rd</sup> Ed)**

**-The Hardware/Software Interface**

**by**

**David A. Patterson**

**John L. Hennessy**

# **Chapter Five**

## **The Processor: Datapath and Control**

### **Part C**

# Translation of Microprogramming to Hardware

- A specification methodology
  - appropriate if hundreds of opcodes, modes, cycles, etc.
  - signals specified symbolically using microinstructions
- Two additional tasks to translate the microprogram
  - assign addresses to the microinstructions and
  - fill in the contents of the dispatch ROMs.
- This process is essentially the same as the process of translating an assembly language program into machine instructions:
  - the fields of the assembly language or microprogram instruction are translated, and
  - labels on the instructions must be resolved to addresses.

# Microcode field to control signal (table)

- Each microcode field translates to a set of control signals to be set.
- This table specifies a value for each of the fields:
  - 22 different values of the fields specify all the required combinations of the 18 control lines.
  - Control lines that are not set which correspond to actions are 0 by default.
  - Multiplexor control lines are set to 0 if the output matters.
  - If a multiplexor control line is not explicitly set, its output is a don't care and is not used.

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 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, IRWrite	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using 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.

# Sequencing Fields and Dispatch Table

- The sequencing field can have four values:
  - **Fetch (meaning go to the Fetch state),**
  - **Dispatch 1,**
  - **Dispatch 2, and**
  - **Seq.**
- These four values are encoded to set the 2-bit address control

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

- **Fetch = 0, Dispatch 1 = 1, Dispatch 2 = 2, Seq = 3.**

# Dispatch Specification

- specify the contents of the dispatch tables
  - to relate the dispatch entries of the sequence field to the symbolic labels in the microprogram.
  - the dispatch tables.
    - **The first column in each table indicates the value of Op, which is the address used to access the dispatch ROM.**
    - **The second column shows the symbolic name of the opcode.**
    - **The third column indicates the value at that address in the ROM.**

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

# Microcode Assembler

- A microcode assembler would use:
  - the encoding of the sequencing field,
  - the contents of the symbolic dispatch tables in Figure C.5.2,

Microcode dispatch table 1		
Opcode field	Opcode name	Value
000000	R-format	Rformat1
000010	jmp	JUMP1
000100	beq	BEQ1
100011	lw	Mem1
101011	sw	Mem1

Microcode dispatch table 2		
Opcode field	Opcode name	Value
100011	lw	LW2
101011	sw	SW2

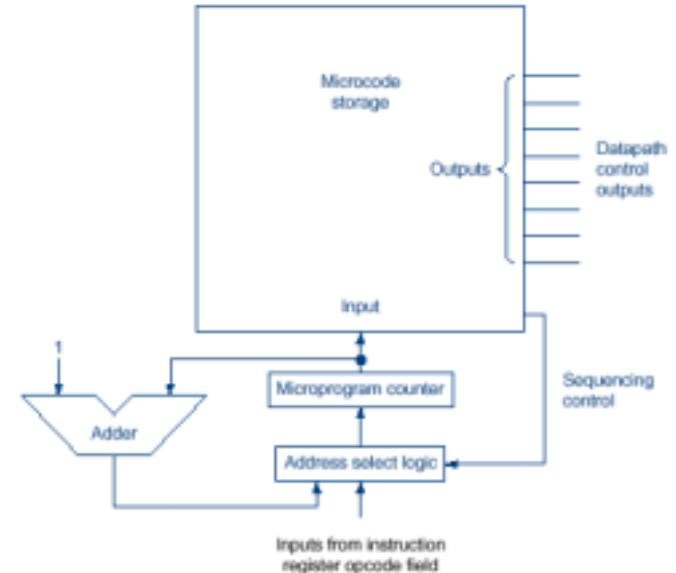
- the specification (in Microcode table)
- the actual microprogram

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

- to generate the microinstructions.

