Lecture 4: MIPS Instruction Set

- Today’s topics:
  - MIPS instructions
  - Code examples

HW 1 due today/tomorrow!
Common Principles

- Amdahl’s Law

- Energy: performance improvements typically also result in energy improvements – less leakage

- 90-10 rule: 10% of the program accounts for 90% of execution time

- Principle of locality: the same data/code will be used again (temporal locality), nearby data/code will be touched next (spatial locality)
Recap

• Knowledge of hardware improves software quality: compilers, OS, threaded programs, memory management

• Important trends: growing transistors, move to multi-core and accelerators, slowing rate of performance improvement, power/thermal constraints, long memory/disk latencies

• Reasoning about performance: clock speeds, CPI, benchmark suites, performance and power equations

• Next: assembly instructions
Instruction Set

• Understanding the language of the hardware is key to understanding the hardware/software interface

• A program (in say, C) is compiled into an executable that is composed of machine instructions – this executable must also run on future machines – for example, each Intel processor reads in the same x86 instructions, but each processor handles instructions differently

• Java programs are converted into portable bytecode that is converted into machine instructions during execution (just-in-time compilation)

• What are important design principles when defining the instruction set architecture (ISA)?
A Basic MIPS Instruction

C code: 

```
a = b + c ;
```

Assembly code: (human-friendly machine instructions)
```
add   a, b, c      #  a is the sum of b and c
```

Machine code: (hardware-friendly machine instructions)
```
0000001000110010000100000000100000
```

Translate the following C code into assembly code:
```
a = b + c + d + e;
```
Instruction Set

- Important design principles when defining the instruction set architecture (ISA):
  - keep the hardware simple – the chip must only implement basic primitives and run fast
  - keep the instructions regular – simplifies the decoding/scheduling of instructions

We will later discuss RISC vs CISC
Example

C code  \[ a = b + c + d + e; \]
translates into the following assembly code:

\[
\begin{align*}
\text{add} & \quad a, b, c & \quad \text{add} & \quad a, b, c \\
\text{add} & \quad a, a, d & \quad \text{or} & \quad \text{add} & \quad f, d, e \\
\text{add} & \quad a, a, e & \quad \text{add} & \quad a, a, f \\
\end{align*}
\]

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code
- Some sequences are better than others... the second sequence needs one more (temporary) variable \[ f \]
Subtract Example

C code \[ f = (g + h) \, - \, (i + j); \]
translates into the following assembly code:

\[
\begin{align*}
\text{add} & \, \, \, t0, \, g, \, h & \text{add} & \, \, \, f, \, g, \, h \\
\text{add} & \, \, \, t1, \, i, \, j & \text{or} & \text{sub} \, \, \, f, \, f, \, i \\
\text{sub} & \, \, \, f, \, t0, \, t1 & \text{sub} & \, \, \, f, \, f, \, j \\
\end{align*}
\]

• Each version may produce a different result because floating-point operations are not necessarily associative and commutative... more on this later
Operands

• In C, each “variable” is a location in memory

• In hardware, each memory access is expensive – if variable \( a \) is accessed repeatedly, it helps to bring the variable into an on-chip scratchpad and operate on the scratchpad (registers)

• To simplify the instructions, we require that each instruction (add, sub) only operate on registers

• Note: the number of operands (variables) in a C program is very large; the number of operands in assembly is fixed... there can be only so many scratchpad registers
Registers

• The MIPS ISA has 32 registers (x86 has 8 registers) – Why not more? Why not less?

• Each register is 32 bits wide (modern 64-bit architectures have 64-bit wide registers)

• A 32-bit entity (4 bytes) is referred to as a word

• To make the code more readable, registers are partitioned as $s0-$s7 (C/Java variables), $t0-$t9 (temporary variables)...

   add  $s0, $s1, $s2
Binary Stuff

- 8 bits = 1 Byte, also written as $8b = 1B$
- 1 word = 32 bits = 4B
- $1\text{KB} = 1024 \text{ B} = 2^{10} \text{ B}$
- $1\text{MB} = 1024 \times 1024 \text{ B} = 2^{20} \text{ B}$
- $1\text{GB} = 1024 \times 1024 \times 1024 \text{ B} = 2^{30} \text{ B}$
- A 32-bit memory address refers to a number between 0 and $2^{32} - 1$, i.e., it identifies a byte in a 4GB memory
Memory Operands

- Values must be fetched from memory before (add and sub) instructions can operate on them

Load word
lw  $t0, memory-address

Store word
sw  $t0, memory-address

How is memory-address determined?
Memory Address

- The compiler organizes data in memory... it knows the location of every variable (saved in a table)... it can fill in the appropriate mem-address for load-store instructions

```c
int a, b, c, d[10]
```
Memory Organization

$gp$ points to area in memory that saves global variables

$gp$ points to area in memory that saves global variables.
Memory Instruction Format

- The format of a load instruction:

  \[ \text{lw} \quad \text{any register}, \quad a \text{ constant that is added to the register in parentheses} \]
Memory Instruction Format

- The format of a store instruction:

```
sw $t0, 8($t3)
```

- Source register
- Destination address
- Any register
- A constant that is added to the register in parentheses
Example

int a, b, c, d[10];

addi $gp, $zero, 1000  # assume that data is stored at
# base address 1000; placed in $gp;
# $zero is a register that always
# equals zero
lw  $s1, 0($gp)       # brings value of a into register $s1
lw  $s2, 4($gp)       # brings value of b into register $s2
lw  $s3, 8($gp)       # brings value of c into register $s3
lw  $s4, 12($gp)      # brings value of d[0] into register $s4
lw  $s5, 16($gp)      # brings value of d[1] into register $s5
Example

Convert to assembly:

Example

Convert to assembly:


Assembly (same assumptions as previous example):

\[
\begin{align*}
\text{lw} & \quad $s0, 0($gp) \quad \# \text{ a is brought into } $s0 \\
\text{lw} & \quad $s1, 20($gp) \quad \# \text{ d[2] is brought into } $s1 \\
\text{add} & \quad $s2, $s0, $s1 \quad \# \text{ the sum is in } $s2 \\
\text{sw} & \quad $s2, 24($gp) \quad \# \text{ $s2 is stored into d[3]}
\end{align*}
\]

Assembly version of the code continues to expand!