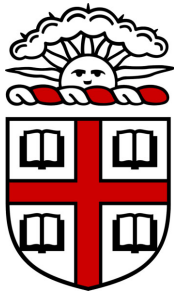
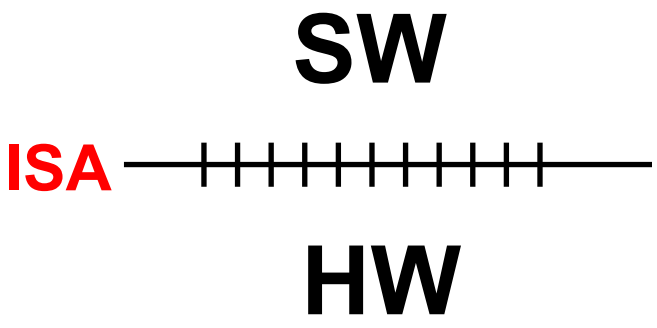
EN164: Design of Computing Systems
Topic 03: Instruction Set Architecture Design

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Spring 2012



ISA is the HW/SW interface

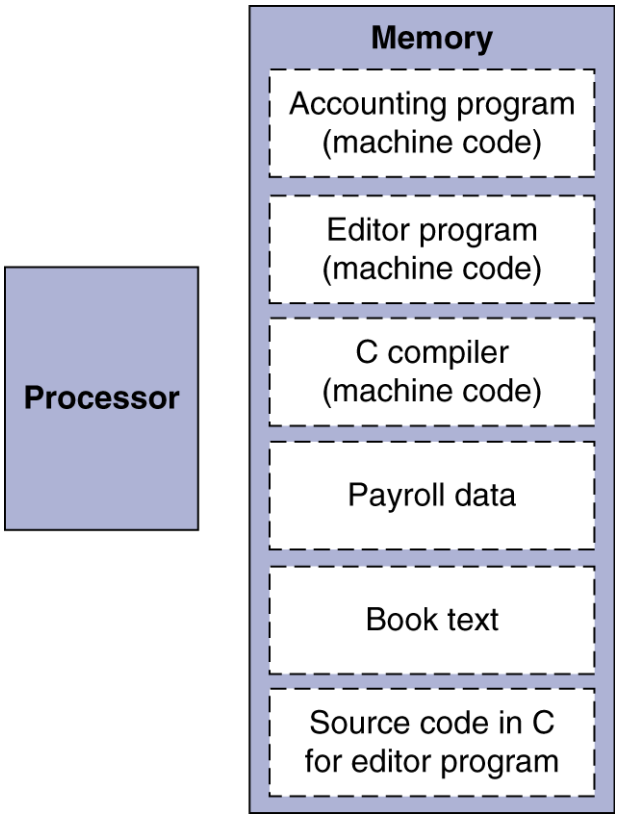


ISA choice determines:

- program size (& memory size)
- complexity of hardware (CPI and f)
- execution time for different applications and domains
- power consumption
- die area (cost)

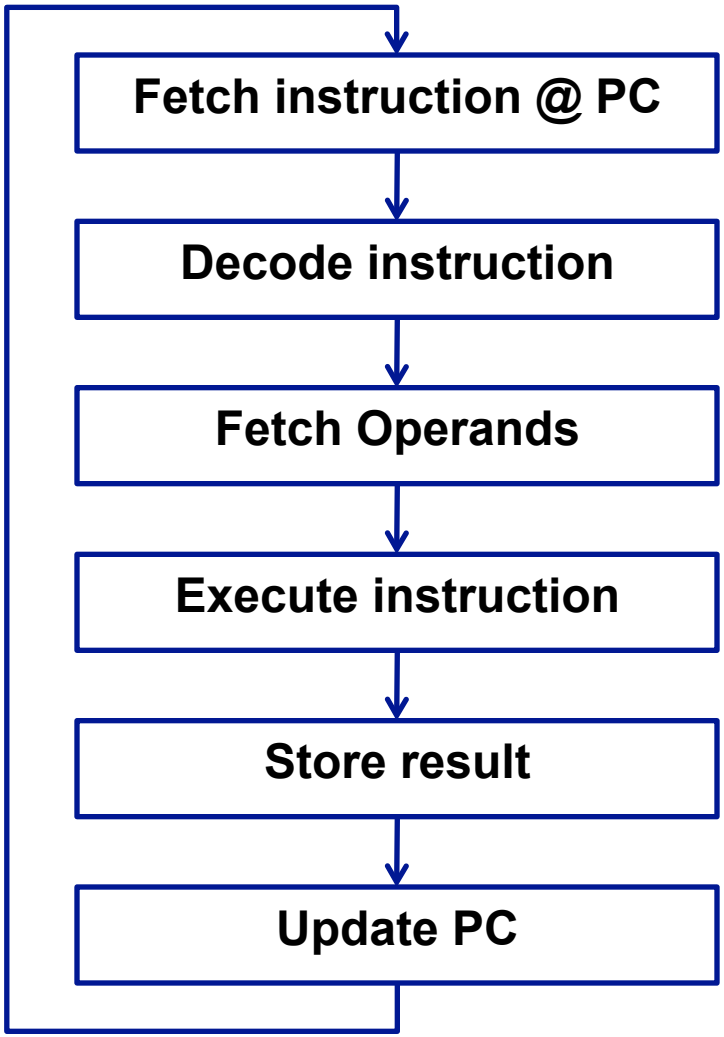
Stored program concept (von Neumann model)

The BIG Picture



- Instructions represented in binary numbers, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Steps in execution of a program



- What is instruction format / size?
- how is it decoded?
- Where are the operands located? What are their sizes?
- What are supported operations?
- How to determine the successor instruction?

