MIPS architecture and language

These transparencies are based on those provided with the following book:
David Money Harris and Sarah L. Harris, “Digital Design and Computer Architecture”, 2nd Edition, 2012, Elsevier

Assembly Language

- **Instructions:** commands in a computer's language
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)
- **MIPS** architecture:
  - Developed by John Hennessy and his colleagues at Stanford in the 1980's.
  - Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco

Once you've learned one architecture, it's easy to learn others

Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

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

Instructions: Addition

C Code

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

MIPS assembly code

```
add a, b, c
```

- **add:** mnemonic indicates operation to perform
- **b, c:** source operands (on which the operation is performed)
- **a:** destination operand (to which the result is written)

Instructions: Subtraction

- Similar to addition - only mnemonic changes

C Code

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

MIPS assembly code

```
sub a, b, c
```

- **sub:** mnemonic
- **b, c:** source operands
- **a:** destination operand

Design Principle 1

Simplicity favors regularity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Easier to encode and handle in hardware

Multiple Instructions

- More complex code is handled by multiple MIPS instructions.

C Code

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

MIPS assembly code

```
add t, b, c  # t = b + c
sub a, t, d  # a = t - d
```

- What follows the '#' is a **comment**

Design Principle 2

Make the common case fast

- MIPS includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions
- MIPS is a ***reduced instruction set computer*** **(RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are ***complex instruction set computers*** **(CISC)**

Operands

- Operand location: physical location in computer
  - Registers
  - Memory
  - Constants (also called *immediates*)

Operands: Registers

- MIPS has 32 32-bit registers
- Registers are faster than memory
- MIPS is called “32-bit architecture” because it operates on 32-bit data

Design Principle 3

Smaller is Faster

- MIPS includes only a small number of registers

MIPS Register Set

Name	Register Number	Usage
`$0`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2-3	function return values
`$a0-$a3`	4-7	function 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	function return address

Operands: Registers

- Registers:
  - `$` before name
  - Example: `$0`, “register zero”, “dollar zero”
- Registers used for specific purposes:
  - `$0` always holds the constant value 0.
  - the *saved registers*, `$s0-$s7`, used to hold variables
  - the *temporary registers*, `$t0 - $t9,` used to hold intermediate values during a larger computation
  - discuss others later

Instructions with Registers

- Revisit add instruction

C Code

```
a = b + c
```

MIPS assembly code

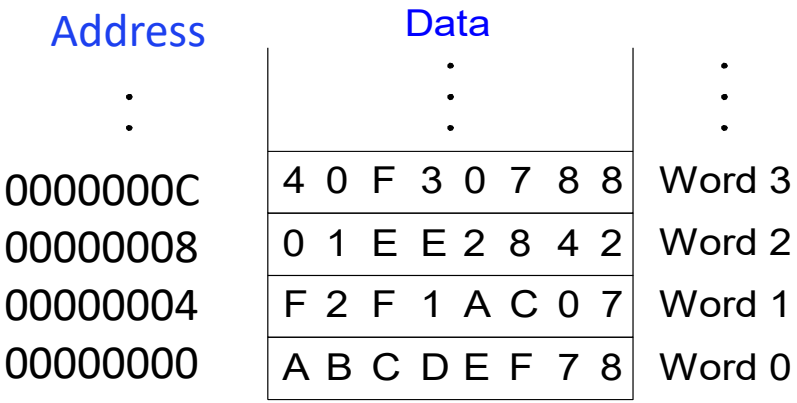
```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
```

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables kept in registers

Reading Byte-Addressable Memory

- The address of a memory word is always a multiple of 4. For example,
  - the address of memory word #2 is $2 \times 4 = 8$
  - the address of memory word #10 is $10 \times 4 = 40$ (0x28)
- **MIPS is byte-addressed**



Reading Memory

- Memory read called ***load***
- **Mnemonic:** *load word* (`lw`)
- **Format:**

```
lw $s0, 8($t1)
```

- **Address calculation:**
  - add *base address* (`$t1`) to the *offset* (8)
  - address = (`$t1` + 8)
- **Result:**
  - `$s0` holds the value at address (`$t1` + 8)

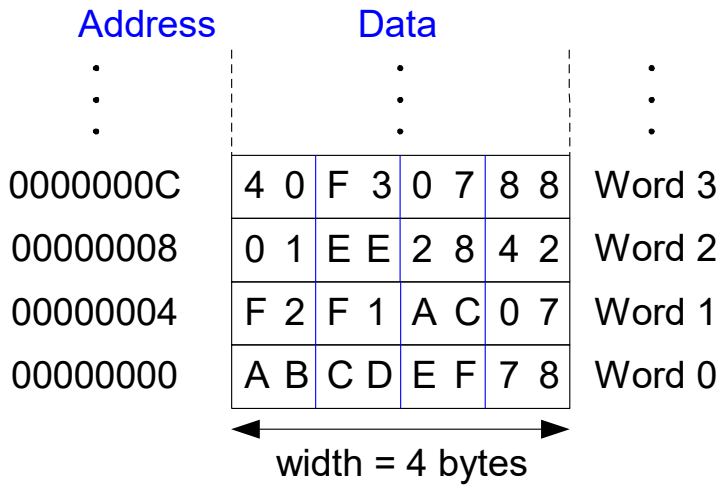
Any register may be used as base address

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 4 into `$s3`.
- `$s3` holds the value 0xF2F1AC07 after load

MIPS assembly code

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



Writing Memory

- Memory write are called ***store***
- **Mnemonic:** *store word* (`sw`)

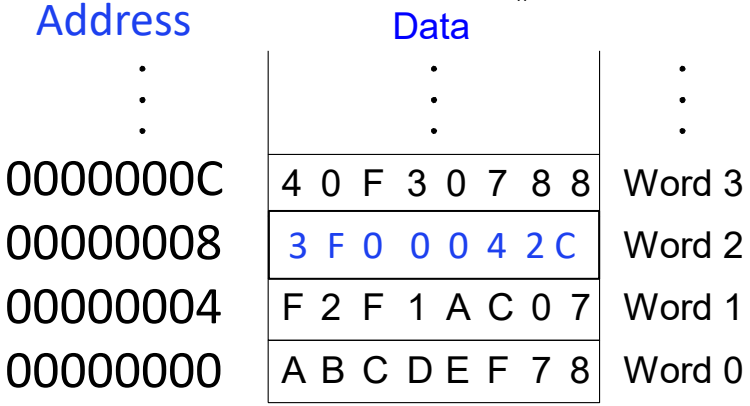
Writing Memory

- **Example:** Write (store) the value in `$t4` into memory address 8
  - add the base address (`$0`) to the offset (0x8)
  - address: ($0 + 0x8) = 8

Offset can be written in decimal (default) or hexadecimal

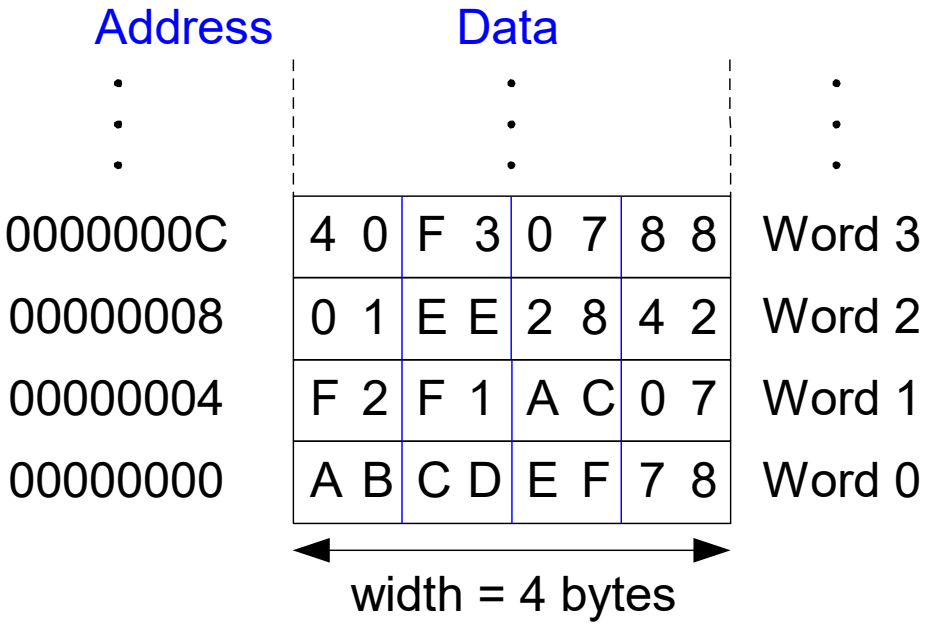
Assembly code

```
sw $t4, 0x8($0)   # write the value in $t4
                  # to memory address 8
```



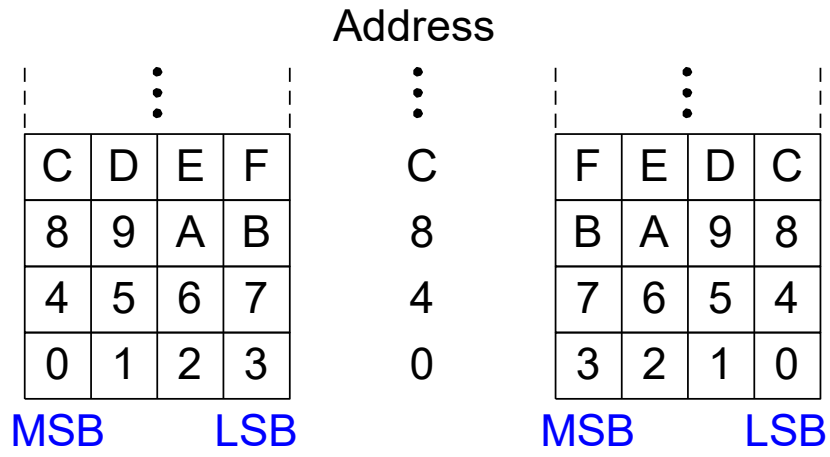
Byte-Addressable Memory

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



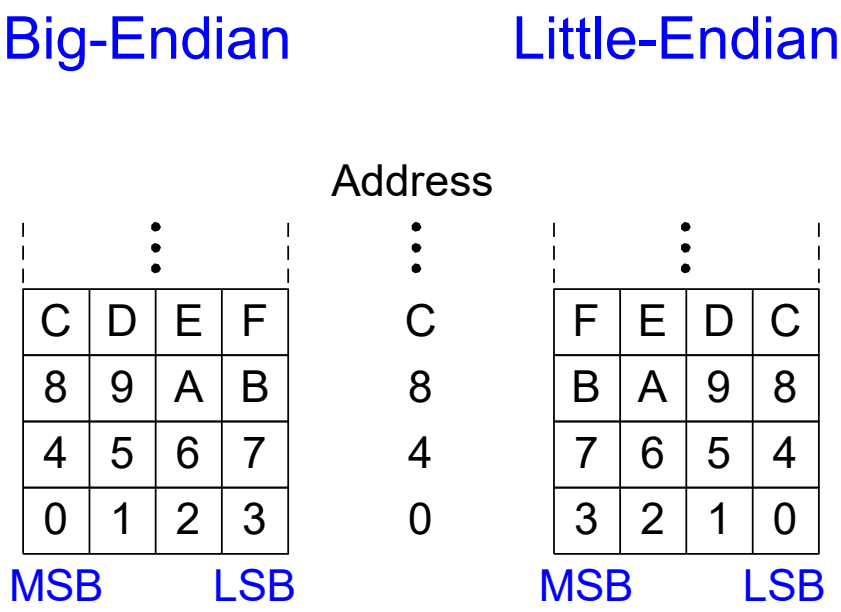
Big-Endian & Little-Endian Memory

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



Big-Endian & Little-Endian Memory

- Jonathan Swift's *Gulliver's Travels*: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- It doesn't really matter which addressing type used – except when the two systems need to share data!



Big-Endian & Little-Endian Example

- Suppose `$t0` initially contains 0x23456789
- After following code runs on big-endian system, what value is `$s0`?

