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 \( \mu \) instruction:
    contains sequencer control + values for control points
  • limited state transitions:
    branch to zero, next sequential, branch to \( \mu \) instruction address from dispatch ROM

° Horizontal \( \mu \) Code: one control bit in \( \mu \) instruction for every control line in datapath

° Vertical \( \mu \) Code: groups of control-lines coded together in \( \mu \) 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 Failing”) 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

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.

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

What happens to Instruction with Exception?

° MIPS architecture defines the instruction as having no effect if the instruction causes an exception.

° When get to virtual memory we will see that certain classes of exceptions must prevent the instruction from changing the machine state.

° This aspect of handling exceptions becomes complex and potentially limits performance => why it is hard

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 than 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

**Saving State**

° Push it onto the stack
  - Vax, 68k, 80x86

° Save it in special registers
  - MIPS EPC, BadVAddr, Status, Cause

° Shadow Registers
  - M68k
  - Save state in a shadow of the internal pipeline registers

**Additions to MIPS ISA to support Exceptions?**

° Exception state is kept in "coprocessor 0".
  - EPC—a 32-bit register used to hold the address of the affected instruction (register 14 of coprocessor 0).
  - 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 bits 5 to 2 of this register encodes the two possible exception sources mentioned above: undefined instruction=0 and arithmetic overflow=1 (register 13 of coprocessor 0).
  - BadVAddr - register contained memory address at which memory reference occurred (register 8 of coprocessor 0)
  - Status - interrupt mask and enable bits (register 12 of coprocessor 0)
  - Control signals to write EPC, Cause, BadVAddr, and Status
  - Be able to write exception address into PC, increase mux to add as input 01000000 00000000 00000000 01000000two (8000 0080 hex)
  - May have to undo PC = PC + 4, since want EPC to point to offending instruction (not its successor); PC = PC - 4
Recap: Details of Status register

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

Recap: Details of Cause register

<table>
<thead>
<tr>
<th>Status</th>
<th>Pending</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
</table>

- **Pending interrupt** 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
- **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

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
But: What has to change in our µ-sequencer?

- Need concept of branch at micro-code level

Example: Can easily use with for non-ideal memory

Example: Can easily use with for non-ideal memory

Summary: Microprogramming one inspiration for RISC

- 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)

Administrative Issues: Result of Midterm I

- Exam Average: 62, Standard Dev: 13.5
  - People had trouble with square-root problem
  - This was very much like Divide!
    - Large shift register moving left
  - Some issues with microcode
Square root example: consider remainder shifting \textit{LEFT}

Starting: \( M = R_0 = 01110110 \) and \( S_0 = 0000 \)

Try: \( N_1 = 1000 - 1000 \)
\[ \left( 2 \times S_0 + 1000 \right) \times 1000 \]
\( R_1 = 00110110 \Rightarrow S_1 = S_0 + 1000 = 1000 \)

Try: \( N_2 = 0100 - 10100 \)
\[ \left( 2 \times S_1 + 0100 \right) \times 0100 \]
\( R_2 = 00110110 \) (unchanged)

Try: \( N_3 = 0010 - 10010 \)
\[ \left( 2 \times S_2 + 0010 \right) \times 0010 \]
\( R_3 = 00010010 \Rightarrow S_3 = S_2 + 0010 = 1010 \)

Try: \( N_4 = 0001 - 10101 \)
\[ \left( 2 \times S_3 + 0001 \right) \times 0001 \]
\( R_4 = 00010010 \) (unchanged)

Result < 0 \( \Rightarrow S_4 = S_3 = 1010 \)
\( R_4 = 00010010 \) (unchanged)

Final result: \( = 1010_2 \) with \( 10010_2 \) remainder

or: \( = 10 \) with 18 remainder!

