Lecture 2: MIPS Instruction Set

- Today's topic:
  - MIPS instructions
- Reminder: sign up for the mailing list cs3810
- Reminder: set up your CADE accounts (EMCB 224)

Recap

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

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

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)?

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

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)

00000010001100100100000000100000

Translate the following C code into assembly code:

a = b + c + d + e;

Example

C code   a = b + c + d + e;

translates into the following assembly code:

add a, b, c		add a, b, c
add a, a, d	or	add f, d, e
add a, a, e		add a, a, f

- 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);

Assembly code translation with only add and sub instructions:

Subtract Example

C code f = (g + h) – (i + j);
translates into the following assembly code:

```
add  t0, g, h              add  f, g, h
add  t1,  i, j       or    sub   f, f, i
sub  f,   t0, t1           sub   f, f, j
```

- 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

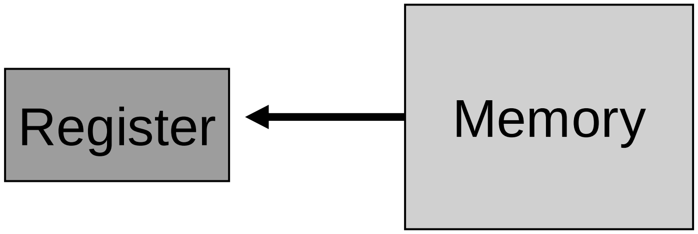
Registers

- The MIPS ISA has 32 registers (x86 has 8 registers) – Why not more? Why not less?
- Each register is 32-bit 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)…

