# Chapter Four – I (4/4)

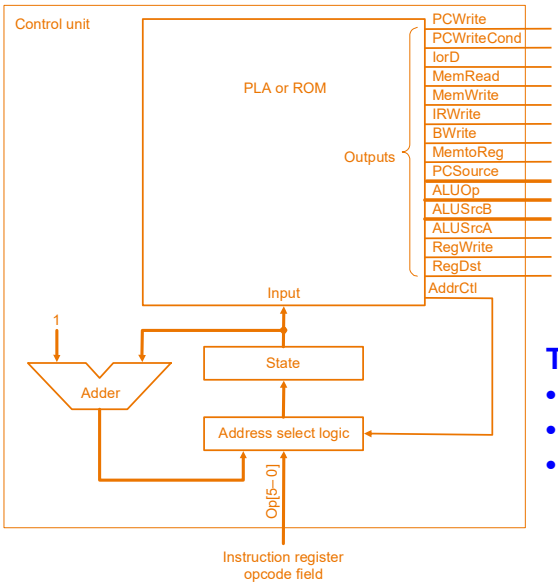


## Microprogramming

- Control is the hard part of processor design
  - Datapath is fairly regular and well-organized
  - Memory is highly regular
  - Control is irregular and global
- Microprogramming:
  - A Particular Strategy for Implementing the Control Unit of a processor by "programming" at the level of register transfer operations
- Microarchitecture:
  - Logical structure and functional capabilities of the hardware as seen by the microprogrammer

