

CS152
Computer Architecture and Engineering
Lecture 13: Microprogramming and Exceptions

March 3, 1995

Dave Patterson (patterson@cs) and
Shing Kong (shing.kong@eng.sun.com)

Slides available on <http://http.cs.berkeley.edu/~patterson>

cs 152 μ prog..1

©DAP & SIK 1995

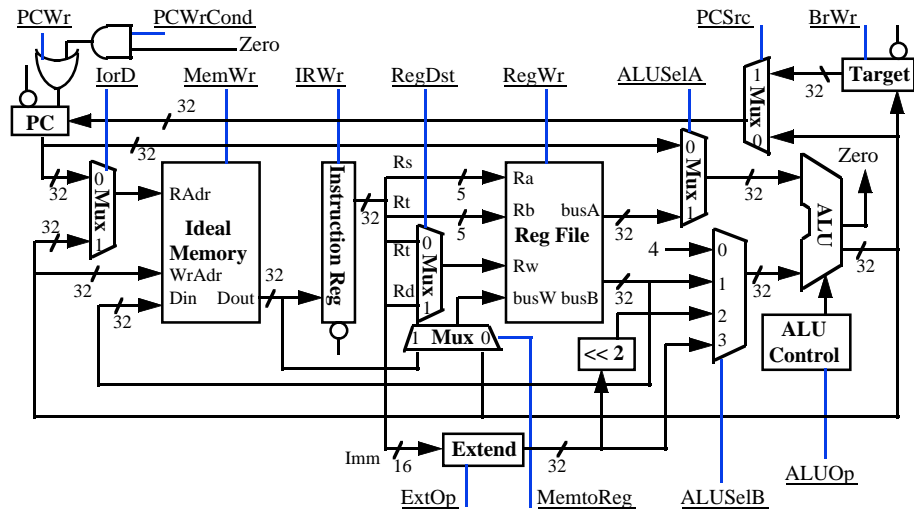
Review of a Multiple Cycle Implementation

- The root of the single cycle processor's problems:
 - The cycle time has to be long enough for the slowest instruction
- Solution:
 - Break the instruction into smaller steps
 - Execute each step (instead of the entire instruction) in one cycle
 - Cycle time: time it takes to execute the longest step
 - Keep all the steps to have similar length
 - This is the essence of the multiple cycle processor
- The advantages of the multiple cycle processor:
 - Cycle time is much shorter
 - Different instructions take different number of cycles to complete
 - Load takes five cycles
 - Jump only takes three cycles
 - Allows a functional unit to be used more than once per instruction

