The Big Picture: Where are We Now?

- The Five Classic Components of a Computer
  - Processor
  - Control
  - Datapath
  - Memory
  - Input
  - Output

- Today's Topics:
  - Microprogrammed control
  - Administrivia; Courses
  - Microprogram it yourself
  - Exceptions
  - Intro to Pipelining (if time permits)

Review: Multicycle Datapath

- Critical Path?

Review: Control Specification for multicycle proc

ir <= mem[pc]  "instruction fetch"

A <= R[rs]  B <= R[rt]

S <= A fun B  G <= A or ZX

S <= A + SX  S <= A + SX

M <= MEM[S]  BPC <= PC + 4

R[rd] <= S  R[to] <= B

R[to] <= S  PC <= PC + 4

R[to] <= M  PC <= PC + 4

2/25/04 ©UCB Spring 2004 CS152 / Kubiatowicz Lec9.3

2/25/04 ©UCB Spring 2004 CS152 / Kubiatowicz Lec9.4
Review: 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

Review: Controller Design

- The state diagrams that arise define the controller for an instruction set processor are highly structured
- Use this structure to construct a simple “microsequencer”
- Control reduces to programming this very simple device
  ⇒ microprogramming

Example: Jump-Counter

“Zero” “Increment” “Load”

None of above: Do nothing (for wait states)
Our Microsequencer

Our Microsequencer

Microprogram Control Specification

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>μPC</td>
<td>Equal</td>
</tr>
<tr>
<td></td>
<td>Next</td>
</tr>
<tr>
<td></td>
<td>IR PC</td>
</tr>
<tr>
<td></td>
<td>Ops</td>
</tr>
<tr>
<td></td>
<td>Exec</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
</tr>
<tr>
<td></td>
<td>Write-Back</td>
</tr>
<tr>
<td></td>
<td>Wr Dst</td>
</tr>
<tr>
<td>0000</td>
<td>?</td>
</tr>
<tr>
<td>0001</td>
<td>x</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>x</td>
</tr>
<tr>
<td>0110</td>
<td>x</td>
</tr>
<tr>
<td>1010</td>
<td>x</td>
</tr>
<tr>
<td>1000</td>
<td>x</td>
</tr>
<tr>
<td>1011</td>
<td>x</td>
</tr>
<tr>
<td>1100</td>
<td>x</td>
</tr>
</tbody>
</table>

Microprogram Control Specification

Alternative multicycle datapath (book)

Alternative multicycle datapath (book)

New Finite State Machine (FSM) Spec

New Finite State Machine (FSM) Spec

Q: How improve to do something in state 0001?
### Finite State Machine (FSM) Spec

- **Instruction Fetch**
  - \( IR \leftarrow MEM[PC] \)
  - \( PC \leftarrow PC + 4 \)

- **Decode**
  - **R-type**
    - \( ALUout \leftarrow PC + SX \)
    - \( R[rd] \leftarrow ALUout \)
    - \( R[rt] \leftarrow M \)
  - **LW**
    - \( ALUout \leftarrow A + SX \)
    - \( M \leftarrow MEM[ALUout] \)
  - **SW**
    - \( ALUout \leftarrow A + SX \)
    - \( MEM[ALUout] \leftarrow B \)
  - **BEQ**
    - If \( A = B \) then
      - \( PC \leftarrow ALUout \)

### Microprogramming

- **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
  - Overkill when ISA matches datapath 1:1

- **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

### “Macroinstruction” Interpretation

- User program plus Data
  - This can change!
  - One of these is mapped into one of these

- AND microsequence
  - E.g., Fetch, Calcl Operand Addr, Fetch Operand(s), Calculate, Save Answer(s)
Administrivia

° Lab 3 design document to go over in section
  • EMail preliminary design document to TA by 9pm tonight
  • Email final version to TA by midnight Thursday
  • Sections back in normal rooms
  • This is a complicated lab – may need to give updates as we get the boards installed
° Change to Lab specification
  • Your ChipScope should be set up to trigger on a PC entered from board switches
° Homework quiz still next Wednesday
  • Homework is shorter this time
  • Make sure to check on assigned problems
° Midterm I, Wednesday March 10th (Two Weeks)
  • 5:30 - 8:30 in 306 Soda Hall
  • Bring a Calculator!
  • One 8 1/2 by 11 page (both sides) of notes
  • Make up exam?