Example of an instruction

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

```
add $t0, $s1, $s2
```

assembly language

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

machine language

ISA design choices

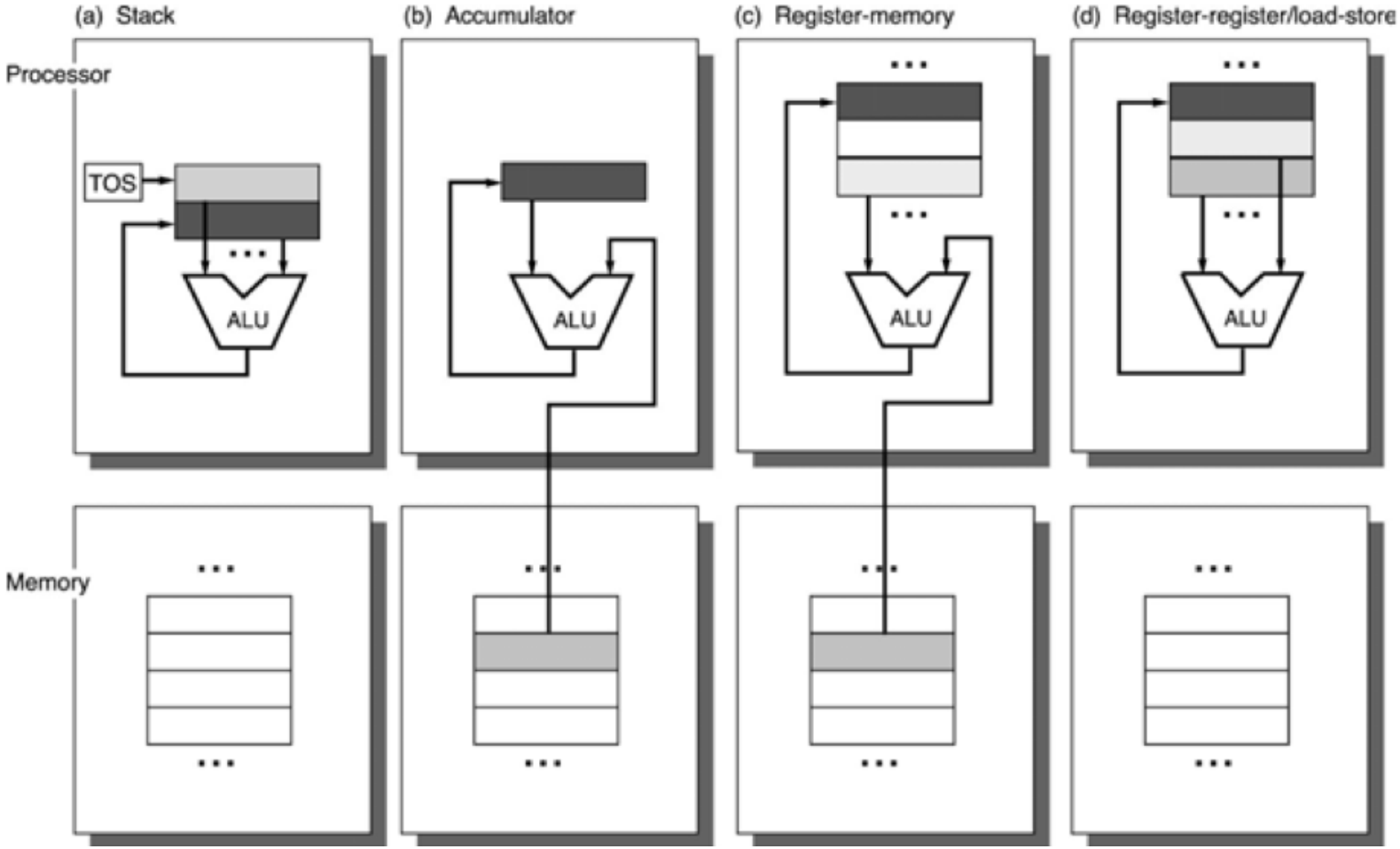
- Number, size (fixed/variable) and format of instructions
- Operations supported (arithmetic, logical, string, floating point, jump, etc)
- Operands supported (bytes, words, signed, unsigned, floating, etc)
- Operand storage (accumulator, stack, registers, memory)
- Addressing modes

Typical operations

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP Add, Subtract, Multiply, Divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
	search, translate

[slide from M. Martin]

Classification of ISAs



[Figure from D. Brooks -- Harvard]

These ISAs give different characteristics in terms of size of programs, number of instructions and CPI.

Examples of ISA

Instruction sequence for C = A + B for the four ISAs

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1, A Load R2, B Add R3, R1, R2 Store C, R3

Some architectures (e.g. x86) support hybrid ISAs for different classes of instructions and/or for backward compatibility.

What makes a good ISA?

- Efficiency of hardware implementation
- Convenience of programming / compiling
- Matches target applications (or generality)
- Compatibility and portability

Four design principles for ISA

1. Simplicity favors regularity
2. Smaller is faster
3. Make the common case fast
4. Good design demands good compromises

*ISA design is an **art**!*

Popular ISAs

- x86 from Intel (laptops, servers)
- ARM (mobile devices)
- MIPS (embedded devices)
- Power and PowerPC from IBM (servers, old Macs)
- and many others still spoken and dead ISAs

MIPS architecture: Example of Reduced Instruction Set Computer (RISC) architecture

- Used as the example throughout the course
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
- 32 bit and 64 bit available
- Will implement and boot a subset MIPS processor in lab

Arithmetic operations

High-level code

```
a = b + c;
a = b - c;
```

MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
sub $s0, $s1, $s2
```

- add: mnemonic indicates what operation to perform
- $s1, $s1: source operands on which the operation is performed
- $s0: destination operand to which the result is written

- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Register operands

- Memory is slow.
- Most architectures have a small set of (fast) registers
- MIPS has a 32 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
  - Written with a $ before their name
- Assembler names
  - $0 always hold value 0
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

MIPS registers

Name	Register Number	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	procedure return values
$a0-$a3	4-7	procedure arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	procedure return address

Format of R-type instructions

R-Type

- *Register-type*
- 3 register operands:
  - `rs`, `rt`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode* (0 for R-type instructions)
  - `funct`: the *function*
    together, the opcode and function tell the computer what operation to perform
  - `shamt`: the *shift amount* for shift instructions, otherwise it's 0

R-Type examples

Assembly Code