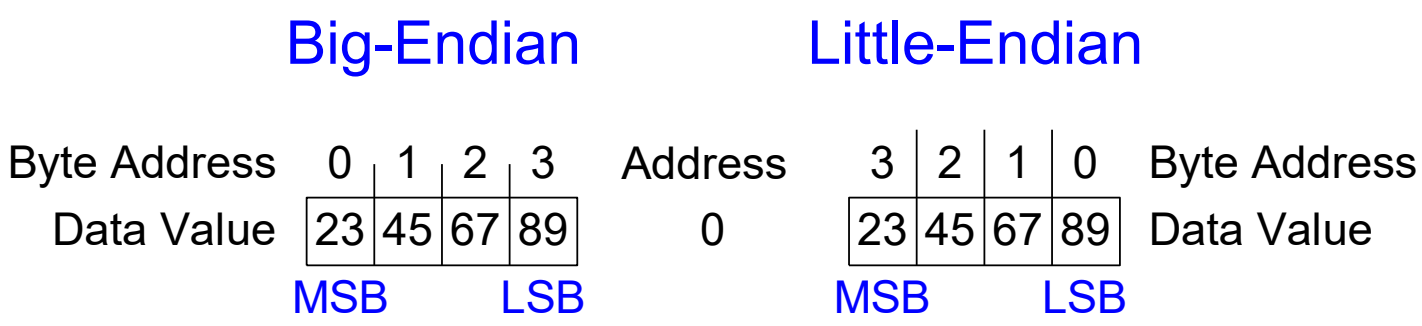
```
sw $t0, 0($0)
lb $s0, 1($0)
```

Big-Endian & Little-Endian Example

- Suppose `$t0` initially contains 0x23456789
- After following code runs on big-endian system, what value is `$s0`?

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

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



Word Alignment

- In the MIPS architecture, word addresses for `lw` e `sw` must be word aligned. That is, the address must be divisible by 4.
- Thus, the instruction

```
lw $s0, 7($0)
```

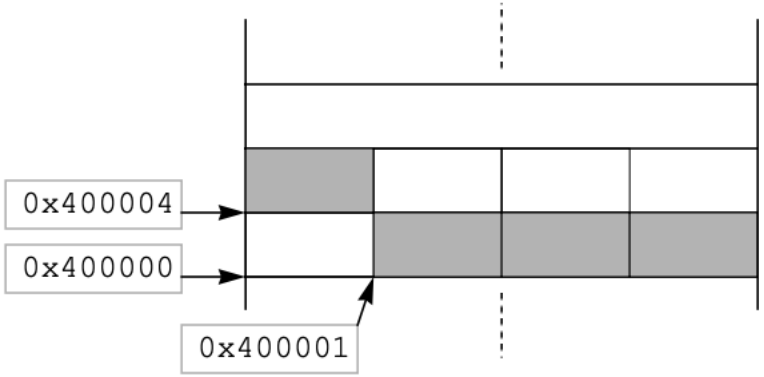
- is an illegal instruction.

Reading Byte-Addressable Memory

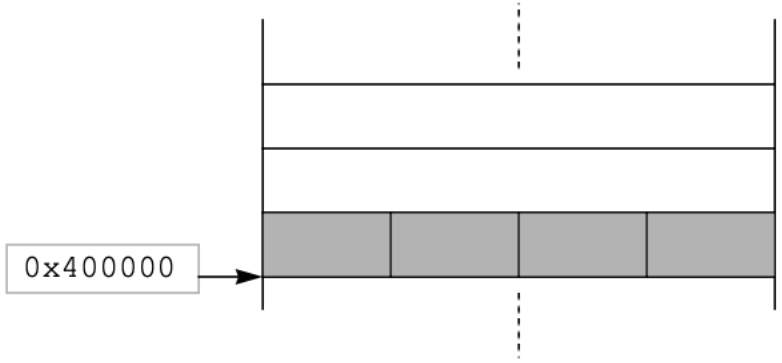
- The MIPS assembler allows user to control data alignment using the .align directive
- If the programmer does not use the directive, the data will be automatically aligned in memory at the proper boundaries
- **An unaligned data in memory may require multiple memory accesses and special processing**
- All instructions are of the same size (word), they must be aligned in memory.

Reading Byte-Addressable Memory

Data	Size (bytes)	Store at address
byte	$1 = 2^0$	Any address
half-word	$2 = 2^1$	Multiple of 2
word	$4 = 2^2$	Multiple of 4
double	$8 = 2^3$	Multiple of 8



Unaligned word stored at address 0x400001



Aligned word stored at address 0x400000

Design Principle 4

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).

Operands: Constants/Immediates

- `lw` and `sw` use constants or *immediates*
- *immediate*ly available from instruction
- 16-bit two's complement number
- `addi`: add immediate
- Subtract immediate (`subi`) necessary?

C Code

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

MIPS assembly code

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

Machine Language

- Binary representation of instructions
- Computers only understand 1's and 0's
- 32-bit instructions
  - Simplicity favors regularity: 32-bit data & instructions
- 3 instruction formats:
  - **R-Type:** register operands
  - **I-Type:** immediate operand
  - **J-Type:** for jumping (discuss later)

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*
    with opcode, tells computer what operation to perform
  - `shamt`: the *shift amount* for shift instructions, otherwise it's 0

R-Type

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
```

I-Type

- *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 opcode

I-Type

I-Type Examples

Assembly Code	op	rs	rt	imm
`addi $s0, $s1, 5`	8	17	16	5
`addi $t0, $s3, -12`	8	19	8	-12
`lw $t2, 32($0)`	35	0	10	32
`sw $s1, 4($t1)`	43	9	17	4
	6 bits	5 bits	5 bits	16 bits

Field Values

Note the differing order of registers in 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	

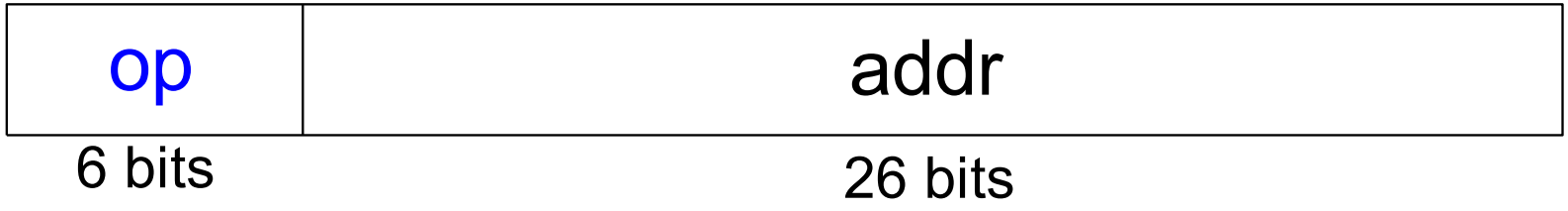
Sign Extension

- I-type instructions have a 16-bit immediate field, but the immediates are used in 32-bit operations. For example, lw adds a 16-bit offset to a 32-bit base register
- What should go in the upper half of the 32 bits?
- For positive immediates, the upper half should be all 0's, but for negative immediates, the upper half should be all 1's
- This is called *sign extension*
- An N-bit two's complement number is sign-extended to an M-bit number (M>N) by copying the sign bit (most significant bit) of the N-bit number into al of the upper bits of the M-bit. Sign-extending a two's complement number does not change its value.

Machine Language: J-Type

- *Jump-type*
- 26-bit address operand (`addr`)
- Used for jump instructions (`j`)

J-Type

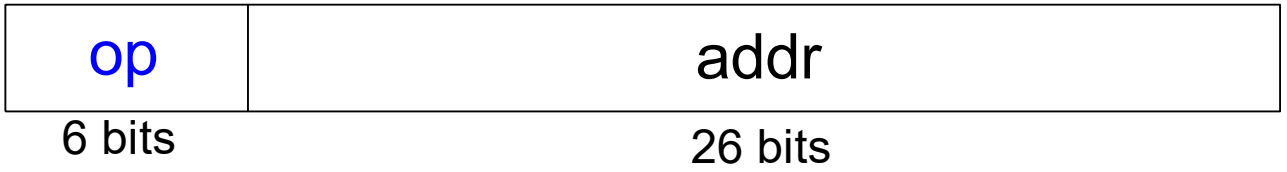


Review: Instruction Formats

R-Type

I-Type

J-Type



Homework #1

- Which is the machine instruction correspondent to the following assembly instruction?

```
add $t0, $t1, $t2
```

Homework #2

- Which is the machine instruction correspondent to the following assembly instruction?

addi $t0, $t1, 0x1234

Homework #3

- Which is the assembly instruction correspondent to the following machine instruction?

  1000 1100 0000 1000 0000 0000 0001 0100

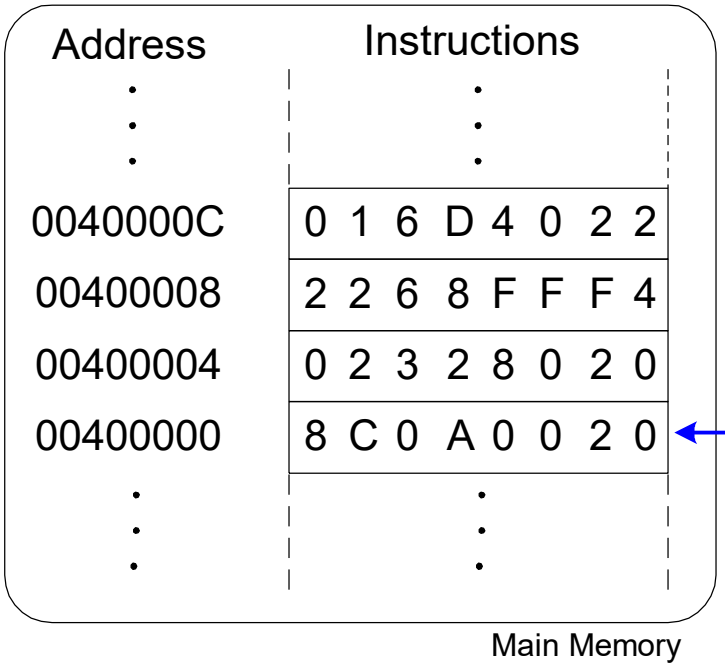
Power of the Stored Program

- 32-bit instructions & data stored in memory
- Sequence of instructions: only difference between two applications
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- Program Execution:
  - Processor *fetches* (reads) instructions from memory in sequence
  - Processor performs the specified operation

The Stored Program

Assembly Code	Machine Code
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`

Stored Program

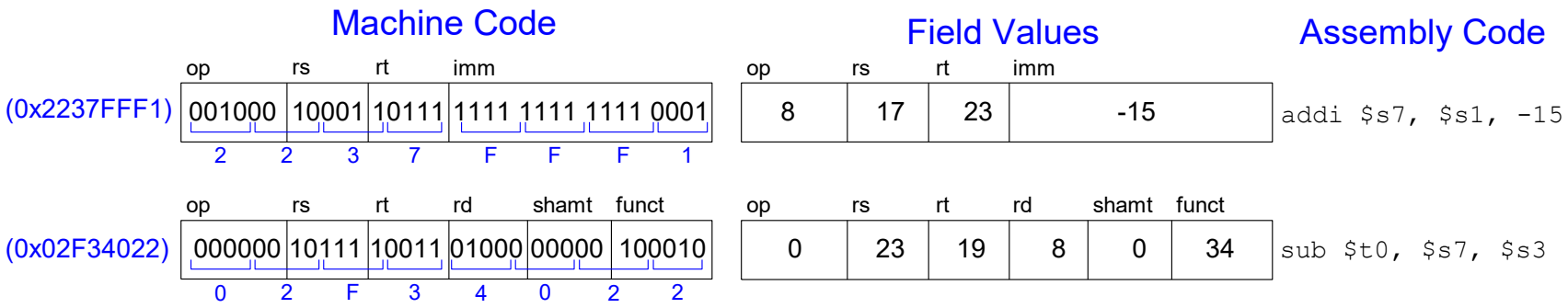


In MIPS programs, the instructions are normally stored starting at address `0x00400000`

Program Counter (PC): keeps track of current instruction

Interpreting Machine Code

- Start with opcode: tells how to parse rest
- If opcode all 0's
  - R-type instruction
  - Function bits tell operation
- Otherwise
  - opcode tells operation



Programming

- High-level languages:
  - e.g., C, Java, Python
  - Written at higher level of abstraction
- Common high-level software constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

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.

Logical Instructions

- **`and`, `or`, `xor`, `nor`**
  - `and`: useful for **masking** bits
    - Masking all but the least significant byte of a value:
      0xF234012F AND 0x000000FF = 0x0000002F
  - `or`: useful for **combining** bit fields
    - Combine 0xF2340000 with 0x000012BC:
      0xF2340000 OR 0x000012BC = 0xF23412BC
  - `nor`: useful for **inverting** bits:
    - A NOR $0 = NOT A
- **`andi`, `ori`, `xori`**
  - 16-bit immediate is *zero-extended* (*not* sign-extended)
  - `nori` not needed

Zero extension

- Most MIPS instructions (like `addi`, `lw` and `sw`) do sign extension to support both positive and negative immediates
- An exception to this rule is that logical operations (`andi`, `ori`, `xori`) place 0's in the upper half.
- That is called *zero extension*.

Homework

- The "**`nori`**" is not part of the MIPS instruction set, because the same functionality can be implemented using existing instructions.

- Write a short assembly code snippet that has the following functionality

