Lecture 4: MIPS Instruction Set

- Today's topics:

  - Chapter 1 wrap-up
  - MIPS instructions
  - Code examples

HW 1 due today/tomorrow!
HW 2 posted later today
Piazza signup link

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)

00000010001100100100000000100000

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

MIPS x86 (Intel, AMD)

We will later discuss RISC vs CISC

Reduced Complex

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

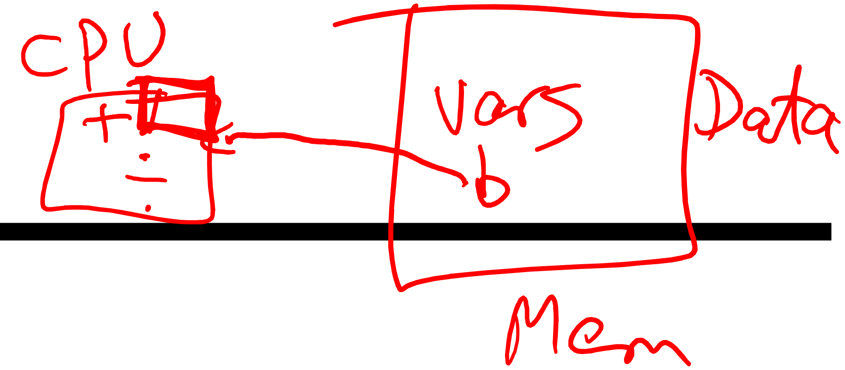
C code f = (g + h) – (i + j);

