ECE/CS 552: Microprogramming and Exceptions

Instructor: Mikko H Lipasti

Fall 2010
University of Wisconsin-Madison

Lecture notes based on set created by Mark Hill.

Microprogramming

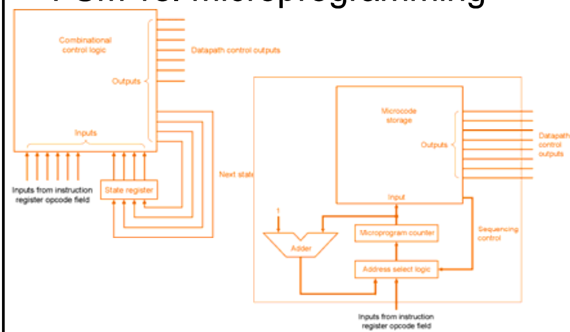
- Alternative way of specifying control
- FSM
  - State – bubble
  - Control signals in bubble
  - Next state given by signals on arc
  - Not a great language for specifying complex events
- Instead, treat as a programming problem

Microprogramming

- Datapath remains the same
- Control is specified differently but does the same
- Each cycle a microprogram field specifies required control signals

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 vs. Microprogramming



Benefits of Microprogramming

- More disciplined control logic
  - Easier to debug
- Enables family of machines with same ISA
- Enables more complex ISA (benefit?)
- Writeable control store allows late fixes
- But, in the late 1980's
  - CAD tools and PLAs offer similar discipline
  - Caches make memory almost as fast as control store

State of the Art

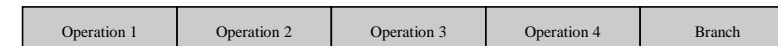
- Specify control
  - FSM – does not scale easily
  - Microprogram – works
  - VHDL/Verilog – preferred
- Specify control in VHDL/Verilog
  - CAD compile to PLA
  - Could use ROM or RAM

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

Intellectual Heritage

- Microprogramming seems dead
- But what if technology shifts:
  - Control store is faster than caches?
- Also, "Very Long Instruction Word" or VLIW
  - Origins in microcode optimization research at Yale
    - Josh Fisher, VLIW startup company: Multiflow Trace
    - Now used in Transmeta Crusoe, Intel IA-64, TI DSP family
  - Explicitly Parallel Instruction-set Computing (EPIC)
    - Microcode specifies parallel hardware operations
    - Can generalize to express any parallel computation
    - Simple hardware (like microcode engine); complex software



Exceptions

- What happens?
  - Instruction fetch page fault
  - Illegal opcode
  - Privileged opcode
  - Arithmetic overflow
  - Data page fault
  - I/O device status change
  - Power-on/reset

Exceptions

- For some, we could test for the condition
  - Arithmetic overflow
  - I/O device ready (polling)
- But most tests uselessly say "no"
- Solution:
  - Surprise "procedure call"
  - Called an exception

Exceptions: Big Picture

- Two types:
  - Interrupt (asynchronous) or
  - Trap (synchronous)
- Hardware handles initial reaction
- Then invokes a software exception handler
  - By convention, at e.g. 0xC00
  - O/S kernel provides code at the handler address

Exceptions: Hardware

- Sets state that identifies cause of exception
  - MIPS: in exception_code field of Cause register
- Changes to kernel mode for dangerous work ahead
- Disables interrupts
  - MIPS: recorded in status register
- Saves current PC (MIPS: exception PC)
- Jumps to specific address (MIPS: 0x80000080)
  - Like a surprise JAL – so can't clobber $31

Exceptions: Software

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

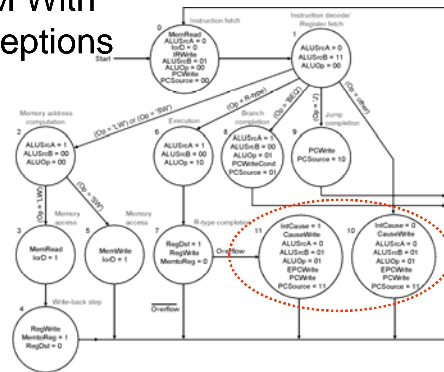
Exceptions: Software, cont'd

- Handle specific exception
- Jump to clean-up to resume user program
- Restore registers
- Reset flag that detects incorrect entry
- Atomically
  - Restore previous mode (user vs. supervisor)
  - Enable interrupts
  - Jump back to program (using EPC)

Implementing Exceptions

- We worry only about hardware, not s/w
- IntCause
  - 0 undefined instruction
  - 1 arithmetic overflow
- Changes to the datapath
  - Detect exception
  - Additional source for next PC
  - Storage for exception cause, return address, spare register
- New states in control FSM

FSM With Exceptions



Implementing Exceptions

- New arcs in FSM just like regular arcs
- FSM more complex if must add many arcs
- Critical path may get worse
- Alternative: vectored interrupts
  - PC = base = f(cause)
  - E.g. PC = 0x80 + intcause << 7 # 32 instrs
  - Faster
  - More hardware, more space

Review

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

- We will use *pipelining* to achieve last row