```
add $s0, $s1, $s2

sub $t0, $t3, $t5
```

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Note the order of registers in the assembly code:

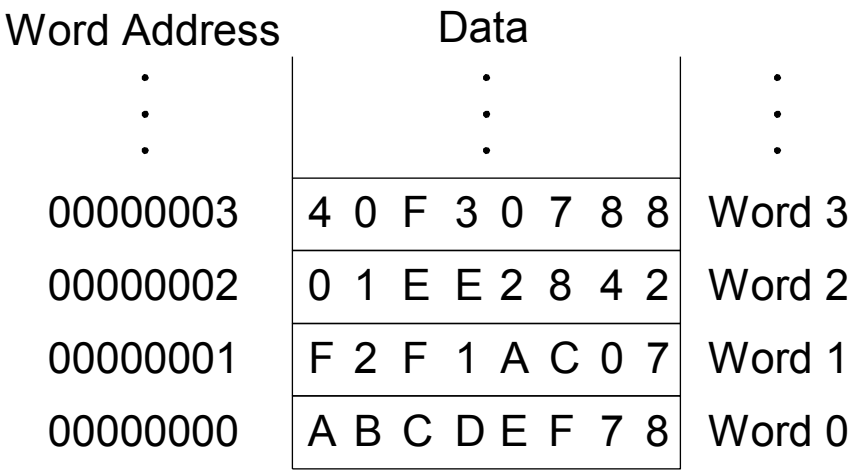
`add rd, rs, rt`

Memory operands

- Too much data to fit in only 32 registers
- Memory is large, but slow
- Commonly used variables kept in registers
- Using a combination of registers and memory, a program can access a large amount of data fairly quickly
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory

Word-addressable memory

- Each 32-bit data word has a unique address

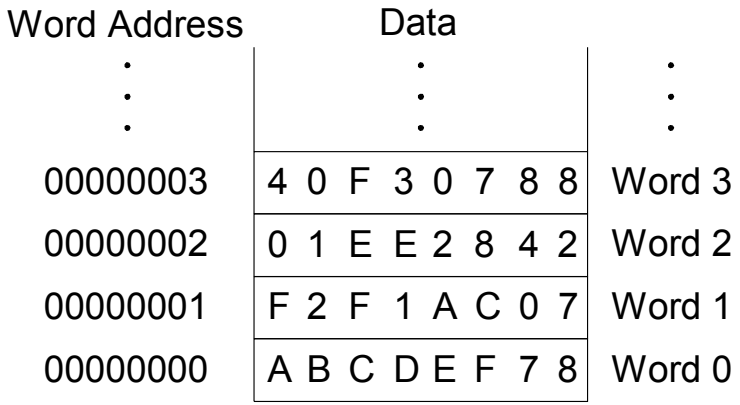


Reading word-addressable memory

- Memory reads are called *loads*
- Mnemonic: *load word* (`lw`)
- Example: read a word of data at memory address 1 into `$s3`
- Memory address calculation:
  - add the *base address* (`$0`) to the *offset* (1)
  - address = ($0 + 1) = 1
- Any register may be used to store the base address.
- `$s3` holds the value 0xF2F1AC07 after the instruction completes.

Assembly code

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

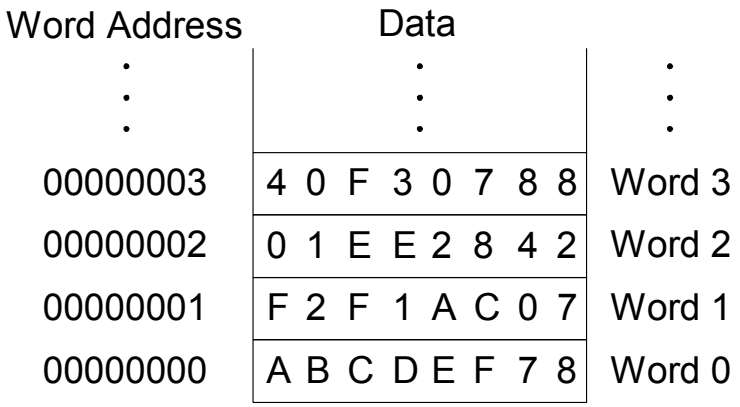


Writing word-addressable memory

- Memory writes are called *stores*
- Mnemonic: *store word* (`sw`)
- Example: Write (store) the value held in `$t4` into memory address 7
- Offset can be written in decimal (default) or hexadecimal
- Memory address calculation:
  - add the base address (`$0`) to the offset (0x7)
  - address: ($0 + 0x7) = 7
- Any register may be used to **store** the base address

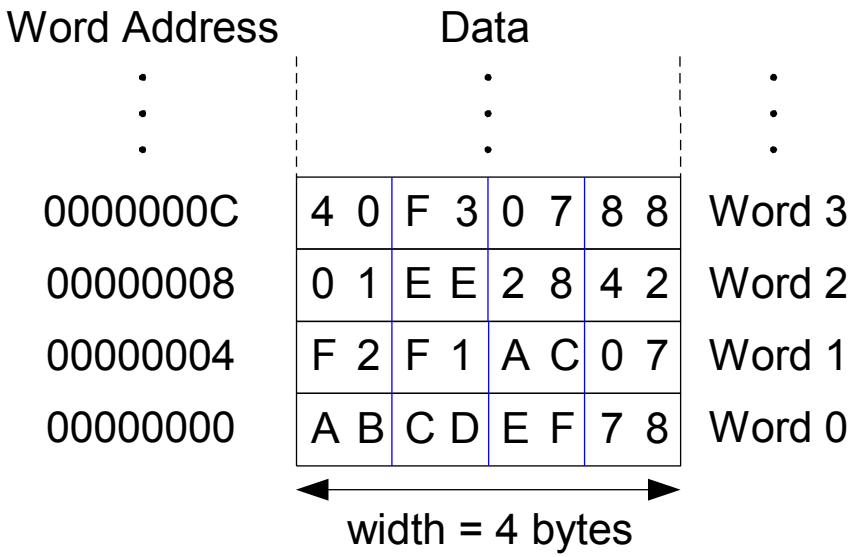
Assembly code

```
sw $t4, 0x7($0)   # write the value in $t4
                  # to memory word 7
```



Byte-addressable memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (`lb`) and store byte (`sb`)
- Each 32-bit words has 4 bytes, so the word address increments by 4

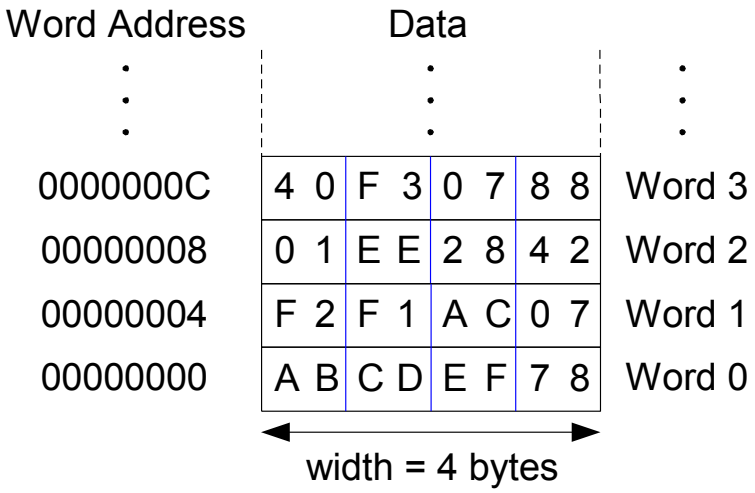


Reading byte-addressable memory

- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is 2 × 4 = 8
  - the address of memory word 10 is 10 × 4 = 40 (0x28)
- Load a word of data at memory address 4 into `$s3`.
- `$s3` holds the value 0xF2F1AC07 after the instruction completes.
- MIPS is byte-addressed, not word-addressed.

MIPS assembly code

