Recap: Microprogramming

- Microprogramming is a convenient method for implementing structured control state diagrams:
  - Random logic replaced by microPC sequencer and ROM
  - Each line of ROM called a μ-instruction: contains sequencer control + values for control points
  - Limited state transitions: branch to zero, next sequential, branch to μ-instruction address from dispatch ROM

- Horizontal μ-Code: one control bit in μ-instruction for every control line in datapath
- Vertical μ-Code: groups of control-lines coded together in μ-instruction (e.g., possible ALU dest)

- Control design reduces to Microprogramming
  - Part of the design process is to develop a “language” that describes control and is easy for humans to understand

Exceptions

- Exception = unprogrammed control transfer
  - System takes action to handle the exception
    - Must record the address of the offending instruction
    - Record any other information necessary to return afterwards
  - Returns control to user
  - Must save & restore user state

- Allows construction of a “user virtual machine”
**Two Types of Exceptions: Interrupts and Traps**

° **Interrupts**
  - caused by external events:
    - Network, Keyboard, Disk I/O, Timer
  - asynchronous to program execution
    - Most interrupts can be disabled for brief periods of time
    - Some (like “Power Falling”) are non-maskable (NMI)
  - may be handled between instructions
  - simply suspend and resume user program

° **Traps**
  - caused by internal events
    - exceptional conditions (overflow)
    - errors (parity)
    - faults (non-resident page)
  - synchronous to program execution
    - condition must be remedied by the handler
    - instruction may be retried or simulated and program continued or program may be aborted

° **Precise Interrupts**
  - Precise ⇒ state of the machine is preserved as if program executed up to the offending instruction
    - All previous instructions completed
    - Offending instruction and all following instructions act as if they have not even started
    - Same system code will work on different implementations
    - Position clearly established by IBM
    - Difficult in the presence of pipelining, out-of-order execution, ...
    - MIPS takes this position

° **Imprecise** ⇒ system software has to figure out what is where and put it all back together

° Performance goals often lead designers to forsake precise interrupts
  - system software developers, user, markets etc. usually wish they had not done this

° Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

**Big Picture: user / system modes**

° By providing two modes of execution (user/system) it is possible for the computer to manage itself
  - operating system is a special program that runs in the privileged mode and has access to all of the resources of the computer
  - presents “virtual resources” to each user that are more convenient that the physical resources
    - files vs. disk sectors
    - virtual memory vs physical memory
  - protects each user program from others
  - protects system from malicious users.
  - OS is assumed to “know best”, and is trusted code, so enter system mode on exception.

° Exceptions allow the system to taken action in response to events that occur while user program is executing:
  - Might provide supplemental behavior (dealing with denormal floating-point numbers for instance).
  - “Unimplemented instruction” used to emulate instructions that were not included in hardware (i.e. MicroVax)

° **Addressing the Exception Handler**

° **Traditional Approach: Interrupt Vector**
  - PC <- MEM[ IV_base + cause || 00]
  - 370, 68000, Vax, 80x86, ...

° **RISC Handler Table**
  - PC <- IT_base + cause || 0000
  - saves state and jumps
  - Sparc, PA, M68K, ...

° **MIPS Approach: fixed entry**
  - PC <- EXC_addr
  - Actually very small table
    - RESET entry
    - TLB
    - other

iv_base
cause
handler
code

iv_base
cause
handler entry code
Saving State

- Push it onto the stack
  - Vax, 68k, 80x86
- Shadow Registers
  - M88k
  - Save state in a shadow of the internal pipeline registers
- Save it in special registers
  - MIPS EPC, BadVaddr, Status, Cause

Additions to MIPS ISA to support Exceptions

- Exception state is kept in "coprocessor 0":
  - Use mfc0 read contents of these registers
  - Every register is 32 bits, but may be only partially defined

  - BadVAddr (register 8)
    - register contained memory address at which memory reference occurred
  - Status (register 12)
    - interrupt mask and enable bits
  - Cause (register 13)
    - the cause of the exception
    - Bits 5 to 2 of this register encodes the exception type (e.g. undefined instruction=10 and arithmetic overflow=12)
  - EPC (register 14)
    - address of the affected instruction (register 14 of coprocessor 0).

