CMPE 411
Computer Architecture

Lecture 2

Instructions Semantics & Representation

September 5, 2017

www.csee.umbc.edu/~younis/CMPE411/CMPE411.htm

Lecture's Overview

❑ *Previous Lecture:*

- What computer architecture is and why it is important to study
- Organization and anatomy of computers
- The impact of microelectronics technology on computers
- The evolution of the computer industry and generations

❑ *This Lecture:*

- Instruction set architecture and CPU operations
- Instruction types, operands and operations
- Decision making and repetition of instruction execution
- Supporting procedure and context switching

Introductions

- To command computer's hardware, you must speak its language
- The words of a machine's language are called instructions, and its vocabulary is called instruction set
- Once you learn one machine language, it is easy to pick up others:
  - ➔There are few fundamental operations that all computers must provide
  - ➔All designers have the same goal of finding a language that simplifies building the hardware and the compiler while maximizing performance and minimizing cost
- Learning how instructions are represented leads to discovering the secret of computing: “the stored-program concept”
- The MIPS instruction set is used as a case study
  - MIPS is commercialized by Imagination Technologies (http://www.imgtec.com/)
  - Popular choice in the embedded core market
    - Applications in consumer electronics, printers storage devices, digital cameras, etc.

Instruction Set Architecture (ISA)

- ISA: An abstract interface between the hardware and the lowest level software of a machine for encompassing all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on.

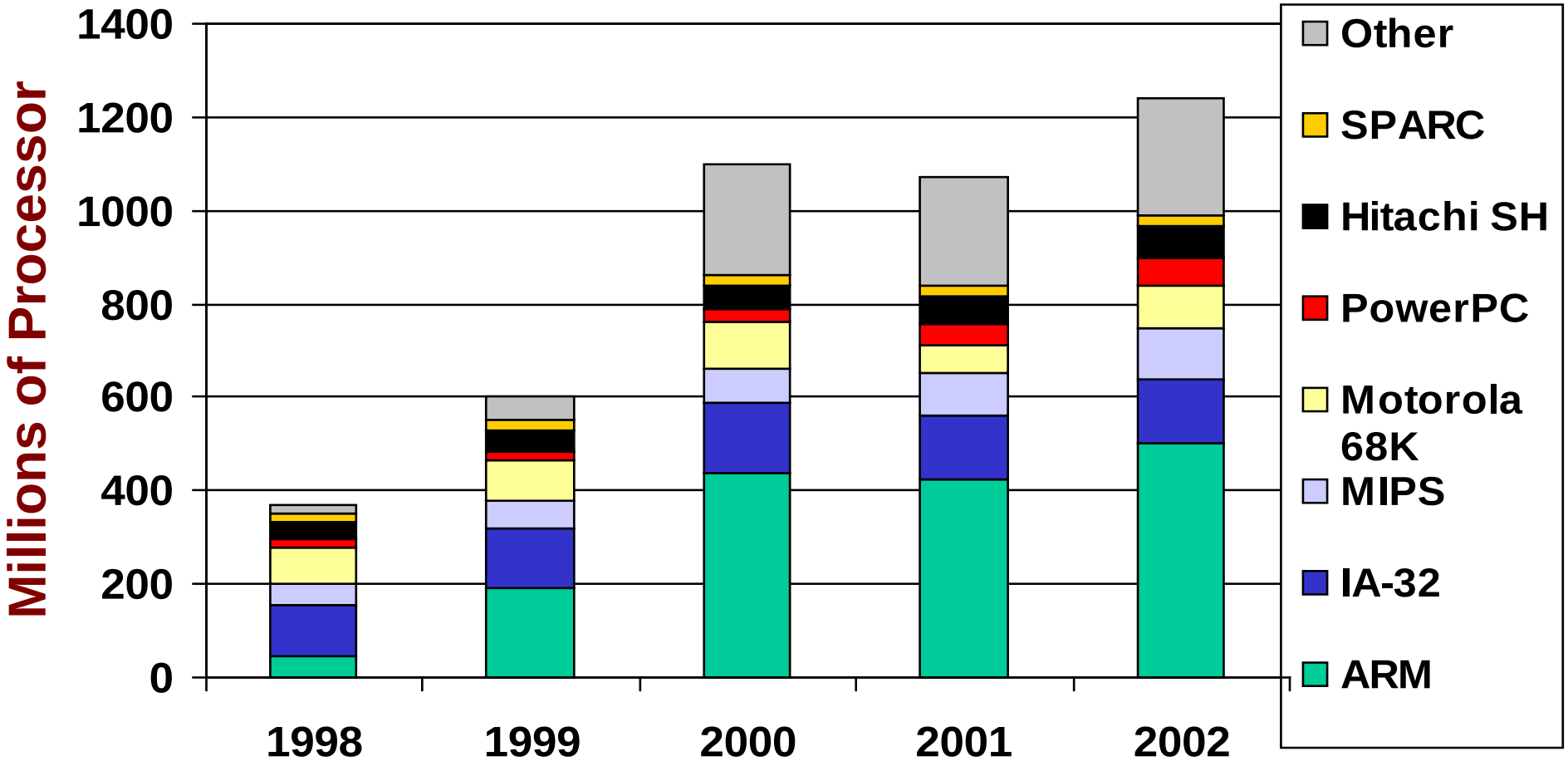
  "... the attributes of a [computing] system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls, the logic design, and the physical implementation."
  – Amdahl, Blaauw, and Brooks, 1964

