# Lecture 4: MIPS Instruction Set

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

# 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:

```
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 $\quad$ 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)…

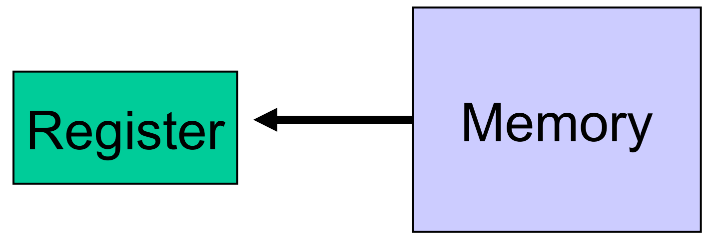
# Binary Stuff

- 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

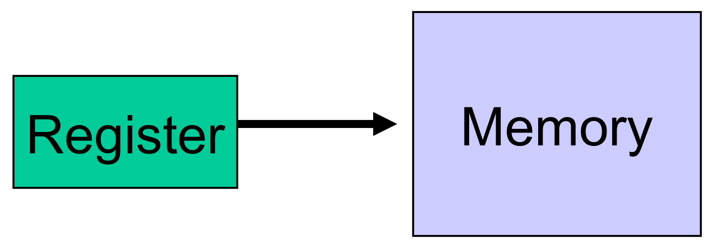
# Memory Operands

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

Load word
lw  $t0, memory-address



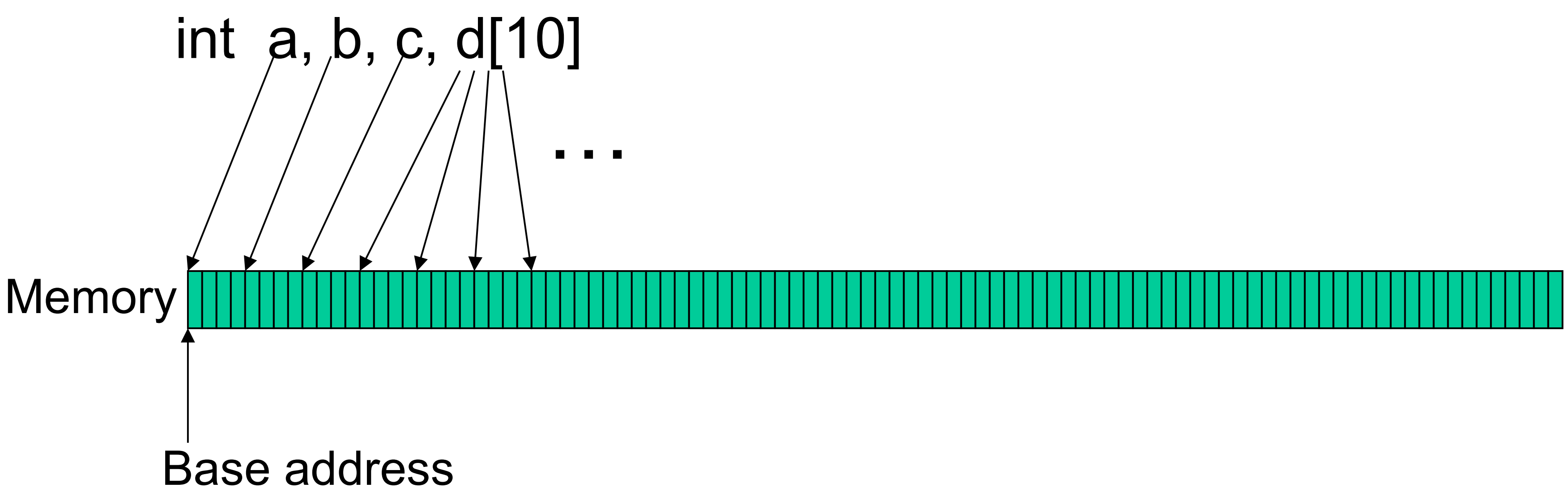

Store word
sw  $t0, memory-address




How is memory-address determined?

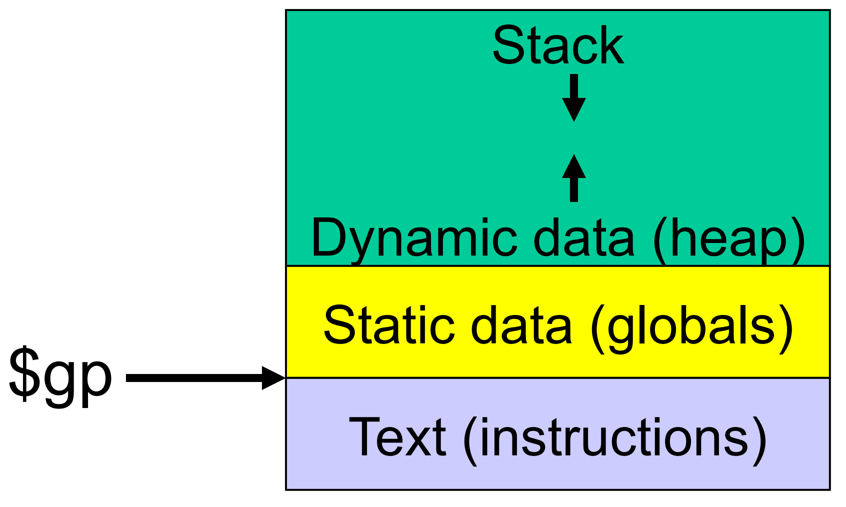
# Memory Address

- 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 brackets

# 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 brackets

# 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;

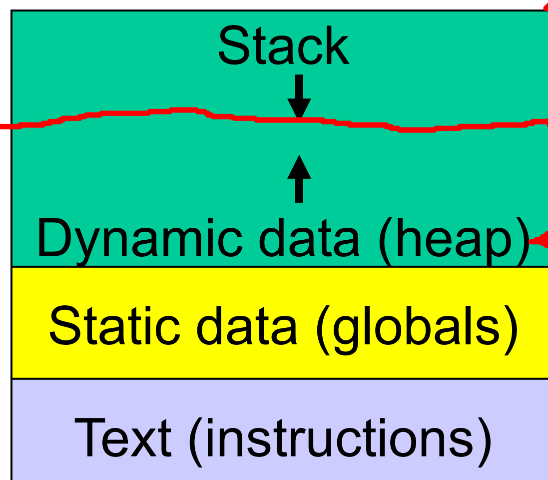
Assembly (same assumptions as previous example):