- Control signals to write BadVAddr, Status, Cause, and EPC
- Be able to write exception address into PC (8000 0080 hex)
- May have to undo PC = PC + 4, since want EPC to point to offending instruction (not its successor): PC = PC - 4

Details of Status register

<table>
<thead>
<tr>
<th>Status</th>
<th>Mask</th>
<th>old</th>
<th>prev</th>
<th>current</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Mask = 1 bit for each of 5 hardware and 3 software interrupt levels
  - 1 => enables interrupts
  - 0 => disables interrupts
- k = kernel/user
  - 0 => was in the kernel when interrupt occurred
  - 1 => was running user mode
- e = interrupt enable
  - 0 => interrupts were disabled
  - 1 => interrupts were enabled
- When interrupt occurs, 6 LSB shifted left 2 bits, setting 2 LSB to 0
  - run in kernel mode with interrupts disabled

Details of Cause register

<table>
<thead>
<tr>
<th>Status</th>
<th>15</th>
<th>10</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
</table>
| Pending | 5   | hardware levels: bit set if interrupt occurs but not yet serviced
- handles cases when more than one interrupt occurs at same time, or while records interrupt requests when interrupts disabled
| Code    |     |     |    |    |
| Exception Code encodes reasons for interrupt
- 0 (INT) => external interrupt
- 4 (ADDRL) => address error exception (load or instr fetch)
- 5 (ADDRS) => address error exception (store)
- 6 (IBUS) => bus error on instruction fetch
- 7 (DBUS) => bus error on data fetch
- 8 (Syscall) => Syscall exception
- 9 (BKPT) => Breakpoint exception
- 10 (RI) => Reserved Instruction exception
- 12 (OVF) => Arithmetic overflow exception
### Part of the handler in trap_handler.s

```asm
.entry:  ; Exceptions/interrupts come here

.set noat
.move $k1 $at  ; $at = Saved $at
.set at
.sw  $v0 $a1  ; $v0 = Saved $a1
.mcfs $k0 $t3  ; Cause
              ; $t3 = Grab the cause register
.li  $v0 4    ; syscall 4 (print_str)
    la $a0 __m1_syscall
.li  $v0 1    ; syscall 1 (print_int)
    srl $a0 $k0 2  ; Shift Cause reg
    mfc0 $k1 $t3  ; Cause

.set noat
.move $at $k1  ; Restore $at
.set at
.rfe         ; Return from exception handler
.addiu $k0 $k0 4  ; Return to next instruction
.jr  $k0
```

### Example: How Control Handles Traps in our FSD

- **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**—detected on ALU ops such as signed add
  - Used to save PC and enter exception handler

- **External Interrupt**—flagged by asserted interrupt line
  - Again, must save PC and enter exception handler

- **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.

### How add traps and interrupts to state diagram?

The diagram illustrates how traps and interrupts are added to the state diagram of our FSD, with specific transitions for *undefined* and *overflow* conditions, as well as a new branch at micro-code level.

### But: What has to change in our µ-sequence?

- **Need concept of** branch **at micro-code level**

**Example: Can easily use with for non-ideal memory**

- `IR <= MEM[PC]` ("instruction fetch")
- `~wait`
- `A <= R[rs]`
- `B <= R[rt]`
- `S <= A fun B`
- `R[rd] <= S`
- `PC <= PC + 4`

- `S <= A or ZX`
- `R[rt] <= S`
- `PC <= PC + 4`

- `S <= A + SX`
- `M <= MEM[S]`
- `MEM[S] <= B`
- `~wait`

- `SW`
  - `MEM[S] <= B`
  - `~wait`
  - `wait`

- `LW`
  - `S <= A or ZX`
  - `M <= MEM[S]`
  - `MEM[S] <= B`

- `BEQ`
  - `PC <= Next(PC)`

- `ORi`
  - `S <= A + SX`
  - `R[rt] <= M`
  - `PC <= PC + 4`

- `S <= A + SX`

- `SW`

**Administrative Issues**

- **Midterm I: Not graded yet!**
  - Sorry!