cs 152 μ prog..2

©DAP & SIK 1995

Review: Multiple Cycle Datapath

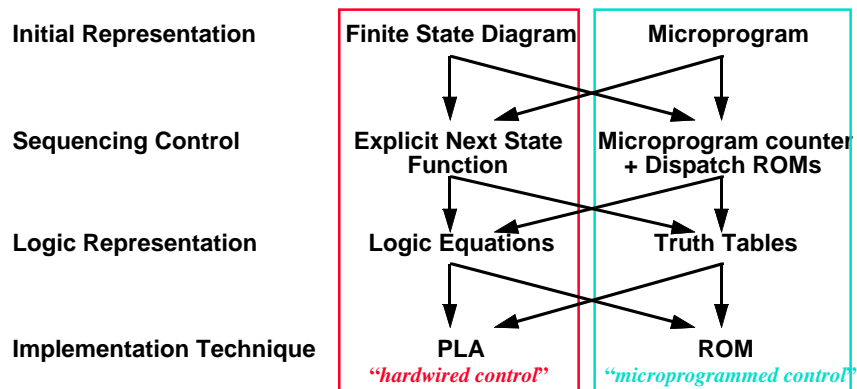


cs 152 μprog..3

©DAP & SIK 1995

Overview of the Two Lectures

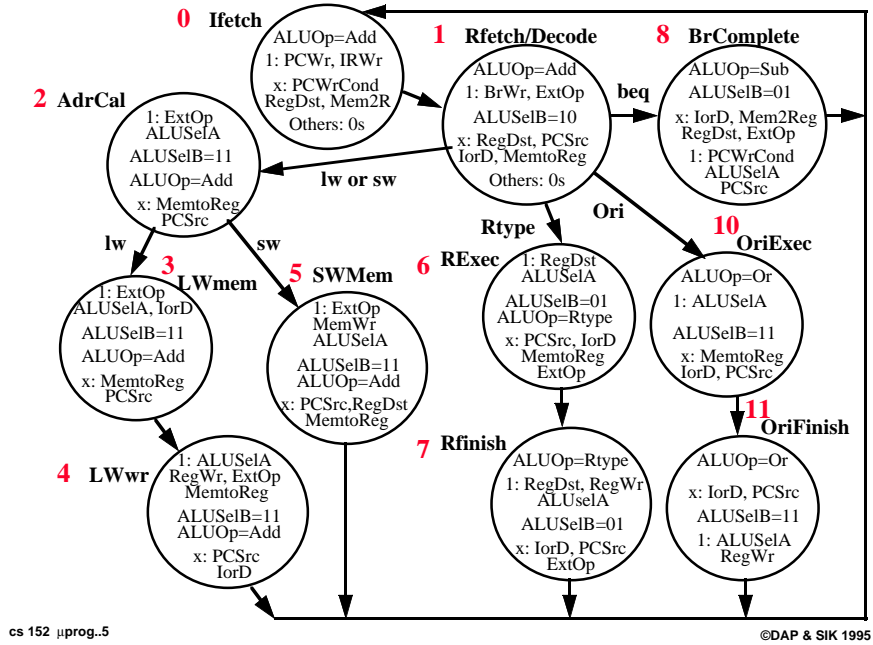
- Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.



cs 152 μprog..4

©DAP & SIK 1995

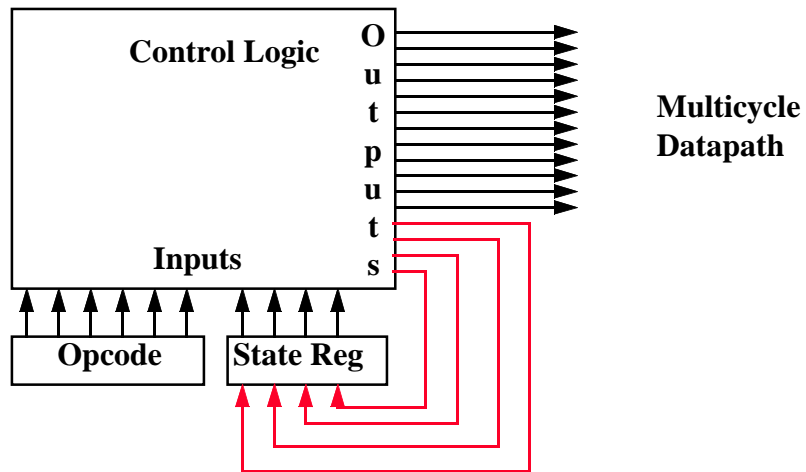
Initial Representation: Finite State Diagram



cs 152 µprog..5

©DAP & SIK 1995

Sequencing Control: Explicit Next State Function



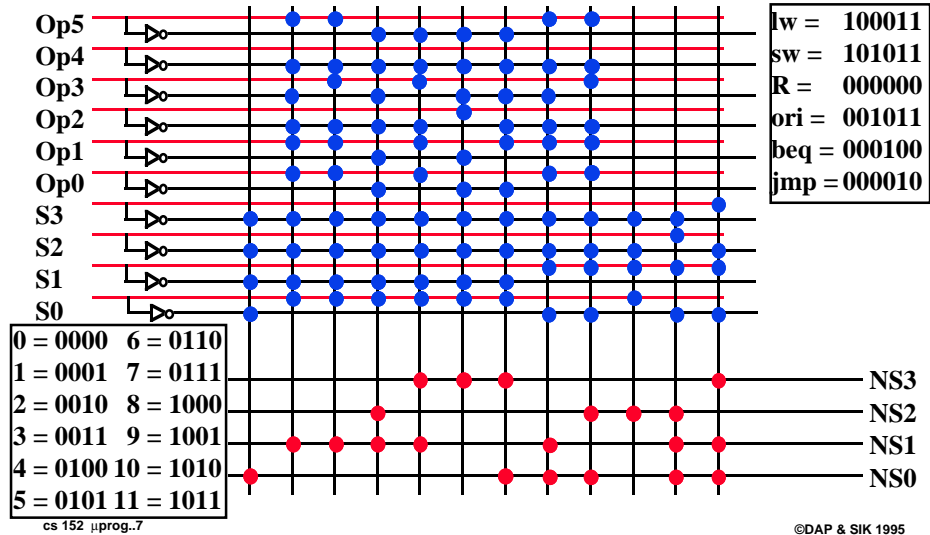
- Next state number is encoded just like datapath controls