```
$t0 = $t1 NOR 0xF234
```

- Use as few instructions as possible.

Logical Instructions Example 1

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Result

$s3								
$s4								
$s5								
$s6								

Logical Instructions Example 1

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

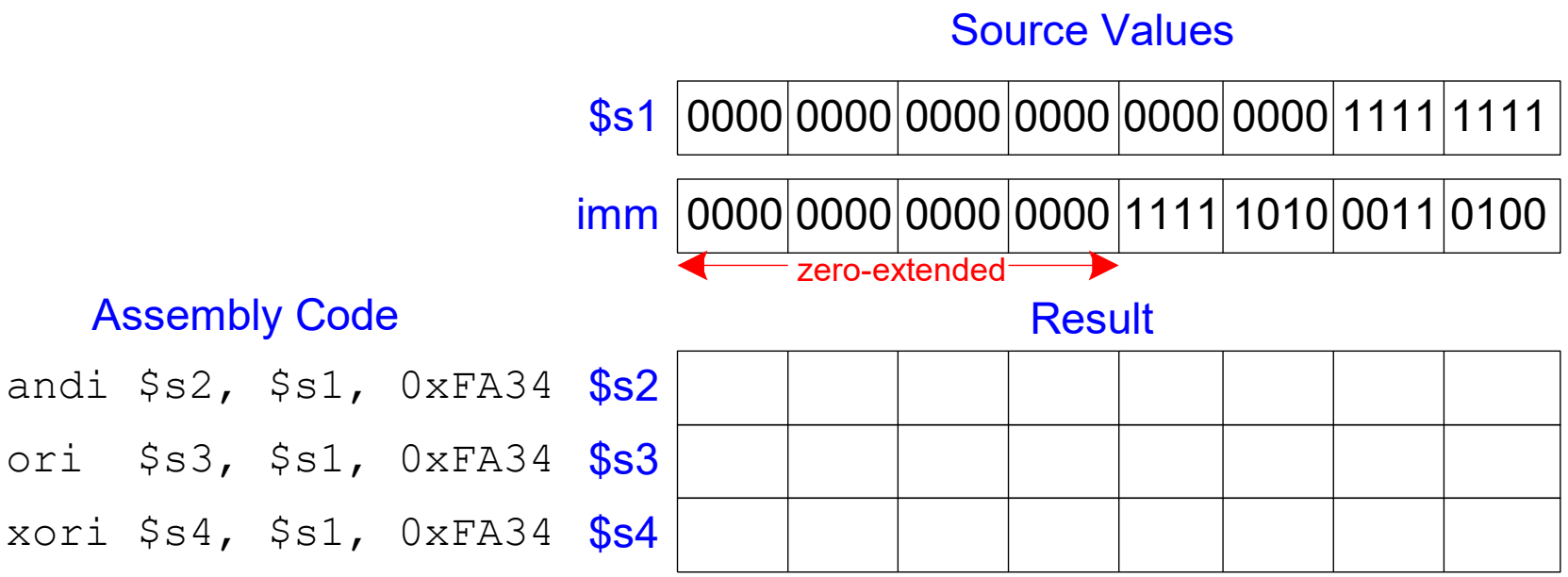
Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

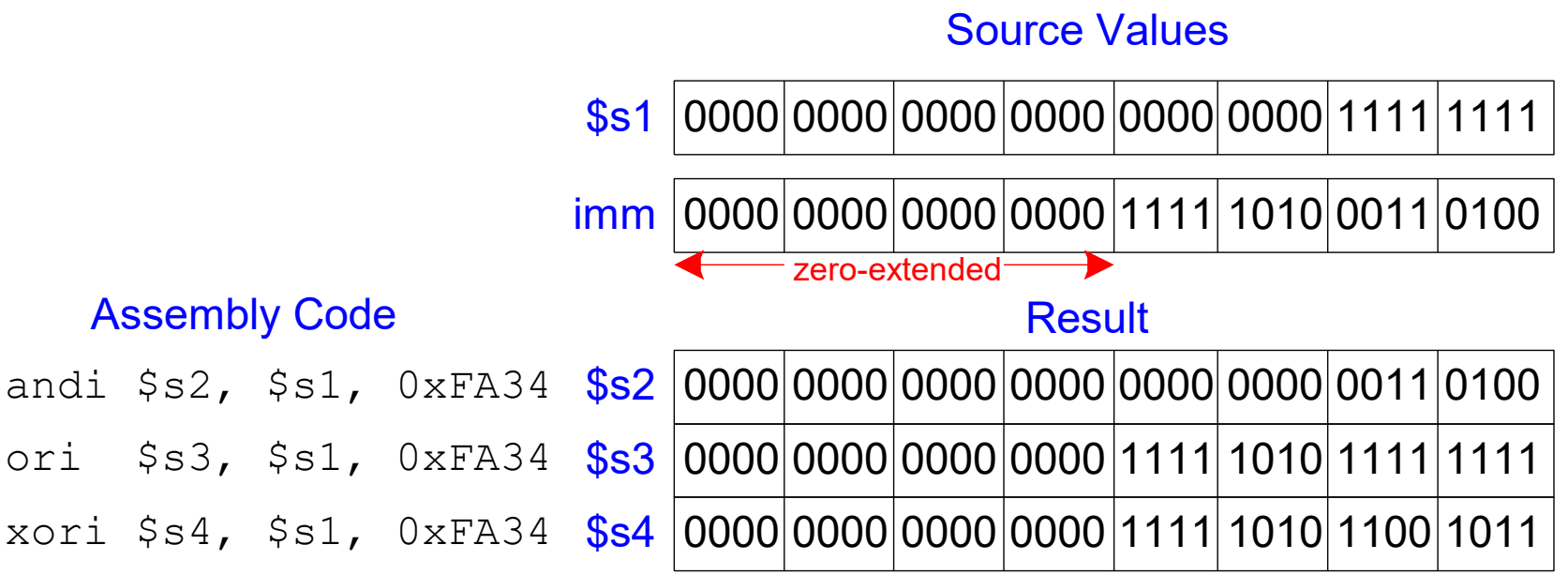
Result

$s3	0100	0110	1010	0001	0000	0000	0000	0000
$s4	1111	1111	1111	1111	1111	0000	1011	0111
$s5	1011	1001	0101	1110	1111	0000	1011	0111
$s6	0000	0000	0000	0000	0000	1111	0100	1000

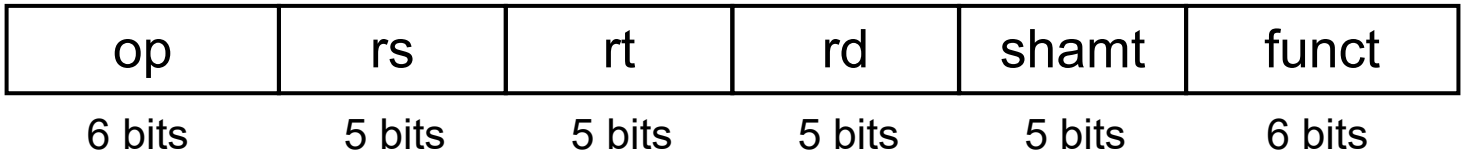
Logical Instructions Example 2



Logical Instructions Example 2



Shift Instructions

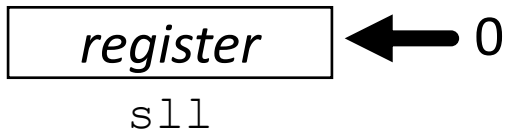


- `sll:` shift left logical
  - **Example:** `sll $t0, $t1, 5  # $t0 <= $t1 << 5`
- `srl:` shift right logical
  - **Example:** `srl $t0, $t1, 5  # $t0 <= $t1 >> 5`
- `sra:` shift right arithmetic
  - **Example:** `sra $t0, $t1, 5  # $t0 <= $t1 >>> 5`

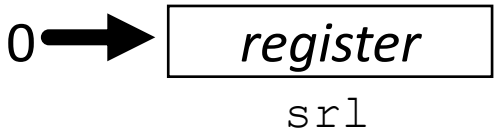
- Shamt field: how many positions to shift.

Logical Shift Instructions

- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i

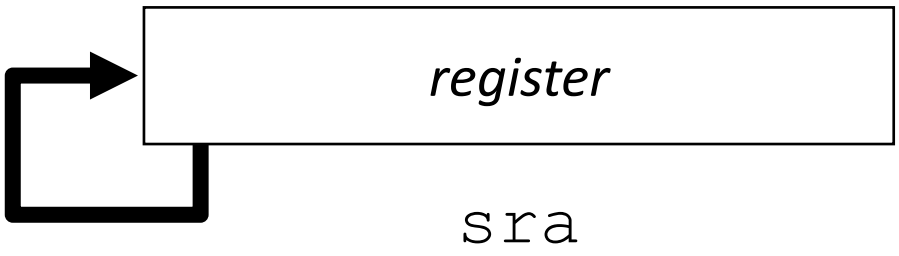


- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)



Arithmetic Shift Instructions

- Right shifts can be either
  - logical: (0's shift into the most significant bits)
  - arithmetic: the sign bit shifts into the most significant bits
- `sra` by *i* bits divides by 2^i



Variable Shift Instructions

- `sllv:` shift left logical variable
  - **Example:** `sllv $t0, $t1, $t2 # $t0 <= $t1 << $t2`
- `srlv:` shift right logical variable
  - **Example:** `srlv $t0, $t1, $t2 # $t0 <= $t1 >> $t2`
- `srav:` shift right arithmetic variable
  - **Example**: `srav $t0, $t1, $t2 # $t0 <= $t1 >>> $t2`

Shift Instructions

Assembly Code	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 2`	0	0	17	8	2	0
`srl $s2, $s1, 2`	0	0	17	18	2	2
`sra $s3, $s1, 2`	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Field Values

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Generating Constants

- 16-bit constants using `addi`:

C Code

```
// int is a 32-bit signed word
a = 0x4f3c;
```

MIPS assembly code

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

- 32-bit constants using load upper immediate (`lui`) and `ori`:

C Code

```
// int is a 32-bit signed word
a = 0xFEDC8765;
```

MIPS assembly code

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

Multiplication, Division

- Special registers: `lo`, `hi`
- 32 × 32 multiplication, 64 bit result
  - `mult $s0, $s1`
  - Result in {`hi`, `lo`}
- 32-bit division, 32-bit quotient, remainder
  - `div $s0, $s1`
  - Quotient in `lo`
  - Remainder in `hi`
- Moves from `lo`/`hi` special registers
  - `mflo $s2`
  - `mfhi $s3`

Pseudo-instructions

- Also called *macroinstructions*
- Pseudo-instructions are not real instructions implemented in hardware. They are created to make the program more readable
- When converted to machine code, pseudo-instructions are translated to one or more MIPS instructions.

Load immediate

- `li  $t0, 4`
  translates to
- `ori  $t0, $0, 4`

- but what should
  `li  $t0, 90000`
  `li  $t0, -5`
  translate to?

Load immediate (II)

- So

```
li   $t0, 90000
```

- translates to

```
lui   $at, 1     #load upper 16 bits
                 #(equal to 65536)
ori   $t0, $at, 24464
                 #lower 16 bits
                 #for 90000
```

- The special register `$at` should only be used for pseudo-instructions.

Load address

- `la  $t0, label`

- translates to

```
lui   $at, n  # load upper 16 bits
              # of label
ori   $t0, $at, m
              #lower 16 bits
              # of label
```

Multiply

3-input multiplication/division pseudo-instructions also exist

Example:

- `mul rd, rs, rt`

  writes the 32-bit result of `rs` x `rt` in `rd`

  translates to

```
mult rs, rt
mflo rd
```

Division

Example:

- `div rd, rs, rt`

  writes the quotient of `rs/rt` into `rd`

  translates to

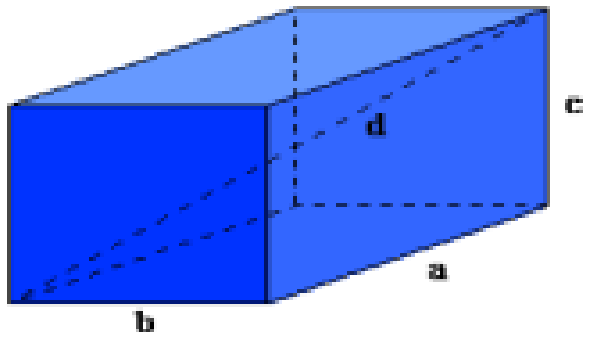
```
        bne rt, $0, ok
        break $0
ok:     div rs, rt
        mflo rd
```

Example (I)

The following is an example program to compute the volume and surface area of a rectangular parallelepiped.

The formulas for the volume and surface area are as follows:

```
volume = aSide*bSide*cSide
surfaceArea = 2( aSide*bSide + aSide*cSide +
                 bSide*cSide)
```



Example (II)

```
#  Example to compute the volume and surface area
#  of a rectangular parallelepiped.

