REP MOVNS (DEST SRC)

IF AddressSize = 16
    THEN
        Use CX for CountReg;
    ELSE IF AddressSize = 64 and REX.W used
        THEN Use RCX for CountReg; Fl;
    ELSE
        Use ECX for CountReg;
    Fl;
WHILE CountReg ≠ 0
    DO
        Service pending interrupts (if any);
        Execute associated string instruction;
        CountReg ← (CountReg - 1);
        IF CountReg = 0
            THEN exit WHILE loop; Fl;
        IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
            or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
            THEN exit WHILE loop; Fl;
    OD;

How many instructions does this take in ARM and MIPS?
Small Semantic Gap Examples in VAX

- **FIND FIRST**
  - Find the first set bit in a bit field
  - Helps OS resource allocation operations

- **SAVE CONTEXT, LOAD CONTEXT**
  - Special context switching instructions

- **INSQUEUE, REMQUEUE**
  - Operations on doubly linked list

- **INDEX**
  - Array access with bounds checking

- **STRING Operations**
  - Compare strings, find substrings, ...  

- **Cyclic Redundancy Check Instruction**

- **EDITPC**
  - Implements editing functions to display fixed format output

Small versus Large Semantic Gap

- **CISC vs. RISC**
  - Complex instruction set computer $\rightarrow$ complex instructions
    - Initially motivated by “not good enough” code generation
  - Reduced instruction set computer $\rightarrow$ simple instructions
    - John Cocke, mid 1970s, IBM 801
      - Goal: enable better compiler control and optimization

- **RISC motivated by**
  - Memory stalls (no work done in a complex instruction when there is a memory stall?)
    - When is this correct?
  - Simplifying the hardware $\rightarrow$ lower cost, higher frequency
  - Enabling the compiler to optimize the code better
    - Find fine-grained parallelism to reduce stalls
How High or Low Can You Go?

- Very large semantic gap
  - Each instruction specifies the complete set of control signals in the machine
  - Compiler generates control signals
  - Open microcode (John Cocke, circa 1970s)
    - Gave way to optimizing compilers

- Very small semantic gap
  - ISA is (almost) the same as high-level language
  - Java machines, LISP machines, object-oriented machines, capability-based machines
A Note on ISA Evolution

- ISAs have evolved to reflect/satisfy the concerns of the day

- Examples:
  - Limited on-chip and off-chip memory size
  - Limited compiler optimization technology
  - Limited memory bandwidth
  - Need for specialization in important applications (e.g., MMX)

- Use of translation (in HW and SW) enabled underlying implementations to be similar, regardless of the ISA
  - Concept of dynamic/static interface
  - Contrast it with hardware/software interface
Effect of Translation

- One can translate from one ISA to another ISA to change the semantic gap tradeoffs

- Examples
  - Intel’s and AMD’s x86 implementations translate x86 instructions into programmer-invisible microoperations (simple instructions) in hardware
  - Transmeta’s x86 implementations translated x86 instructions into “secret” VLIW instructions in software (code morphing software)

- Think about the tradeoffs
ISA-level Tradeoffs: Instruction Length

- **Fixed length**: Length of all instructions the same
  + Easier to decode single instruction in hardware
  + Easier to decode multiple instructions concurrently
  -- Wasted bits in instructions *(Why is this bad?)*
  -- Harder-to-extend ISA (how to add new instructions?)

- **Variable length**: Length of instructions different (determined by opcode and sub-opcode)
  + Compact encoding *(Why is this good?)*
    Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. *How?*
  -- More logic to decode a single instruction
  -- Harder to decode multiple instructions concurrently

- **Tradeoffs**
  - Code size (memory space, bandwidth, latency) vs. hardware complexity
  - ISA extensibility and expressiveness
  - Performance? Smaller code vs. difficult decode
ISA-level Tradeoffs: Uniform Decode

- **Uniform decode**: Same bits in each instruction correspond to the same meaning
  - Opcode is always in the same location
  - Ditto operand specifiers, immediate values, ...
  - Many “RISC” ISAs: Alpha, MIPS, SPARC
  + Easier decode, simpler hardware
  + Enables parallelism: generate target address before knowing the instruction is a branch
    -- Restricts instruction format (fewer instructions?) or wastes space