```
lw $s3, 4($0)  # read word at address 4 into $s3
```

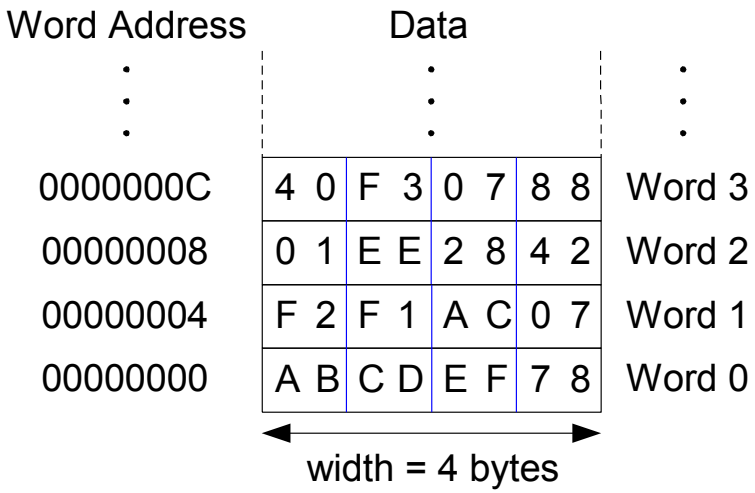


Writing byte-addressed memory

- Example: stores the value held in `$t7` into memory address 0x2C (44)

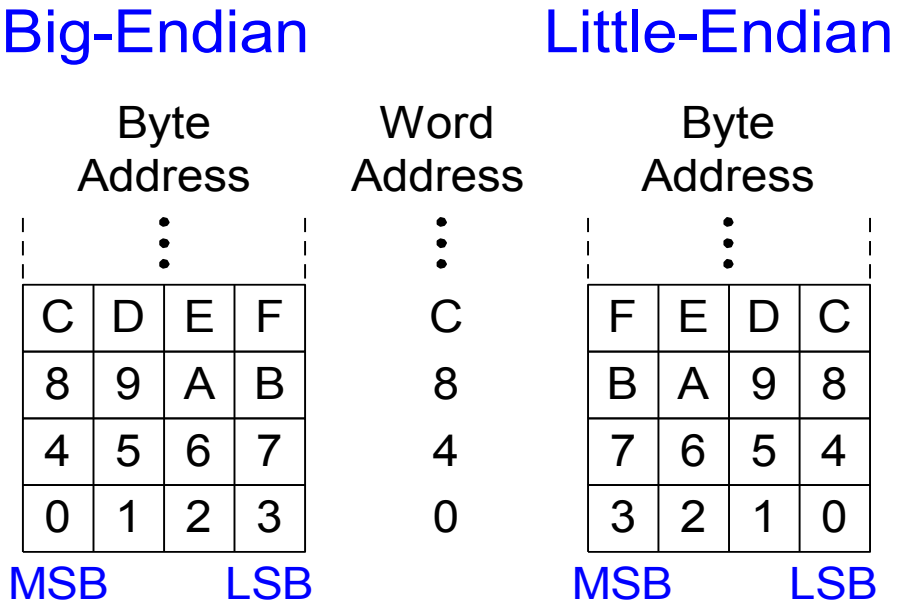
MIPS assembly code

```
sw $t7, 44($0)  # write $t7 into address 44
```



Big-Endian and little-Endian

- How to number bytes within a word?
- Word address is the same for big- or little-endian
- Little-endian: byte numbers start at the little (least significant) end
- Big-endian: byte numbers start at the big (most significant) end

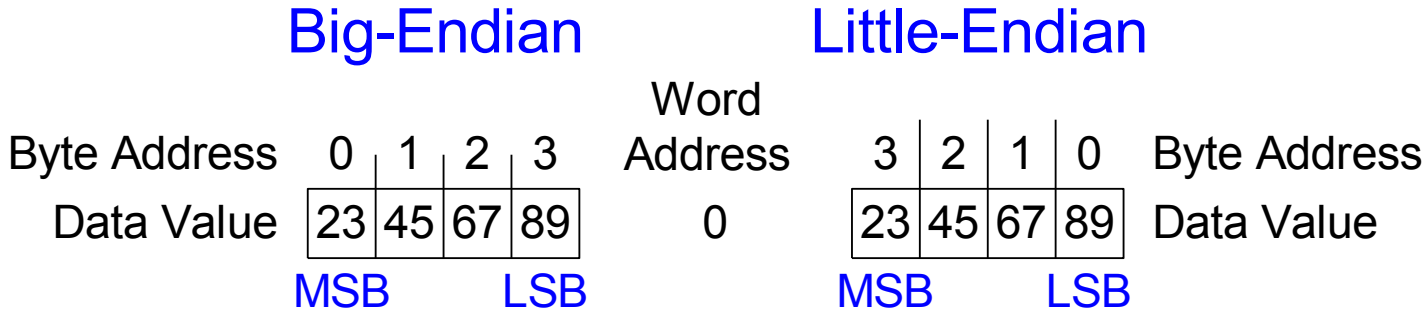


Big- and little- Endian example

- Suppose `$t0` initially contains 0x23456789. After the following program is run on a big-endian system, what value does `$s0` contain? In a little-endian system?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

- Big-endian: 0x00000045
- Little-endian: 0x00000067



Design principle 4 in action

Good design demands good compromises

- Multiple instruction formats allow flexibility
  - `add`, `sub`: use 3 register operands
  - `lw`, `sw`: use 2 register operands and a constant
- Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster).

Constant/Immediate operands

- lw and sw illustrate the use of constants or *immediates*
- Called immediates because they are *immediate*ly available from the instruction
- Immediates don't require a register or memory access.
- The add immediate (addi) instruction adds an immediate to a variable (held in a register).
- An immediate is a 16-bit two's complement number.
- Is subtract immediate (subi) necessary?

High-level code

