CENG311 Computer Architecture

Instructions: Language of the Computer

IZTECH, Fall 2023

12 October 2023

From a High-Level Language to the Language of Hardware

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

Assembly language program (for MIPS)

```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```

Assembler

Binary machine language program (for MIPS)

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

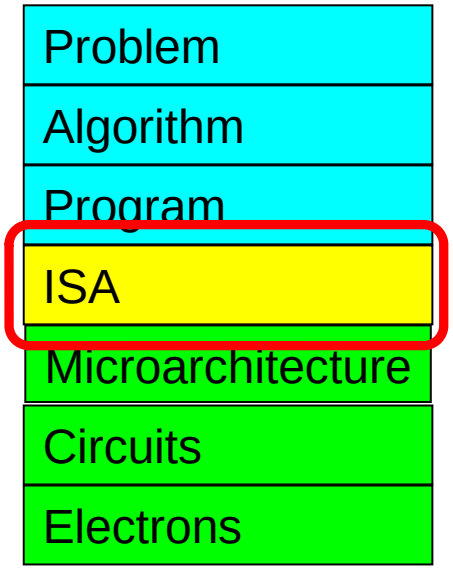
Instruction Set Architecture (ISA)

Instructions

- Opcodes, Addressing Modes, Data Types
- Instruction Types and Formats
- Registers, Condition Codes

Memory

- Address space, Addressability, Alignment
- Virtual memory management

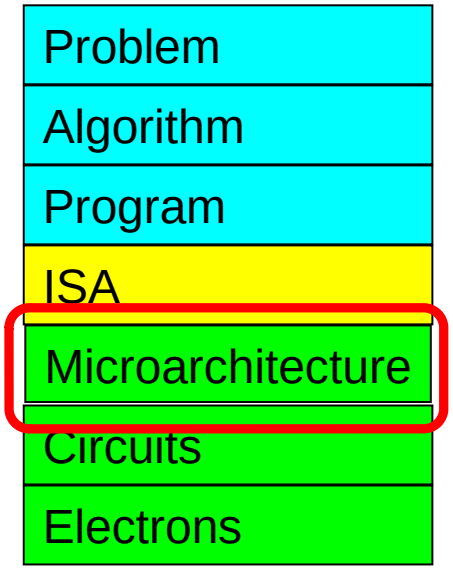


Microarchitecture

Implementation of the ISA under specific design constraints and goals

Anything done in hardware without exposure to software

- Pipelining
- In-order versus out-of-order instruction execution
- Memory access scheduling policy
- Speculative execution
- Superscalar processing (multiple instruction issue)
- Caching: Levels, size, associativity, replacement
- Prefetching



ISA vs Microarchitecture

ISA: specifies how the programmer sees instructions to be executed

- Interface between software and hardware
- What the software developer needs to know to write and debug programs

Microarchitecture: How the underlying implementation actually executes instructions

- Specific implementation of an ISA
- Not visible to the software

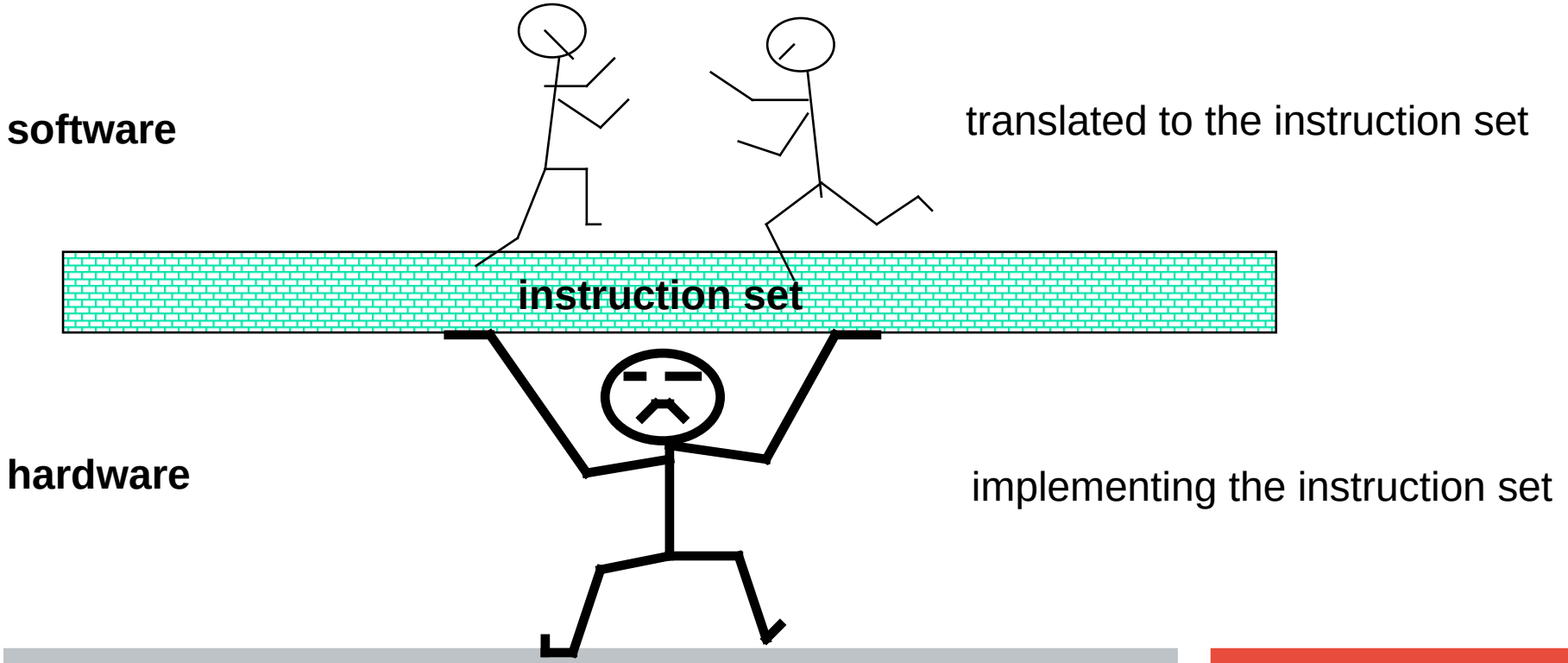
Add instruction vs Adder implementation

Instruction Set

Computer’s language: instructions

Its vocabulary: instruction set (e.g., x86, MIPS)

Interface between processor and low-level software



MIPS Instruction Set

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Arithmetic Operations

Add and subtract, three operands

Two sources and one destination