Review: Testing: Make sure that things work

° Testing methodologies
  • Understand what correct behavior is when you design things
    - Collect vectors for later use
  • Build monitor modules to check assertions of correct values
  • Produce a regression test
    - Set of tests to run each time something changes
° Types of test (Doug Clark):
  • Directed Vectors – test explicit behavior
  • Random Vectors – apply random values or orderings to device
  • Daemons – continuous error insertion
° Monitor modules:
  • Check to see if invariants are maintained during long running simulations

Review: Monitor Modules: Passthrough testing

module monitorsum32 (carry, sum, A, B);
  input [31:0] A, B;
  output [31:0] sum;
  output carry;
  reg [31:0] predsum;
  reg precarry;

  // The “real” adder
  sum32 mysum (carry, sum, A, B);

  `ifndef synthesis // This checker code only for simulation
    always @(A or B)
    begin
      #100 //wait for output to settle (don’t make too long!)
      [predcarry, predsum] = A + B;
      if ((carry != predcarry) || (sum != predsum))
        $display(">>> Mismatch: 0x%x+0x%x->0x%x carry %x", A_B,sum,carry);
    end
  `endif
endmodule

Lab3 version: monitors and benches

° Important design idea:
  • Set up your processor so that you can take the same design and both simulate and push it to board
° Testing Ideas:
  • wrap testing infrastructure around devices under test (DUT)
  • Include test vectors that are supposed to detect errors in implementation. Even strange ones…
  • Can (and probably should in later labs) include assert statements to check for “things that should never happen”
Designing a Microinstruction Set

1) Start with list of control signals
2) Group signals together that make sense (vs. random): called "fields"
3) Place fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)
4) To minimize the width, encode operations that will never be used at the same time
5) Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals
   • Use computers to design computers

Recap: Multicycle datapath (book)
3&4) Microinstruction Format: unencoded vs. encoded fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Width</th>
<th>Control Signals Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU Control</td>
<td>4</td>
<td>wide</td>
</tr>
<tr>
<td>SRC1</td>
<td>2</td>
<td>narrow</td>
</tr>
<tr>
<td>SRC2</td>
<td>5</td>
<td>ALUSElB, ExtOp</td>
</tr>
<tr>
<td>ALU Destination</td>
<td>3</td>
<td>RegWrite, MemtoReg, RegDst</td>
</tr>
<tr>
<td>Memory</td>
<td>3</td>
<td>MemRead, MemWrite, lirD</td>
</tr>
<tr>
<td>Memory Register</td>
<td>1</td>
<td>IRWrite</td>
</tr>
<tr>
<td>PCWrite Control</td>
<td>3</td>
<td>Add</td>
</tr>
<tr>
<td>Sequencing</td>
<td>3</td>
<td>AddrCt</td>
</tr>
<tr>
<td>Total width</td>
<td>24</td>
<td>15 bits</td>
</tr>
</tbody>
</table>

Quick check: what do these fieldnames mean?

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>RegWrite</th>
<th>MemToReg</th>
<th>RegDest</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ALU</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>ALU</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>ALU</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>ALU</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SRC2:</th>
<th>Code</th>
<th>Name</th>
<th>ALUSElB</th>
<th>ExtOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>---</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>4</td>
<td>00</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>rt</td>
<td>01</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>ExtShft</td>
<td>10</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Extend</td>
<td>11</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>Extend</td>
<td>11</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Step 5—Group into Fields, Order and Assign Names