  – Enables implementations of varying cost and performance to run identical software

- ABI (application binary interface): The user portion of the instruction set plus the operating system interfaces used by application programmers. Defines a standard for binary portability across computers.

* Slide is courtesy of Mary Jane Irwin

ISA Type Sales



PowerPoint “comic” bar chart with approximate values

* Slide is courtesy of Mary Jane Irwin

Operations of the Computer Hardware

"There must certainly be instructions for performing the fundamental arithmetic operations."

Burkes, Goldstine and Von Neumann, 1947

- Assembly language is a symbolic representation of what the processor actually understand
- MIPS assembler allows only one instructions/line and ignore comments following # until end of line
- Example:

  Translation of a segment of a C program to MIPS assembly instructions:

  C: f = (g + h) - (i + j)

  MIPS:

```
add     t0, g, h    # temp. variable t0 contains "g + h"
add     t1, i, j    # temp. variable t1 contains "i + j"
sub     f, t0, t1   # f = t0 - t1 = (g + h) - (i + j)
```

Operands of the Computer Hardware

- Registers are the bricks of computer construction
- Registers are hardware design primitives that are also visible to programmer
- The size of the registers (referred to as *word*) in MIPS architecture is 32 bits
- Unlike variables of a programming language, the number of registers is limited (MIPS has 32 registers)
- Effective use of registers is a key to program performance
- The MIPS convention is to use 2 character names following a '$' to represent registers, e.g. $s0, $s1, .. for variables and $t0, $t1, … for temp registers
- Example:

  Translation of a segment of a C program to MIPS assembly instructions:

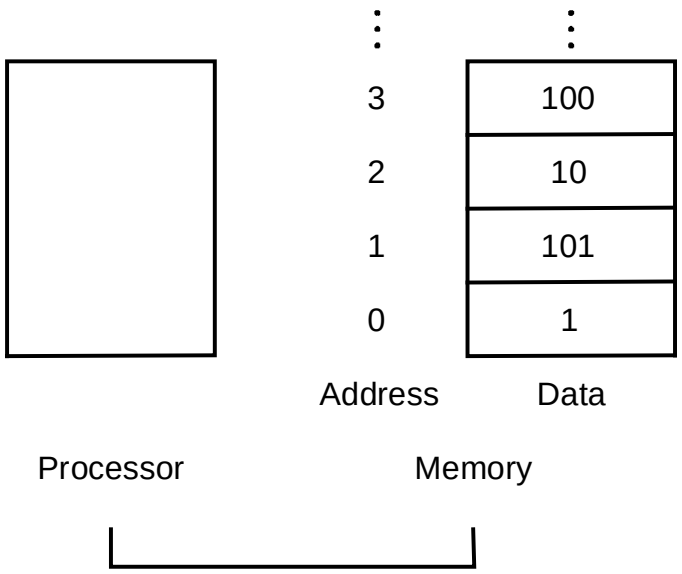
  C: f = (g + h) - (i + j)

  MIPS:

```
add     $t0, $s1, $s2    # temp. variable t0 contains "g + h"
add     $t1, $s3, $s4    # temp. variable t1 contains "i + j"
sub     $s0, $t0, $t1    # f = t0 - t1 = (g + h) - (i + j)
```