- **Get started reading Chapter 6**
  - Complete chapter on Pipelining...

- **Sections:**
  - Monday (10/15) sections ⇒ Cory 119.
    - You will demonstrate your processors on a mystery program
    - Report still due at midnight
  - Following week ⇒ Usual place

- **Lab Reports:**
  - Up to you to do a good job of summarizing your work
  - Part of grade will be on quality of your writing
    - Put code and schematics in appendices appropriately referenced
    - Use actual wordprocessor (Microsoft Word online)

**Question #1: Why do microcoding?**

- If simple instruction could execute at very high clock rate...
- If you could even write compilers to produce microinstructions...
- If most programs use simple instructions and addressing modes...
- If microcode is kept in RAM instead of ROM so as to fix bugs ...
- If same memory used for control memory could be used instead as cache for “macroinstructions”...
- Then why not skip instruction interpretation by a microprogram and simply compile directly into lowest language of machine? (microprogramming is overkill when ISA matches datapath 1-1)

**Recall: Performance Evaluation**

- **What is the average CPI?**
  - state diagram gives CPI for each instruction type
  - workload gives frequency of each type

<table>
<thead>
<tr>
<th>Type</th>
<th>CPI, for type</th>
<th>Frequency</th>
<th>CPI, x freq,</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>4</td>
<td>40%</td>
<td>1.6</td>
</tr>
<tr>
<td>Load</td>
<td>5</td>
<td>30%</td>
<td>1.5</td>
</tr>
<tr>
<td>Store</td>
<td>4</td>
<td>10%</td>
<td>0.4</td>
</tr>
<tr>
<td>branch</td>
<td>3</td>
<td>20%</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Average CPI: 4.1
Question #2: Can we get CPI < 4.1?

- Seems to be lots of “idle” hardware
  - Why not overlap instructions???

The Big Picture: Where are We Now?
- The Five Classic Components of a Computer
  - Processor
    - Control
  - Memory
  - Input
  - Output

- Next Topics:
  - Pipelining by Analogy
  - Pipeline hazards

Pipelining is Natural!
- Laundry Example
  - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes

Sequential Laundry
- Sequential laundry takes 6 hours for 4 loads
  - If they learned pipelining, how long would laundry take?
**Pipelined Laundry: Start work ASAP**

- Pipelined laundry takes 3.5 hours for 4 loads

**Pipelining Lessons**

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced speedup = Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
- Stall for Dependences

**The Five Stages of Load**

- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

**Note: These 5 stages were there all along!**
Pipelining

- Improve performance by increasing throughput

Ideal speedup is number of stages in the pipeline. Do we achieve this?

Basic Idea

- What do we need to add to split the datapath into stages?

Graphically Representing Pipelines

- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

Conventional Pipelined Execution Representation
**Single Cycle, Multiple Cycle, vs. Pipeline**

**Single Cycle Implementation:**
- Load
- Store
- Waste

**Multiple Cycle Implementation:**
- Cycle 1
- Cycle 2
- Cycle 3
- Cycle 4
- Cycle 5
- Cycle 6
- Cycle 7
- Cycle 8
- Cycle 9
- Cycle 10

**Pipeline Implementation:**
- Load
- Store
- Ifetch
- Reg
- Exec
- Mem
- Wr

---

**Why Pipeline?**

- Suppose we execute 100 instructions
  - Single Cycle Machine
    - \(45 \text{ ns/cycle} \times 1 \text{ CPI} \times 100 \text{ inst} = 4500 \text{ ns}\)
  - Multicycle Machine
    - \(10 \text{ ns/cycle} \times 4.6 \text{ CPI} \text{ (due to inst mix)} \times 100 \text{ inst} = 4600 \text{ ns}\)
  - Ideal pipelined machine
    - \(10 \text{ ns/cycle} \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle drain}) = 1040 \text{ ns}\)

