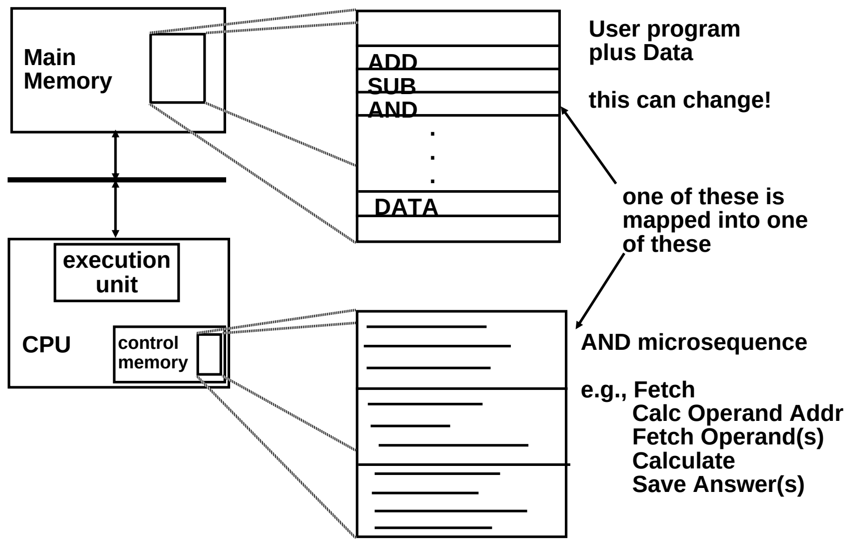
Computer System Architecture

Processor Part IV

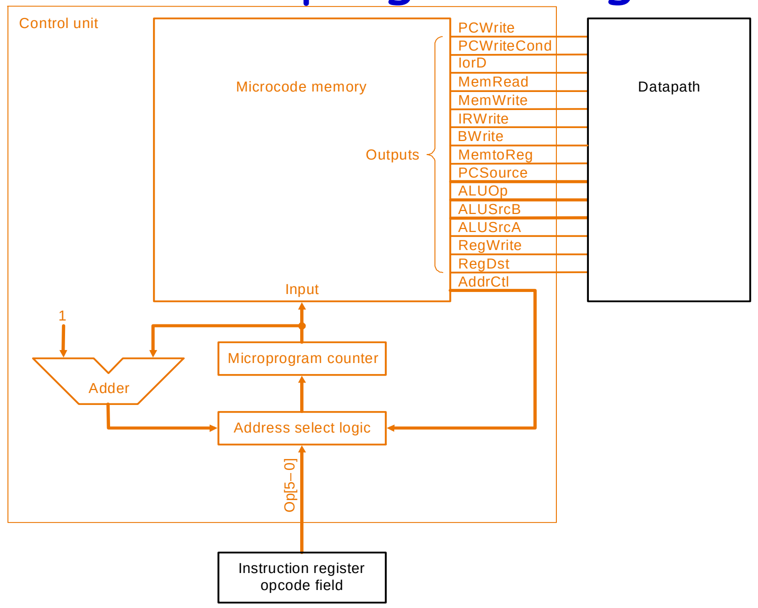
Chalermek Intanagonwiwat

Slides courtesy of John Hennessy and David Patterson

"Macroinstruction" Interpretation



Microprogramming



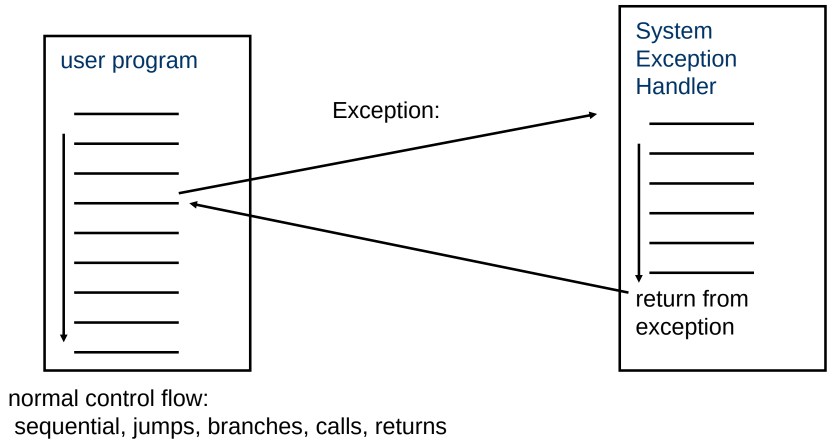
Microprogramming Pros and Cons

- 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)

Microprogramming Pros and Cons (cont.)

- Generality
  - Can implement multiple instruction sets on same machine.
  - Can tailor instruction set to application.
- Compatibility
  - Many organizations, same instruction set
- Slow

Exceptions



Exceptions (cont.)

- Exception = unprogrammed control transfer
  - system takes action to handle the exception
    - must record the address of the offending instruction
  - returns control to user
  - must save & restore user state

Two Types of Exceptions

- Interrupts
  - caused by external events
  - asynchronous to program execution
  - may be handled between instructions
  - simply suspend and resume user program

Two Types of Exceptions (cont.)