Large Structure Representation

- Given the limited number of registers, data structures such as arrays are kept in memory
- The data transfer instructions that move data from/to memory to/from a register are traditionally called *load/store* ("*lw*" and "*sw*" in MIPS standing for load/store word)
- MIPS uses indirect memory reference using a base register and a constant increment



- Example: Compiling an assignment when an operand is in memory

  Let's assume that A is an array of 100 words and that the compiler has associated the variables g and h with the registers $s1 and $s2. Let's assume that the staring address, or base address, of the array is in $s3. The following is the compiler translation of a segment of a C program to MIPS assembly instructions:

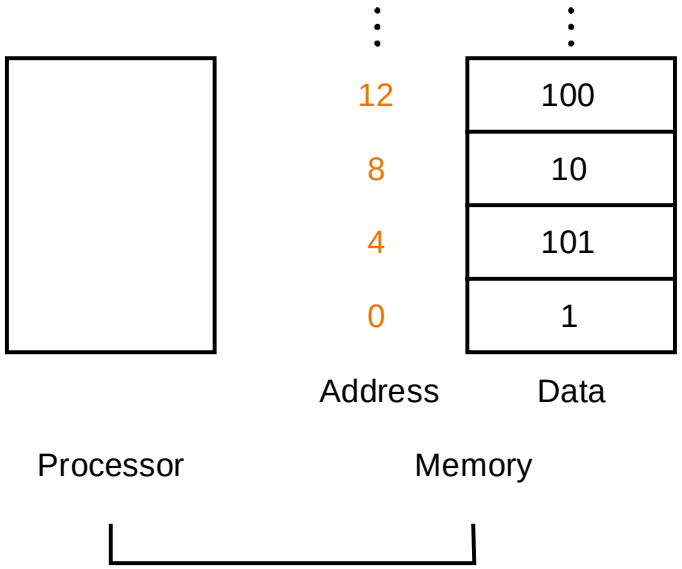
```
C:      g = h + A[8];

MIPS:   lw    $t0, 32($s3)    # Temporary reg. $t0 gets A[8]
        add   $s1, $s2, $t0   # g = h + A[8]
```

Compilation & Address formation

❑ Compiler Role:

- ➔ Associates variables with registers
- ➔ Allocates data structures like arrays in memory
- ➔ Places the proper staring address into the data transfer instructions
- ➔ keeps most frequently used variables in registers to save on load/store operations (spilling registers)
- ➔ keeps operands in registers to achieve the highest performance



❑ Memory Addressing

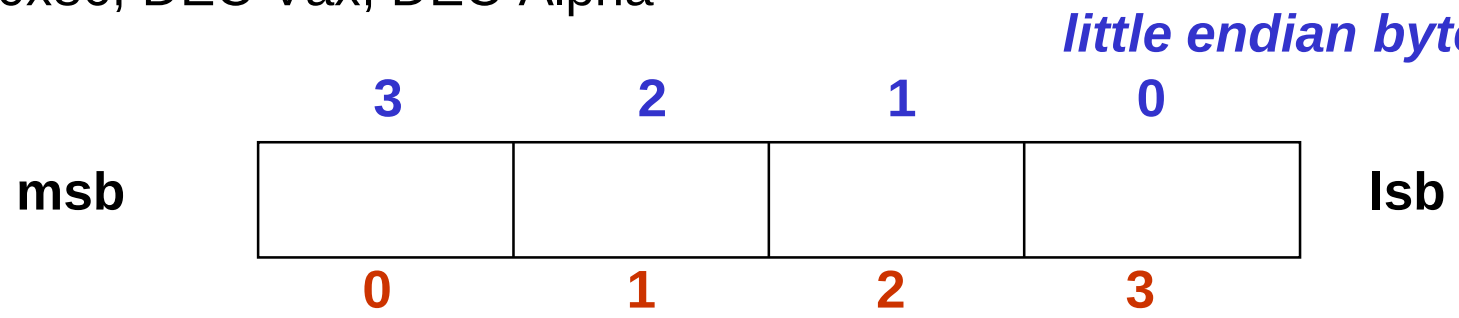
- ➔ The address of a word matches the byte address of one of its 4 bytes
- ➔ The addresses of sequential words differ by 4 (word size in byte)
  ➔ words' addresses are multiple of 4 (alignment restriction)
- ➔ Byte addresses affects array index calculation: the offset to be added to the base register $s3 must be 4 × 8 in order to access A[8]

```
lw        $t0, 32($s3)        # Temporary reg. $t0 gets A[8]
```