- **Non-uniform decode**
  - E.g., opcode can be the 1st-7th byte in x86
  + More compact and powerful instruction format
  -- More complex decode logic
x86 vs. Alpha Instruction Formats

- **x86:**

  ![x86 Instruction Format Diagram]

  - Instruction Prefixes: Up to four prefixes of 1 byte each (optional)
  - Opcode: 1-, 2-, or 3-byte opcode
  - ModR/M: 1 byte (if required)
  - SIB: 1 byte (if required)
  - Displacement: Address displacement of 1, 2, or 4 bytes or none
  - Immediate: Immediate data of 1, 2, or 4 bytes or none

- **Alpha:**

  ![Alpha Instruction Format Diagram]

  - Opcode: RA, RB, Function, RC
  - Number: Disp, RC
MIPS Instruction Format

- **R-type, 3 register operands**
  
<table>
<thead>
<tr>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>6-bit</td>
</tr>
</tbody>
</table>
  
- **I-type, 2 register operands and 16-bit immediate operand**
  
<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>
  
- **J-type, 26-bit immediate operand**
  
<table>
<thead>
<tr>
<th>opcode</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>26-bit</td>
</tr>
</tbody>
</table>

- **Simple Decoding**
  - 4 bytes per instruction, regardless of format
  - must be 4-byte aligned (2 lsb of PC must be 2b’00)
  - format and fields easy to extract in hardware
Figure 4-1: ARM instruction set formats
A Note on Length and Uniformity

- Uniform decode usually goes with fixed length

- In a variable length ISA, uniform decode can be a property of instructions of the same length
  - It is hard to think of it as a property of instructions of different lengths
A Note on RISC vs. CISC

- Usually, ...

- RISC
  - Simple instructions
  - Fixed length
  - Uniform decode
  - Few addressing modes

- CISC
  - Complex instructions
  - Variable length
  - Non-uniform decode
  - Many addressing modes
ISA-level Tradeoffs: Number of Registers

- Affects:
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file

- Large number of registers:
  + Enables better register allocation (and optimizations) by compiler → fewer saves/restores
  -- Larger instruction size
  -- Larger register file size
ISA-level Tradeoffs: Addressing Modes

- Addressing mode specifies how to obtain an operand of an instruction
  - Register
  - Immediate
  - Memory (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, ...)

- More modes:
  + help better support programming constructs (arrays, pointer-based accesses)
  -- make it harder for the architect to design
  -- too many choices for the compiler?
    - Many ways to do the same thing complicates compiler design
x86 vs. Alpha Instruction Formats

- **x86:**

<table>
<thead>
<tr>
<th>Instruction Prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to four prefixes of 1 byte each (optional)</td>
<td>1-, 2-, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes or none</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

- **Alpha:**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>Disp</td>
</tr>
<tr>
<td>RA RB</td>
<td>Disp</td>
</tr>
<tr>
<td>RA RB</td>
<td>Function RC</td>
</tr>
</tbody>
</table>
### Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>Mod</th>
<th>R/M</th>
<th>Value of ModR/M Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX]</td>
<td>00</td>
<td>00</td>
<td>08 18 20 28 30 38</td>
</tr>
<tr>
<td>[ECX]</td>
<td>01</td>
<td>00</td>
<td>03 11 19 21 29 31</td>
</tr>
<tr>
<td>[EDX]</td>
<td>02</td>
<td>00</td>
<td>05 13 21 23 29 33</td>
</tr>
<tr>
<td>[EBX]</td>
<td>03</td>
<td>00</td>
<td>07 15 23 25 33 37</td>
</tr>
<tr>
<td>[ESP]</td>
<td>04</td>
<td>00</td>
<td>09 17 25 27 35 39</td>
</tr>
<tr>
<td>[EBP]</td>
<td>05</td>
<td>00</td>
<td>0B 19 27 29 35 3F</td>
</tr>
<tr>
<td>[ESI]</td>
<td>06</td>
<td>00</td>
<td>0D 1F 27 2F 35 3F</td>
</tr>
<tr>
<td>[EDI]</td>
<td>07</td>
<td>00</td>
<td>0F 1F 2F 3F 3F 3F</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The [-][-] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table.
### Table 2.3. 32-Bit Addressing Forms with the SIB Byte