# ------------------------------------------------
#  Data Declarations
.data

aSide:          .word     73
bSide:          .word     14
cSide:          .word     16

volume:         .word     0
surfaceArea:    .word     0
```

Example (III)

```
# ------------------------------------------------
#  Text/code section

.text
.globl     main
main:

# -----
#  Load variables into registers.


      lw    $t0, aSide
      lw    $t1, bSide
      lw    $t2, cSide
```

Example (IV)

```
# ----
#  Find volume of a rectangular paralllelpiped.
#     volume = aSide * bSide * cSide

      mul   $t3, $t0, $t1
      mul   $t4, $t3, $t2
      sw    $t4, volume

# -----
#  Find surface area of a rectangular parallelepiped.
#     surfaceArea = 2*(aSide*bSide+aSide*cSide+bSide*cSide)

      mul   $t3, $t0, $t1          # aSide * bSide
      mul   $t4, $t0, $t2          # aSide * cSide
      mul   $t5, $t1, $t2          # bSide * cSide
      add   $t6, $t3, $t4
      add   $t7, $t6, $t5
      mul   $t7, $t7, 2
      sw    $t7, surfaceArea

# -----
#  Done, terminate program.

      li    $v0, 10                # call code for terminate
      syscall                      # system call (terminate)
.end main
```

Branching

- Execute instructions out of sequence
- Types of branches:
  - **Conditional**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
  - **Unconditional**
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)

Conditional Branching (beq)

```
# MIPS assembly
 addi  $s0, $0, 4          # $s0 = 0 + 4 = 4
 addi  $s1, $0, 1          # $s1 = 0 + 1 = 1
 sll   $s1, $s1, 2         # $s1 = 1 << 2 = 4
 beq   $s0, $s1, target    # branch is taken
 addi  $s1, $s1, 1         # not executed
 sub   $s1, $s1, $s0       # not executed

target:                    # label
 add   $s1, $s1, $s0       # $s1 = 4 + 4 = 8
```

Labels indicate instruction location. They can't be reserved words and must be followed by colon (:)

The Branch Not Taken (bne)

```
# MIPS assembly
  addi      $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi      $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll       $s1, $s1, 2         # $s1 = 1 << 2 = 4
  bne       $s0, $s1, target    # branch not taken
  addi      $s1, $s1, 1         # $s1 = 4 + 1 = 5
  sub       $s1, $s1, $s0       # $s1 = 5 - 4 = 1

target:
  add       $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Unconditional Branching (j)

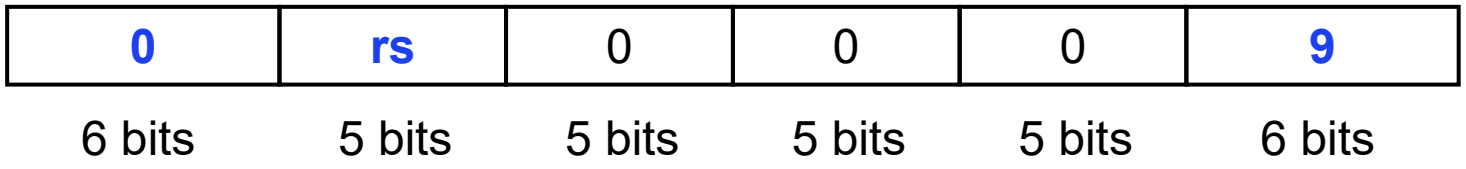
```
# MIPS assembly
  addi      $s0, $0, 4          # $s0 = 4
  addi      $s1, $0, 1          # $s1 = 1
  j         target              # jump to target
  sra       $s1, $s1, 2         # not executed
  addi      $s1, $s1, 1         # not executed
  sub       $s1, $s1, $s0       # not executed

  target:
  add       $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Unconditional Branching (`jr`)

```
# MIPS assembly
0x00002000        addi $s0, $0, 0x2010
0x00002004        jr   $s0
0x00002008        addi $s1, $0, 1
0x0000200C        sra  $s1, $s1, 2
0x00002010        lw   $s3, 44($s1)
```

`jr` is an **R-type** instruction.



High-Level Code Constructs

- `if` statements
- `if/else` statements
- `while` loops
- `for` loops

If Statement

C Code

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

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```

If Statement

C Code

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

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
    bne $s3, $s4, L1
    add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

Assembly tests opposite case (`i != j`) of high-level code (`i == j`)

If/Else Statement

C Code

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

MIPS assembly code

If/Else Statement

C Code

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

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
      bne $s3, $s4, L1
      add $s0, $s1, $s2
      j   done
L1:   sub $s0, $s0, $s3
done:
```

If/Else Statement (II)

C Code

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

MIPS assembly code

```
# $s0 = f
# $s3 = i, $s4 = j
      beq $s3, $s4, L1
      addi $s0, $s0, 1
      j   done
L1:   addi $s0, $s0, -1
done:
```

While Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

MIPS assembly code

While Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

MIPS assembly code

```
# $s0 = pow, $s1 = x

        addi $s0, $0, 1
        add  $s1, $0, $0
        addi $t0, $0, 128
while:  beq  $s0, $t0, done
        sll  $s0, $s0, 1
        addi $s1, $s1, 1
        j    while
done:
```

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

For Loops

```
for (initialization; condition; loop operation)
  statement
```

- **`initialization:`** executes before the loop begins
- **`condition:`** is tested at the beginning of each iteration
- **`loop operation:`** executes at the end of each iteration
- **`statement:`** executes each time the condition is met

For Loops

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

For Loops

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
        add  $s1, $0, $0
        add  $s0, $0, $0
        addi $t0, $0, 10
for:    beq  $s0, $t0, done
        add  $s1, $s1, $s0
        addi $s0, $s0, 1
        j    for
done:
```

Magnitude comparison

- 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;`
- Used in combination with **beq**, **bne**

```
slt $t0, $s1, $s2 # if ($s1 < $s2)
bne $t0, $0, L    #   branch to L
```

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example

```
$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001

slt  $t0, $s0, $s1  # signed
```

  - −1 < +1 ⇒ $t0 = 1

```
sltu $t0, $s0, $s1  # unsigned
```

  - +4,294,967,295 > +1 ⇒ $t0 = 0

Less Than Comparison

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

MIPS assembly code

Less Than Comparison

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

$t1 = 1 if i < 101

Branch pseudo-instructions

- To make the programmer job easier, some pseudo-instructions exist to support conditional branches
- Example: branch if less than (**`blt`**)

  The `blt` instruction compares 2 registers, treating them as signed integers, and takes a branch if one register is less than another

  Hence, the pseudo-instruction

```
blt $8, $9, label
```

  translates to

```
slt $1, $8, $9        # $1: $at
bne $1, $0, label
```

Branch pseudo-instructions

- Here's the list of pseudo-instructions for conditional branches
  - `bge $t0, $s0, L1`

    ```
    slt $at, $t0, $s0
    beq $at, $0, L1
    ```
  - `blt $t0, $s0, L1`

    ```
    slt $at, $t0, $s0
    bne $at, $0, L1
    ```
  - `bgt $t0, $s0, L1`

    ```
    slt $at, $s0, $t0
    bne $at, $0, L1
    ```
  - `ble $t0, $s0, L1`

    ```
    slt $at, $s0, $t0
    beq $at, $0, L1
    ```

Branch pseudo-instructions (II)

- Here's the list of pseudo-instructions for conditional branches with *unsigned values*
  - `bgeu $t0, $s0, L1`

    ```
    sltu $at, $t0, $s0
    beq $at, $0, L1
    ```
  - `bltu $t0, $s0, L1`

    ```
    sltu $at, $t0, $s0
    bne $at, $0, L1
    ```
  - `bgtu $t0, $s0, L1`

    ```
    sltu $at, $s0, $t0
    bne $at, $0, L1
    ```
  - `bleu $t0, $s0, L1`

    ```
    sltu $at, $s0, $t0
    beq $at, $0, L1
    ```

If-then-else: <

C Code

```
int a, b, c;

if (a < b)
{
  c = 5;
}
else
{
  c = 8;
}
```

MIPS assembly code

```
# $t1 = a, $t2 = b, $t3 = c
main:
    addi        $t1, $0, 6
    addi        $t2, $0, 5

    slt $t4, $t1, $t2  # if (a < b)
    beq $t4, $0, L1    # else

    addi $t3, $0, 5    # then
    j     L2

L1: addi $t3, $0, 8    # else

L2:
```

If-then-else: >

C Code

```
int a, b, c;

if (a > b)
{
  c = 5;
}
else
{
  c = 8;
}
```

MIPS assembly code

```
# $t1 = a, $t2 = b, $t3 = c
main:
    addi      $t1, $0, 6
    addi      $t2, $0, 5

    slt $t4, $t2, $t1  # if (b < a)
    beq $t4, $0, L1    # else

    addi $t3, $0, 5    # then
    j    L2

L1: addi $t3, $0, 8    # else

L2:
```

If-then-else: >=

C Code

```
int a, b, c;

if (a >= b)
{
  c = 5;
}
else /* a < b */
{
  c = 8;
}
```

MIPS assembly code

```
# $t1 = a, $t2 = b, $t3 = c
main:
    addi        $t1, $0, 6
    addi        $t2, $0, 5

    slt $t4, $t1, $t2  # if (a < b)
    beq $t4, $0, L1

    addi $t3, $0, 8    # (a < b)
    j     L2

L1: addi $t3, $0, 5    # (a >= b)

L2:
```

If-then-else: <=

C Code

```
int a, b, c;

if (a <= b)
{
  c = 5;
}
else /* a > b */
{
  c = 8;
}
```

MIPS assembly code

```
# $t1 = a, $t2 = b, $t3 = c
main:
    addi        $t1, $0, 6
    addi        $t2, $0, 5

    slt $t4, $t2, $t1  # if (b < a)
    beq $t4, $0, L1

    addi $t3, $0, 8    # (b < a)
    j     L2

L1: addi $t3, $0, 5    # (a <= b)

L2:
```

Example (I)

- The following is an example program to find the sum of squares from 1 to n.
- For example, the sum of squares for n=10 is as follows:

$$1^2 + 2^2 + \cdots + 10^2 = 385$$

Example (II)

```
#  Example program to compute the sum of squares.
# ------------------------------------------------
#  Data Declarations

.data
n:                .word 10
sumOfSquares:     .word 0

# ------------------------------------------------
#  text/code section
```

Example (III)

```
.text
.globl     main
main:

