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
Graph Coloring Example

\[ A = B + C \]
\[ C = D + E \]
\[ B = A - F \]

We need 5 registers

Compilers and Data Allocation

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

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
- Displacement
  - Add R4, 100(R1)
  - 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

Addressing Modes for MIPS

- Memory is byte-addressable with 64-bit addresses
- Instruction has a mode bit to select Little Endian or Big Endian
- Memory accesses to byte, half-word, word, or double word
- Memory accesses must be aligned
Instruction Format for MIPS

I-type instructions

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

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

R-type instructions

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

ALU operations with registers
rd ← rs op rt
Read/write special registers
Move data between GPR and FPR

Load/Store Operations

<table>
<thead>
<tr>
<th>Opcode</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(R2)</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 16, 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

- DADD R1, R2, R3
- DADDU R1, R3, #4
- DSUB R1, R2, R3
- DSUBU
- DMUL R1, R2, R3
- DMULT
- DDIV
- DDIVU
- DMADD

Add, add unsigned, add immediate, add immediate unsigned
Subtract, subtract unsigned
Multiply, multiply unsigned, divide, divide unsigned, multiply and add

Logical Operations

- AND R1, R2, R3
- ANDI
- OR R1, R2, R3
- ORI
- XOR R1, R2, R3
- XORI

And, And immediate
Or, Or immediate
Xor, Xor immediate
Load and Shift Operations

- **LUI**
  - Load upper immediate – bits 32 to 47, then sign extend
- **DSLL R1, R2, #5**
  - Shift left and right logical, shift right arithmetic
- **DSRL**
- **DSRA**
  - Shift left and right logical variable (shift value in register)
- **DSLLV R1, R3, R4**
  - Shift left and right logical variable
- **DSRLV**
- **DSRAV**

Compare Operations

- **SLT R1, R2, R3**
  - Set less than
- **SLTI**
  - Set less than immediate
- **SLTU**
  - Set less than unsigned
- **SLTIU**
  - Set less than immediate unsigned

Branch Operations

- **BEQZ R1, #600**
  - Branch if zero, branch if not zero
- **BNEZ**
  - Branch if equal, branch if not equal
- **BEQ R1, R2, #100**
  - Branch if equal, branch if not equal
- **BNE**
  - 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
  - \( \text{PC} = \text{PC} + 4 + \text{immediate shifted left by 2} \)

Jump Operations

- **J #4000**
  - Jump, \( \text{PC}_{36..63} = \text{immediate shifted left by 2} \)
- **JR**
  - Jump register indirect, \( \text{PC} = \text{value of register} \)
- **JAL**
  - Jump and link, save \( \text{PC} \) in \( R31 \)
  - \( \text{PC}_{36..63} = \text{immediate shifted left by 2} \)
- **JALK**
  - Jump and link, save \( \text{PC} \) in \( R31 \)
  - \( \text{PC} = \text{value of register} \)

Conditional Move Operations

- **MOVN R1, R2, R4**
  - Move GPR to another GPR if third GPR is negative
- **MOVZ**
  - Or zero

Exception Handling Operations

- **TRAP #4000**
  - Transfer to operating system at given address
  - \( \text{PC} = \text{immediate, save \( PC \)} \)
- **ERET**
  - Return to user mode after exception and restore it
FP Arithmetic Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD D F1, F2, F3</td>
<td>Add single precision FP, double precision FP</td>
</tr>
<tr>
<td>ADD S</td>
<td>Add pairs of single precision FP stored in the same register</td>
</tr>
<tr>
<td>SUB D F1, F2, F3</td>
<td>Subtract single precision FP, double precision FP</td>
</tr>
<tr>
<td>SUB S</td>
<td>Subtract pairs of single precision FP stored in the same register</td>
</tr>
<tr>
<td>MUL D F1, F2, F3</td>
<td>Multiply single precision FP, double precision FP</td>
</tr>
<tr>
<td>MUL S</td>
<td>Multiply pairs of single precision FP stored in the same register</td>
</tr>
<tr>
<td>DIV D F1, F2, F3</td>
<td>Divide single precision FP, double precision FP</td>
</tr>
<tr>
<td>DIV S</td>
<td>Divide pairs of single precision FP stored in the same register</td>
</tr>
<tr>
<td>MADD D F1, F2, F3</td>
<td>Multiply and add single precision FP, double precision FP</td>
</tr>
<tr>
<td>MADD S</td>
<td>Multiply and add pairs of single precision FP</td>
</tr>
</tbody>
</table>

FP Move Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT X Y F1, F2</td>
<td>Convert from X to Y type</td>
</tr>
<tr>
<td>MOV S F2, F3</td>
<td>Move from one FPR to another</td>
</tr>
<tr>
<td>MOV D</td>
<td>Single or double precision</td>
</tr>
<tr>
<td>MFC1 F2, R3</td>
<td>Move single precision number from FPR to GPR</td>
</tr>
<tr>
<td>MTC1 R4, F1</td>
<td>Move single precision number from GPR to FPR</td>
</tr>
</tbody>
</table>

Pipelining – the Idea

- Similar to assembly line in a factory
- Divide instruction into smaller tasks
- Each task is performed on subset of resources
- Overlap the execution of multiple tasks by completing different tasks from different instructions in parallel
- Ideally this should lead to throughput of one instruction per clock cycle
Pipelining Overview

- Each task is called pipe stage or pipe segment
- All stages must be able to proceed at the same time:
  - Stage duration is called the processor cycle
  - It is determined by the slowest stage
- Goal of pipelining is to increase throughput – number of instructions completed per clock cycle
- In ideally balanced pipeline with $n$ stages:

$$\begin{align*}
T_{\text{instruction}} &= \frac{T_{\text{instruction unpipelined}}}{n} \\
\text{Speedup} &= n
\end{align*}$$

Homework

- Due 9/28 by the end of the class
- 2.10, 2.11, 2.12
  - Hint for 2.10: Assume that only load and store access memory for data
  - Hint for 2.11: To classify instructions as ALU, jump, conditional branch or load/store think like this:
    - Is instruction changing PC unconditionally (jump inst.)
    - Is instruction changing PC on a condition (branch)
    - Is instruction moving data between mem and reg (load/store)
    - Otherwise it is ALU
  - Hint for 2.12: Assume that all loads and stores are displacement loads and stores