---

**Can pipelining get us into trouble?**

- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource two different ways at the same time
    - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
  - control hazards: attempt to make a decision before condition is evaluated
    - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in branch instructions
  - data hazards: attempt to use item before it is ready
    - E.g., one sock of pair in dryer and one in washer; can’t fold until get sock from washer through dryer
    - instruction depends on result of prior instruction still in the pipeline

- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards
Single Memory is a Structural Hazard

Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazards limit performance

- Example: if 1.3 memory accesses per instruction and only one memory access per cycle then
  - average CPI ≥ 1.3
  - otherwise resource is more than 100% utilized

Control Hazard Solution #1: Stall

- Stall: wait until decision is clear
- Impact: 2 lost cycles (i.e. 3 clock cycles per branch instruction) => slow
- Move decision to end of decode
  - save 1 cycle per branch

Control Hazard Solution #2: Predict

- Predict: guess one direction then back up if wrong
- Impact: 0 lost cycles per branch instruction if right, 1 if wrong (right - 50% of time)
  - Need to “Squash” and restart following instruction if wrong
  - Produce CPI on branch of \( (1 \times .5 + 2 \times .5) = 1.5 \)
  - Total CPI might then be: \( 1.5 \times .2 + 1 \times .8 = 1.1 \) (20% branch)
- More dynamic scheme: history of 1 branch (- 90%)
Control Hazard Solution #3:Delayed Branch

° **Delayed Branch:** Redefine branch behavior (takes place after next instruction)
° **Impact:** 0 clock cycles per branch instruction if can find instruction to put in “slot” (-50% of time)
° As launch more instruction per clock cycle, less useful

---

Data Hazard on r1:

• Dependencies backwards in time are hazards

Data Hazard Solution:

• “Forward” result from one stage to another

---
Forwarding (or Bypassing): What about Loads?

- Dependencies backwards in time are hazards

Forwarding (or Bypassing): What about Loads

- Dependencies backwards in time are hazards

Designing a Pipelined Processor

- Go back and examine your datapath and control diagram
- associated resources with states
- ensure that flows do not conflict, or figure out how to resolve
- assert control in appropriate stage

Control and Datapath: Split state diag into 5 pieces

```
IR ← Mem[PC]; PC ← PC+4;
A ← R[rs]; B← R[rt]
S ← A + B;
S ← A or ZX;
R[rd] ← S;
S ← A + SX;
M ← Mem[S]
R[rt] ← S;
Mem[S] ← B
R[rd] ← S;
```

```
Next PC Inst. Mem IR Reg File Exec
A B D M
Mem Access Data Mem
```
Summary: Pipelining
° Reduce CPI by overlapping many instructions
  • Average throughput of approximately 1 CPI with fast clock
° Utilize capabilities of the Datapath
  • start next instruction while working on the current one
  • limited by length of longest stage (plus fill/flush)
  • detect and resolve hazards
° What makes it easy
  • all instructions are the same length
  • just a few instruction formats
  • memory operands appear only in loads and stores
° What makes it hard?
  • structural hazards: suppose we had only one memory
  • control hazards: need to worry about branch instructions
  • data hazards: an instruction depends on a previous instruction

Summary: Where this class is going
° We’ll build a simple pipeline and look at these issues
  • Lab 5 ⇒ Pipelined Processor
  • Lab 6 ⇒ With caches
° We’ll talk about modern processors and what’s really hard:
  • exception handling
  • trying to improve performance with out-of-order execution, etc.
  • Trying to get CPI < 1 (Superscalar execution)

Summary
° Microprogramming is a fundamental concept
  • implement an instruction set by building a very simple processor and interpreting the instructions
  • essential for very complex instructions and when few register transfers are possible
  • Control design reduces to Microprogramming
° 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
  • Providing clean interrupt model gets hard with pipelining!
° Precise Exception ⇒ state of the machine is preserved as if program executed up to the offending instruction
  • All previous instructions completed
  • Offending instruction and all following instructions act as if they have not even started

10/12/01 ©UCB Fall 2001 CS152 / Kubiatowicz Lec12.49

10/12/01 ©UCB Fall 2001 CS152 / Kubiatowicz Lec12.50