```
add a, b, c  # a gets b + c
```

All arithmetic operations have this form

```
sub, mult, div
```

Arithmetic operations occur only on registers in MIPS instructions

Arithmetic Example

C code:

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

Compiled MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

Register Operands

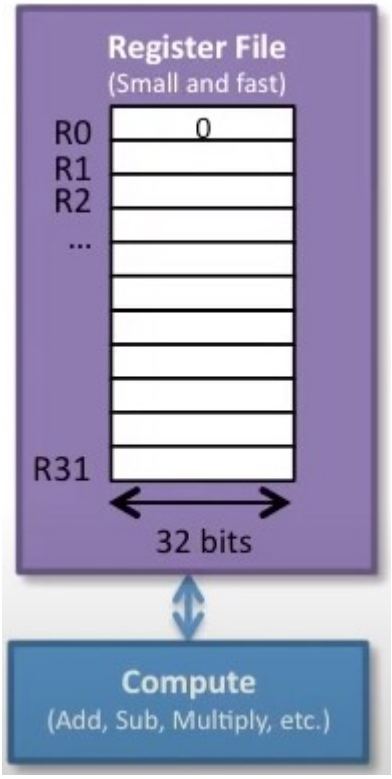
Arithmetic instructions use register operands

MIPS has a 32 × 32-bit register file

- Used for frequently accessed data
- Numbered 0 to 31

Register names

- $t0, $t1, …, $t9 for temporary values
- $s0, $s1, …, $s7 for saved variables



Register Operand Example

C code:

f = (g + h) - (i + j);

f, …, j in $s0, …, $s4

Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Memory Operands

Main memory used for composite data

- Arrays, structures, dynamic data

To apply arithmetic operations