# Sequencer-based control unit

- **Complex instructions: the "next state" is often current state + 1**

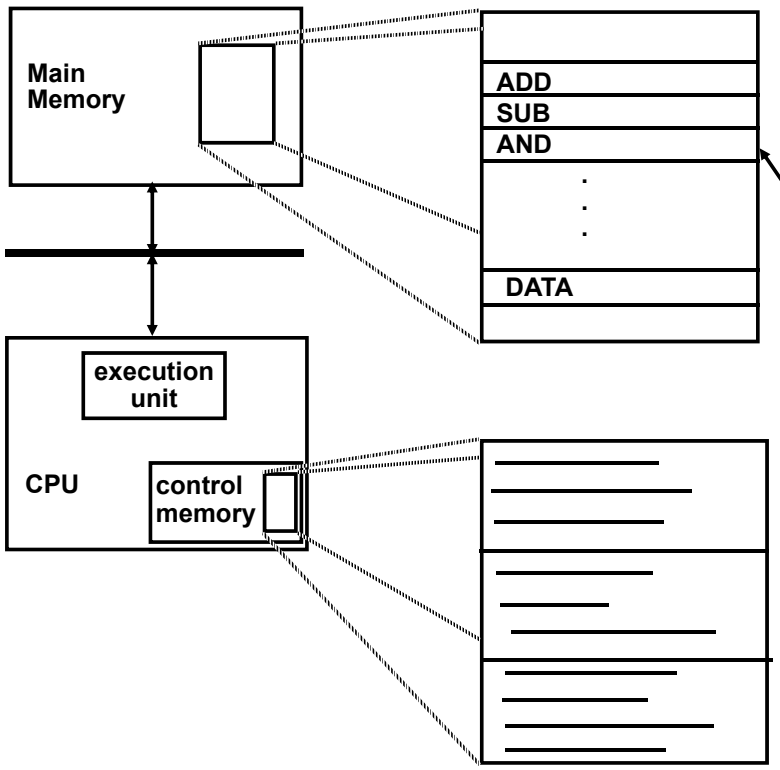



Multicycle
Datapath

**Types of "branching"**
- **Set state to 0**
- **Dispatch (state 1)**
- **Use incremented state number**



# "Macroinstruction" Interpretation




**User program plus Data**

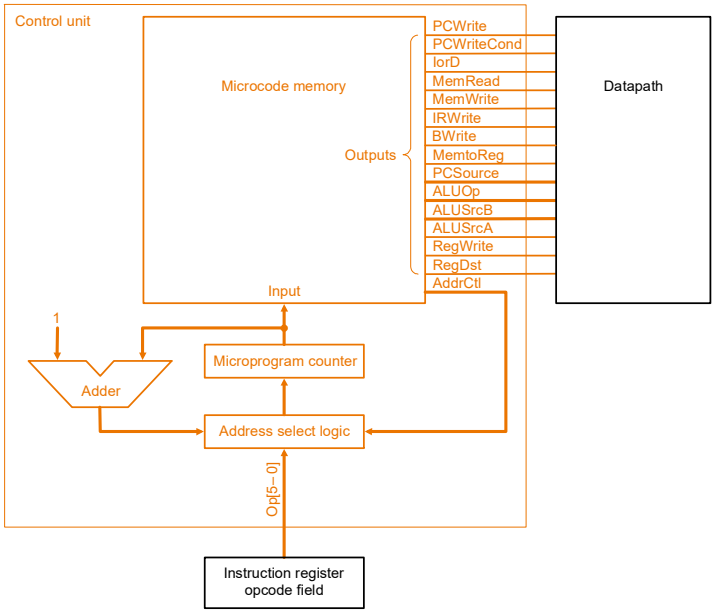
**this can change!**

**one of these is mapped into one of these**

**AND microsequence**

**e.g., Fetch**
**Calc Operand Addr**
**Fetch Operand(s)**
**Calculate**
**Save Answer(s)**

# Microprogramming




- **What are the "microinstructions" ?**



# 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**

# 1&2) List of control signals, grouped into fields

**Single Bit Control**

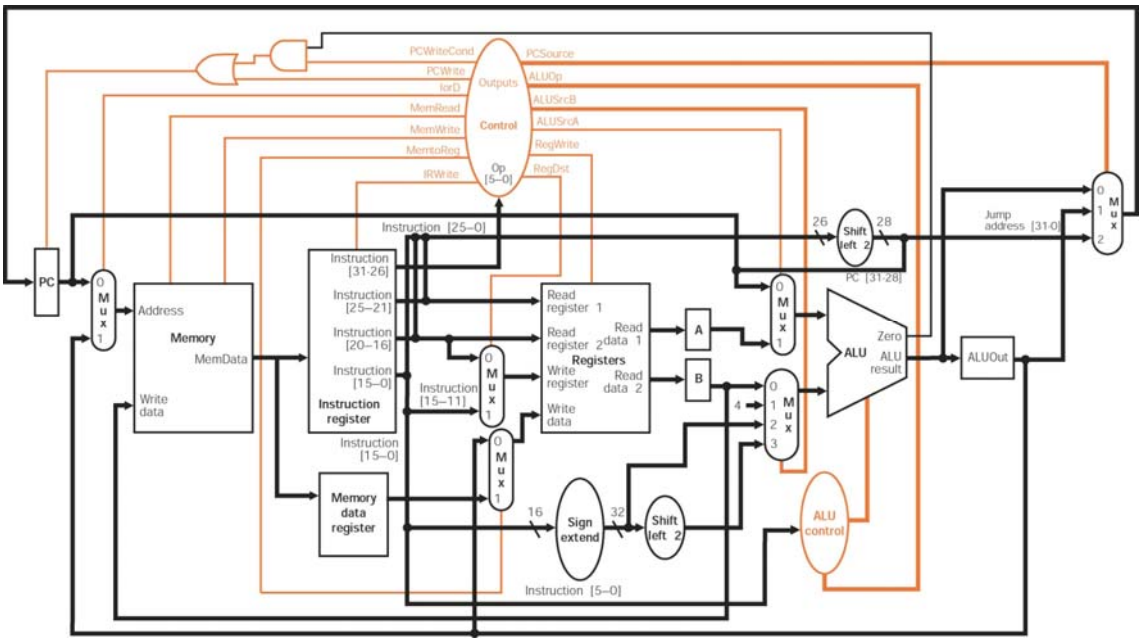
| *Signal name* | *Effect when deasserted* | *Effect when asserted* |
|---|---|---|
| ALUSelA | 1st ALU operand = PC | 1st ALU operand = Reg[rs] |
| RegWrite | None | Reg. is written |
| MemtoReg | Reg. write data input = ALU | Reg. write data input = memory RegDst |
| | Reg. dest. no. = rt | Reg. dest. no. = rd |
| TargetWrite | None | Target reg. = ALU |
| MemRead | None | Memory at address is read |
| MemWrite | None | Memory at address is written |
| IorD | Memory address = PC | Memory address = ALU |
| IRWrite | None | IR = Memory |
| PCWrite | None | PC = PCSource |
| PCWriteCond | None | IF ALUzero then PC = PCSource |