cs 152 µprog..6

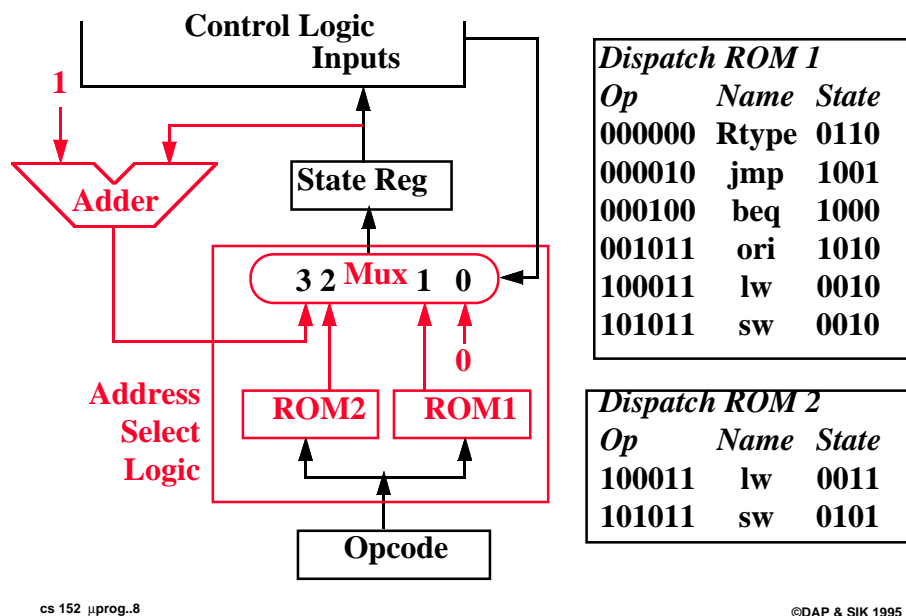
©DAP & SIK 1995

Implementation Technique: Programmed Logic Arrays

- Each output line the logical OR of logical AND of input lines or their complement: AND minterms specified in top AND plane, OR sums specified in bottom OR plane



Sequencer-based control unit details

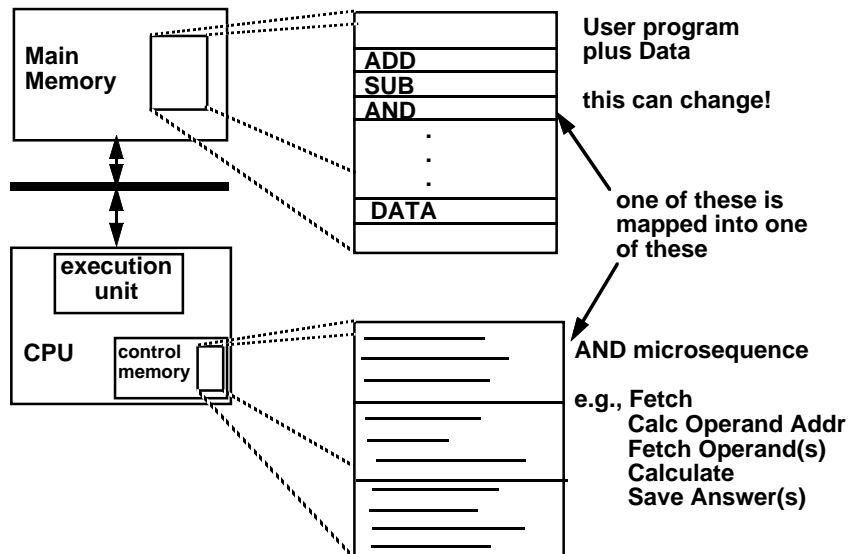


Implementing Control with a ROM

- Instead of a PLA, use a ROM with one word per state (“Control word”)