- ➔ Machines that use the address of the leftmost byte as the word address is called "*Big Endian*" and those that use rightmost bytes called "*Little Endian*"

Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - The memory address of a word must be a multiple of 4 (alignment restriction)
- **Big Endian:** leftmost byte is word corresponds to low address
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian:** rightmost byte is word corresponds to low address
  - Intel 80x86, DEC Vax, DEC Alpha



Affects only byte order storage in memory

* Slide is courtesy of Mary Jane Irwin

Compiling Using a Variable Array Index

Let's assume that A is an array of 100 words and that the compiler has associated the variables g, h and i with the registers $s1, $s2 and $s4. Let's assume that the starting address, or base address, of the array is in $s3. The following is the compiler translation of a segment of a C program to MIPS assembly instructions: *G = h + A[i];*

First we have to ensure word alignment:

```
add     $t1, $s4, $s4     # Temp reg $t1 = 2 * i
add     $t1, $ t1, $ t1   # Temp reg $t1 = 4 * i
```

To get the address of *A[i]*, we need to add *$t1* to the base of *A* in *$s3*:

```
add     $t1, $ t1, $ s3   # $t1 = address of A[i] (4 * i + $s3)
```

Now we can use that address to load A[i] into a temporary register:

```
lw      $t0, 0($t1)       # Temporary register $t0 gets A[i]
```

Finally add *A[i]* to *h* and place the sum in *g*:

```
add     $s1, $s2, $t0     # g = h + A[i]
```

Instruction Representation

- Humans are taught to think in base 10 (decimal) but numbers may be represented in any base (123 in base 10 = 1111011 in binary or base 2)
- Numbers are stored in computers as a series of high and low electronic signals (binary numbers)
- Binary digits are called bits and considered the atom of computing
- Each piece of an instruction is a number and placing these numbers together forms the instruction
- Assembler translate the assembly symbolic instructions into machine language instructions (machine code)
- Example:

Assembly: add $t0, $s1, $s2

M/C language (decimal):	0	17	18	8	0	32
M/C language (binary):	000000	10001	10010	01000	00000	100000
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Note: MIPS compiler by default maps $s0,…,$s7 to reg. 16-23 and $t0,…,$t7 to reg. 8-15

MIPS Instruction format

❑ Register-format instructions:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

op: Basic operation of the instruction, traditionally called opcode
rs: The first register source operand
rt: The second register source operand
rd: The register destination operand, it gets the result of the operation
shmat: Shift amount (explained in future lectures)
funct: This field selects the specific variant of the operation of the op field

❑ Immediate-type instructions:

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

- Some instructions need longer fields than provided for large value constant
- The 16-bit address means a load word instruction can load a word within a region of $\pm 2^{15}$ bytes of the address in the base register
- Example: lw $t0, 32($s3) # Temporary register $t0 gets A[8]

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32	N/A
sub	R	0	reg	reg	reg	0	34	N/A
lw	I	35	reg	reg	N/A	N/A	N/A	address
sw	I	43	reg	reg	N/A	N/A	N/A	address