# Implementation of the Microprogram

- A typical implementation of a microcode controller:
  - an explicit incrementer to compute the default sequential next state and
  - would place the microcode in a read-only memory.
- The microinstructions are assembled directly from the microprogram.
- The microprogram counter, which replaces the state register of a finite state machine controller, determines how the next microinstruction is chosen.
- The address select logic contains the dispatch tables as well as the logic to select from among the alternative next states; the selection of the next microinstruction is controlled by the sequencing control outputs from the control logic.
- The combination of the current microprogram counter, incrementer, dispatch tables, and address select logic forms a sequencer that selects the next microinstruction.
- The microcode storage may consist either of read-only memory (ROM) or may be implemented by a PLA. PLAs may be more efficient in VLSI implementations, while ROMs may be easier to change.



# Exceptions

- **Hardest Part of Control (and Implementation)**
  - **Exception**
    - Unexpected event from within the processor
    - (ex) arithmetic overflow
  - **Interrupt**
    - An exception that comes from outside of the processor
    - (ex) I/O device seeking communication with the processor
- **Confusion over the terms**
  - **Most**
    - Interrupt=interrupt and/or exception
  - **MIPS convention**
    - Exception = cause is either internal or external
    - Interrupt = event is externally caused

# Exceptions and Our Interests in this chapter

Type of Event	Cause	Exp / Int
I/O Device Request	External	Interrupt
Invoke OS from User Program	Internal	Exception
Arithmetic Overflow	Internal	Exception
Using Undefined Instruction	Internal	Exception
Hardware Malfunctions	Either	Exp or Int

- **Our interests**
  - **Control Implementation for detecting two types of exceptions**
  - **Exceptions that arise from the portions of the instruction set we already have already discussed**
- **Importance of exceptions during the design of the control unit**
  - **Detection of exceptional conditions**
  - **Taking appropriate action**

# Exception Handling

- **Two types of exceptions**
  - Execution of undefined instruction
  - Arithmetic Overflow
- **Basic Action of the machine upon exception condition**
  - Save the offending instruction in EPC (exception program counter): 32-bit Register
  - Transfer control to OS at some specified address
- **OS actions**
  - **Initial action**
    - Providing some service to user program, or
    - Taking some predefined action in response to an over flow, or
    - Stopping the execution of the program and error reporting
  - **Next Action**
    - Terminate (or continue) execution using EPC (to determine where to restart the execution)