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
```

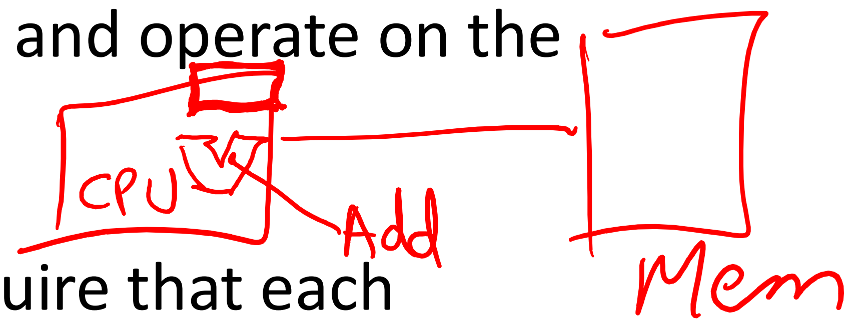
- Each version may produce a different result because floating-point operations are not necessarily associative and commutative… more on this later

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 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

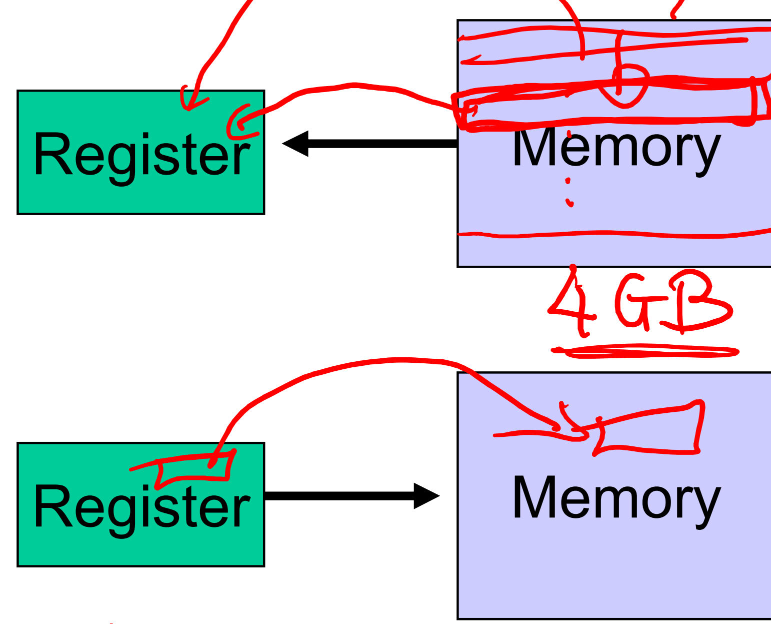
- 8 bits = 1 Byte, also written as 8b = 1B
- 1 word = 32 bits = 4B
- 1KB = 1024 B = 2^{10} B
- 1MB = 1024 x 1024 B = 2^{20} B
- 1GB = 1024 x 1024 x 1024 B = 2^{30} B
- A 32-bit memory address refers to a number between 0 and $2^{32} – 1$, i.e., it identifies a byte in a 4GB memory

Memory Operands

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

Load word
lw $t0, memory-address

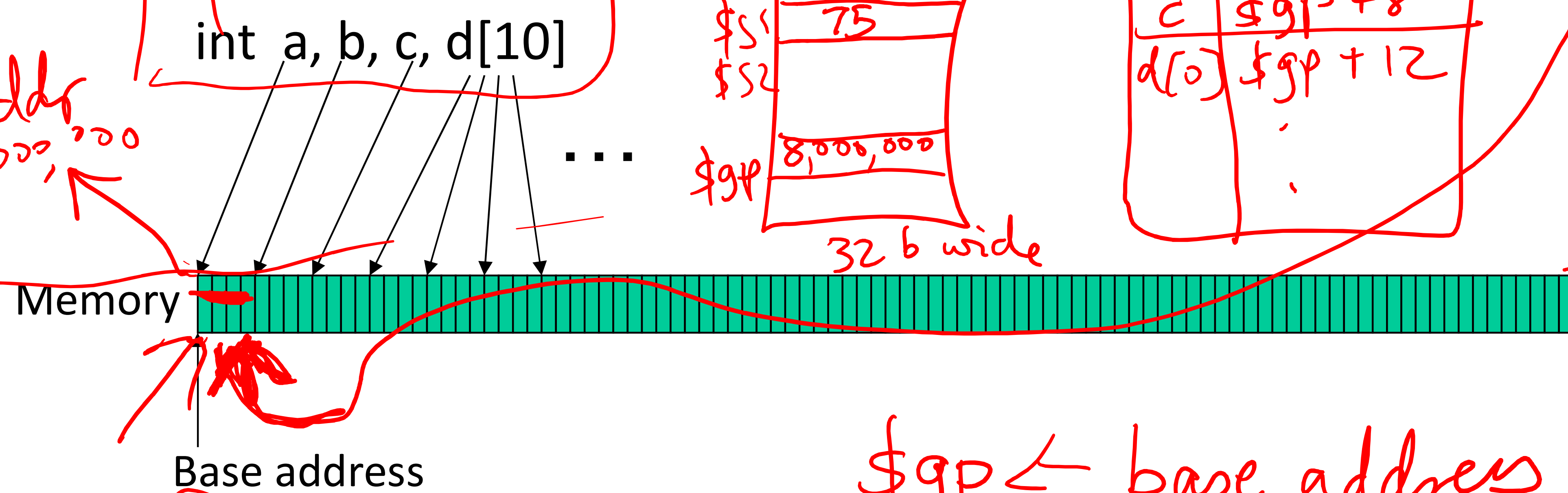
Store word
sw $t0, memory-address



How is memory-address determined?

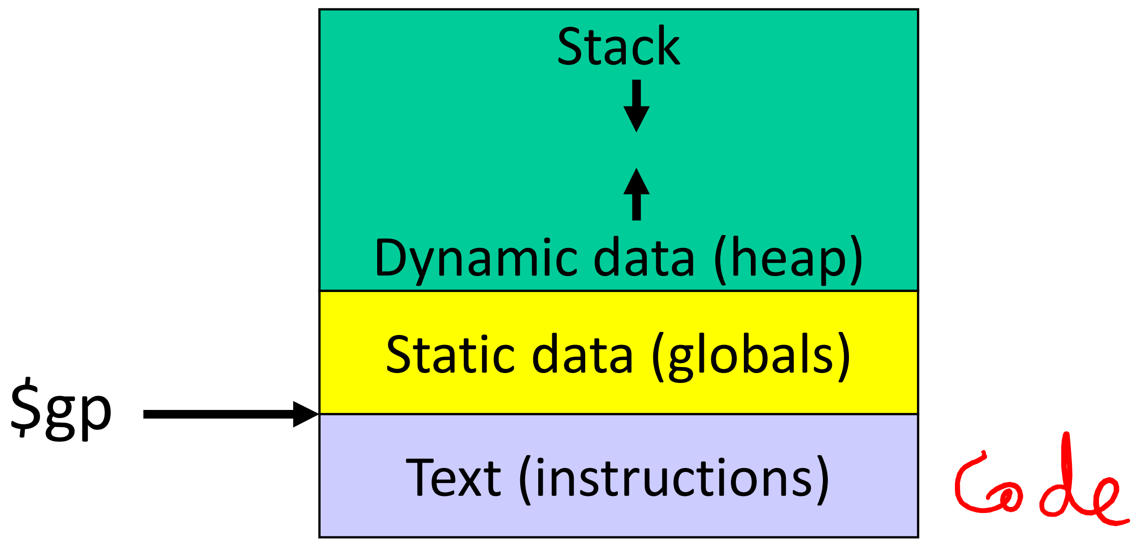
Memory Address

- 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



Memory Organization

$gp points to area in memory that saves global variables



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 parentheses

example

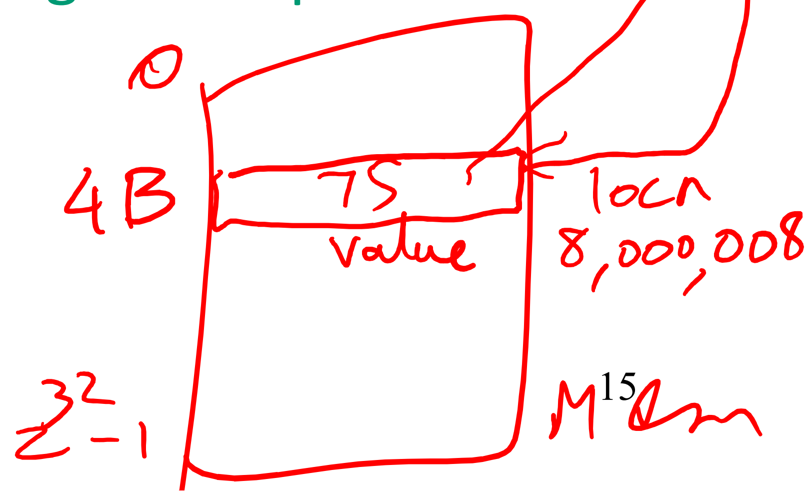
8,000,000 Rd contents of $t3

Add 8

8,000,008 → This is mem addr

Placed in $t0

75 is value placed in $t0



Memory Instruction Format

- The format of a store instruction:

source register

destination address

sw $t0, 8($t3)

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:

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

Example

Convert to assembly:

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

Assembly (same assumptions as previous example):

```
lw   $s0, 0($gp)     # a is brought into $s0
lw   $s1, 20($gp)    # d[2] is brought into $s1
add  $s2, $s0, $s1   # the sum is in $s2
sw   $s2, 24($gp)    # $s2 is stored into d[3]
```

Assembly version of the code continues to expand!