**Multiple Bit Control**

| *Signal name* | *Value* | *Effect* |
|---|---|---|
| ALUOp | 00 | ALU adds |
| | 01 | ALU subtracts |
| | 10 | ALU does function code |
| | 11 | ALU does logical OR |
| ALUSelB | 000 | 2nd ALU input = Reg[rt] |
| | 001 | 2nd ALU input = 4 |
| | 010 | 2nd ALU input = sign extended IR[15-0] |
| | 011 | 2nd ALU input = sign extended, shift left 2 IR[15-0] |
| | 100 | 2nd ALU input = zero extended IR[15-0] |
| PCSource | 00 | PC = ALU |
| | 01 | PC = Target |
| | 10 | PC = PC+4[29-26] : IR[25–0] << 2 |



# Multicycle Datapath with Control Lines

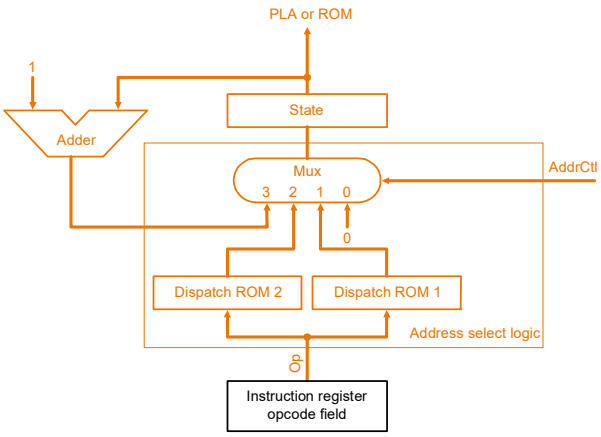


# Start with list of control signals, cont'd

- For next state function (next microinstruction address), use Sequencer-based control unit from last lecture
  - Called "microPC" or "μPC" vs. state register

- Signal Sequencing

| Value | Effect |
|---|---|
| 00 | Next μaddress = 0 |
| 01 | Next μaddress = dispatch ROM 1 |
| 10 | Next μaddress = dispatch ROM 2 |
| 11 | Next μaddress = μaddress + 1 |



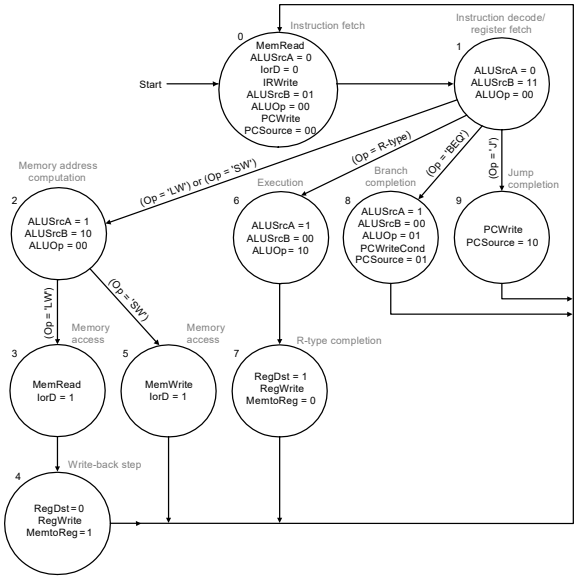



# Details

| Dispatch ROM 1 | | |
|---|---|---|
| Op | Opcode name | Value |
| 000000 | R-format | 0110 |
| 000010 | jmp | 1001 |
| 000100 | beq | 1000 |
| 100011 | lw | 0010 |
| 101011 | sw | 0010 |

