Microprogramming

Alternative way of specifying control

FSM

- State -- bubble
- control signals in bubble
- next state given by signals on arc
- not a great language to specify when things are complex

Treat as a programming problem

Microprogramming

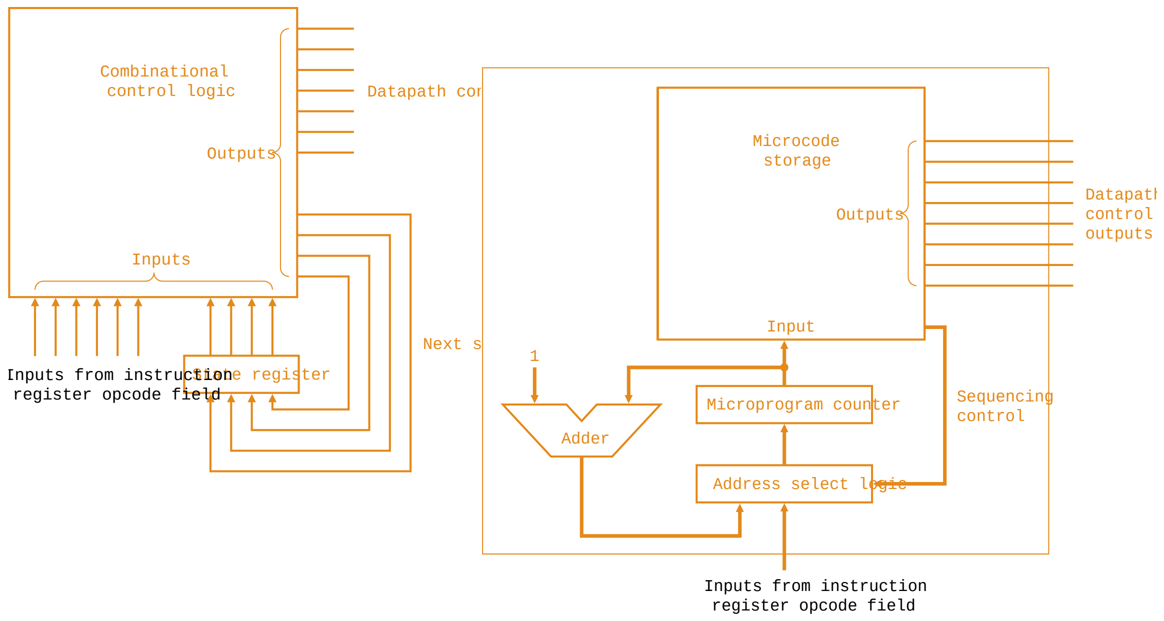
Datapath remains the same

Only control is specified differently but does the same

Each cycle specify required control signal via microprogram field

label	**alu**	**src1**	**src2**	**reg**	**memory**	**pcwrite**	**next?**
fetch	add	pc	4	read pc	alu	alu	+1
	add	pc	extshft	read			dispatch 1
mem1	add	A	extend				dispatch 2
lw2					read alu		+1
				write mdr			fetch

FSM (left) & Microprogramming (right)



Potential Benefits of Microprogramming

More disciplined control logic - easier to debug

Enables family of machines with same ISA (IBM 360/370)

Enables more complex instruction set

Writable control-store allows in-the-field fixes

But in the 1990s:

CAD tools and PLAs offer similar discipline

Caches make memory almost as fast as control store

Complex ISA - hardwired+micro-ops (e.g., Pentium Pro)

State of the Art

Specify control

- FSM - does not scale
- microprogram - works
- vhdl/verilog - preferred

Implement control

- random logic - only if CAD tools generate
- PLAs - mostly generated by CAD tools
- Control store + update - why accept this contraint?

State of the Art

Specify control in verilog/vhdl

CAD compile to PLA, but could used ROM or RAM

Microprogramming implementation seems dead

- because it unnecessarily constrains CAD’s targets

But what if technology makes control store faster than caches?