```
a = a + 4;
b = a - 12;
```

MIPS assembly code

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```

I-Type format instructions

- *Immediate-type*
- 3 operands:
  - `rs`, `rt`: register operands
  - `imm`: 16-bit two's complement immediate
- Other fields:
  - `op`: the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by the opcode

I-Type

I-Type examples

Assembly Code

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw   $t2, 32($0)
sw   $s1,  4($t1)
```

Field Values

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4
6 bits	5 bits	5 bits	16 bits

Note the differing order of registers in the assembly and machine codes:

```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

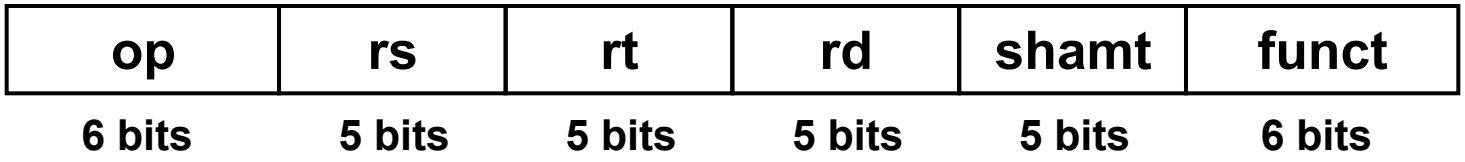
Logical operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

- Useful for extracting and inserting groups of bits in a word

Shift operations

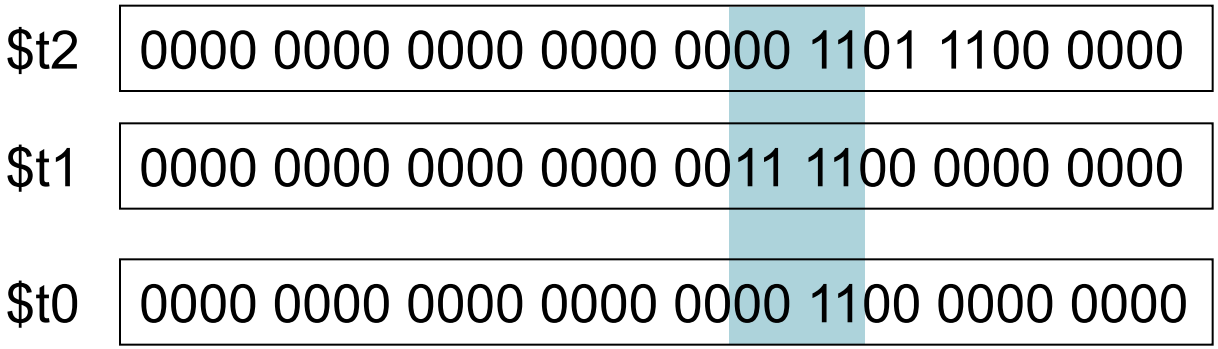


- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

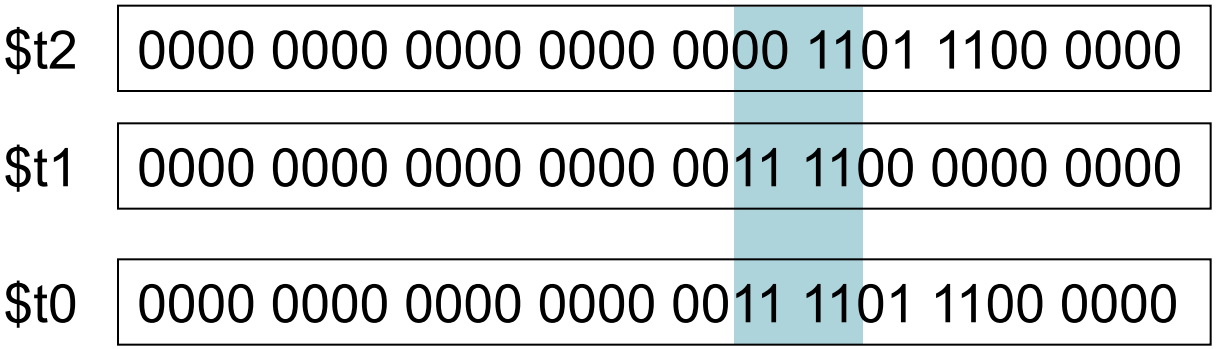
```
and $t0, $t1, $t2
```



OR operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```



NOT operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Conditional operators

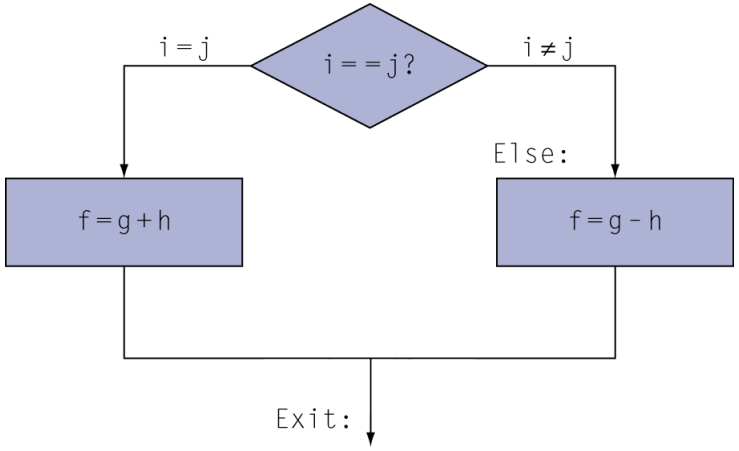
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Compiling if statement

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j   Exit
Else:   sub $s0, $s1, $s2
Exit:   …
```

Compiling loop statements

- C code:

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

  - i in $s3, k in $s5, address of save in $s6
- Compiled MIPS code:

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

More conditional operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with `beq`, `bne`

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Branch instruction design

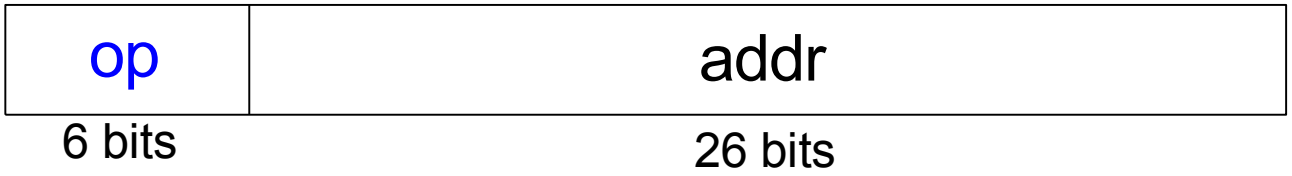
- Why not `blt`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