State number	Control Word Bits 18-2	Control Word Bits 1-0
0	10010100000001000	11
1	0000000010011000	01
2	0000000000010100	10
3	0011000000010100	11
4	00110010000010110	00
5	00101000000010100	00
6	0000000001000100	11
7	0000000001000111	00
8	0100000100100100	00
9	1000001000000000	00
10	...	11
11	...	00

Macroinstruction Interpretation



Variations on Microprogramming

- **Horizontal Microcode**

- control field for each control point in the machine



- **Vertical Microcode**

- compact microinstruction format for each class of microoperation

branch: μ seq-op μ addr

execute: ALU-op A,B,R

memory: mem-op S, D

Microprogramming Pros and Cons

- **Ease of design**
- **Flexibility**
 - Easy to adapt to changes in organization, timing, technology
 - Can make changes late in design cycle, or even in the field
- **Can implement very powerful instruction sets (just more control memory)**
- **Generality**
 - Can implement multiple instruction sets on same machine. (Emulation)
 - Can tailor instruction set to application.
- **Compatibility**
 - Many organizations, same instruction set
- **Costly to implement**
- **Slow**

Outline of Today's Lecture

- **Recap (5 minutes)**
- **Microinstruction Format Example (15 minutes)**
- **Questions and Administrative Matters (5 minutes)**
- **Do-it-yourself Microprogramming (25 minutes)**
- **Break (5 minutes)**
- **Exceptions (25 minutes)**

Designing a Microinstruction Set

- **Start with list of control signals**
- **Group signals together that make sense: called "fields"**
- **Places fields in some logical order (ALU operation & ALU operands first and microinstruction sequencing last)**
- **Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals**
- **To minimize the width, encode operations that will never be used at the same time**

Start with list of control signals, grouped into fields

<u>Signal name</u>	<u>Effect when deasserted</u>	<u>Effect when asserted</u>
ALUSelA	1st ALU operand = PC	1st ALU operand = Reg[rs]
RegWrite	None	Reg. is written
MemtoReg	Reg. write data input = ALU	Reg. write data input = memory
RegDst	Reg. dest. no. = rt	Reg. dest. no. = rd
TargetWrite	None	Target reg. = ALU
MemRead	None	Memory at address is read
MemWrite	None	Memory at address is written
lorD	Memory address = PC	Memory address = ALU
IRWrite	None	IR = Memory
PCWrite	None	PC = PCSource
PCWriteCond	None	IF ALUzero then PC = PCSource

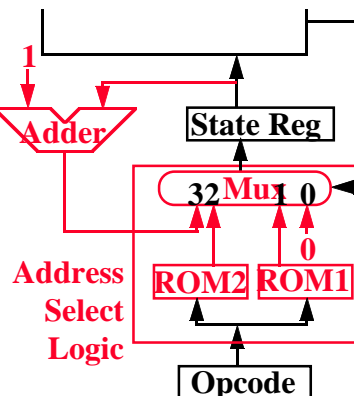
<u>Signal name</u>	<u>Value</u>	<u>Effect</u>
ALUOp	00	ALU adds
	01	ALU subtracts
	10	ALU does function code
	11	ALU does logical OR
ALUSelB	000	2nd ALU input = Reg[rt]
	001	2nd ALU input = 4
	010	2nd ALU input = sign extended IR[15-0]
	011	2nd ALU input = sign extended, shift left 2 IR[15-0]
	100	2nd ALU input = zero extended IR[15-0]
PCSource	00	PC = ALU
	01	PC = Target
	10	PC = PC+4[29-26] : IR[25-0] << 2

cs 152 μprog..15 ©DAP & SIK 1995

Start with list of control signals, cont'd

- For next state function (next microinstruction address), use Sequencer-based control unit from last lecture

<u>Signal</u>	<u>Value</u>	<u>Effect</u>
Sequen	00	Next μaddress = 0
-cing	01	Next μaddress = dispatch ROM 1
	10	Next μaddress = dispatch ROM 2
	11	Next μaddress = μaddress + 1