| Dispatch ROM 2 | | |
|---|---|---|
| Op | Opcode name | Value |
| 100011 | lw | 0011 |
| 101011 | sw | 0101 |

| State number | Address-control action | Value of AddrCtl |
|---|---|---|
| 0 | Use incremented state | 3 |
| 1 | Use dispatch ROM 1 | 1 |
| 2 | Use dispatch ROM 2 | 2 |
| 3 | Use incremented state | 3 |
| 4 | Replace state number by 0 | 0 |
| 5 | Replace state number by 0 | 0 |
| 6 | Use incremented state | 3 |
| 7 | Replace state number by 0 | 0 |
| 8 | Replace state number by 0 | 0 |
| 9 | Replace state number by 0 | 0 |



# 3) Microinstruction Format

- **Unencoded (Horizontal) vs. Encoded (Vertical) fields**

| Field Name | Width | | Control Signals Set |
|---|---|---|---|
| | wide | narrow | |
| ALU Ctrl | 3 | 2 | ALUOp |
| SRC1 | 2 | 1 | ALUSrcA |
| SRC2 | 4 | 2 | ALUSrcB |
| Reg Ctrl | 3 | 2 | RegWrite, MemtoReg, RegDst. |
| Memory | 3 | 2 | MemRead, MemWrite, IorD |
| PCWrite Ctrl | 3 | 2 | PCWrite, PCWriteCond, PCSource |
| Sequencing | 4 | 2 | AddrCtl |
| Total width | 22 | 13 | bits |



# 4) Legend of Fields and Symbolic Names

| Field name | Value | Signals active | Comment |
|---|---|---|---|
| ALU control | Add | ALUOp = 00 | Cause the ALU to add. |
| | Subt | ALUOp = 01 | Cause the ALU to subtract; this implements the compare for branches. |
| | Func code | ALUOp = 10 | Use the instruction's function code to determine ALU control. |
| SRC1 | PC | ALUSrcA = 0 | Use the PC as the first ALU input. |
| | A | ALUSrcA = 1 | Register A is the first ALU input. |
| SRC2 | B | ALUSrcB = 00 | Register B is the second ALU input. |
| | 4 | ALUSrcB = 01 | Use 4 as the second ALU input. |
| | Extend | ALUSrcB = 10 | Use output of the sign extension unit as the second ALU input. |
| | Extshft | ALUSrcB = 11 | Use the output of the shift-by-two unit as the second ALU input. |
| Register control | Read | | Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B. |
| | Write ALU | RegWrite, RegDst = 1, MemtoReg = 0 | Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data. |
| | Write MDR | RegWrite, RegDst = 0, MemtoReg = 1 | Write a register using the rt field of the IR as the register number and the contents of the MDR as the data. |
| Memory | Read PC | MemRead, IorD = 0 | Read memory using the PC as address; write result into IR (and the MDR) |
| | Read ALU | MemRead, IorD = 1 | Read memory using the ALUOut as address; write result into MDR. |
| | Write ALU | MemWrite, IorD = 1 | Write memory using the ALUOut as address, contents of B as the data. |
| PC write control | ALU | PCSource = 00 PCWrite | Write the output of the ALU into the PC. |
| | ALUOut-cond | PCSource = 01, PCWriteCond | If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut. |
| | jump address | PCSource = 10, PCWrite | Write the PC with the jump address from the instruction. |
| Sequencing | Seq | AddrCtl = 11 | Choose the next microinstruction sequentially. |
| | Fetch | AddrCtl = 00 | Go to the first microinstruction to begin a new instruction. |
| | Dispatch 1 | AddrCtl = 01 | Dispatch using the ROM 1. |
| | Dispatch 2 | AddrCtl = 10 | Dispatch using the ROM 2. |