Summary of instruction formats

R-Type

I-Type

J-Type



Procedure calls

Procedure calling conventions:

- Caller:
  - passes **arguments** to callee.
  - jumps to the callee
- Callee:
  - **performs the procedure**
  - **returns the result** to caller
  - **returns to the point of call**
  - **must not overwrite** registers or memory needed by the caller

MIPS conventions:

- Call procedure: jump and link (`jal`)
- Return from procedure: jump register (`jr`)
- Argument values: $a0 - $a3
- Return value: $v0 -$v1

Procedure calls

High-level code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

MIPS assembly code

```
0x00400200 main: jal  simple
0x00400204       add  $s0, $s1, $s2
...

0x00401020 simple: jr $ra
```

`void` means that `simple` doesn't return a value.

`jal:` jumps to `simple` and saves `PC`+4 in the return address register (`$ra`). In this case, `$ra` = 0x00400204 after `jal` executes.

`jr $ra:` jumps to address in `$ra`, in this case 0x00400204.

Input arguments and return values

High-level code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;                // return value
}
```

Input arguments and return values

MIPS assembly code

```
main:
   ...
   addi $a0, $0, 2     # argument 0 = 2
   addi $a1, $0, 3     # argument 1 = 3
   addi $a2, $0, 4     # argument 2 = 4
   addi $a3, $0, 5     # argument 3 = 5
   jal  diffofsums     # call procedure
   add  $s0, $v0, $0   # y = returned value
   ...

# $s0 = result
diffofsums:
   add $t0, $a0, $a1   # $t0 = f + g
   add $t1, $a2, $a3   # $t1 = h + i
   sub $s0, $t0, $t1   # result = (f + g) - (h + i)
   add $v0, $s0, $0    # put return value in $v0
   jr  $ra             # return to caller
```

MIPS conventions:

- Argument values: `$a0` - `$a3`
- Return value: `$v0`

Potential problems in procedural calling

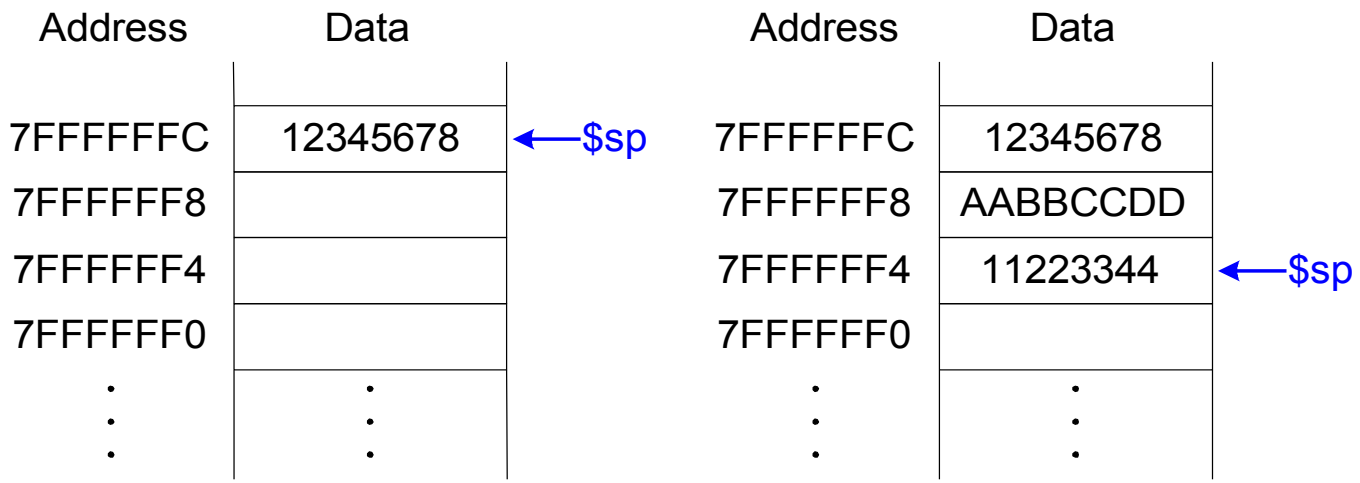
MIPS assembly code

```
# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

- `diffofsums` overwrote 3 registers: `$t0`, `$t1`, and `$s0`
- `diffofsums` can use the *stack* to temporarily store registers

The stack

- Memory used to temporarily save variables
- Like a stack of dishes, last-in-first-out (LIFO) queue
- *Expands*: uses more memory by growing down (from higher to lower memory addresses) when more space is needed
- *Contracts*: uses less memory when the space is no longer needed
- Stack pointer: `$sp`, points to top of the stack



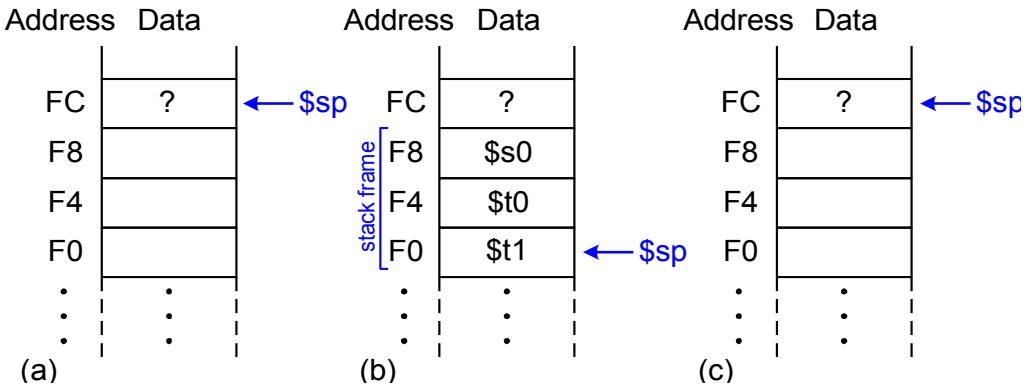
How procedures use the stack

- Called procedures must have no other unintended side effects.
- But `diffofsums` overwrites 3 registers: `$t0`, `$t1`, `$s0`

```
# MIPS assembly
# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

Storing register values on the stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12   # make space on stack
                       # to store 3 registers
  sw   $s0, 8($sp)     # save $s0 on stack
  sw   $t0, 4($sp)     # save $t0 on stack
  sw   $t1, 0($sp)     # save $t1 on stack
  add  $t0, $a0, $a1   # $t0 = f + g
  add  $t1, $a2, $a3   # $t1 = h + i
  sub  $s0, $t0, $t1   # result = (f + g) - (h + i)
  add  $v0, $s0, $0    # put return value in $v0
  lw   $t1, 0($sp)     # restore $t1 from stack
  lw   $t0, 4($sp)     # restore $t0 from stack
  lw   $s0, 8($sp)     # restore $s0 from stack
  addi $sp, $sp, 12    # deallocate stack space
  jr   $ra             # return to caller
```