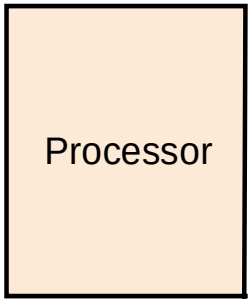
The Stored Program Concept

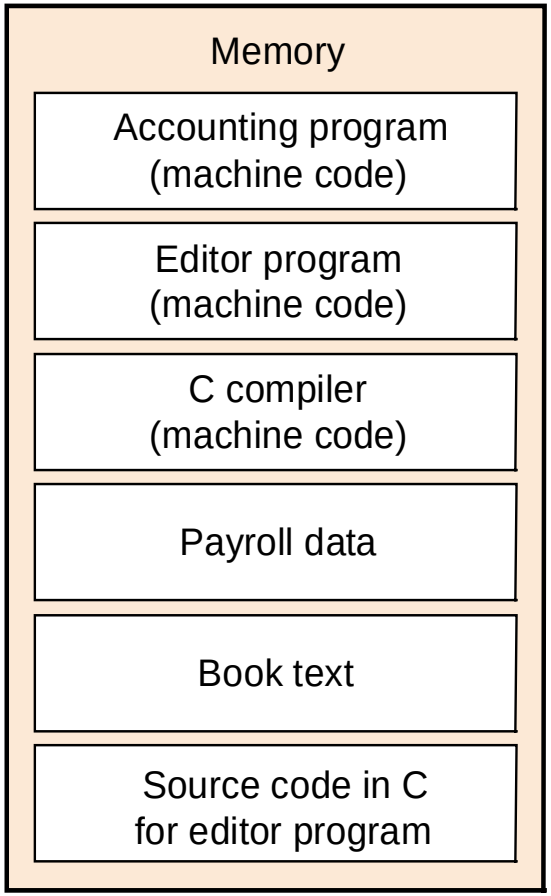
- Learning how instructions are represented leads to discovering the secret of computing: “the stored-program concept”
- Today’s computers are built based on two key principles :
  - ➔ Instructions are represented as numbers
  - ➔ Programs can be stored in memory to be read or written just like numbers

The power of the concept:

memory can contain:

- ➔ the source code for an editor
- ➔ the compiled m/c code for the editor
- ➔ the text that the compiled program is using
- ➔ the compiler that generated the code





Instructions for Making Decisions

- What distinguish a computer from a simple calculator is its ability to make decisions
- Based on input data and the values created during the computation, different instructions are executed
- Decisions making instructions is commonly represented in programming languages using *if-statement* combined with *go-to*
- MIPS assembly language includes two conditional branching instructions:

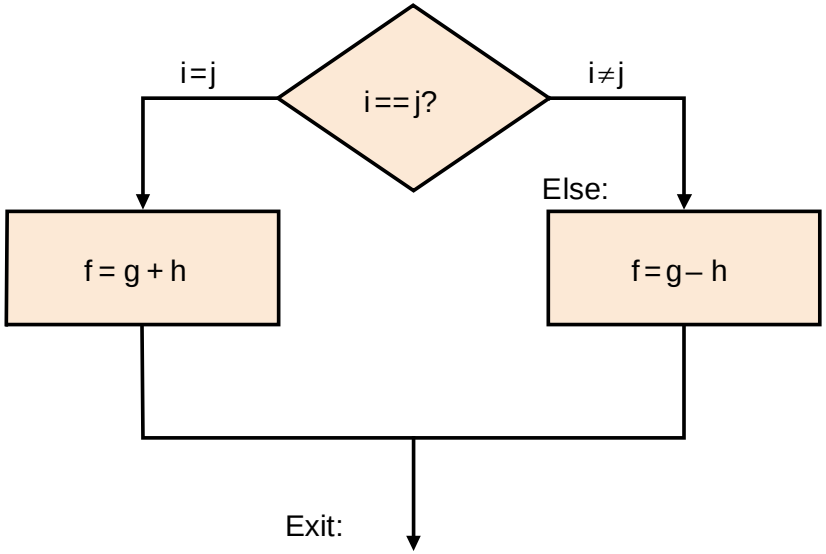
```
beq register1, register2, L1   # go to L1 if (register1) = (register2)
bne register1, register2, L1   # go to L1 if (register1) ≠ (register2)
```

- Compilers frequently create branches and labels although they do not appear in high level programs

Compiling if-then-else

Assuming the five variables f, g, h, i, and j correspond to the five registers $s0 through $s4, what is the compiler MIPS code for the following C if statement:

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



MIPS:

```
        bne   $s3, $s4, Else     # go to Else if i ≠ j
        add   $s0, $s1, $s2      # f = g + h (skipped if i ≠ j)
        j     Exit
Else:   sub   $s0, $s1, $s2      # f = g - h (skipped if i = j)
Exit:
```

Compiling a *while* Loop

Assume that i, j and k correspond to $s3 through $s5, and the base of the array "save" is in $s6. what is the compiler MIPS code for the following C segment:

while (save[i] == k) i = i + j;

MIPS:

The first step is to load save[i] into a temporary register

```
Loop:   add     $t1, $s3, $s3          # Temp reg $t1 = 2 * i
        add     $t1, $ t1, $ t1        # Temp reg $t1 = 4 * i
        add     $t1, $ t1, $ s6        # $t1 = address of save[i]
        lw      $t0, 0($t1)            # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if save[i] ≠ k

