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.

<table>
<thead>
<tr>
<th>Initial Representation</th>
<th>Finite State Diagram</th>
<th>Microprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequencing Control</td>
<td>Explicit Next State</td>
<td>Microprogram counter + Dispatch ROMs</td>
</tr>
<tr>
<td></td>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>Logic Representation</td>
<td>Logic Equations</td>
<td>Truth Tables</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation Technique</td>
<td>PLA</td>
<td>ROM</td>
</tr>
<tr>
<td></td>
<td>“hardwired control”</td>
<td>“microprogrammed control”</td>
</tr>
</tbody>
</table>

Recap: “MacroinSTRUCTION” Interpretation

- User program plus Data
- this can change!
- one of these is mapped into one of these
- AND microsequence
- e.g., Fetch
  - Calc Operand Addr
  - Fetch Operand(s)
  - Calculate
  - Save Answer(s)

Recap: Micro-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 “microsequence”
  - Each state in previous diagram becomes a “microinstruction”
  - Microinstructions often taken sequentially
- Control reduces to programming this device
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)

Recap: Horizontal vs. Vertical Microprogramming

NOTE: previous organization is not TRUE horizontal microprogramming; register decoders give flavor of encoded microoperations

Most microprogramming-based controllers vary between:
- **Horizontal** organization (1 control bit per control point)
  - more control over the potential parallelism of operations in the datapath
  - uses up lots of control store

- **Vertical** organization (fields encoded in the control memory and must be decoded to control something)
  - easier to program, not very different from programming a RISC machine in assembly language
  - extra level of decoding may slow the machine down

Recap: Designing a Microinstruction Set

1) Start with list of control signals
2) Group signals together that make sense (vs. random): called “fields”
3) Places fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)
4) 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
5) To minimize the width, encode operations that will never be used at the same time

Alternative datapath (book): Multiple Cycle Datapath

- Miminizes Hardware: 1 memory, 1 adder
1&2) Start with list of control signals, grouped into fields

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted</th>
<th>Effect when asserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUSelA</td>
<td>1st ALU operand = PC</td>
<td>1st ALU operand = Reg[rs]</td>
</tr>
<tr>
<td>RegWrite</td>
<td>None</td>
<td>Reg. is written</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>Reg. write data input = ALU</td>
<td>Reg. write data input = memory</td>
</tr>
<tr>
<td>RegDst</td>
<td>Reg. dest. no. = rt</td>
<td>Reg. dest. no. = rd</td>
</tr>
<tr>
<td>MemRead</td>
<td>None</td>
<td>Memory at address is read, MDR &lt;= Mem[addr]</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Memory at address is written</td>
</tr>
<tr>
<td>IorD</td>
<td>Memory address = PC</td>
<td>Memory address = S</td>
</tr>
<tr>
<td>IrWrite</td>
<td>None</td>
<td>IR &lt;= Memory</td>
</tr>
<tr>
<td>PCWrite</td>
<td>None</td>
<td>PC &lt;= PCsSource</td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>None</td>
<td>IF ALUzero then PC &lt;= PCsSource</td>
</tr>
<tr>
<td>PCSource</td>
<td>PCSource = ALU</td>
<td>PCSource = ALUout</td>
</tr>
<tr>
<td>ExtOp</td>
<td>Zero Extended</td>
<td>Sign Extended</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signal name Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td>ALUSelB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>

Quick check: what do these fieldnames mean?

**Destination:**

<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>---</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>rd ALU</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>rt ALU</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>rt MEM</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**SRC2:**

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>001</td>
<td>4</td>
<td>00</td>
<td>X</td>
</tr>
<tr>
<td>010</td>
<td>rt</td>
<td>01</td>
<td>X</td>
</tr>
<tr>
<td>011</td>
<td>ExtShft</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>Extend</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>Extend0</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

3) 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>2</td>
</tr>
<tr>
<td>SRC1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SRC2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>ALU Destination</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Memory</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Memory Register</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PCWrite Control</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Sequencing</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Total width</td>
<td>24</td>
<td>15 bits</td>
</tr>
</tbody>
</table>
### Alternative Datapath (book): Multiple Cycle Datapath

- **Mimines Hardware:** 1 memory, 1 adder

#### Finite State Machine (FSM) Spec

- **IR <= MEM[PC]**
- **PC <= PC + 4**
- **R-type**
  - **ALUout <= A fun B**
  - **R[rd] <= ALUout**
- **BEQ:**
  - If A = B then
  - **PC <= ALUout**

### Recap: Specific Sequencer from last lecture

- **Sequencer-based control unit from last lecture**
  - Called “microPC” or “µPC” vs. state register

#### Control Value Effect

<table>
<thead>
<tr>
<th>Control Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Next µaddress = 0</td>
</tr>
<tr>
<td>01</td>
<td>Next µaddress = dispatch ROM</td>
</tr>
<tr>
<td>10</td>
<td>Next µaddress = µaddress + 1</td>
</tr>
</tbody>
</table>

#### ROM:

- **R-type:** 000000 0100
- **BEQ:** 000100 0011
- **ori:** 001101 0110
- **LW:** 100011 1000
- **SW:** 101011 1011

### Microprogram it yourself!

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>ALU Dest.</th>
<th>Memory</th>
<th>Mem. Req.</th>
<th>PC Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch: Add</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>IR</td>
<td>ALU</td>
<td>Seq</td>
<td>Dispatch</td>
</tr>
<tr>
<td>RItype:Func</td>
<td>rs</td>
<td>rt</td>
<td>rd ALU</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ: Subt</td>
<td>rs</td>
<td>rt</td>
<td>ALUoutCond</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Microprogram it yourself!