# Microprogramming

- **A specification methodology**
  - **appropriate if hundreds of opcodes, modes, cycles, etc.**
  - **signals specified symbolically using microinstructions**

| Label | ALU control | SRC1 | SRC2 | Register control | Memory | PCWrite control | Sequencing |
|---|---|---|---|---|---|---|---|
| Fetch | Add | PC | 4 | | Read PC | ALU | Seq |
| | Add | PC | Extshft | Read | | | Dispatch 1 |
| Mem1 | Add | A | Extend | | | | Dispatch 2 |
| LW2 | | | | | Read ALU | | Seq |
| | | | | Write MDR | | | Fetch |
| SW2 | | | | | Write ALU | | Fetch |
| Rformat1 | Func code | A | B | | | | Seq |
| | | | | Write ALU | | | Fetch |
| BEQ1 | Subt | A | B | | | ALUOut-cond | Fetch |
| JUMP1 | | | | | | Jump address | Fetch |

- ***Will two implementations of the same architecture have the same microcode?***
- ***What would a microassembler do?***



# 5) Maximally vs. Minimally Encoded

- **No encoding:**
  - **1 bit for each datapath operation**
  - **faster, requires more memory (logic)**
  - **used for Vax 780 — an astonishing 400K of memory!**
- **Lots of encoding:**
  - **send the microinstructions through logic to get control signals**
  - **uses less memory, slower**
- **Historical context of CISC:**
  - **Too much logic to put on a single chip with everything else**
  - **Use a ROM (or even RAM) to hold the microcode**
  - **It's easy to add new instructions**

# 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



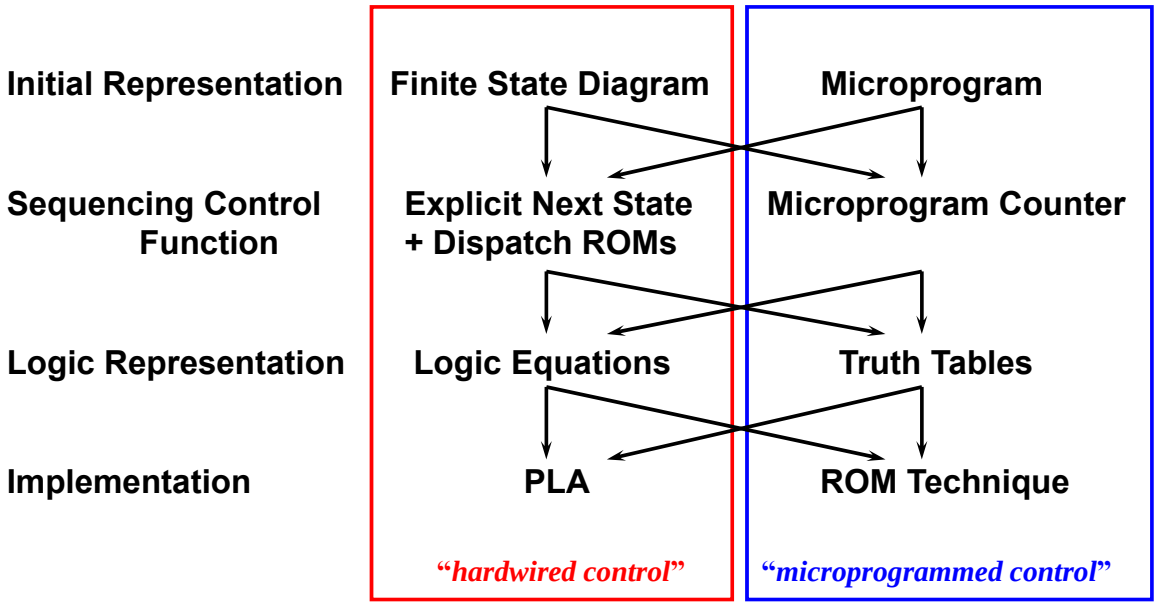
# Historical Perspective

- In the ‘60s and ‘70s microprogramming was very important for implementing machines
- This led to more sophisticated ISAs and the VAX
- In the ‘80s RISC processors based on pipelining became popular
- Pipelining the microinstructions is also possible!
- Implementations of IA-32 architecture processors since 486 use:
  - “hardwired control” for simpler instructions
    (few cycles, FSM control implemented using PLA / random logic)
  - “microcoded control” for more complex instructions
    (large numbers of cycles, central control store)

- The IA-64 architecture uses a RISC-style ISA and can be implemented without a large central control store