```
        bne     $t0, $s5, Exit         # go to Exit if save[i] ≠ k
```

The next instruction add j to i:

```
        add     $s3, $s3, $s4          # i =  i + j
```

Finally reaching the loop end

```
        j       Loop          # go back to the beginning of loop
Exit:
```

Supporting Procedures

- A subroutine is a tool to structure programs in order to make them easier to understand and facilitate reuse
- Execution of a procedure follows the following steps:
  1. Put parameters in a place accessible to the procedure
  2. Transfer control to the procedure
  3. Acquire the storage resources needed for the procedure
  4. Perform the desired task
  5. Place the results value in a place accessible to the calling program
  6. Return control to the point of origin
- MIPS assembler allocate registers $a0-$a3 for argument passing, $v0-$v1 for return values and $ra for return address
- a Jump-and-link *jal* instruction save the address of next instruction in a register $ra and branch to the procedure
- The hardware provides a program counter to trace instruction flow and manage transfer of control
- The *jal* instruction stores PC+4 in $ra for use by the *jr* instruction as a return address upon procedure completion

Parameters Passing

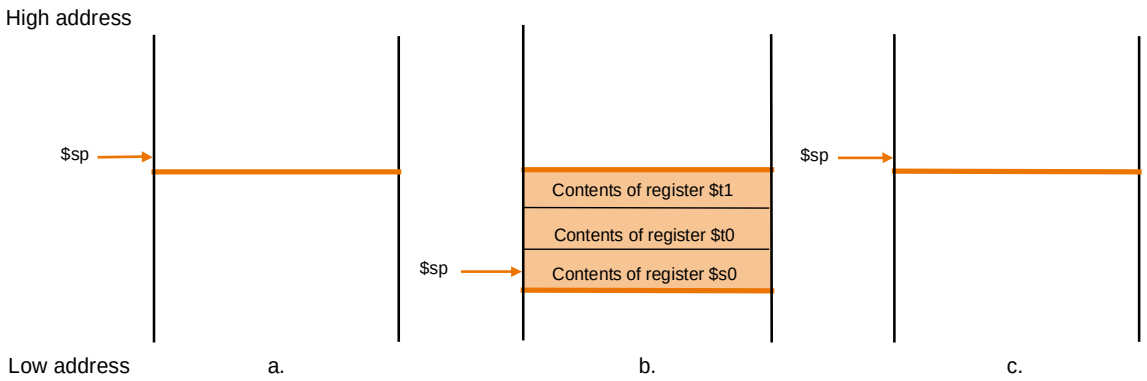
- Registers can be used for passing small number of parameters
- A stack is used to spill registers of the current context and make room for the called procedure to run and to allow for large parameters to be passed
- A stack traditionally grows from high addresses to low addresses
- Hardware dedicates a register as stack pointer for stack memory reference
- Upon returning from a procedure the caller needs to retrieve the old values of registers (machine status) to resume execution of using its context
- There is no "*Push*" and "*Pop*" instructions in MIPS requiring manual stack management (adjusting the pointer)
- For nested procedure calls, original parameters and *$ra* has to be stored in by the caller in the stack

Preserved	Not preserved
Save registers: $s0-$s7	Temporary registers: $t0-$t9
Stack pointer register: $sp	Argument registers: $a0-$a3
Return address register: $ra	Return value registers: $v0-$v1
Stack above the stack pointer	Stack below the stack pointer

An Example

Let's convert the following C segment into a procedure:

