Encoding The Instruction Set

- Previous design decisions affect the size of the instruction
- Size of the compiled program
- Ease of decoding

Encoding The opcode Field

- Depends on whether every operation can be combined with every addressing mode
- If it can separate address specifier is needed for each operand
- If it can’t opcode can signify the addressing mode

Instruction Set Design Trade-offs

- More registers are better for compiler optimization
- More addressing modes bring faster operation
- More registers and addressing modes make instructions longer
- Shorter instructions and instructions with similar CPI are better for pipelining

Instruction Formats

- Variable
  - As few bits as possible to encode instructions
  - Decoding might be complicated

- Fixed
  - Always same number bits to encode instructions
  - Easy decoding

- Hybrid
CISC vs. RISC

- Complex Instruction Set Computer (CISC)
  - Instructions are highly specialized
  - Support for a variety of instructions, addressing modes, etc.
  - Different CPI and instruction size
- Reduced Instruction Set Computer (RISC)
  - Short, simple instructions, support for a few addressing modes
  - More complex instructions must be programmed
  - Same low CPI

Reduced Code Size

- Important for embedded applications
- Design hybrid version of instruction set with both 16-bit and 32-bit instructions
  - 16-bit instructions are simpler, support fewer operations and addressing modes
- Compressed code
  - Instruction cache contains full instructions
  - Memory contains compressed instructions
  - On cache miss, instruction is fetched and decompressed

Role of Compilers

- Compiler generates object code in machine language from the high-level language such as C
- Instruction set is compiler’s target
- In addition to generating the code, compiler optimizes the code to make it:
  - Shorter – 25% to 90%
  - Faster
  - Susceptible to pipelining

Compilation

- Compiler makes two to four passes through the code:
  - In each pass it performs one of the optimizations
  - The optimizations are optional and may be skipped to achieve faster compilation
  - This is still sequential – if compiler could go back and repeat steps it might discover better optimizations but this would increase time and complexity
- Compiler design goals:
  - Correctness
  - Speed of compilation

Front End

- Transforms high-level language into common intermediate representation
- When a new language becomes popular only front-end needs to be rewritten
High-Level Optimizations

- Transform the code to take advantage of parallelism and increase speed of execution:
  - Loop unrolling – expand body of the loop to encompass several iterations thus eliminating number of conditional branches
    ```
    for (i = 0; i < 100; i++)
    {
        g ();
    }
    ```
    ```
    for (i = 0; i < 100; i+=2)
    {
        g ();
        g ();
    }
    ```
  - Procedure inlining – eliminates conditional branch
  - Prefetch insertion – prefetch array references in loops

Global Optimizations

- Global and local optimizations
  - Global common subexpression elimination – locates several expressions that compute same value and replaces the second with the temporary variable
  - Local optimization is done only within basic block
  - Copy propagation: if A=X replace all later references to A with X
  - Register allocation
    - Goal is to allocate most accessed variables to registers
    - Since number of registers is limited, must choose variables that do not conflict and assign them to same register

Graph Coloring

- Assign variables to graph nodes
- Variables A and B conflict if A is accessed both before and after B is accessed (or at the same time)
- Undirected edge represents a conflict between nodes
- Color nodes so that no two adjacent nodes have the same color
- If number of colors is less than number of registers then all variables can be assigned to registers
- NP complete problem but there are heuristics that run in near-linear time

Graph Coloring Example

- A=B+C
- C=D+E
- B=A-F
Graph Coloring Example

A = B + C
C = D + E
B = A - F

We need 5 registers

Code Generator
- Takes advantage of design features of a specific architecture
- Reorder instructions to improve pipeline performance
- Replace multiplication with addition and shifts

Compilers and Data Allocation
- Within a program data is allocated at:
  - Stack – local scalar variables and activation records
  - Global area – global variables and constants
  - Heap – dynamic objects accessed with pointers
- Stack variables best for register allocation
- Heap variables should not be allocated into registers
- Aliased variables should not be allocated into registers
- Global variables should be allocated if they are accessed frequently

Compilers and Architecture
- Provide regularity: operations, data typed and addressing modes should be orthogonal
- E.g., do not restrict the use of GPR
- Provide primitives not solutions – special features that match kernels or high-level languages are often unusable
- Simplify trade-offs among alternatives
  - Compilers strive to generate efficient code
  - Specify benefits and costs of each alternative
  - Make use of everything that is known at compile time