# -----
#  Compute sum of squares from 1 to n.

      lw    $t0, n                  #
      li    $t1, 1                  # loop index (1 to n)
      li    $t2, 0                  # sum

sumLoop:
      mul   $t3, $t1, $t1           # index^2
      add   $t2, $t2, $t3

      add   $t1, $t1, 1
      ble   $t1, $t0, sumLoop

      sw    $t2, sumOfSquares

# -----
#  Done, terminate program.

      li    $v0, 10                 # call code for terminate
      syscall                       # system call
.end main
```

Switch case Statement

C Code

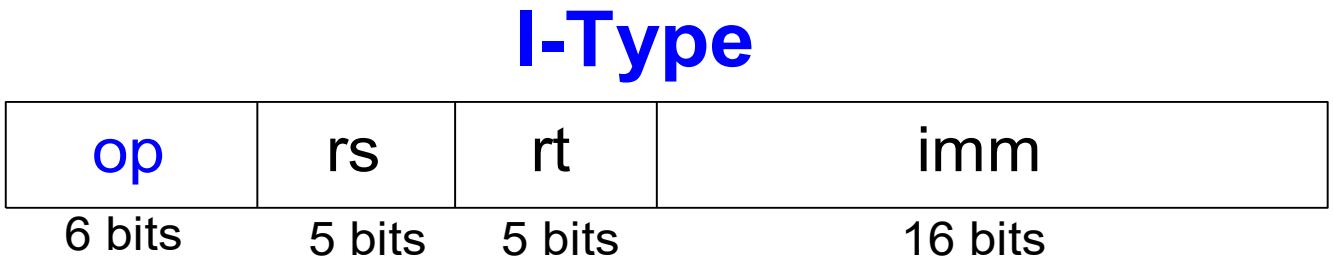
```
switch (k)
{
   case 0: f = g + h;
           break;
   case 1: f = g - h;
           break;
   case 2: f = 0;
}
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s5 = k
.data
JumpTable:
    .word L0, L1, L2
    .text
    .globl main
main:
        la $t4, JumpTable
        sll $t1, $s5, 2
        add $t1, $t1, $t4
        lw $t0, 0($t1)
        jr $t0
   L0:  add $s0, $s1, $s2
        j next
   L1:  sub $s0, $s1, $s2
        j next
   L2:  add $s0, $0, $0
   next:
```

Address generation

- Considering the instructions `beq` and `bne`



- The immediate field contains a signed 16-bit value specifying the number of words away from the current Program Counter address to the location symbolically specified by the label.

- `PC = PC + se imm << 2`

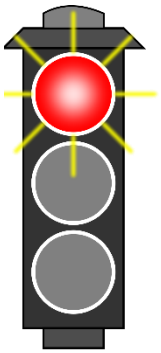
- `se`: sign extension

Far jumps

- Conditional branches present a limit into the target address representation

I-Type


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



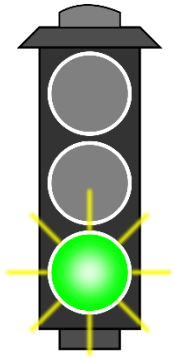
```
                bne $t0, $t1, L1
                sub $s0, $s1, $s2
                . . .
                . . .
                . . .
                j L2
L1:             add $s0, $s1, $s2
L2:
```

Let assume that this is non representable due to the far jump

Far jumps: solution

- The far jump is executed by means of an unconditional jump.

```
if (i == j)                    beq $t0, $t1, LAB1
  {                            j L1
  f = g - h;           LAB1:   sub $s0, $s1, $s2
  . . .                        . . .
  . . .                        . . .
  . . .                        . . .
  }                            j L2
else                   L1:     add $s0, $s1, $s2
  f = g + h;           L2:
```

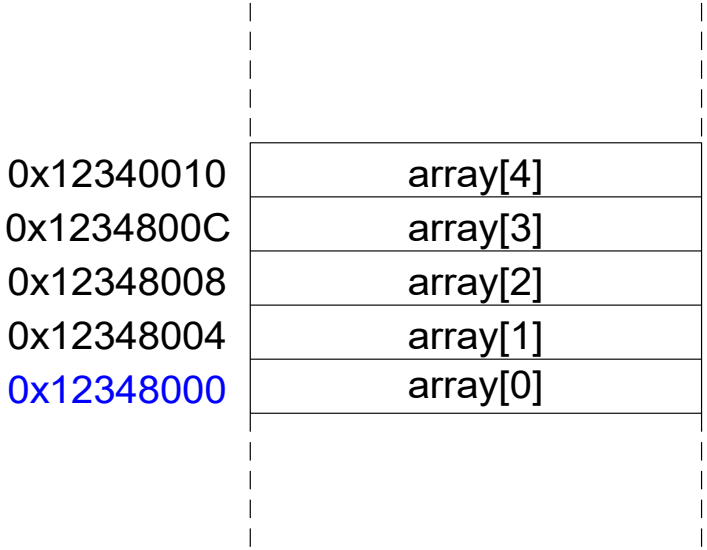


Arrays

- Access large amounts of similar data
- **Index**: access each element
- **Size**: number of elements

Arrays

- 5-element array
- **Base address** = 0x12348000 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register



Accessing Arrays

```
// C Code
   int array[5];
   array[0] = array[0] * 2;
   array[1] = array[1] * 2;
```

Accessing Arrays

```
// C Code
   int array[5];
   array[0] = array[0] * 2;
   array[1] = array[1] * 2;

# MIPS assembly code
# $s0 = array base address
  lui  $s0, 0x1234          # 0x1234 in upper half of $s0
  ori  $s0, $s0, 0x8000     # 0x8000 in lower half of $s0

  lw   $t1, 0($s0)          # $t1 = array[0]
  sll  $t1, $t1, 1          # $t1 = $t1 * 2
  sw   $t1, 0($s0)          # array[0] = $t1

  lw   $t1, 4($s0)          # $t1 = array[1]
  sll  $t1, $t1, 1          # $t1 = $t1 * 2
  sw   $t1, 4($s0)          # array[1] = $t1
```

Arrays Using For Loops

```
// C Code
   int array[1000];
   int i;

   for (i=0; i < 1000; i = i + 1)
        array[i] = array[i] * 8;

# MIPS assembly code
# $s0 = array base address, $s1 = i
```

Arrays Using For Loops

```
# MIPS assembly code
# $s0 = array base address, $s1 = i
# initialization code
  lui  $s0, 0x23B8         # $s0 = 0x23B80000
  ori  $s0, $s0, 0xF000    # $s0 = 0x23B8F000
  addi $s1, $0, 0          # i = 0
  addi $t2, $0, 1000       # $t2 = 1000

loop:
  slt  $t0, $s1, $t2       # i < 1000?
  beq  $t0, $0, done       # if not then done
  sll  $t0, $s1, 2         # $t0 = i * 4 (byte offset)
  add  $t0, $t0, $s0       # address of array[i]
  lw   $t1, 0($t0)         # $t1 = array[i]
  sll  $t1, $t1, 3         # $t1 = array[i] * 8
  sw   $t1, 0($t0)         # array[i] = array[i] * 8
  addi $s1, $s1, 1         # i = i + 1
  j    loop                # repeat
done:
```

Bi-dimensional array

```
// C Code
int matrix[2][4];
```

A	B	C	D
E	F	G	H

- The elements of a matrix are stored in memory in order, starting from the first row (*stored by rows*)

matrix[0][0]	**A**
matrix[0][1]	**B**
matrix[0][2]	**C**
matrix[0][3]	**D**
matrix[1][0]	**E**
matrix[1][1]	**F**
matrix[1][2]	**G**
matrix[1][3]	**H**

Access to a row in a matrix

C Code

```
int matrix[5][4];
int i;

for (i=0; i < 4; i++)
    matrix[3][i]++;
```

MIPS assembly code

```
.data
matrix:
    .word 8, 10, 11, 1
    .word 7, 5, 9, 2
    .word 6, 11, 0, 3
    .word 4, 3, 8, 4
    .word 0, 1, 2, 5
 .text
    .globl main
main:
    la $t0, matrix
    li $t1, 3                   # row number 3
    li $t2, 4                   # number of columns
    mul $t3, $t2, $t1           # number of words
                                # in the first 3 rows
    sll $t3, $t3, 2             # number of bytes
                                # in the first 3 rows
    add $t5, $t0, $t3           # $t5: initial address
                                # of row with index 3
    add $t4, $0, $0             # $t4 = 0 (i)
Lab1:
    lw $t6, ($t5)               # matrix[3][i]
    addi $t6, $t6, 1            # ++
    sw $t6, ($t5)
    addi $t5, $t5, 4            # update address
    addi $t4, $t4, 1
    bne $t4, $t2, lab1
```

Access to a column in a matrix

C Code

```
int matrix[5][4];
int i;

for (i=0; i < 5; i++)
    matrix[i][2]++;
```

MIPS assembly code

```
.data
matrix:
    .word 8, 10, 11, 1
    .word 7, 5, 9, 2
    .word 6, 11, 0, 3
    .word 4, 3, 8, 4
    .word 0, 1, 2, 5
 .text
    .globl main
main:
    la $t0, matrix
    addi $t3, $0, 2          # column number 2
    sll $t3, $t3, 2          # number of bytes
    add $t5, $t0, $t3        # $t5: initial address
                             # of matrix[0][2]
    addi $t2, $0, 5          # number of rows
    add $t4, $0, $0          # $t4 = 0 (i)
Lab1:
    lw $t1, ($t5)            # matrix[i][2]
    addi $t1, $t1, 1         # ++
    sw $t1, ($t5)
    addi $t5, $t5, 16        # update address
    addi $t4, $t4, 1
    bne $t4, $t2, lab1
```

ASCII Code

- *American Standard Code for Information Interchange*
- Each text character has unique byte value
  - For example, S = 0x53, a = 0x61, A = 0x41
  - Lower-case and upper-case differ by 0x20 (32)

Cast of Characters

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Example with strings

C Code

```
char chararray[5];
int i;
for (i=0 ; i != 5 ; i++)
 chararray[i] =
 chararray[i] + 'a' - 'A';
```

MIPS assembly code

```
.data
charrarray:
    .ascii "salve"
    .text
    .globl main
# $s0 = base address of chararray
# $s1 = i
main:
        la $s0, chararray
        addi $s1, $0, 0
        addi $t0, $0, 5
loop:   beq $s1, $t0, done
        add $t1, $s0, $s1
        lb $t2, 0($t1)
        addi $t2, $t2, -32
        sb $t2, 0($t1)
        addi $s1, $s1, 1
        j loop
done:
```

System Calls

- Some OS services are provided, including operations for I/O: read, write, file open, file close, etc.
- The arguments for the syscall are placed in $a0-$a3
- The type of syscall is identified by placing the appropriate number in $v0
- $v0 is also used for the syscall's return value

Examples

procedure	code $v0	operation
read int	5	$v0 contains the number
print int	1	$a0 contains number to print
print string	4	$a0 address of string
exit	10	End of program
print char	11	$a0 contains char to print
read char	12	$v0 contains the char

Example: Print Routine

```
.data
str:     .asciiz   "the answer is "

.text

li      $v0, 4    # 4 is the code for printing a string
la      $a0, str  # the syscall expects the string
                  # address as a parameter

syscall

li       $v0, 1   # 1 is the code for printing an int
li       $a0, 5   # the syscall expects the integer as a parameter

syscall
```

Store the string in memory and null-terminate it

Example: Print Routine

- Write an assembly program to prompt the user for two numbers and print the sum of the two numbers.

Example: Print Routine

```
.text
.globl main
main:
li   $v0, 4
la  $a0, str1
syscall
li   $v0, 5
syscall
add  $t0, $v0, $0
li   $v0, 5
syscall
add  $t1, $v0, $0
li   $v0, 4
la  $a0, str2
syscall
li    $v0, 1
add  $a0, $t1, $t0
syscall
```