Protocol for preserving registers

Preserved *Callee-Saved*	Nonpreserved *Caller-Saved*
`$s0 - $s7`	`$t0 - $t9`
`$ra`	`$a0 - $a3`
`$sp`	`$v0 - $v1`
stack above `$sp`	stack below `$sp`

Multiple procedure calls

```
proc1:
   addi $sp, $sp, -4    # make space on stack
   sw   $ra, 0($sp)     # save $ra on stack
   jal  proc2
   ...
   lw   $ra, 0($sp)     # restore $s0 from stack
   addi $sp, $sp, 4     # deallocate stack space
   jr  $ra              # return to caller
```

Storing saved registers on the stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4   # make space on stack to
                      # store one register
  sw  $s0, 0($sp)     # save $s0 on stack
                      # no need to save $t0 or $t1
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  lw  $s0, 0($sp)     # restore $s0 from stack
  addi $sp, $sp, 4    # deallocate stack space
  jr  $ra             # return to caller
```

Recursive procedure calls

High-level code

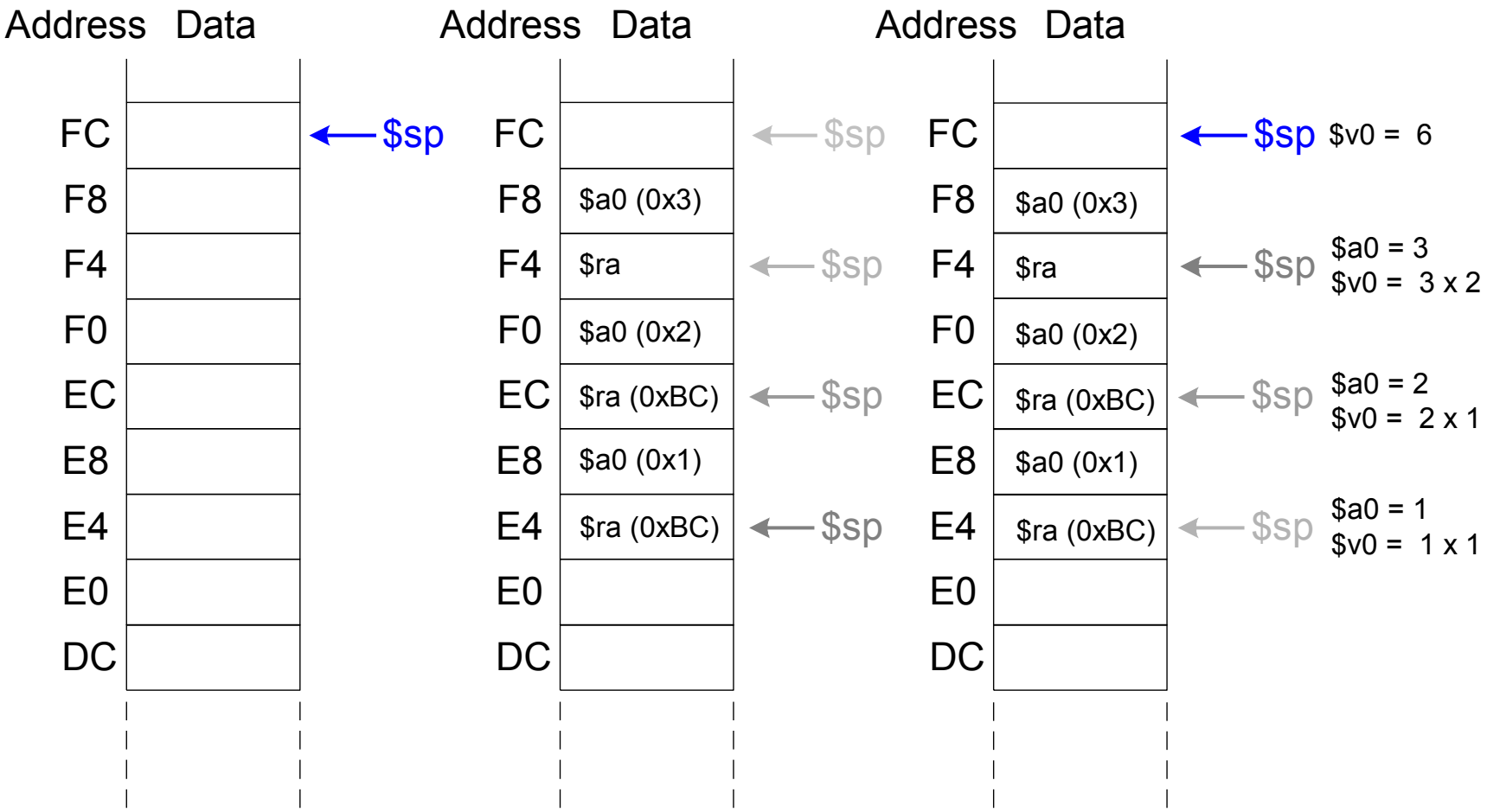
```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