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Dest.</th>
<th>Memory</th>
<th>Mem. Reg.</th>
<th>PC Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch: Add</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>IR</td>
<td>ALU</td>
<td>Seq</td>
<td>Dispatch</td>
</tr>
<tr>
<td>Rtype: Func</td>
<td>rs</td>
<td>rt</td>
<td>rd ALU</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ori:</td>
<td>Or</td>
<td>rs</td>
<td>Extend</td>
<td>rt ALU</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td>Read ALU</td>
<td>Seq</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>Sw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td>Write ALU</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Beq:</td>
<td>Subt.</td>
<td>rs</td>
<td>rt</td>
<td>ALUOutCond.</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Administrivia

° Midterm I next Wednesday
  - 5:30 - 8:30 in 277 Cory (Honest!)
  - Bring a Calculator!
  - One 8 1/2 by 11 page (both sides) of notes
  - Make up exam: 5:30 – 8:30 in 606 Soda Hall
° Materials through Chapter 5, Appendix A, B & C
° Review session this Sunday 7:00 306 Soda
° Afterwards: Pizza and refreshments at LaVals
° Lab 4 breakdown due by midnight tonight
  - EMail to your TA
  - Get moving on it! This is a complicated lab.
° Now, start reading Chapter 6

Lab4: start using test benches

° Idea: 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"

Administrivia: Courses to consider during Telebears

° General Philosophy
  - Take courses from great teachers (HKN ratings helps find them)
    - http://www-hkn.eecs.berkeley.edu/toplevel/coursesurveys.html
  - Take variety of undergrad courses now to get introduction to areas; can learn advanced material on own later once know vocabulary
    - Who knows what you will work on over a 40 year career?
° CS169 Software Engineering
  - Everyone writes programs, even hardware designers
  - Often programs are written in groups ⇒ learn skill in school
° EE122 Introduction to Communication Networks
  - World is getting connected; communications must play major role
° CS162 Operating Systems
  - All special-purpose hardware will run a layer of software that uses processes and concurrent programming; CS162 is the closest thing
An Alternative MultiCycle DataPath

- In each clock cycle, each Bus can be used to transfer from one source
- μ-instruction can simply contain B-Bus and W-Dst fields

Load

- Execute
- Mem
- Write-back

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**

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</td>
</tr>
<tr>
<td>Hardware malfunctions</td>
<td>Either</td>
<td>Exception or</td>
</tr>
<tr>
<td></td>
<td></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 \(\Rightarrow\) why it is hard

Precise Interrupts

- Precise \(\Rightarrow\) 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 \(\Rightarrow\) 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 \(\leftarrow\) MEM[ iv_base + cause || 00]
  - 370, 68000, Vax, 80x86, ...
- RISC Handler Table
  - PC \(\leftarrow\) IT_base + cause || 0000
  - saves state and jumps
  - Sparc, PA, M88K, ...
- MIPS Approach: fixed entry
  - PC \(\leftarrow\) 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>15</th>
<th>8</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask</td>
<td></td>
<td>k</td>
<td>k</td>
<td>e</td>
<td>k</td>
<td>e</td>
<td>k</td>
<td>e</td>
</tr>
<tr>
<td>old</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>prev</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>current</td>
<td></td>
<td></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

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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></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

```asm
.ktext 0x80000080
entry:
  .set noat
  move $k1 $at # Save $at
  .set at
  sw $v0 a1 # Not re-entrant and we can't trust $sp
  sw $a0 a2
  mfld $k0 $13 # Cause
    .set noat
    move $at $k1 # Save $at
    .set at
    sw $v0 $a2 # Not re-entrant and we can't trust $sp
    sw $a0 $a3
    mfld $k0 $14 # Get the return address (EPC)
    .set noat
    move $at $k1 # Restore $at
    .set at
    rfe # Return from exception handler
    addiu $k0 $k0 4 # Return to next instruction
    jr $k0

set: lw $v0 a1
    lw $a0 a2
    mfld $k0 $14 # EPC
    .set noat
    move $at $k1 # Save $at
    .set at
    sw $v0 $a2 # Not re-entrant and we can't trust $sp
    sw $a0 $a3
    mfld $k0 $14 # Get the return address (EPC)
    .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?

But: What has to change in our \( \mu \)-sequencer?

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

**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)