```
.data
str1:  .asciiz  “Enter 2 numbers:”
str2:  .asciiz  “The sum is ”
```

Function Calls

- **Caller:** calling function (in this case, `main`)
- **Callee:** called function (in this case, `sum`)

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Function Conventions

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

MIPS Function Conventions

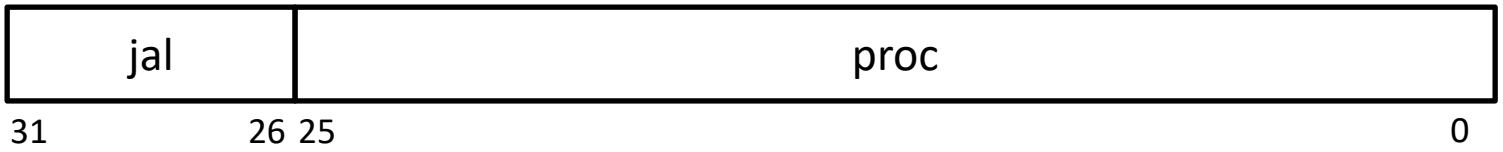
- **Call Function:** jump and link (`jal`)
- **Return** from function: jump register (`jr`)
- **Arguments**: `$a0 – $a3`
- **Return value**: `$v0`

Jump and Link

- `jal:` J-type instruction

Example:

```
jal proc
```



It executes the following operations:

- It modifies the register `PC` in order to jump to `proc`
- It copies the *return address* into the register `$ra`.

Function Calls

C Code

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

void simple() {
  return;
}
```

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

Function Calls

C 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
```

Function Calls

C 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
```

jal: $ra = PC + 4 = **0x00400204**
jumps to simple (**0x00401020**)

jr $ra: jumps to address in $ra (**0x00400204**)

Input Arguments & Return Value

MIPS conventions:

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

Input Arguments & Return Value

C 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 & Return Value

MIPS assembly code

```
# $s0 = y

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 Function
  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
```

Input Arguments & Return Value

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
```

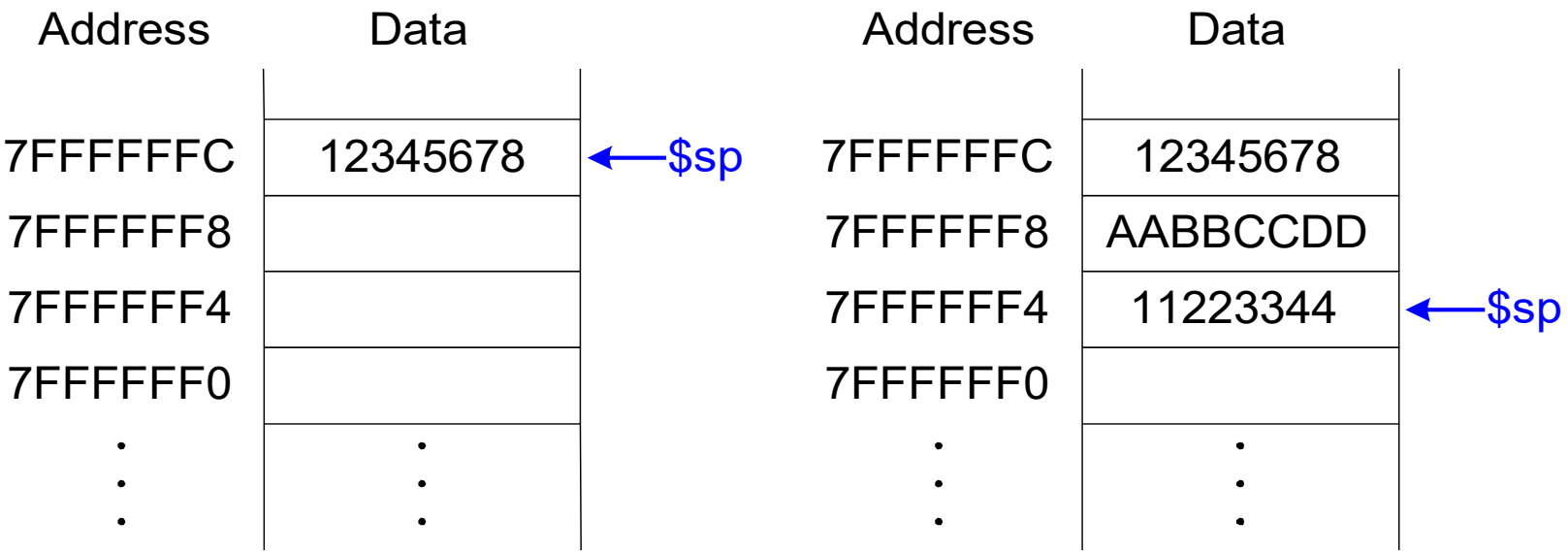
- `diffofsums` overwrote 3 registers: `$t0`, `$t1`, `$s0`
- `diffofsums` can use the *stack* to temporarily store these registers

The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands (or push)*****:** uses more memory when more space needed
- ***Contracts (or pop)*****:** uses less memory when the space is no longer needed

The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: `$sp` points to top of the stack



After 2 more push operations

How Functions use the Stack

- Called functions must have no 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
```

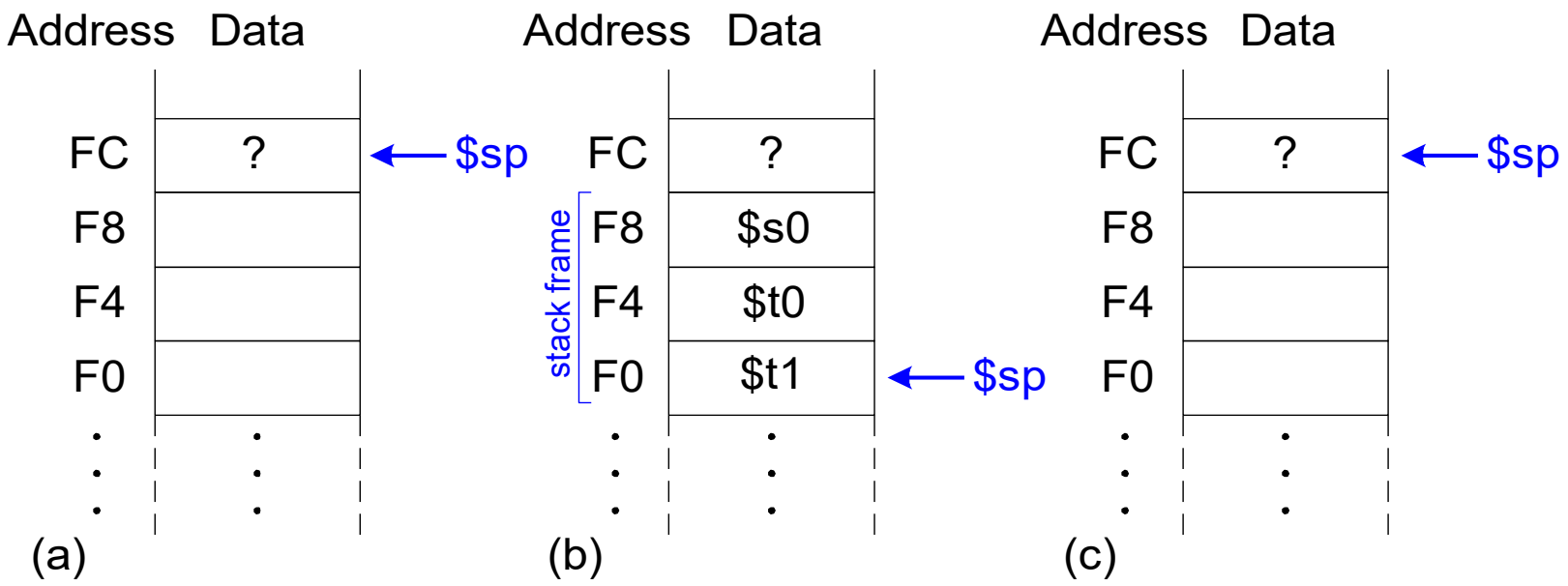
Stack frame

- The stack space that a function allocates for itself is called its *stack frame*
- Each function should access only its own stack frame, not the frames belonging to other functions.

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
```

The stack during `diffofsums` Call



Before `diffofsum` During `diffofsum` After `diffofsum`

Registers

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

Leaf and nonleaf functions

- A *leaf* function does not call other functions
- A *nonleaf* function call other functions

Nonleaf function

- A nonleaf function has to save nonpreserved registers on the stack before it calls another function, and then restores those registers afterward
- The caller saves any non-preserved register (`$t0-$t9` and `$a0-$a3`) that are needed after the call
- The callee saves any of the preserved registers (`$s0-Ss7` and `$ra`) that intends to modify.

Leaf function

- A leaf function may not save non-preserved registers (in particular `$t0`–`$t9`).

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
```

Multiple Function 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 $ra from stack
   addi $sp, $sp, 4     # deallocate stack space
   jr  $ra              # return to caller
```

Example: C sort

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

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

- v in `$a0`, k in `$a1`, temp in `$t0`

The procedure swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

The sort procedure

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

- v in `$a0`, n in `$a1`, i in `$s0`, j in `$s1`

The procedure body

```
          move $s2, $a0           # save $a0 into $s2
          move $s3, $a1           # save $a1 into $s3
          move $s0, $zero         # i = 0
for1tst:  slt  $t0, $s0, $s3      # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
          beq  $t0, $zero, exit1  # go to exit1 if $s0 ≥ $s3 (i ≥ n)
          addi $s1, $s0, -1       # j = i - 1
for2tst:  slti $t0, $s1, 0        # $t0 = 1 if $s1 < 0 (j < 0)
          bne  $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
          sll  $t1, $s1, 2        # $t1 = j * 4
          add  $t2, $s2, $t1      # $t2 = v + (j * 4)
          lw   $t3, 0($t2)        # $t3 = v[j]
          lw   $t4, 4($t2)        # $t4 = v[j + 1]
          slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
          beq  $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3
          move $a0, $s2           # 1st param of swap is v (old $a0)
          move $a1, $s1           # 2nd param of swap is j
          jal  swap               # call swap procedure
          addi $s1, $s1, -1       # j -= 1
          j    for2tst            # jump to test of inner loop
exit2:    addi $s0, $s0, 1        # i += 1
          j    for1tst            # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

The full procedure

```
sort:   addi $sp,$sp, -20       # make room on stack for 5 registers
        sw $ra, 16($sp)         # save $ra on stack
        sw $s3, 12($sp)         # save $s3 on stack
        sw $s2, 8($sp)          # save $s2 on stack
        sw $s1, 4($sp)          # save $s1 on stack
        sw $s0, 0($sp)          # save $s0 on stack
        …                       # procedure body
        …
exit1:  lw $s0, 0($sp)          # restore $s0 from stack
        lw $s1, 4($sp)          # restore $s1 from stack
        lw $s2, 8($sp)          # restore $s2 from stack
        lw $s3, 12($sp)         # restore $s3 from stack
        lw $ra,16($sp)          # restore $ra from stack
        addi $sp,$sp, 20        # restore stack pointer
        jr $ra                  # return to calling routine
```