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Values for Field</th>
<th>Function of Field with Specific Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td></td>
<td>ALU adds</td>
</tr>
<tr>
<td>Subt.</td>
<td></td>
<td>ALU subtracts</td>
</tr>
<tr>
<td>Func</td>
<td></td>
<td>ALU does function code</td>
</tr>
<tr>
<td>Or</td>
<td></td>
<td>ALU does logical OR</td>
</tr>
<tr>
<td>SRC1</td>
<td>PC</td>
<td>1st ALU input &lt;= Reg[rs]</td>
</tr>
<tr>
<td>SRC2</td>
<td>2nd ALU input</td>
<td>2nd ALU input &lt;= Reg[rs]</td>
</tr>
<tr>
<td>ExtSrc</td>
<td>2nd ALU input</td>
<td>ExtSrc 2nd ALU input &lt;= Reg[rs]</td>
</tr>
<tr>
<td>ExtShft</td>
<td>2nd ALU input</td>
<td>ExtShft 2nd ALU input &lt;= Reg[rs]</td>
</tr>
<tr>
<td>rt</td>
<td>2nd ALU input</td>
<td>rt 2nd ALU input &lt;= Reg[rs]</td>
</tr>
<tr>
<td>dest(ination)</td>
<td>rd ALU</td>
<td>Reg[rd] &lt;= ALUout</td>
</tr>
<tr>
<td>Memreg</td>
<td>IR</td>
<td>IR &lt;= Mem</td>
</tr>
<tr>
<td>Memreg</td>
<td>PCWrite</td>
<td>PC &lt;= Mem</td>
</tr>
<tr>
<td>Memreg</td>
<td>Write ALU</td>
<td>Write memory using ALUout for addr</td>
</tr>
<tr>
<td>Memreg</td>
<td>Mem(ory)</td>
<td>Read memory using PC</td>
</tr>
<tr>
<td>Memreg</td>
<td>Read ALU</td>
<td>Read memory using ALUout for addr</td>
</tr>
<tr>
<td>Memreg</td>
<td>MemToReg</td>
<td>Read memory using ALUout for addr</td>
</tr>
<tr>
<td>Memreg</td>
<td>RegWrite</td>
<td>RegWrite Mem</td>
</tr>
<tr>
<td>Memreg</td>
<td>MemtoReg</td>
<td>MemtoReg RegWrite Mem</td>
</tr>
<tr>
<td>Memreg</td>
<td>RegDest</td>
<td>RegDest RegWrite Mem</td>
</tr>
<tr>
<td>Memreg</td>
<td>MemSource</td>
<td>MemSource RegWrite MemSource</td>
</tr>
<tr>
<td>Memreg</td>
<td>MemSource</td>
<td>RegWrite MemSource RegWrite Mem</td>
</tr>
<tr>
<td>Seq(encing)</td>
<td>Fetch</td>
<td>Go to next sequential microinstruction</td>
</tr>
<tr>
<td>Seq(encing)</td>
<td>Dispatch</td>
<td>Dispatch using ROM</td>
</tr>
</tbody>
</table>

Specific Sequencer for our Microcoding

Sequencer-based control unit from last lecture

- Called “microPC” or “μPC” vs. state register

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>fetch</td>
<td>Next μaddress = 0</td>
</tr>
<tr>
<td>01</td>
<td>dispatch</td>
<td>Next μaddress = dispatch ROM</td>
</tr>
<tr>
<td>10</td>
<td>seq</td>
<td>Next μaddress = μaddress + 1</td>
</tr>
</tbody>
</table>

ROM:

```
000000: Rtype (0100)
000100: BEQ (0010)
001101: ORI (0110)
100011: LW (1000)
101011: SW (1011)
```

 Opcode: Dispatch state

```
000000: Rtype (0100)
000100: BEQ (0010)
001101: ORI (0110)
100011: LW (1000)
101011: SW (1011)
```

μAddress Select Logic

```
000000: Rtype (0100)
000100: BEQ (0010)
001101: ORI (0110)
100011: LW (1000)
101011: SW (1011)
```

Opcode: Dispatch state

```
000000: Rtype (0100)
000100: BEQ (0010)
001101: ORI (0110)
100011: LW (1000)
101011: SW (1011)
```
Recap: Finite State Machine (FSM) Spec

IR <= MEM[PC] 0000
PC <= PC + 4

ALUout <= PC + SX 0001

R-type
ALUout <= A + SX
0100
ALUout <= A or ZK
0110
ALUout <= A + SX
1000
ALUout <= M
1010

Ori
ALUout <= A fun B
0101

LW
ALUout <= A + SX
1110

BEQ
PC <= M
1100

Memory
Write-back

“decode”

Execute

Microprogram it yourself!