Microinstruction Format

<u>Field Name</u>	<u>Width</u>	<u>Control Signals Set</u>
ALU Control	2	ALUOp
SRC1	1	ALUSelA
SRC2	3	ALUSelB
ALU Destination	4	RegWrite, MemtoReg, RegDst, TargetWrite
Memory	3	MemRead, MemWrite, IorD
Memory Register	1	IRWrite
PCWrite Control	4	PCWrite, PCWriteCond, PCSource
Sequencing	2	AddrCtl
Total	20	

cs 152 μprog..17

©DAP & SIK 1995

Legend of Fields and Symbolic Names

<u>Field Name</u>	<u>Values for Field</u>	<u>Function of Field with Specific Value</u>
ALU	Add	ALU adds
	Func code	ALU subtracts
	Subt.	ALU does function code
SRC1	Or	ALU does logical OR
	PC	1st ALU input = PC
	rs	1st ALU input = Reg[rs]
SRC2	4	2nd ALU input = 4
	Extend	2nd ALU input = sign ext. IR[15-0]
	Extend0	2nd ALU input = zero ext. IR[15-0]
	Extshft	2nd ALU input = sign ex., sl IR[15-0]
ALU destination	rt	2nd ALU input = Reg[rt]
	Target	Target = ALU
Memory	rd	Reg[rd] = ALU
	Read PC	Read memory using PC
	Read ALU	Read memory using ALU output
Memory register	Write ALU	Write memory using ALU output
	IR	IR = Mem
	Write rt	Reg[rt] = Mem
PC write	Read rt	Mem = Reg[rt]
	ALU	PC = ALU output
	Target-cond.	IF ALU Zero then PC = Target
Sequencing	jump addr.	PC = PCSource
	Seq	Go to sequential μinstruction
	Fetch	Go to the first microinstruction
	Dispatch i	Dispatch using ROMi (1 or 2)

cs 152 μprog..18

©DAP & SIK 1995

Microprogram it yourself!

<u>Label</u>	<u>ALU</u>	<u>SRC1</u>	<u>SRC2</u>	<u>ALU Dest.</u>	<u>Memory</u>	<u>Mem. Req.</u>	<u>PC Write</u>	<u>Sequencing</u>
Fetch	Add	PC	4		Read PC	IR	ALU	Seq

Questions and Administrative Matters

- “Midterm” for instructors and TAs: constructive criticism by Friday
 - Please put your name, as I want to hear from everyone
 - If you want to submit an anonymous form, just take a second copy
 - Be careful what you wish for, it may come true
 - Return in class today, right after 5 minute break (take another if don't have one)
- Lecture next Wednesday March 8 is moved from 306 Soda to ?? because of a conference that day
- Lab 4 progress report in discussion section March 8-10; project also due in discussion sections March 15 to 17; everyone needs to be there for both meetings
- Read the course newsgroup to keep uptodate on latest news

Break (5 Minutes)

- Turn in class surveys!

Exceptions and Interrupts

- Control is hardest part of the design
- Hardest part of control is exceptions and interrupts
 - events other than branches or jumps that change the normal flow of instruction execution
 - exception is an unexpected event from within the processor; e.g., arithmetic overflow
 - interrupt is an unexpected event from outside the processor; e.g., I/O
- MIPS convention: exception means any unexpected change in control flow, without distinguishing internal or external; use the term interrupt only when the event is externally caused.

<u>Type of event</u>	<u>From where?</u>	<u>MIPS terminology</u>
I/O device request	External	Interrupt
Invoke OS from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or Interrupt

How are Exceptions Handled?