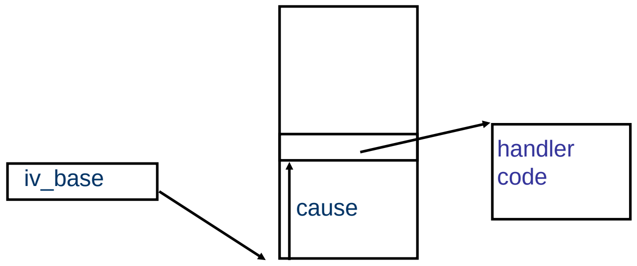
- Traps
  - caused by internal events
    - exceptional conditions (overflow)
    - errors (parity)
    - faults (non-resident page)
  - synchronous to program execution
  - condition must be remedied by the handler
  - instruction may be retried or simulated and program continued or program may be aborted

MIPS Convention

Type of event	*From where?*	*MIPS terminology*
I/O device request	**External**	**Interrupt**
Invoke OS from user program	**Internal**	**Exception**
Arithmetic overflow	**Internal**	**Exception**
Using an undefined instruction	**Internal**	**Exception**

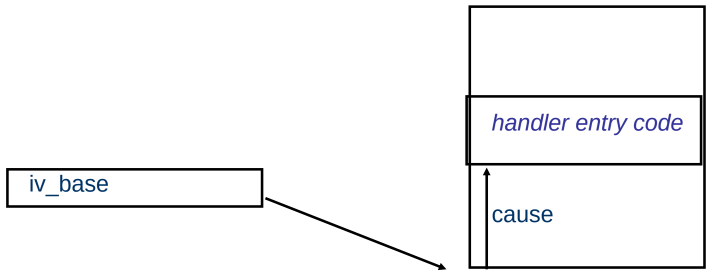
Addressing the Exception Handler

- Traditional Approach: Interupt Vector
  - PC <- MEM[IV_base + cause || 00]
  - E.g., Vax, 80x86



Addressing the Exception Handler (cont.)

- RISC Handler Table
  - PC <– IV_base + cause || 0000
  - saves state and jumps
  - E.g., Sparc



Addressing the Exception Handler (cont.)

- MIPS Approach: fixed entry
  - PC <- EXC_addr

Saving State

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

Additions to MIPS ISA to support Exceptions?

- EPC–a 32-bit register used to hold the address of the affected instruction
- 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

Additions to MIPS ISA to support Exceptions? (cont.)

- BadVAddr - register contained memory address at which memory reference occurred
- Status - interrupt mask and enable bits
- Control signals to write EPC , Cause, BadVAddr, and Status

Additions to MIPS ISA to support Exceptions? (cont.)

- Be able to write exception address into PC, increase mux to add as input 01000000 00000000 00000000 01000000$_{two}$ (8000 0080$_{hex}$)
- May have to undo PC = PC + 4, since want EPC to point to offending instruction (not its successor); PC = PC - 4

Precise Interrupts

- Precise => state of the machine is preserved as if program executed upto the offending instruction
  - Same system code will work on different implementations of the architecture
  - Difficult in the presence of pipelining, out-ot-order execution, ...
  - MIPS takes this position

Precise Interrupts (cont.)

- 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

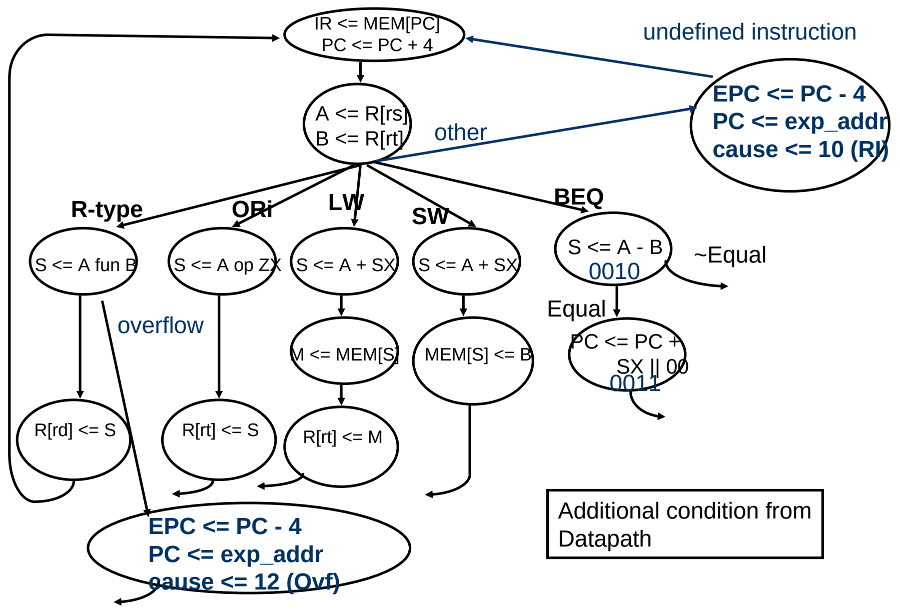
How Control Detects Exceptions in our FSM

- 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.

How Control Detects Exceptions in our FSM (cont.)

- Arithmetic overflow
  - Chapter 4 (HW/SW book) included logic in the ALU to detect overflow, and a signal called Overflow is provided as an output from the ALU.
  - This signal is used in the modified finite state machine to specify an additional possible next state

Modification to the Control Specification



Summary

- Specialize state-diagrams easily captured by microsequencer
  - simple increment & “branch” fields
  - datapath control fields
- Control design reduces to Microprogramming
- Exceptions are the hard part of control

Summary (cont.)

- Need to find convenient place to detect exceptions and to branch to state or microinstruction that saves PC and invokes the operating system
- As we get pipelined CPUs that support page faults on memory accesses which means that the instruction cannot complete AND you must be able to restart the program at exactly the instruction with the exception, it gets even harder

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 …

Summary: Microprogramming one inspiration for RISC (cont.)

- 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?