Recursive Function Call

High-level code

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

Recursive Function Call

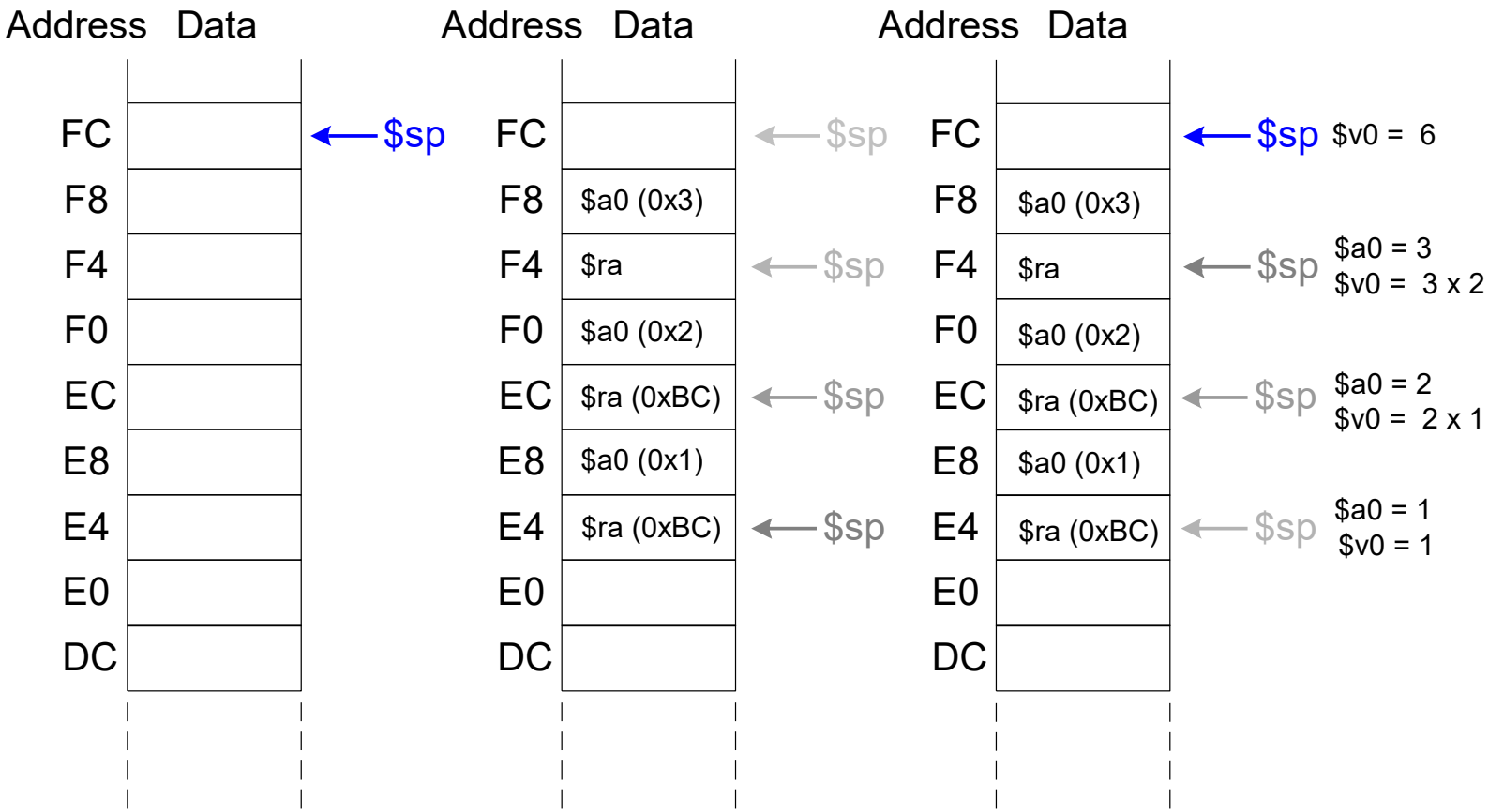
MIPS assembly code

```
0x90 factorial: addi $sp, $sp, -8   # make room
0x94            sw   $a0, 4($sp)    # store $a0
0x98            sw   $ra, 0($sp)    # store $ra
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            addi $sp, $sp, 8    # restore $sp
0xB0            jr   $ra            # return
0xB4       else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial      # recursive call
0xBC            lw   $ra, 0($sp)    # restore $ra
0xC0            lw   $a0, 4($sp)    # restore $a0
0xC4            addi $sp, $sp, 8    # restore $sp
0xC8            mul  $v0, $a0, $v0  # n * factorial(n-1)
0xCC            jr   $ra            # return
```

Stack During Recursive Call

High level calling code

```
{
int y;
…
y = factorial(3);
…
}
```



Additional arguments

- Functions may have more than 4 input arguments
- The stack is used to store these temporary values
- By MIPS convention, if a function has more than 4 arguments, the first 4 are passed in the argument registers and the additional arguments are passed on the stack, just above $sp
- The caller must expand the stack to make room for the additional arguments.

Example

- Res = addem (num1, num2, num3, num4, num5); /* num1+num2+num3+num4+num5 */

```
.data
num1: .word 1
num2: .word 2
num3: .word 3
num4: .word 4
num5: .word 5
sum: .word 0
```

```
.text
.globl main
.ent main
main:
lw $a0, num1
lw $a1, num2
lw $a2, num3
lw $a3, num4
lw $t0, num5

addi $sp, $sp, -4
sw $t0, 0($sp)

jal addem

sw $v0, sum
addi $sp, $sp, 4

li $v0, 10
syscall
.end main
```

```
.globl addem
.ent addem

addem:

addi $v0, $0, 0

add $v0, $v0, $a0
add $v0, $v0, $a1
add $v0, $v0, $a2
add $v0, $v0, $a3

lw $t0, ($sp)

add $v0, $v0, $t0

jr $ra
.end addem
```

Local Variables

- Local variables are declared within a function and can be accessed only within that function
- Local variables are stored in $t0-$t9; if there are too many local variables, they can also be stored in the function's stack frame
- In particular, local arrays are stored in the stack.

Example

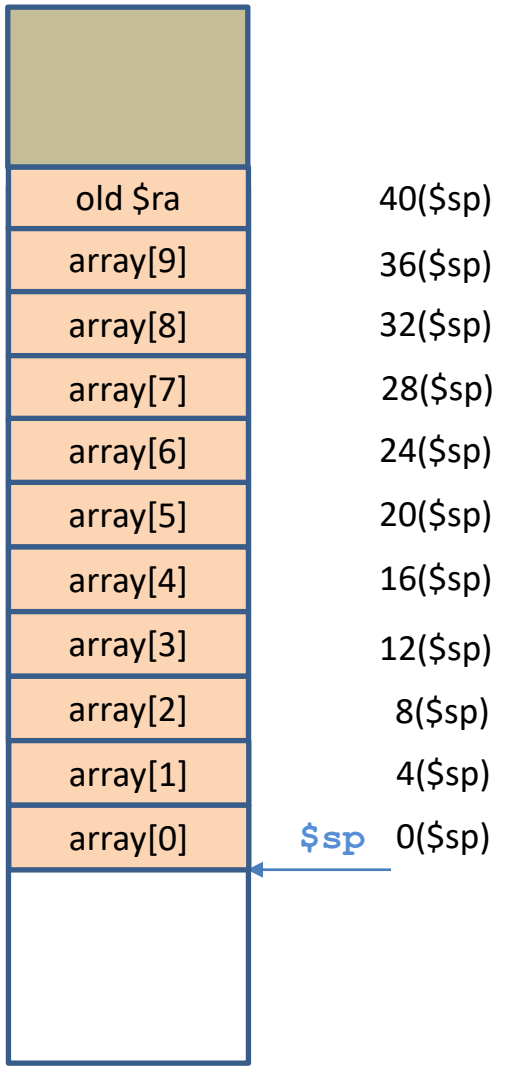
- Write a procedure using an array of 10 integers as local variable and calling 3 procedures to read, reverse and print the array.

High-level code

```
void proc1()
{
int array[10];
read(array, 10);
reverse(array, 10);
print(array, 10);
}
```

Example

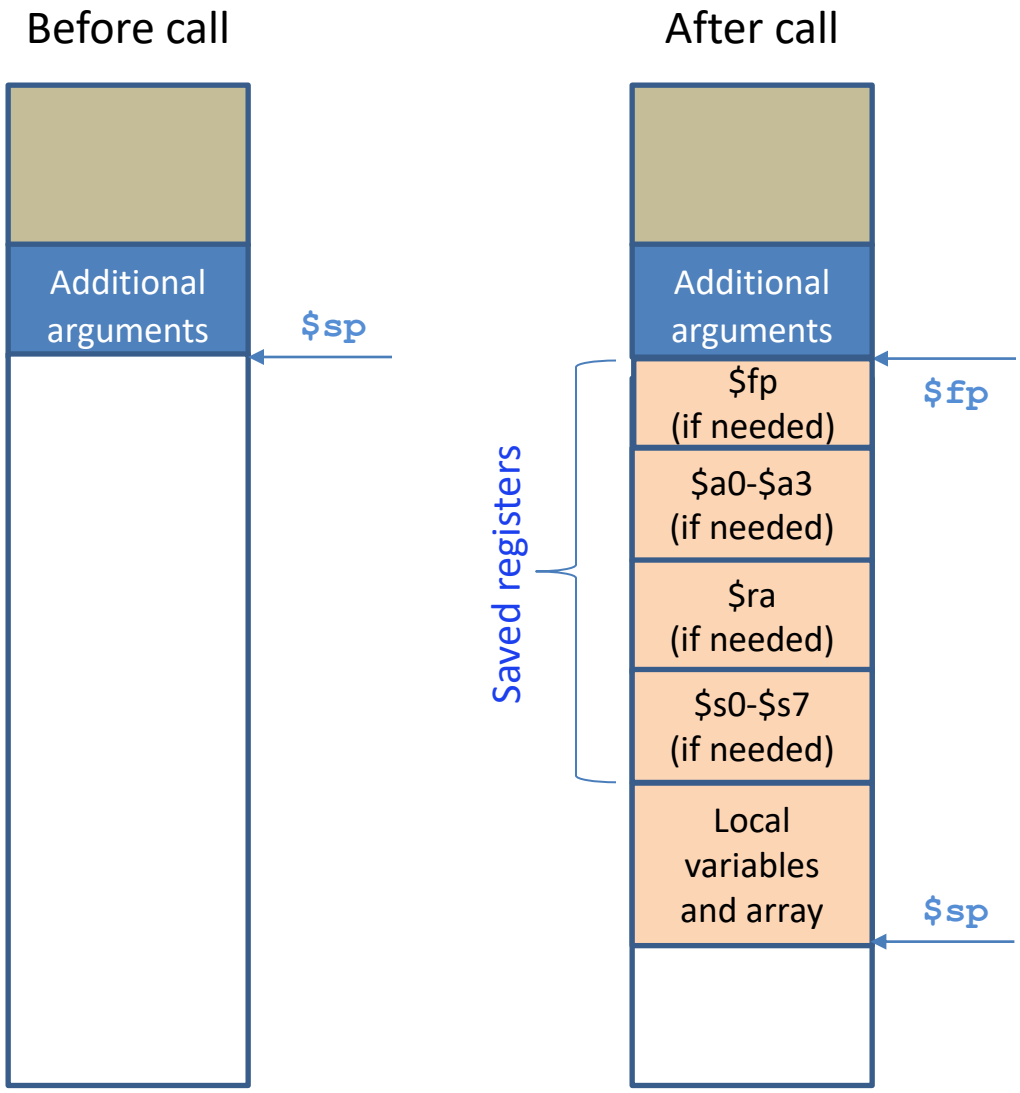
```
proc1:
addi  sp, $sp, -44 # allocate stack frame = 44 bytes
sw    $ra, 40($sp) # save $ra on the stack
move  $a0, $sp     # $a0 = address of array on the stack
li    $a1, 10      # $a1 = 10
jal   read         # call function read
move  $a0, $sp     # $a0 = address of array on the stack
li    $a1, 10      # $a1 = 10
jal   reverse      # call function reverse
move  $a0, $sp     # $a0 = address of array on the stack
li    $a1, 10      # $a1 = 10
jal   print        # call function print
lw    $ra, 40($sp) # load $ra from the stack
addi $sp, $sp, 44 # Free stack frame = 44 bytes
jr    $ra          # return to caller
```



Frame pointer Register

- During execution of a given module it is possible for the stack pointer to move dynamically
- Since the location of all items in a stack frame is based on the stack pointer, it is useful to define a fixed point in each stack frame and save the address of this reference point in a register called *frame pointer (`$fp`)*
- The Frame Pointer is a preserved register: this necessitates storing the old frame pointer in each stack frame (*callee saved*).

Stack Frame


Before call
Additional arguments
$sp
After call
Additional arguments
$fp
(if needed)
$a0-$a3
(if needed)
$ra
(if needed)
$s0-$s7
(if needed)
Saved registers
Local variables and array
$fp
$sp

Function Call Summary

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

Addressing Modes

How do we address the operands?