MIPS
- Follows design recommendations from previous discussions
- Simple load-store instruction set
- Designed for pipelining efficiency – fixed instruction set encoding
- Efficiency as compiler target
- Very popular
Registers for MIPS
- 32 general purpose registers 64-bit long R0…R31
- 32 floating point registers 64-bit long F0…F31
- Value of R0 is always 0
- A few special registers can be transferred to and from GPR
- Instructions for moving contents between GPR and FPR

Data Types for MIPS
- 8-, 16-, 32- and 64-bit integers
- 32- and 64-bit floating point
- Operands smaller than 64 are either 0 or sign-extended and loaded into 64-bit registers
- All operations are performed on 64-bit registers

Addressing Modes for MIPS
- Immediate
  - Add R4, #3
    \[ \text{Reg}[R4] \leftarrow \text{Reg}[R4] + 3 \]
- Displacement
  - Add R4, 100(R1)
    \[ \text{Reg}[R4] \leftarrow \text{Reg}[R4] + \text{Mem}[\text{Reg}[R1]+100] \]
  - If we use 0 for displacement we get register indirect mode
  - If we use R0 for register we get memory direct (absolute) mode
- Since only two addressing modes exist they are encoded in opcode

Instruction Format for MIPS

I-type instructions

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
</tbody>
</table>

- Loads and stores of bytes, half-words, words, double words
- \( rt \leftarrow rs \) op immediate
- Conditional branches
  - \( PC \leftarrow f(rs, \text{immediate}) \)
  - \( PC \leftarrow f(rs, rt, \text{immediate}) \)
- Jump register
  - \( PC \leftarrow rs \)
- ALU operations with register and immediate
  - \( rt \leftarrow rs \) op immediate

R-type instructions

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>

- ALU operations with registers
  - \( rd \leftarrow rs \) op rt
- Read/write special registers
- Move data between GPR and FPR
Instruction Format for MIPS

J-type instructions

<table>
<thead>
<tr>
<th>opcode</th>
<th>Offset to PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>26</td>
</tr>
</tbody>
</table>

Jump, trap, return from exception

PC ← PC + offset

Load/Store Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, 30(R2)</td>
<td>Load double word, store double word</td>
</tr>
<tr>
<td>SD 1000(R0)</td>
<td>Use either of four addressing modes</td>
</tr>
<tr>
<td>LW R1, 60(R3)</td>
<td>Load word, store word</td>
</tr>
<tr>
<td>LWU</td>
<td>Load unsigned word</td>
</tr>
<tr>
<td>SW 1000(R0)</td>
<td>Use either of four addressing modes</td>
</tr>
<tr>
<td>LH R1, 40(R3)</td>
<td>Load half word, store half word</td>
</tr>
<tr>
<td>LHU</td>
<td>Load unsigned half word</td>
</tr>
<tr>
<td>SH 1000(R0)</td>
<td>Use either of four addressing modes</td>
</tr>
<tr>
<td>LB R1, 10(R1)</td>
<td>Load byte, store byte</td>
</tr>
<tr>
<td>LBU</td>
<td>Load unsigned byte</td>
</tr>
<tr>
<td>SB 1000(R0)</td>
<td>Use either of four addressing modes</td>
</tr>
<tr>
<td>LS R1, 40(R3)</td>
<td>Load single prec. FP, store single prec FP</td>
</tr>
<tr>
<td>S S</td>
<td>Use either of four addressing modes</td>
</tr>
<tr>
<td>LD R5, 20(R1)</td>
<td>Load double prec. FP, store double prec FP</td>
</tr>
<tr>
<td>S D</td>
<td>Use either of four addressing modes</td>
</tr>
</tbody>
</table>

Arithmetic Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DADD R1, R2, R3</td>
<td>Add, add unsigned, add immediate, add immediate unsigned</td>
</tr>
<tr>
<td>DADDU</td>
<td>Add immediate, add immediate unsigned</td>
</tr>
<tr>
<td>DADDI R1, R3, #4</td>
<td>Add immediate, add immediate unsigned</td>
</tr>
<tr>
<td>DSUB R1, R2, R3</td>
<td>Subtract, subtract unsigned</td>
</tr>
<tr>
<td>DSUBU</td>
<td>Subtract, subtract unsigned</td>
</tr>
<tr>
<td>DMUL R1, R2, R3</td>
<td>Multiply, multiply unsigned, divide, divide unsigned</td>
</tr>
<tr>
<td>DMULU</td>
<td>Multiply, multiply unsigned, divide, divide unsigned</td>
</tr>
<tr>
<td>DMULU</td>
<td>Multiply and add</td>
</tr>
<tr>
<td>DMU</td>
<td>Multiply and add</td>
</tr>
<tr>
<td>DDIV</td>
<td>Divide, divide unsigned</td>
</tr>
<tr>
<td>DDIVU</td>
<td>Divide, divide unsigned</td>
</tr>
</tbody>
</table>