<table>
<thead>
<tr>
<th>Addr</th>
<th>ALU SRC1 SRC2 Dest.</th>
<th>Memory</th>
<th>Mem</th>
<th>Reg.</th>
<th>PC</th>
<th>Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Add PC 4</td>
<td>Read</td>
<td>IR</td>
<td>ALU</td>
<td>Seq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>Add PC Extshft</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dispatch</td>
</tr>
<tr>
<td>0010</td>
<td>Subt. rs rt</td>
<td>ALUOutCond.</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>Func rs rt</td>
<td>Seq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>0101</td>
<td>rd ALU</td>
<td>Seq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>0110</td>
<td>Or rs Extend0</td>
<td>Seq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>0111</td>
<td>rt ALU</td>
<td>Seq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>1000</td>
<td>Add rs Extend</td>
<td>Seq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>1001</td>
<td>Read ALU</td>
<td>Seq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>1010</td>
<td>rt MEM</td>
<td>Seq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>1011</td>
<td>Add rs Extend</td>
<td>Seq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>1100</td>
<td>Write ALU</td>
<td>Seq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
</tbody>
</table>

Review: Overview of Control

- 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.

Initial Representation

- Finite State Diagram
- Microprogram

Sequencing Control

- Explicit Next State Function
- Microprogram counter + Dispatch ROMs

Logic Representation

- Logic Equations
- Truth Tables

Implementation Technique

- PLA “hardwired control”
- ROM “microprogrammed control”

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

Precise Exceptions

° 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

° Two modes of execution (user/system) :
  • operating system runs in 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, M88K, ...

° 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
- 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>k</th>
<th>e</th>
<th>k</th>
<th>e</th>
<th>k</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</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>
<tbody>
<tr>
<td>Pending</td>
<td>Code</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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
Part of the handler in trap_handler.s

entry:
⇐ Exceptions/interrupts come here
.set noat
.move $k1 $at # Save $at
.set at
.sw $v0 x1 # Not re-entrant and we can’t trust $sp
.sw $a0 x2
.mfc0 $k0 $13 # Cause ⇐ Grab the cause register
li $v0 4 # syscall 4 (print_str)
li $a0 __m1_syscall
li $v0 1 # syscall 1 (print_int)
.srl $a0 $k0 2 # shift Cause reg syscall
ret: lw $v0 $s1
lw $a0 $s2
.mfc0 $k0 $14 # EPC ⇐ Get the return address (EPC)
.set noat
.move $at $k1 # Restore $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?

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

° Need concept of branch at micro-code level

Interrupts are precise because user-visible state committed after exceptions flagged!
Example: Can easily use with for non-ideal memory

IR <= MEM[PC]
“instruction fetch”
~wait

A <= R[rs]
B <= R[rt]
“decode / operand fetch”
wait

S <= A + B
R[rd] <= S
PC <= PC + 4

S <= A + SX
R[rd] <= S
PC <= PC + 4

S <= A or ZX
R[rd] <= S
PC <= PC + 4

S <= A + SX
R[rd] <= MEM[S]
PC <= Next(PC)

S <= A + S
PC <= Next(PC)

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>
<tr>
<td></td>
<td></td>
<td></td>
<td>Average CPI: 4.1</td>
</tr>
</tbody>
</table>

Question #2: Can we get CPI < 4.1?
- Seems to be lots of “idle” hardware
  - Why not overlap instructions???
Legacy Software and Microprogramming

- IBM bet company on 360 Instruction Set Architecture (ISA): single instruction set for many classes of machines
  - (8-bit to 64-bit)
- Stewart Tucker stuck with job of what to do about software compatibility
  - If microprogramming could easily do same instruction set on many different microarchitectures, then why couldn’t multiple microprograms do multiple instruction sets on the same microarchitecture?
  - Coined term “emulation”: instruction set interpreter in microcode for non-native instruction set
  - Very successful: in early years of IBM 360 it was hard to know whether old instruction set or new instruction set was more frequently used

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.
  - Can tailor instruction set to application.
- Compatibility
  - Many organizations, same instruction set
- Costly to implement
- Slow

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
- Design of a Microprogramming language
  1. Start with list of control signals
  2. Group signals together that make sense (vs. random): called “fields”
  3. Place fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)
  4. To minimize the width, encode operations that will never be used at the same time
  5. Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals

Thought: 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)