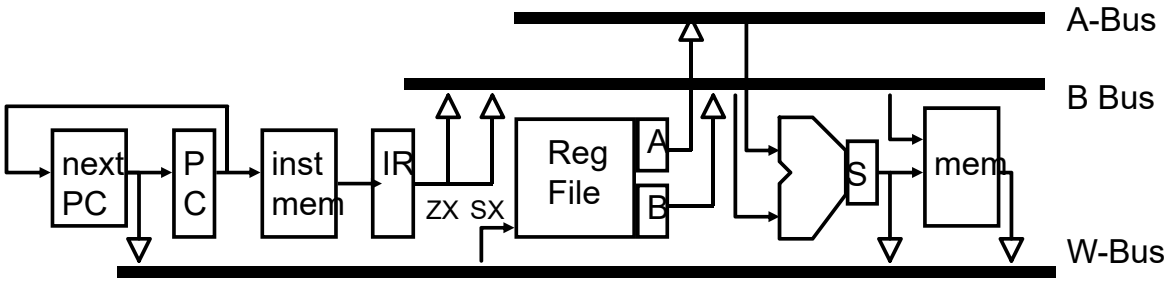
# Overview of Control

- The control can then be implemented with one of several methods using a structured logic technique.






# An Alternative DataPath: "Princeton" Organization



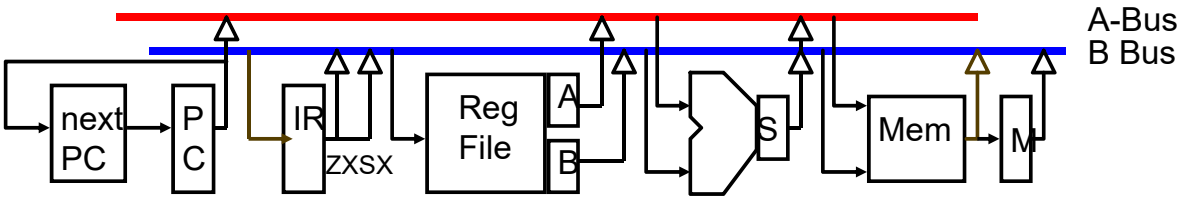

- In each clock cycle, each Bus can be used to transfer from one source
- μ-instruction can simply contain B-Bus and W-Dst fields
- Single memory for instruction and data access

- In this case our state diagram does not change
  - several additional control signals
  - must ensure each bus is only driven by one source on each cycle

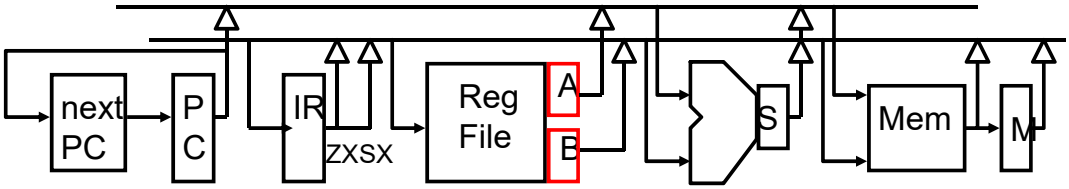
# What about a 2-Bus Microarchitecture (datapath)?