Horizontal vs. Vertical microcode

Horizontal

- fewer and wider micro-instructions
- less encoding
- larger control store - may waste space (control lines)

Vertical

- more and narrower micro-instructions
- dense encoding
- smaller control store - but may need more steps

Exceptions: Background

What happens:

- instruction fetch page fault
- illegal opcode
- privileged opcode
- arithmetic overflow
- data page fault
- I/O device statuc change
- power-on/reset

Exceptions: Background

For some, we could test for the condition

- arithmetic overflow
- I/O device ready

But most tests for other conditions uselessly say “no”

Solution: Generate “surprise procedure calls” called exception

Exceptions: Big Picture

Interrupt (asynchronous) or trap (synchronous) triggers exception

Hardware handles initial reaction

Then invokes a software exception handler

Exceptions: Hardware

- Sets state giving cause of exception
- (MIPS: in exception_code field of Cause register -
    - a coprocessor 0 register)
- Changes to Kernel mode for dangerous work ahead
- Disables interrupts (to prevent infinite looping)
- (MIPS: both the above in Status register -
    - another coprocessor 0 register)
- saves current PC (MIPS: exception PC (EPC)
- jumps to specific address (MIPS: PC = 0x80000080)
- (like a surprise jal - so can't clobber $31)

Exceptions: Software

- Exception handler (MIPS: .ktext beginning at 0x80000080)
- Set flag to detect incorrect entry -exception while in handler
- Save some registers
- Find exception type (MIPS: exception_code in Cause reg)
- E.g., I/O interrupt or syscall
- Jump to specific exception handler ...

Exceptions: Software, cont.

- Handle specific exception
- Jump to clean-up to resume user program
- restore registers
- Reset flag that detects incorrect entry
- Atomically
    - restore previous mode
    - enable interrupts
    - jump back to program (using EPC)

Implementing Exceptions

We worry only about hardware, not software handler

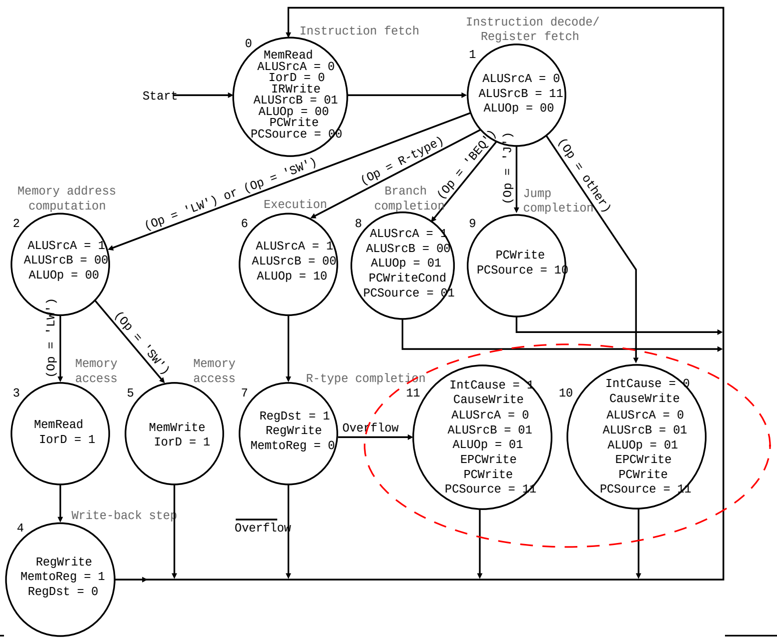
IntCause

- 0 undef instruction
- 1 arithmetic overflow

Changes to the datapath

New states in FSM to deal with exceptions

FSM with Exceptions (Fig. 5.50)



Implementing Exceptions

New arcs in the FSM just like regular arcs

FSM more complex if must add many arcs

Critical path may be worsened

Alternative: vectored interrupts

- PC = base + f(Cause)
- e.g., PC = 0x80 + IntCause << 7 # 32 instructions

+ faster

– more hardware, more space

Review

Type	Control	Datapath	Time (CPI, cycle time)
Single-cycle	comb + end update	No reuse	1 cycle, (imem + reg + ALU + dmem)
Multi-cycle	comb + FSM update	Reuse	[3,5] cycles, Max(imem, reg, ALU, dmem)
We want	?	?	~1 cycle, Max(imem , reg, ALU, dmem)

We will use pipelining (lunch buffet!) to achieve last row