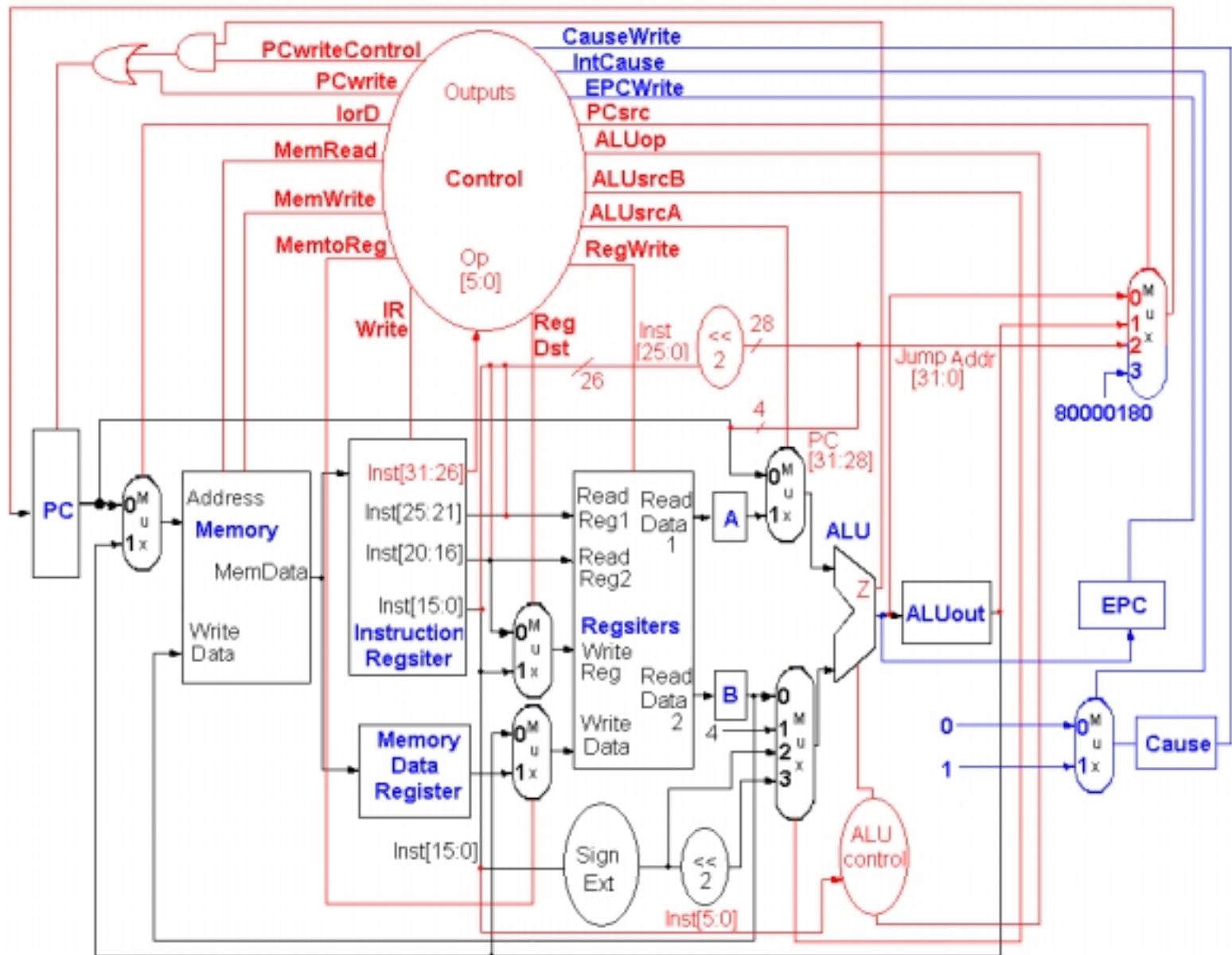
# Exception Control

- Exception Reason Finding
  - 2 main methods used to communicate the reason for exception
  - Cause Register (a status register) : MIPS approach
    - **A field indicates the reason for exception**
  - Vectored Interrupt
    - **Cause of exception determines the transfer address**
    - **(ex)**
      - **Undefined Instruction → 0000 0000**
      - **Arithmetic Overflow → 0000 0020**
    - **OS knows the cause of the exception by the address at which it is initiated**
    - **Addresses are separated by 32 bytes (or 8 instructions)**
- Exception Control (in MIPS) for our case
  - **EPC: 32-bit register which holds the addr of the affected instr**
  - **Cause: 32-bit register to record the cause of the exception**
    - **Since we have only two causes**
    - **Bit0=0 → undefined instruction**
    - **Bit0→1 Arithmetic Overflow**

# Control Signals for Exception Handling

- Additional for the datapath
  - 2 registers
    - **EPC**
    - **Cause**
  - Control Lines
    - **EPCWrite**
    - **CauseWrite**
    - **IntCause** : to set the low-order bit of the Cause register
  - Exception Address
    - **OS entry point for exception handling**
    - **0x8000 0180 (MIPS)**
    - **0x8000 0080 (MIPS Simulator)**
- Writing PC value to EPC
  - Since 4 is already added
  - 4 must be subtracted before writing into EPC