Administrative Issues (continued)

\begin{itemize}
  \item Get started reading Chapter 6!
  \item Complete chapter on Pipelining...
  \item Next week sections => Cory 119
  \item Computers in the News: Merced silicon has finally seen light of day
\end{itemize}

The Big Picture: Where are We Now?

\begin{itemize}
  \item The Five Classic Components of a Computer
\end{itemize}

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
Processor & Input & Memory & Output \\
\hline
Control & & & \\
Datapath & & & \\
\hline
\end{tabular}
\end{center}

\begin{itemize}
  \item Next Topics:
    \begin{itemize}
      \item Pipelining by Analogy
      \item Administrivia; Course road map
    \end{itemize}
\end{itemize}

Pipelining is Natural!

\begin{itemize}
  \item Laundry Example
    \begin{itemize}
      \item Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
    \end{itemize}
  \item Washer takes 30 minutes
  \item Dryer takes 40 minutes
  \item "Folder" takes 20 minutes
\end{itemize}
**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 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!

IR <= MEM[PC]  
PC <= PC + 4

ALUout <= PC + SX  
0000

ALUout <= A op ZX  
0100

If A = B then PC <= ALUout
0010

M <= MEM(ALUout)
1001

MEM(ALUout) <= B
1100

Execute

Memory

Write-back

Decode

Fetch

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

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFetch</td>
</tr>
<tr>
<td>IFetch</td>
</tr>
<tr>
<td>IFetch</td>
</tr>
<tr>
<td>IFetch</td>
</tr>
<tr>
<td>IFetch</td>
</tr>
</tbody>
</table>

### Single Cycle, Multiple Cycle, vs. Pipeline

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

#### Cycle 1:
- Ifetch
- Reg
- Exec
- Mem
- Wr

#### Cycle 2:
- Ifetch
- Reg
- Exec
- Mem
- Wr

#### Cycle 3:
- Ifetch
- Reg
- Exec
- Mem
- Wr

#### Cycle 4:
- Ifetch
- Reg
- Exec
- Mem
- Wr

#### Cycle 5:
- Ifetch
- Reg
- Exec
- Mem
- Wr

#### Cycle 6:
- Ifetch
- Reg
- Exec
- Mem
- Wr

#### Cycle 7:
- Ifetch
- Reg
- Exec
- Mem
- Wr

#### Cycle 8:
- Ifetch
- Reg
- Exec
- Mem
- Wr

#### Cycle 9:
- Ifetch
- Reg
- Exec
- Mem
- Wr

#### Cycle 10:
- Ifetch
- Reg
- Exec
- Mem
- Wr

### Why Pipeline?

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

### Why Pipeline? Because the resources are there!

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst 0</td>
</tr>
<tr>
<td>Inst 1</td>
</tr>
<tr>
<td>Inst 2</td>
</tr>
<tr>
<td>Inst 3</td>
</tr>
<tr>
<td>Inst 4</td>
</tr>
</tbody>
</table>
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)
  • 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
  • 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

° Can always resolve hazards by waiting
  • pipeline control must detect the hazard
  • take action (or delay action) to resolve hazards

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

Single Memory is a Structural Hazard

Detection is easy in this case! (right half highlight means read, left half write)
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 \((1 \times 0.5 + 2 \times 0.5) = 1.5\)
- Total CPI might then be: \(1.5 \times 0.2 + 1 \times 0.8 = 1.1\) (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

Add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11

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

- Can’t solve with forwarding:
  - Must delay/stall instruction dependent on loads

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;
R[rd] <- S;
M <- Mem[S]; Mem[S] <- B;

Pipelined Processor (almost) for slides

What happens if we start a new instruction every cycle?