Recursive procedure calls

MIPS assembly code

```
0x90 factorial:
0x94
0x98
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC
0xB0            jr   $ra           # return
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC
0xC0
0xC4
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```

Stack during recursive calls



Procedural call summary

- Caller
  - Put arguments in $a0-$a3
  - Save any registers that are needed ($ra, maybe $t0-t9)
  - jal callee
  - Restore registers
  - Look for result in $v0
- Callee
  - Save registers that might be disturbed ($s0-$s7)
  - Perform procedure
  - Put result in $v0
  - Restore registers
  - jr $ra

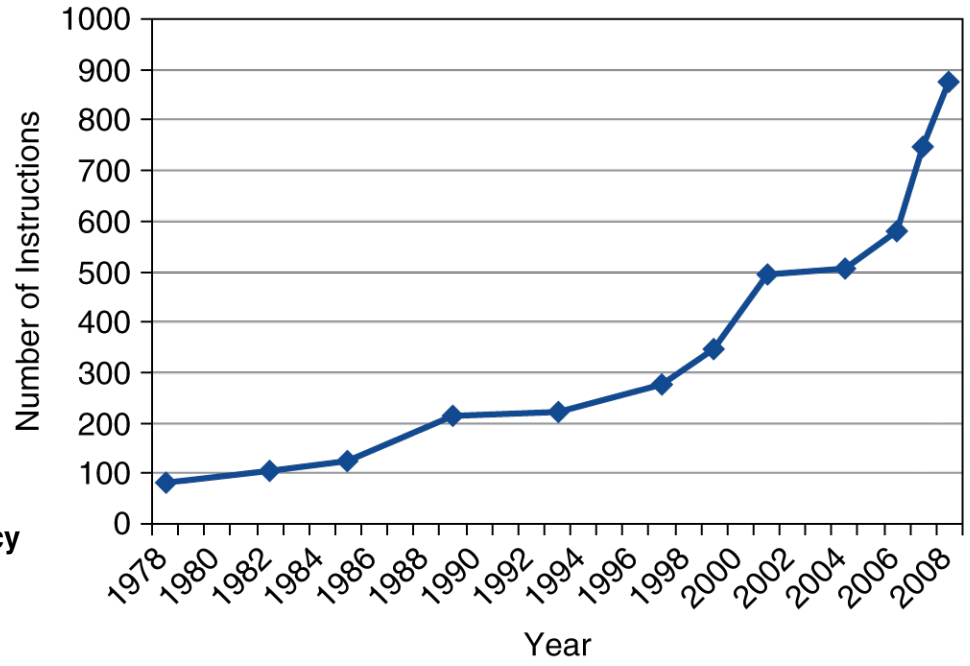
x86 ISA: Example of Complex Instruction Set Computer (CISC) architecture

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accumulate more instructions

° Rank instruction Integer Average Percent total executed

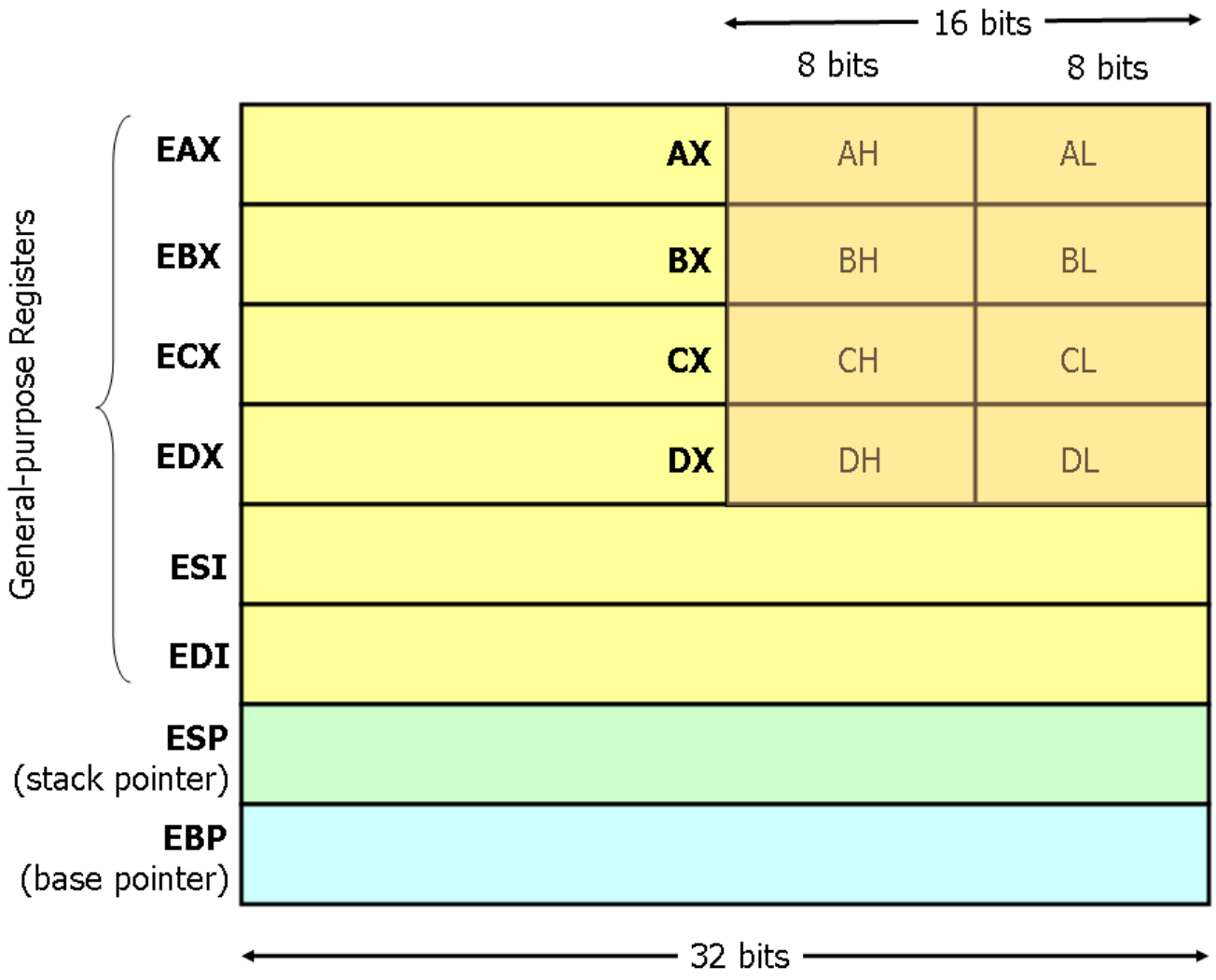
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

° Simple instructions dominate instruction frequency



x86 instruction set

x86 registers



x86 assembly language examples

```
# move the 4 bytes in memory at the address contained in EBX into EAX
mov eax, [ebx]
# move the 4 bytes of data at address ESI+4*EBX into EDX
mov edx, [esi+4*ebx]
# eax = eax + ebx
add eax, ebx
# add ebx to memory content at address memory_location
add memory_location, ebx
# add 10 the data at address EAX
add [eax], 10
# push content of eax into top of stack
push eax
# compare two registers, register – memory or memory memory
cmp eax, ebx
cmp [eax], [ebx]
# conditional jumps
jl, jeq, jg, jge, jle, jne
```