<table>
<thead>
<tr>
<th>r32 (In decimal) Base</th>
<th>(In binary) Base</th>
<th>Scaled Index</th>
<th>SS</th>
<th>Index</th>
<th>Value of SIB Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX] 000</td>
<td>00</td>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
</tr>
<tr>
<td>[ECX] 001</td>
<td>09</td>
<td>0A</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>[EDX] 010</td>
<td>11</td>
<td>12</td>
<td>16</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>[EBX] 011</td>
<td>18</td>
<td>19</td>
<td>1A</td>
<td>1B</td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>100</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>[EBP] 101</td>
<td>28</td>
<td>29</td>
<td>3A</td>
<td>3B</td>
<td></td>
</tr>
<tr>
<td>[ESI] 110</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>[EDI] 111</td>
<td>38</td>
<td>39</td>
<td>3A</td>
<td>3B</td>
<td></td>
</tr>
</tbody>
</table>

| [EAX*+2]              | 01               | 40           | 41  | 42    | 43                                 |
| [ECX*+2]              | 00               | 48           | 49  | 4A    | 4B                                 |
| [EDX*+2]              | 01               | 50           | 51  | 52    | 53                                 |
| [EBX*+2]              | 01               | 58           | 59  | 5A    | 5B                                 |
| none                  | 100              | 60           | 61  | 62    | 63                                 |
| [EBP*+2]              | 101              | 68           | 69  | 6A    | 6B                                 |
| [ESI*+2]              | 110              | 70           | 71  | 72    | 73                                 |
| [EDI*+2]              | 111              | 78           | 79  | 7A    | 7B                                 |

| [EAX*+4]              | 10               | 80           | 81  | 82    | 83                                 |
| [ECX*+4]              | 00               | 88           | 89  | 8A    | 8B                                 |
| [EDX*+4]              | 01               | 90           | 91  | 92    | 93                                 |
| [EBX*+4]              | 01               | 98           | 99  | 9A    | 9B                                 |
| none                  | 100              | A0           | A1  | A2    | A3                                 |
| [EBP*+4]              | 101              | A8           | A9  | AA    | AB                                 |
| [ESI*+4]              | 110              | B0           | B1  | B2    | B3                                 |
| [EDI*+4]              | 111              | B8           | B9  | BA    | BB                                 |

| [EAX*+8]              | 11               | C0           | C1  | C2    | C3                                 |
| [ECX*+8]              | 00               | C8           | C9  | CA    | CB                                 |
| [EDX*+8]              | 01               | DB           | D1  | D2    | D3                                 |
| [EBX*+8]              | 01               | DB           | DA  | DB    | DC                                 |
| none                  | 100              | E0           | E1  | E2    | E3                                 |
| [EBP*+8]              | 101              | EA           | EB  | EC    | ED                                 |
| [ESI*+8]              | 110              | F0           | F1  | F2    | F3                                 |
| [EDI*+8]              | 111              | F8           | F9  | FA    | FB                                 |

### Notes:
1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:
   - **MOD bits**    **Effective Address**
     - 00     [scaled index] + disp32
     - 01     [scaled index] + disp8 + [EBP]
     - 10     [scaled index] + disp32 + [EBP]
X86 SIB-D Addressing Mode

Figure 3-11. Offset (or Effective Address) Computation

\[
\text{Offset} = \text{Base} + (\text{Index} \times \text{Scale}) + \text{Displacement}
\]

x86 Manual Vol. 1, page 3-22  -- see course resources on website
Also, see Section 3.7.3 and 3.7.5
The following addressing modes suggest uses for common combinations of address components.

- **Displacement** — A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.

- **Base** — A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.

- **Base + Displacement** — A base register and a displacement can be used together for two distinct purposes:
  - As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
  - To access a field of a record: the base register holds the address of the beginning of the record, while the displacement is a static offset to the field.

An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.
• (Index * Scale) + Displacement — This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.

• Base + Index + Displacement — Using two registers together supports either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).

• Base + (Index * Scale) + Displacement — Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.
Other Example ISA-level Tradeoffs

- Condition codes vs. not
- VLIW vs. single instruction
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling
- Cache coherence (hardware vs. software)
- ...