```
lw    $s0, 0($gp)      #  a is brought into $s0
lw    $s1, 20($gp)     #  d[2] is brought into $s1
add   $t1, $s0, $s1    #  the sum is in $t1
sw    $t1, 24($gp)     #  $t1 is stored into d[3]
```

Assembly version of the code continues to expand!

# Memory Organization

- The space allocated on stack by a procedure is termed the activation record (includes saved values and data local to the procedure) – frame pointer points to the start of the record and stack pointer points to the end – variable addresses are specified relative to $fp as $sp may change during the execution of the procedure
- $gp points to area in memory that saves global variables
- Dynamically allocated storage (with malloc()) is placed on the heap




runtime

ตัวแปรที่ประกาศไว้ก่อนรัน

สงวนไว้สำหรับโปรแกรม

# Recap – Numeric Representations

- Decimal $35_{10} = 3 \times 10^1 + 5 \times 10^0$
- Binary $00100011_2 = 1 \times 2^5 + 1 \times 2^1 + 1 \times 2^0$
- Hexadecimal (compact representation)

  0x 23 or $23_{hex} = 2 \times 16^1 + 3 \times 16^0$

0-15 (decimal) ☾ 0-9, a-f (hex)

| Dec | Binary | Hex | Dec | Binary | Hex | Dec | Binary | Hex | Dec | Binary | Hex |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0000 | 00 | 4 | 0100 | 04 | 8 | 1000 | 08 | 12 | 1100 | 0c |
| 1 | 0001 | 01 | 5 | 0101 | 05 | 9 | 1001 | 09 | 13 | 1101 | 0d |
| 2 | 0010 | 02 | 6 | 0110 | 06 | 10 | 1010 | 0a | 14 | 1110 | 0e |
| 3 | 0011 | 03 | 7 | 0111 | 07 | 11 | 1011 | 0b | 15 | 1111 | 0f |

ตัวแปรที่คอมไพลเลอร์รู้อยู่แล้ว
add $s0,$s1,$S2
addi $S0,$S1,1000

# 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)
- Putting a constant in a register requires addition to register $zero (a special register that always has zero in it) -- since every instruction requires at least one operand to be a register
- For example, putting the constant 1000 into a register:

addi $s0, $zero, 1000

# 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 |

พวก Immediate

| | | | |
|---|---|---|---|
| *I-type instruction* | | | lw $t0, 32($s3) |
| 6 bits | 5 bits | 5 bits | 16 bits |
| opcode | rs | rt | constant |

sll , srl

Shift Left Logical , Shift Right Logical

# Logical Operations

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

Shift Left = คูณ 2,Shift Right=หารสอง