- Load values from memory into registers
- Store result from register to memory

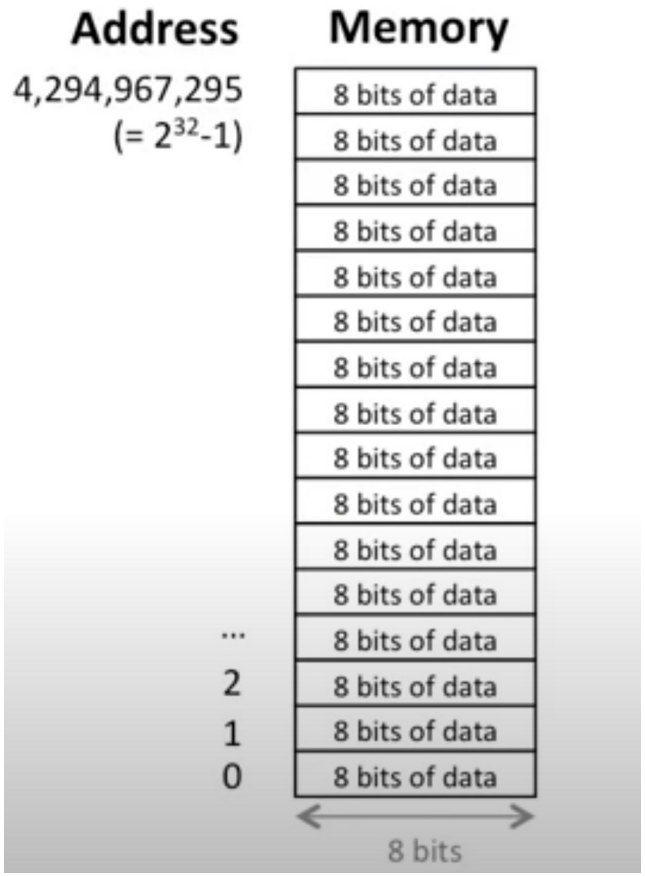
Memory is byte addressed

- Each address identifies an 8-bit byte

Words are aligned in memory

- Address must be a multiple of 4 (1 word = 32 bits)

Memory Addresses



Memory is large 1-dimension array

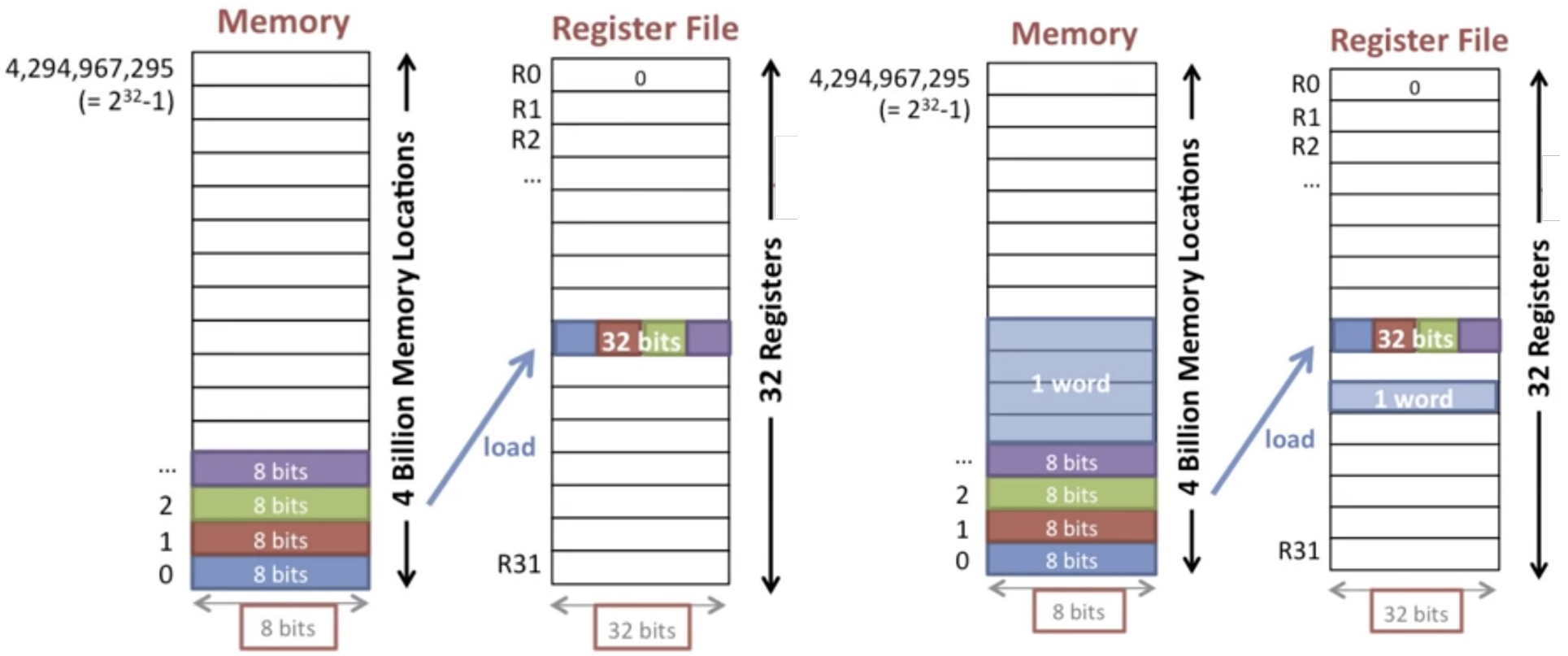
Byte addressed

Each address identifies an 8-bit - 1byte

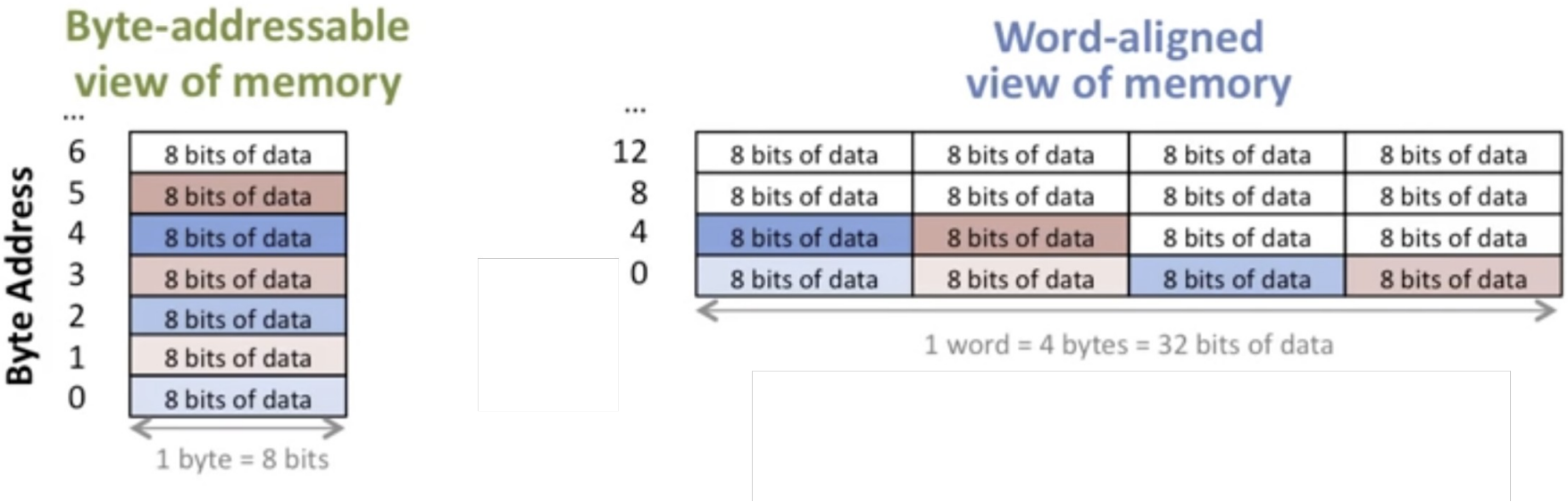
For a 32-bit computer, there are 2^32 memory locations (4GB)

https://www.youtube.com/watch?v=rZev35tJaEY

Memory Load/Store Operations



Byte Addressable/Word Aligned



Load/Store Operations

```
lw    d,off(b)
```

`d` is destination register

`b` is base register, `off` is offset value

b+off forms the memory address to be accessed

The data from memory is available in d, word from memory address b+off

```
sw    s,off(b)
```

Memory Operand Example 1

C code:

g = h + A[8];

g in $s1, h in $s2, base address of A in $s3

Compiled MIPS code:

Index 8 requires offset of 32 (4 bytes per word)

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

Memory Operand Example 2

C code:

A[12] = h + A[8];

h in $s2, base address of A in $s3

Compiled MIPS code:

Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

Register vs Memory

Registers are faster to access than memory

Operating on memory data requires loads and stores

- More instructions to be executed

Compiler must use registers for variables as much as possible

- Only spill to memory for less frequently used variables
- Register optimization is important!

References

Chapter 2.2

Chapter 2.3

(Computer Organization and Design: The Hardware/Software Interface by Hennessy/Patterson, 5th edition)