Many ISA features designed to aid programmers
But, complicate the hardware designer’s job

Virtual memory
- vs. overlay programming
- Should the programmer be concerned about the size of code blocks fitting physical memory?

Addressing modes

Unaligned memory access
- Compile/programmer needs to align data
MIPS: Aligned Access

- LW/SW alignment restriction: 4-byte word-alignment
  - not designed to fetch memory bytes not within a word boundary
  - not designed to rotate unaligned bytes into registers
- Provide separate opcodes for the “infrequent” case
  - LWL rd 6(r0) → byte-6 byte-5 byte-4 D
  - LWR rd 3(r0) → byte-6 byte-5 byte-4 byte-3
- LWL/LWR is slower
- Note LWL and LWR still fetch within word boundary
X86: Unaligned Access

- LD/ST instructions automatically align data that spans a “word” boundary
- Programmer/compiler does not need to worry about where data is stored (whether or not in a word-aligned location)

### 4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.
X86: Unaligned Access

![Diagram of X86 memory access]

**Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory**
Aligned vs. Unaligned Access

- Pros of having no restrictions on alignment

- Cons of having no restrictions on alignment

- Filling in the above: an exercise for you...
Implementing the ISA: Microarchitecture Basics
How Does a Machine Process Instructions?

- What does processing an instruction mean?
- Remember the von Neumann model

\[ A = \text{Architectural (programmer visible) state before an instruction is processed} \]

Process instruction

\[ A' = \text{Architectural (programmer visible) state after an instruction is processed} \]

- Processing an instruction: Transforming A to A’ according to the ISA specification of the instruction
The “Process instruction” Step

- ISA specifies abstractly what A’ should be, given an instruction and A
  - It defines an abstract finite state machine where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no “intermediate states” between A and A’ during instruction execution
    - One state transition per instruction

- Microarchitecture implements how A is transformed to A’
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: multiple state transitions per instruction
    - Choice 1: A \rightarrow A’ (transform A to A’ in a single clock cycle)
    - Choice 2: A \rightarrow A+MS1 \rightarrow A+MS2 \rightarrow A+MS3 \rightarrow A’ (take multiple clock cycles to transform A to A’)

31
A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - No intermediate, programmer-invisible state updates

\[
A = \text{Architectural (programmer visible) state at the beginning of a clock cycle}
\]

Process instruction in one clock cycle

\[
A' = \text{Architectural (programmer visible) state at the end of a clock cycle}
\]
A Very Basic Instruction Processing Engine

- Single-cycle machine

- What is the *clock cycle time* determined by?

- What is the *critical path* of the combinational logic determined by?
Remember: Programmer Visible (Architectural) State

- **Memory**: array of storage locations indexed by an address
- **Registers**: given special names in the ISA (as opposed to addresses)
  - general vs. special purpose
- **Program Counter**: memory address of the current instruction

Instructions (and programs) specify how to transform the values of programmer visible state
Single-cycle vs. Multi-cycle Machines

- **Single-cycle machines**
  - Each instruction takes a single clock cycle
  - All state updates made at the end of an instruction’s execution
  - **Big disadvantage:** The slowest instruction determines cycle time → long clock cycle time

- **Multi-cycle machines**
  - Instruction processing broken into multiple cycles/stages
  - State updates can be made during an instruction’s execution
  - Architectural state updates made only at the end of an instruction’s execution
  - **Advantage over single-cycle:** The slowest “stage” determines cycle time

- Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level
Instruction Processing “Cycle”

- Instructions are processed under the direction of a “control unit” step by step.
- Instruction cycle: Sequence of steps to process an instruction
- Fundamentally, there are six phases:
  - Fetch
  - Decode
  - Evaluate Address
  - Fetch Operands
  - Execute
  - Store Result

- Not all instructions require all six stages (see P&P Ch. 4)
Instruction Processing “Cycle” vs. Machine Clock Cycle

- **Single-cycle machine:**
  - All six phases of the instruction processing cycle take a *single machine clock cycle* to complete

- **Multi-cycle machine:**
  - All six phases of the instruction processing cycle can take *multiple machine clock cycles* to complete
  - In fact, *each phase can take multiple clock cycles to complete*
Instruction Processing Viewed Another Way