Pipelined Datapath (as in book); hard to read

Pipelining the Load Instruction

The five independent functional units in the pipeline datapath are:
- Instruction Memory for the Ifetch stage
- Register File’s Read ports (bus A and busB) for the Reg/Dec stage
- ALU for the Exec stage
- Data Memory for the Mem stage
- Register File’s Write port (bus W) for the Wr stage
### The Four Stages of R-type

- **Ifetch**: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**:
  - ALU operates on the two register operands
  - Update PC
- **Wr**: Write the ALU output back to the register file

### Pipelining the R-type and Load Instruction

- **We have pipeline conflict or structural hazard**:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

### Important Observation

- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage for all instructions:
  - Load uses Register File’s Write Port during its 5th stage
  - R-type uses Register File’s Write Port during its 4th stage

### Solution 1: Insert “Bubble” into the Pipeline

- **Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle**
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.
- **No instruction is started in Cycle 6!**
Solution 2: Delay R-type’s Write by One Cycle

Delay R-type’s register write by one cycle:
- Now R-type instructions also use Reg File’s write port at Stage 5
- Mem stage is a NOOP stage: nothing is being done.

\[\begin{array}{cccccc}
\text{Cycle 1} & \text{Cycle 2} & \text{Cycle 3} & \text{Cycle 4} & \text{Cycle 5} & \text{Cycle 6} & \text{Cycle 7} & \text{Cycle 8} & \text{Cycle 9} \\
\text{R-type} & \text{Ifetch} & \text{Reg/Dec} & \text{Exec} & \text{Mem} & \text{Wr} \\
\end{array}\]

Modified Control & Datapath

\[\begin{array}{cccccc}
\text{IR} \leftarrow \text{Mem[PC]}; \text{PC} \leftarrow \text{PC+4}; \\
\text{A} \leftarrow \text{R[rs]}; \text{B} \leftarrow \text{R[rt]} \\
\text{S} \leftarrow \text{A+B}; \quad \text{S} \leftarrow \text{A or ZB}; \quad \text{S} \leftarrow \text{A+BX}; \quad \text{S} \leftarrow \text{A+BX}; \\
\text{M} \leftarrow \text{Mem[S]} \quad \text{Mem[S]} \leftarrow \text{B} \\
\text{R[rd]} \leftarrow \text{M}; \quad \text{R[rt]} \leftarrow \text{M}; \quad \text{R[rd]} \leftarrow \text{M}; \\
\text{if Cond PC} \leftarrow \text{PC+BX}; \quad \text{if Cond PC} \leftarrow \text{PC+BX}; \\
\end{array}\]

The Four Stages of Store

- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

\[\begin{array}{cccccc}
\text{Cycle 1} & \text{Cycle 2} & \text{Cycle 3} & \text{Cycle 4} & \\
\text{Store} & \text{Ifetch} & \text{Reg/Dec} & \text{Exec} & \text{Mem} & \text{Wr} \\
\end{array}\]

The Three Stages of Beq

- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec:
  - Registers Fetch and Instruction Decode
- Exec:
  - compares the two register operand,
  - select correct branch target address
  - latch into PC

\[\begin{array}{cccccc}
\text{Cycle 1} & \text{Cycle 2} & \text{Cycle 3} & \text{Cycle 4} & \\
\text{Beq} & \text{Ifetch} & \text{Reg/Dec} & \text{Exec} & \text{Mem} & \text{Wr} \\
\end{array}\]
Control Diagram

IR <- Mem[PC]; PC <- PC+4.
A <- R[rs]; B <- R[rt]
S <- A + B;
S <- A or ZX;
S <- A + SX;
S <- A + SX;
M <- S
M <- Mem[S]
S <- A + SX;
R[rd] <- S;
R[r] <- S;
R[rd] <- M;
M <- S
M <- Mem[S]
Mem[S] <- B

Let's Try it Out

10 lw r1, r2(35)
14 addI r2, r2, 3
20 sub r3, r4, r5
24 beq r6, r7, 100
30 ori r8, r9, 17
34 add r10, r11, r12
100 and r13, r14, 15

these addresses are octal
Summary: Pipelining

- **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

- We’ll build a simple pipeline and look at these issues

- We’ll talk about modern processors and what really makes it hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

Summary

- Pipelining is a fundamental concept
  - multiple steps using distinct resources

- Utilize capabilities of the Datapath by pipelined instruction processing
  - start next instruction while working on the current one
  - limited by length of longest stage (plus fill/flush)
  - detect and resolve hazards