- **2-Bus Microarchitecture**

Instruction Fetch



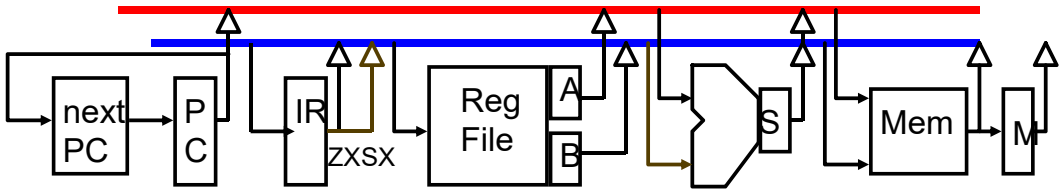

Decode / Operand Fetch



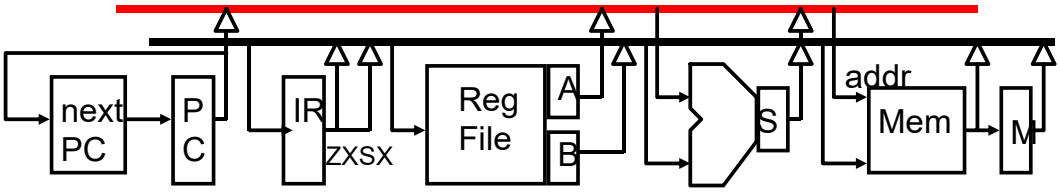



# Operations of 2-Bus Microarchitecture

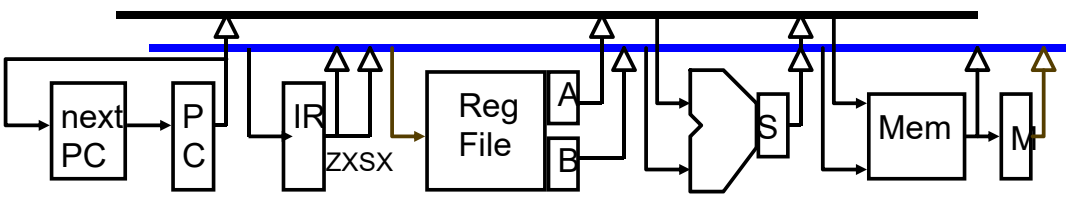
Execute



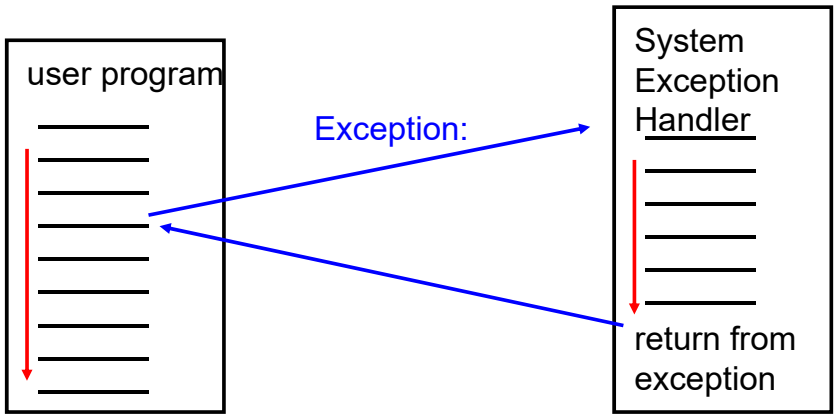

Mem




Write-back



# Exceptions




normal control flow: sequential, jumps, branches, calls, returns

- **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**
- **Allows constuction of a "user virtual machine"**



# 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**

# Two Types of Exceptions

- **Interrupts**
  - **caused by external events**
  - **asynchronous to program execution**
  - **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.**

| Type of event | where? | MIPS term. |
|---|---|---|
| I/O device request | External | Interrupt |
| Invoke OS from user program | Internal | Exception |
| Arithmetic overflow | Internal | Exception |
| Using an undefined instruction | Internal | Exception |
| Hardware malfunctions | Either Exception or Interrupt | |

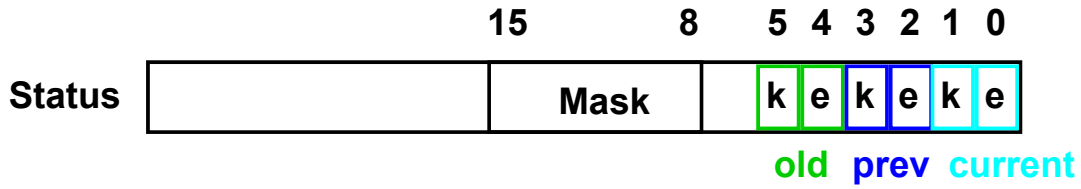
# Additions to MIPS ISA to support Exceptions?

- 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. To simplify the discussion, assume
  - undefined instruction=0
  - arithmetic overflow=1
- Status - interrupt mask and enable bits and determines what exceptions can occur.