- Instructions transform Data (AS) to Data’ (AS’)
- This transformation is done by functional units
  - Units that “operate” on data
- These units need to be told what to do to the data

An instruction processing engine consists of two components
- **Datapath**: Consists of hardware elements that deal with and transform data signals
  - functional units that operate on data
  - hardware structures (e.g. wires and muxes) that enable the flow of data into the functional units and registers
  - storage units that store data (e.g., registers)
- **Control logic**: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data
Single-cycle vs. Multi-cycle: Control & Data

- Single-cycle machine:
  - Control signals are generated in the same clock cycle as data signals are operated on
  - Everything related to an instruction happens in one clock cycle

- Multi-cycle machine:
  - Control signals needed in the next cycle can be generated in the previous cycle
  - Latency of control processing can be overlapped with latency of datapath operation

- We will see the difference clearly in microprogrammed multi-cycle microarchitecture
Many Ways of Datapath and Control Design

- There are many ways of designing the data path and control logic
  
- Single-cycle, multi-cycle, pipelined datapath and control

- Single-bus vs. multi-bus datapaths
  - See your homework 2 question

- Hardwired/combinational vs. microcoded/microprogrammed control
  - Control signals generated by combinational logic versus
  - Control signals stored in a memory structure

- Control signals and structure depend on the datapath design
Flash-Forward: Performance Analysis

- Execution time of an instruction
  - \( \{\text{CPI}\} \times \{\text{clock cycle time}\} \)

- Execution time of a program
  - Sum over all instructions \([\{\text{CPI}\} \times \{\text{clock cycle time}\}]\)
  - \(\{\text{# of instructions}\} \times \{\text{Average CPI}\} \times \{\text{clock cycle time}\}\)

- Single cycle microarchitecture performance
  - \(\text{CPI} = 1\)
  - \(\text{Clock cycle time} = \text{long}\)

- Multi-cycle microarchitecture performance
  - \(\text{CPI} = \text{different for each instruction}\)
    - Average CPI → hopefully small
  - \(\text{Clock cycle time} = \text{short}\)

Now, we have two degrees of freedom to optimize independently
A Single-Cycle Microarchitecture

A Closer Look
Remember...

- Single-cycle machine
Let’s Start with the State Elements

- Data and control inputs

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
For Now, We Will Assume

- “Magic” memory and register file
- Combinational read
  - output of the read data port is a combinational function of the register file contents and the corresponding read select port
- Synchronous write
  - the selected register is updated on the positive edge clock transition when write enable is asserted
  - Cannot affect read output in between clock edges
  - Can affect read output at clock edges (but who cares?)
- Single-cycle, synchronous memory
  - Contrast this with memory that tells when the data is ready
  - i.e., Ready bit: indicating the read or write is done
Instruction Processing

- 5 generic steps (P&H)
  - Instruction fetch (IF)
  - Instruction decode and register operand fetch (ID/RF)
  - Execute/Evaluate memory address (EX/AG)
  - Memory operand fetch (MEM)
  - Store/writeback result (WB)

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
What Is To Come: The Full Datapath

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Single-Cycle Datapath for
Arithmetic and Logical Instructions
R-Type ALU Instructions

- Assembly (e.g., register-register signed addition)
  \[ \text{ADD } rd_{reg} \text{ rs}_{reg} \text{ rt}_{reg} \]

- Machine encoding

```
<table>
<thead>
<tr>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>6-bit</td>
</tr>
</tbody>
</table>
```

- Semantics

\[
\begin{align*}
\text{if } \text{MEM}[\text{PC}] &= \text{ADD } rd \text{ rs } rt \\
GPR[rd] &\leftarrow GPR[rs] + GPR[rt] \\
\text{PC} &\leftarrow \text{PC} + 4
\end{align*}
\]
ALU Datapath

if MEM[PC] == ADD rd rs rt
GPR[rd] ← GPR[rs] + GPR[rt]
PC ← PC + 4

Combination state update logic

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
I-Type ALU Instructions

- **Assembly** (e.g., register-immediate signed additions)
  \[
  \text{ADDI } rt_{\text{reg}} \text{ } rs_{\text{reg}} \text{ } \text{immediate}_{16}
  \]

- **Machine encoding**

<table>
<thead>
<tr>
<th>ADDI</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