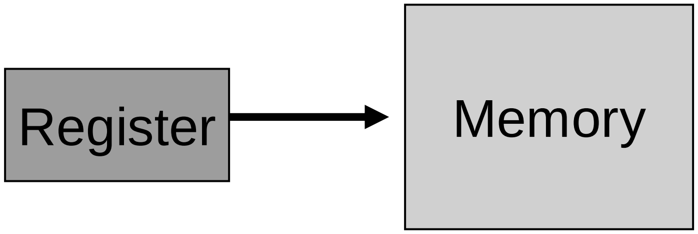
Memory Operands

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

Load word
lw $t0, memory-address



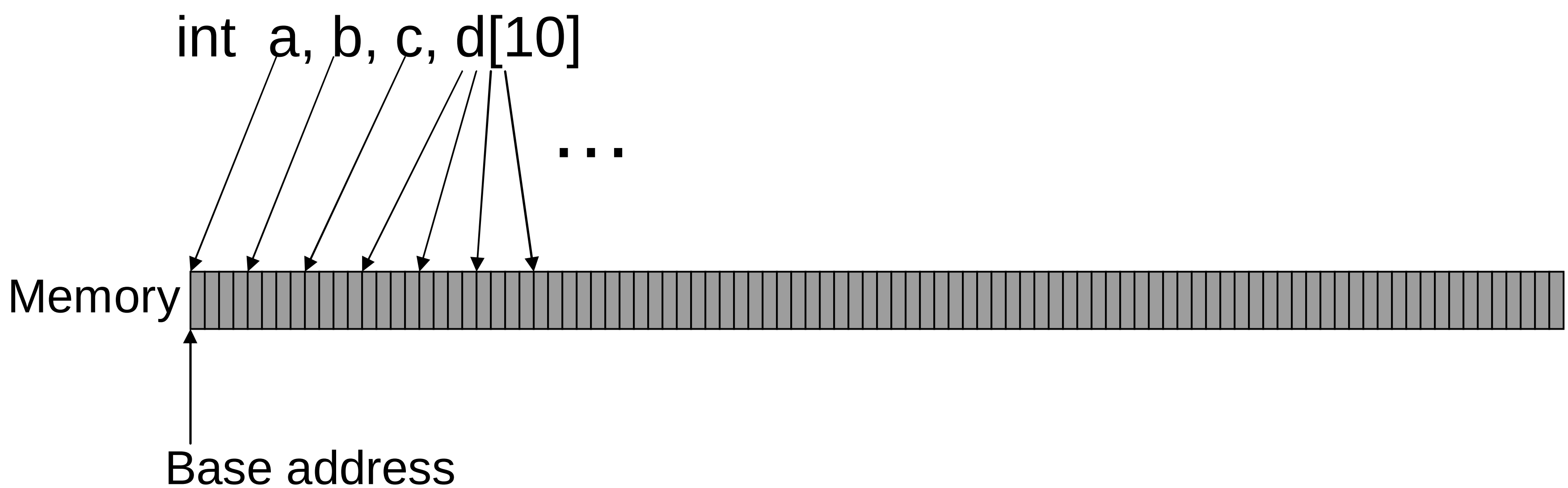
Store word
sw $t0, memory-address



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



Immediate Operands

- An instruction may require a constant as input
- An immediate instruction uses a constant number as one of the inputs (instead of a register operand)

```
addi   $s0, $zero, 1000   # the program has base address
                          #  1000 and this is saved in $s0
                          # $zero is a register that always
                          # equals zero
addi   $s1, $s0, 0        # this is the address of variable a
addi   $s2, $s0, 4        # this is the address of variable b
addi   $s3, $s0, 8        # this is the address of variable c
addi   $s4, $s0, 12       # this is the address of variable d[0]
```

Memory Instruction Format

- The format of a load instruction:

destination register

source address

lw $t0, 8($t3)

any register

a constant that is added to the register in brackets

Example

Convert to assembly:

C code: d[3] = d[2] + a;

Example

Convert to assembly:

C code: d[3] = d[2] + a;

Assembly: # addi instructions as before

```
lw    $t0, 8($s4)     #  d[2] is brought into $t0
lw    $t1, 0($s1)     #   a  is brought into $t1
add   $t0, $t0, $t1   #  the sum is in $t0
sw    $t0, 12($s4)    #  $t0 is stored into d[3]
```

Assembly version of the code continues to expand!

Recap – Numeric Representations

- Decimal 35_{10}
- Binary 00100011_2
- Hexadecimal (compact representation)

  0x 23 or 23_{hex}

0-15 (decimal) → 0-9, a-f (hex)

Instruction Formats

Instructions are represented as 32-bit numbers (one word), broken into 6 fields

R-type instruction add $t0, $s1, $s2

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
opcode	source	source	dest	shift amt	function

I-type instruction lw $t0, 32($s3)

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	constant

Logical Operations

<table>
<tr><th>Logical ops</th><th>C operators</th><th>Java operators</th><th>MIPS instr</th></tr>
<tr><td>Shift Left</td><td><<</td><td><<</td><td>sll</td></tr>
<tr><td>Shift Right</td><td>>></td><td>>>></td><td>srl</td></tr>
<tr><td>Bit-by-bit AND</td><td>&</td><td>&</td><td>and, andi</td></tr>
<tr><td>Bit-by-bit OR</td><td>|</td><td>|</td><td>or, ori</td></tr>
<tr><td>Bit-by-bit NOT</td><td>~</td><td>~</td><td>nor</td></tr>
</table>

Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2: beq register1, register2, L1
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  j L1
  jr $s0

Convert to assembly:

```
if  (i == j)
    f = g+h;
else
    f = g-h;
```

Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2:   beq   register1, register2, L1
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:

```
j     L1
jr    $s0
```

Convert to assembly:

```
if  (i == j)                     bne   $s3, $s4, Else
    f = g+h;                     add   $s0, $s1, $s2
else                             j     Exit
    f = g-h;              Else:  sub   $s0, $s1, $s2
                          Exit:
```

Example

Convert to assembly:

```
while   (save[i] == k)
    i += 1;
```

i and k are in $s3 and $s5 and
base of array save[] is in $s6

Example

Convert to assembly:

```
while   (save[i] == k)
    i += 1;
```

i and k are in $s3 and $s5 and base of array save[] is in $s6

```
Loop:  sll    $t1, $s3, 2
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       addi   $s3, $s3, 1
       j      Loop
Exit:
```

Title

- Bullet