```
int leaf_example (int g, int h,
    int I, int j) {
        int f;
        f= (g + h) - (i +j);
        return f;
}
```

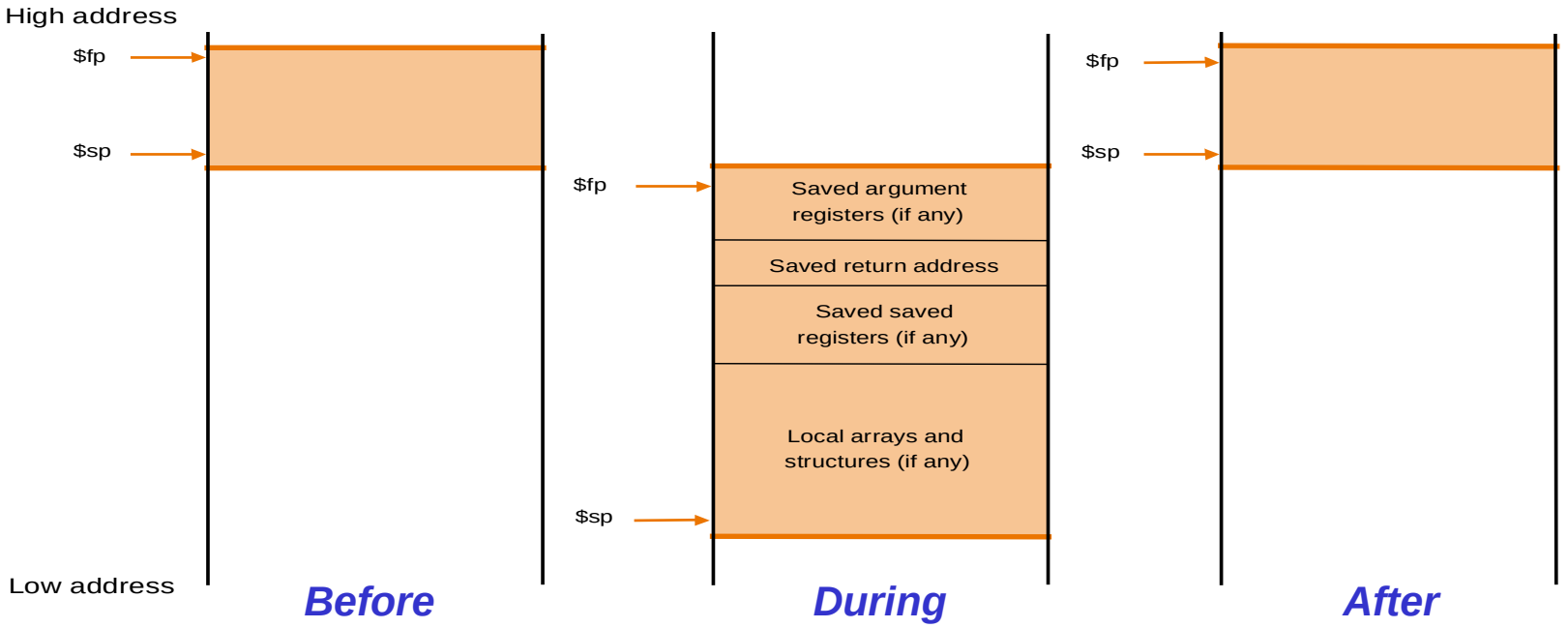


MIPS: Leaf_example:

Saving Caller's Context	sub $sp, $sp, 12	# adjust stack to make room for 3 items
	sw $t1, 8($sp)	# save register $t1 for use afterwards
	sw $t0, 4($sp)	# save register $t0 for use afterwards
	sw $s1, 0($sp)	# save register $s1 for use afterwards
	…..	# procedure body
Restoring Caller's Context	lw $t1, 8($sp)	# restore register $s1 for caller
	lw $t0, 4($sp)	# restore register $t0 for caller
	lw $s1, 0($sp)	# restore register $t1 for caller
	add $sp, $sp, 12	# adjust stack to delete the 3 items
	jr $ra	# jump back to calling routine

Allocating Space for New Data

- The stack is also used to store variables and arrays local to the procedure
- The segment of the stack containing a procedure's saved registers and local variables is called a *procedure frame* or *activation record*
- Some MIPS software uses a frame pointer ($fp) to point to the first word of the frame of a procedure
- A frame pointer offers a stable base reference to local variables since the stack pointer changes throughout the procedure's execution



Conclusion

- ***Summary***
  - ➔ Instruction set architecture and CPU operations
    (The stored-program concept)
  - ➔ Instruction types, operands and operations
    (R-type and I-type MIPS instructions format)
  - ➔ Decision making and repetition of instruction execution
    (*bne*, *beq* and *j* instructions)
  - ➔ Supporting procedure and context switching
    (Stack operations, nested procedure call, *jal* and *jr* instructions)
- ***Next Lecture***
  - ➔ Other styles of MIPS addressing
  - ➔ Program starting steps
  - ➔ An example to put all together

Reading assignment is sections 2.1, …,2.7 in the textbook