  - **I-type**

- **Semantics**

  if \( \text{MEM}[\text{PC}] = \text{ADDI } rt \text{ rs } \text{immediate} \)
  
  \[
  \text{GPR}[rt] \leftarrow \text{GPR}[rs] + \text{sign-extend (immediate)}
  \]
  
  \[
  \text{PC} \leftarrow \text{PC} + 4
  \]
Datapath for R and I-Type ALU Insts.

if MEM[PC] == ADDI rt rs immediate
  GPR[rt] ← GPR[rs] + sign-extend (immediate)
  PC ← PC + 4

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Single-Cycle Datapath for
Data Movement Instructions
Load Instructions

- **Assembly** (e.g., load 4-byte word)
  \[
  \text{LW } r_t \text{ } \text{reg offset}_{16} \text{ (base } \text{reg})
  \]

- **Machine encoding**
  
  \[
  \begin{array}{cccc}
  \text{LW} & \text{base} & \text{rt} & \text{offset} \\
  6\text{-bit} & 5\text{-bit} & 5\text{-bit} & 16\text{-bit} \\
  \end{array}
  \]

- **Semantics**
  
  if \( \text{MEM}[\text{PC}] == \text{LW } r_t \text{ offset}_{16} \text{ (base)} \)
  \[
  \begin{align*}
  \text{EA} &= \text{sign-extend}(\text{offset}) + \text{GPR}[\text{base}] \\
  \text{GPR}[r_t] &\leftarrow \text{MEM}[\text{translate}(\text{EA}) ] \\
  \text{PC} &\leftarrow \text{PC} + 4
  \end{align*}
  \]
LW Datapath

if MEM[PC] == LW rt offset_{16} (base)
EA = sign-extend(offset) + GPR[base]
GPR[rt] ← MEM[ translate(EA) ]
PC ← PC + 4
Store Instructions

- Assembly (e.g., store 4-byte word)
  \[ \text{SW } r_{\text{reg}} \text{ offset}_{16} (\text{base}_{\text{reg}}) \]

- Machine encoding

<table>
<thead>
<tr>
<th>SW</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

- Semantics
  
  if \( \text{MEM}[\text{PC}] == \text{SW } r_{\text{t}} \text{ offset}_{16} (\text{base}) \)
  
  \[ \text{EA} = \text{sign-extend}(\text{offset}) + \text{GPR}[\text{base}] \]
  
  \[ \text{MEM}[\text{translate}(\text{EA})] \leftarrow \text{GPR}[r_{\text{t}}] \]
  
  \[ \text{PC} \leftarrow \text{PC} + 4 \]
if MEM[PC] == SW rt offset_{16} (base) 
EA = sign-extend(offset) + GPR[base] 
MEM[ translate(EA) ] ← GPR[rt] 
PC ← PC + 4
Load-Store Datapath

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Datapath for Non-Control-Flow Insts.

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Single-Cycle Datapath for Control Flow Instructions
Unconditional Jump Instructions

- **Assembly**
  
  J immediate

- **Machine encoding**
  
<table>
<thead>
<tr>
<th>J</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>26-bit</td>
</tr>
</tbody>
</table>

  J-type

- **Semantics**

  if MEM[PC] == J immediate

  target = { PC[31:28], immediate, 2’ b00 }

  PC ← target
Unconditional Jump Datapath

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

if MEM[PC]==J immediate26
  PC = { PC[31:28], immediate26, 2’b00 }

What about JR, JAL, JALR?
Conditional Branch Instructions

- **Assembly (e.g., branch if equal)**
  \[
  \text{BEQ } rs_{\text{reg}} \ rt_{\text{reg}} \ \text{immediate}_{16}
  \]

- **Machine encoding**

- **Semantics (assuming no branch delay slot)**
  
  if MEM[PC] == \text{BEQ } rs \ rt \ \text{immediate}_{16}
  
  target = PC + 4 + \text{sign-extend}(\text{immediate}) \times 4
  
  if GPR[rs] == GPR[rt] then  \text{PC} \leftarrow \text{target}
  
  else  \text{PC} \leftarrow \text{PC} + 4
Conditional Branch Datapath (For You to Fix)

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**

How to uphold the delayed branch semantics?
Putting It All Together

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**