- Machine must save the address of the offending instruction in the EPC (exception program counter)
- Then transfer control to the OS at some specified address
 - OS performs some action in response, then terminates or returns using EPC
- 2 types of exceptions in our current implementation: undefined instruction and an arithmetic overflow
- Which Event caused Exception?
 - Option 1 (used by MIPS): a Cause register contains reason
 - Option 2 Vectored interrupts: address determines cause.
 - addresses separated by 32 instructions
 - E.g.,

<u>Exception Type</u>	<u>Exception Vector Address (in Binary)</u>
Undefined instruction	01000000 00000000 00000000 00000000
Arithmetic overflow	01000000 00000000 00000000 01000000

cs 152 μprog..24

©DAP & SIK 1995

Additions to MIPS ISA to support Exceptions

- EPC—a 32-bit register used to hold the address of the affected instruction.
- Cause—a register used to record the cause of the exception. In the MIPS architecture this register is 32 bits, though some bits are currently unused. Assume that the low-order bit of this register encodes the two possible exception sources mentioned above: undefined instruction=0 and arithmetic overflow=1.
- 2 control signals to write EPC and Cause
- Be able to write exception address into PC, increase mux to add as input 01000000 00000000 00000000 00000000
- May have to undo $PC = PC + 4$, since want EPC to point to offending instruction (not its successor); $PC = PC - 4$

cs 152 μprog..25

©DAP & SIK 1995

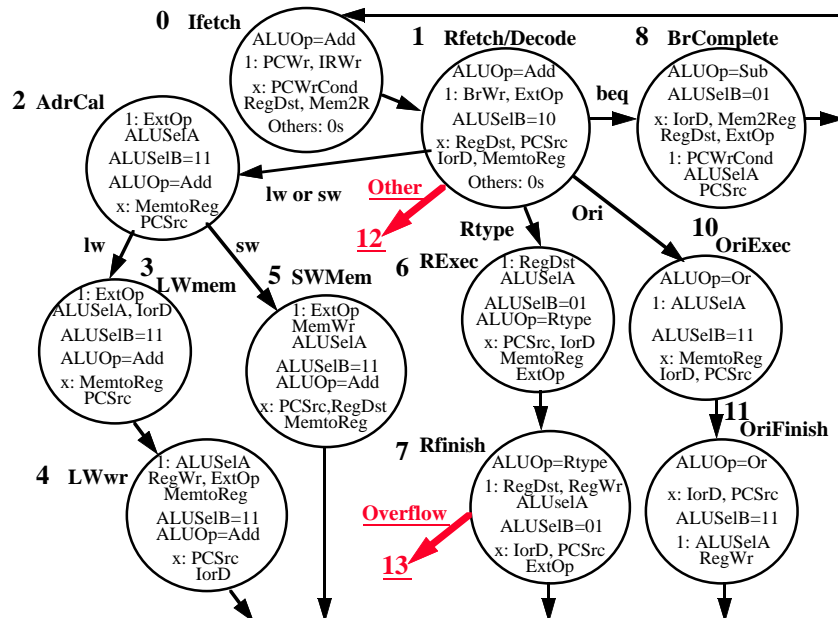
How Control Detects Exceptions

- **Undefined Instruction**—detected when no next state is defined from state 1 for the op value.
 - We handle this exception by defining the next state value for all op values other than lw, sw, 0 (R-type), jmp, beq, and ori as new state 12.
 - Shown symbolically using “other” to indicate that the op field does not match any of the opcodes that label arcs out of state 1.
- **Arithmetic overflow**—Chapter 4 included logic in the ALU to detect overflow, and a signal called Overflow is provided as an output from the ALU. This signal is used in the modified finite state machine to specify an additional possible next state for state 7
- **Note: Challenge in designing control of a real machine is to handle different interactions between instructions and other exception-causing events such that control logic remains small and fast.**
 - Complex interactions makes the control unit the most challenging aspect of hardware design

cs 152 µprog..26

©DAP & SIK 1995

Changes to Finite State Diagram to Detect Exceptions



cs 152 µprog..27

©DAP & SIK 1995

Summary

- **Control is hard part of computer design**
- **Microprogramming specifies control like assembly language programming instead of finite state diagram**
- **Next State function, Logic representation, and implementation technique can be the same as finite state diagram, and vice versa**
- **Exceptions are the hard part of control**
- **Need to find convenient place to detect exceptions and to branch to state or microinstruction that saves PC and invokes the operating system**
- **As we get pipelined CPUs that support page faults on memory accesses which means that the instruction cannot complete AND you must be able to restart the program at exactly the instruction with the exception, it gets even harder**