- Control signals to write EPC , Cause, and Status
- Be able to write exception address into PC, increase mux set PC to exception address ($C000\ 0000_{hex}$).
- May have to undo PC ← PC + 4, since want EPC to point to offending instruction (not its successor); PC ← PC - 4

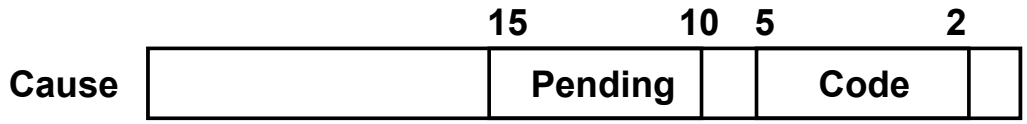


# Details of the Status register




- 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 the Cause register




- 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

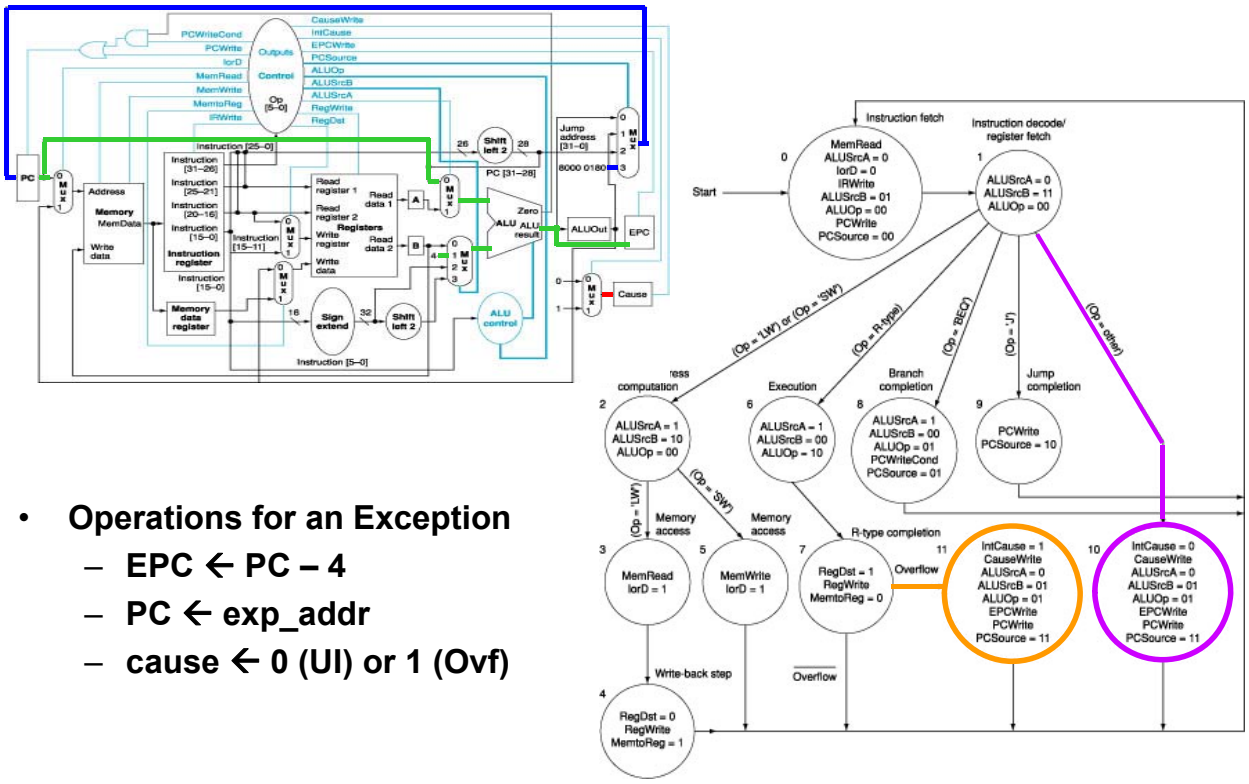


# How Control Detects Exceptions in our FSD

- 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 – 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

- 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

# Modification to the Control Specification




- **Operations for an Exception**
  - **EPC ← PC – 4**
  - **PC ← exp_addr**
  - **cause ← 0 (UI) or 1 (Ovf)**