# Final Multicycle Datapath including Exception



# How Control checks for exceptions

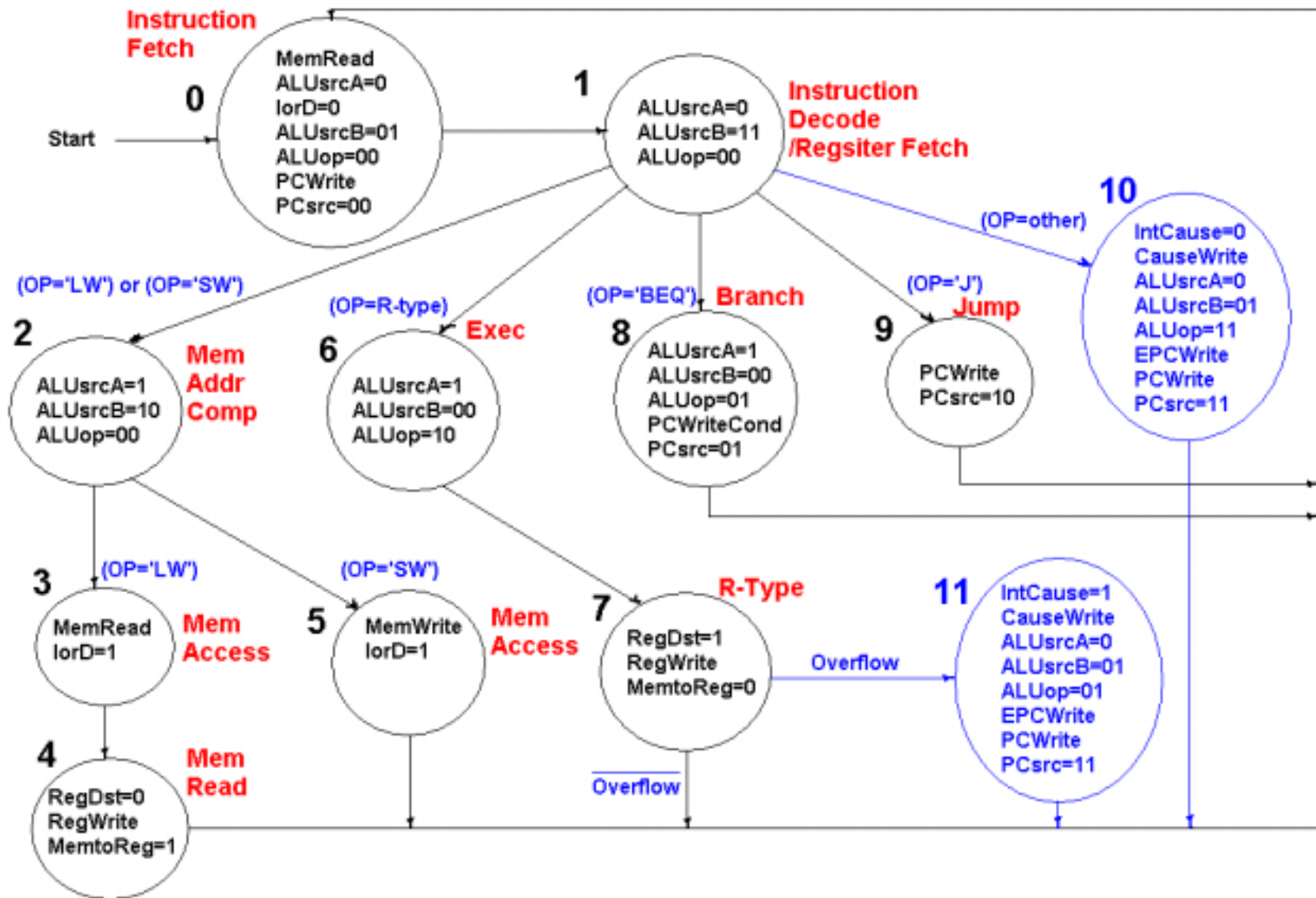
- **Detection of Undefined Instruction**

- When no next state is defined from state 1 for the **op** value.
- Define the next state value for all **op** other than `lw`, `sw`, `0` (R-Type), `j`, and `beq` as state10.

- **Detection of Arithmetic Overflow**

- ALU includes logic to detect overflow
- “Overflow” signal is provided from ALU
- State 11 is assigned to this exception as the next state from state 7

# Complete FSM including exceptions



# 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

# 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

# Chapter 5 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