Logical Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND R1, R2, R3</td>
<td>And, And immediate</td>
</tr>
<tr>
<td>ANDI</td>
<td>And, And immediate</td>
</tr>
<tr>
<td>OR R1, R2, R3</td>
<td>Or, Or immediate</td>
</tr>
<tr>
<td>ORI</td>
<td>Or, Or immediate</td>
</tr>
<tr>
<td>XOR R1, R2, R3</td>
<td>Xor, Xor immediate</td>
</tr>
<tr>
<td>XORI</td>
<td>Xor, Xor immediate</td>
</tr>
</tbody>
</table>

Load and Shift Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUI R1, R2, #5</td>
<td>Load upper immediate – bits 32 to 47, then sign extend</td>
</tr>
<tr>
<td>DSLL R1, R2, #5</td>
<td>Shift left and right logical, shift left and right logical variable (shift value in register)</td>
</tr>
<tr>
<td>DSRL</td>
<td>Shift right arithmetic</td>
</tr>
<tr>
<td>DSLLV R1, R3, R4</td>
<td>Shift right arithmetic variable (shift value in register)</td>
</tr>
<tr>
<td>DSRLV</td>
<td>Shift right arithmetic variable (shift value in register)</td>
</tr>
<tr>
<td>DSRA</td>
<td>Shift right arithmetic</td>
</tr>
</tbody>
</table>

Compare Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLT R1, R2, R3</td>
<td>Set less than, set less than immediate</td>
</tr>
<tr>
<td>SLTI</td>
<td>Set less than, set less than immediate unsigned</td>
</tr>
<tr>
<td>SLTU</td>
<td>Set less than unsigned, set less than immediate unsigned</td>
</tr>
<tr>
<td>SLTRU</td>
<td>Set less than unsigned, set less than immediate unsigned</td>
</tr>
</tbody>
</table>
### Branch Operations

- **BEQZ R1, #600**: Branch if zero, branch if not zero.
- **BNE**: Branch if equal, branch if not equal.
- **BEQ R1, R2, #100**: Branch if equal, branch if not equal.
- **BC1T #500**: Test comparison bit in FP status register and branch if true.
- **BC1F**: Test comparison bit in FP status register and branch if false.

### Jump Operations

- **J #4000**: Jump, PC = immediate_shifted_left_by_2.
- **JR**: Jump register indirect, PC = value of register.
- **JAL**: Jump and link, save PC in R31, PC = immediate_shifted_left_by_2.
- **JALR**: Jump and link, save PC in R31, PC = value of register.

### Conditional Move Operations

- **MOVN R1, R2, R4**: Move GPR to another GPR if third GPR is negative or zero.

### FP Arithmetic Operations

- **ADD .D F1, F2, F3**: Add single precision FP, double precision FP.
- **ADD .S**: Add pairs of single precision FP stored in the same register.
- **SUB .D F1, F2, F3**: Subtract single precision FP, double precision FP.
- **SUB .S**: Subtract pairs of single precision FP stored in the same register.
- **MUL .D F1, F2, F3**: Multiply single precision FP, double precision FP.
- **MUL .S**: Multiply pairs of single precision FP stored in the same register.
- **DIV .D F1, F2, F3**: Divide single precision FP, double precision FP.
- **DIV .S**: Divide pairs of single precision FP stored in the same register.
- **MADD .D F1, F2, F3**: Multiply and add single precision FP, double precision FP.
- **MADD .S**: Multiply and add pairs of single precision FP.

### FP Move Operations

- **CVT.X.Y F1, F2**: Convert from X to Y type.
- **MOV .S F2, F3**: Move from one FPR to another.
- **MOV .D**: Single or double precision.
- **MFC1 F2, R3**: Move single precision number from FPR to GPR.
- **MTC1 R4, F1**: Move single precision number from GPR to FPR.

### Exception Handling Operations

- **TRAP #4000**: Transfer to operating system at given address.
- **PC = immediate, save PC**.
- **ERET**: Return to user mode after exception and restore it.
### FP Compare Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>X and Y can be LT(&lt;), GT(&gt;), LE(£), GE(³), EQ(=), NE(¹)</th>
<th>Compare single or double precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.X.D F1, F2</td>
<td>Compare X and Y, set status register to true or false</td>
<td>X and Y can be LT(&lt;), GT(&gt;), LE(£), GE(³), EQ(=), NE(¹)</td>
<td>Compare single or double precision</td>
</tr>
<tr>
<td>C.X.S</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>