- Register Only
- Immediate
- Base Addressing
- PC-Relative
- Pseudo Direct

Addressing Modes

Register Only

- Operands found in registers
  - **Example:** `add $s0, $t2, $t3`
  - **Example:** `sub $t8, $s1, $0`

Immediate

- 16-bit immediate used as an operand
  - **Example:** `addi $s4, $t5, -73`
  - **Example:** `ori  $t3, $t7, 0xFF`

Addressing Modes

Base Addressing

- Address of operand is:

  `base address + sign-extended immediate`

  - **Example:** `lw  $s4, 72($0)`
    - address = `$0 + 72`
  - **Example:** `sw  $t2, -25($t1)`
    - address = `$t1 - 25`

Addressing Modes

PC-Relative Addressing

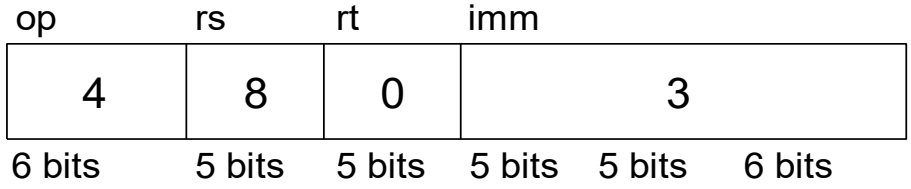
```
0x10              beq   $t0, $0, else
0x14              addi  $v0, $0, 1
0x18              addi  $sp, $sp, i
0x1C              jr    $ra
0x20        else: addi  $a0, $a0, -1
0x24              jal   factorial
```

The 16-bit immediate field gives the number of instructions between the *branch target address* (BTA) and the instruction *after* the branch instruction (the instruction at `PC+4`).

Assembly Code

```
beq $t0, $0, else
(beq $t0, $0, 3)
```

Field Values



Exercise

Calculate the immediate field for the `bne` instruction.

```
0x40        loop:  add   $t1, $a0, $s0
0x44               lb    $t1, 0($t1)
0x48               add   $t2, $a1, $s0
0x4C               sb    $t1, 0($t2)
0x50               addi  $s0, $s0, 1
0x54               bne   $t1, $0, loop
0x58               lw    $s0, 0($sp)
```

Addressing Modes

Pseudo-direct Addressing

```
0x0040005C          jal   sum
...
0x004000A0  sum:    add   $v0, $a0, $a1
```

The processor calculates the *jump target address* (JTA) from the J-type instruction by appending 2 0's and prepending the 4 most significant bits of `PC+4` to the 26-bit address field.

Jump Target Address 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

0 1 0 0 0 2 8

Field Values

op	imm
3	0x0100028
6 bits	26 bits

Machine Code

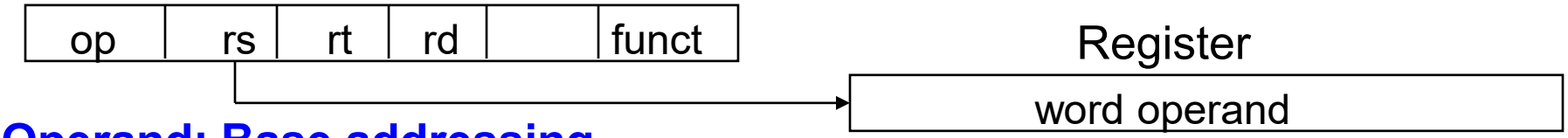
op	addr	
000011	00 0001 0000 0000 0000 0010 1000	(0x0C100028)
6 bits	26 bits	

Limitations

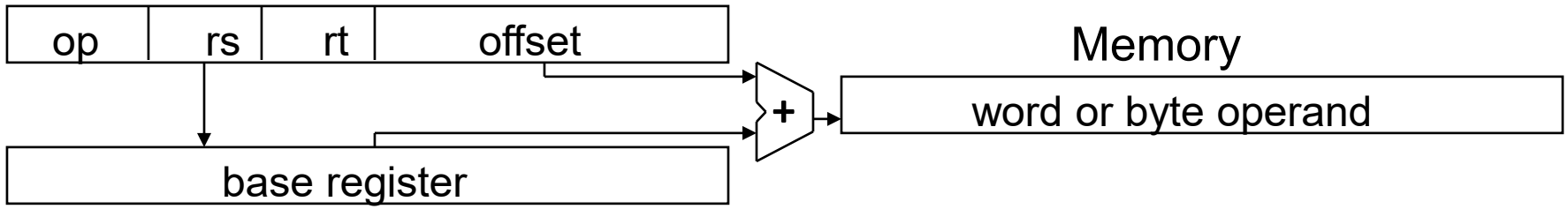
- Because the 4 most significant bits of the JTA are taken from `PC+4` the jump range is limited.

Review: MIPS Addressing Modes

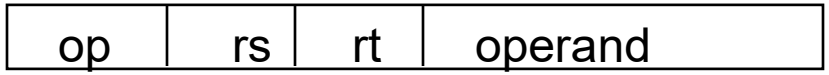
Operand: Register addressing



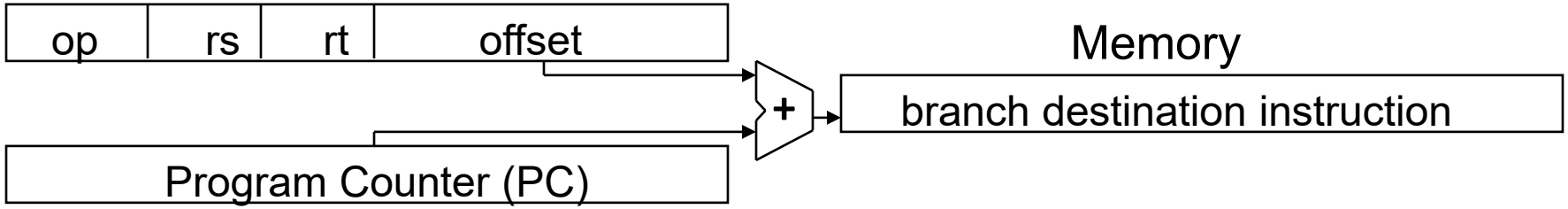
Operand: Base addressing



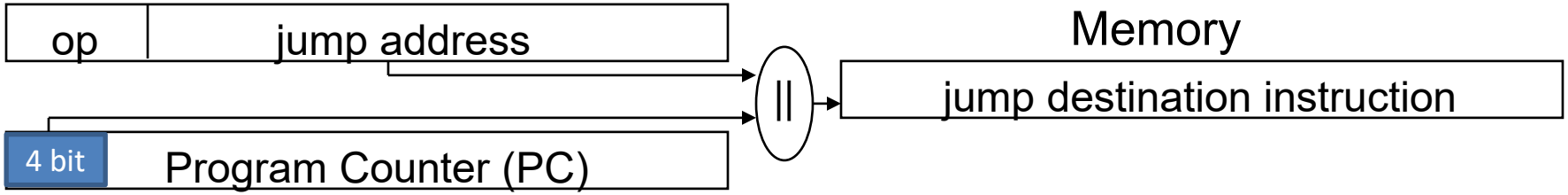
Operand: Immediate addressing



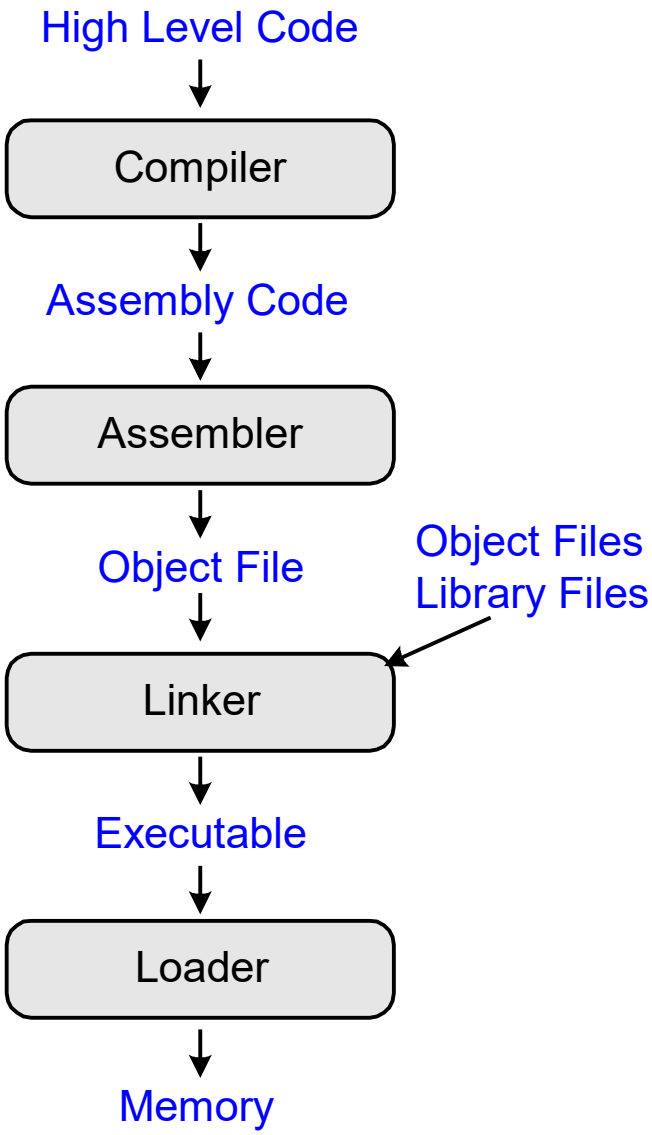
Instruction: PC-relative addressing



Instruction: Pseudo-direct addressing



How to Compile & Run a Program

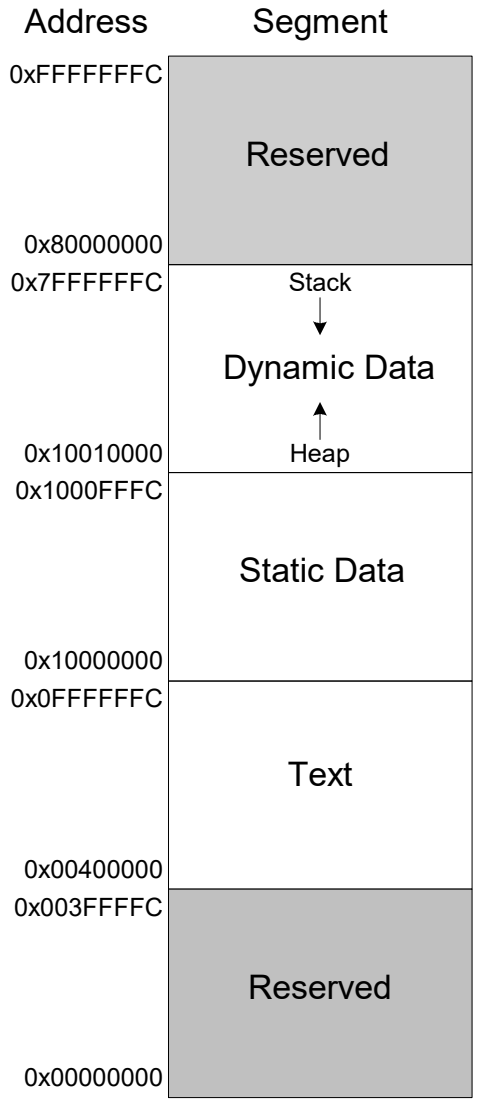


What is Stored in Memory?

- Instructions (also called *text*)
- Data
  - Global/static: allocated before program begins
  - Dynamic: allocated within program

- How big is memory?
  - At most 2^{32} = 4 gigabytes (4 GB)
  - From address 0x00000000 to 0xFFFFFFFF

MIPS Memory Map



Example Program: C Code

```
int f, g, y;  // global variables


int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);

  return y;
}


int sum(int a, int b) {
  return (a + b);
}
```

Example Program: MIPS Assembly

```
int f, g, y;  // global


int main(void)
{


  f = 2;
  g = 3;

  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

```
.data
f:
g:
y:
.text
main:
  addi $sp, $sp, -4    # stack frame
  sw   $ra, 0($sp)     # store $ra
  addi $a0, $0, 2      # $a0 = 2
  sw   $a0, f          # f = 2
  addi $a1, $0, 3      # $a1 = 3
  sw   $a1, g          # g = 3
  jal  sum             # call sum
  sw   $v0, y          # y = sum()
  lw   $ra, 0($sp)     # restore $ra
  addi $sp, $sp, 4     # restore $sp
  jr   $ra             # return to OS
sum:
  add  $v0, $a0, $a1   # $v0 = a + b
  jr   $ra             # return
```

Example Program: Symbol Table

Symbol	Address
`f`	**0x10000000**
`g`	**0x10000004**
`y`	**0x10000008**
`main`	**0x00400000**
`sum`	**0x0040002C**

Example Program: Executable

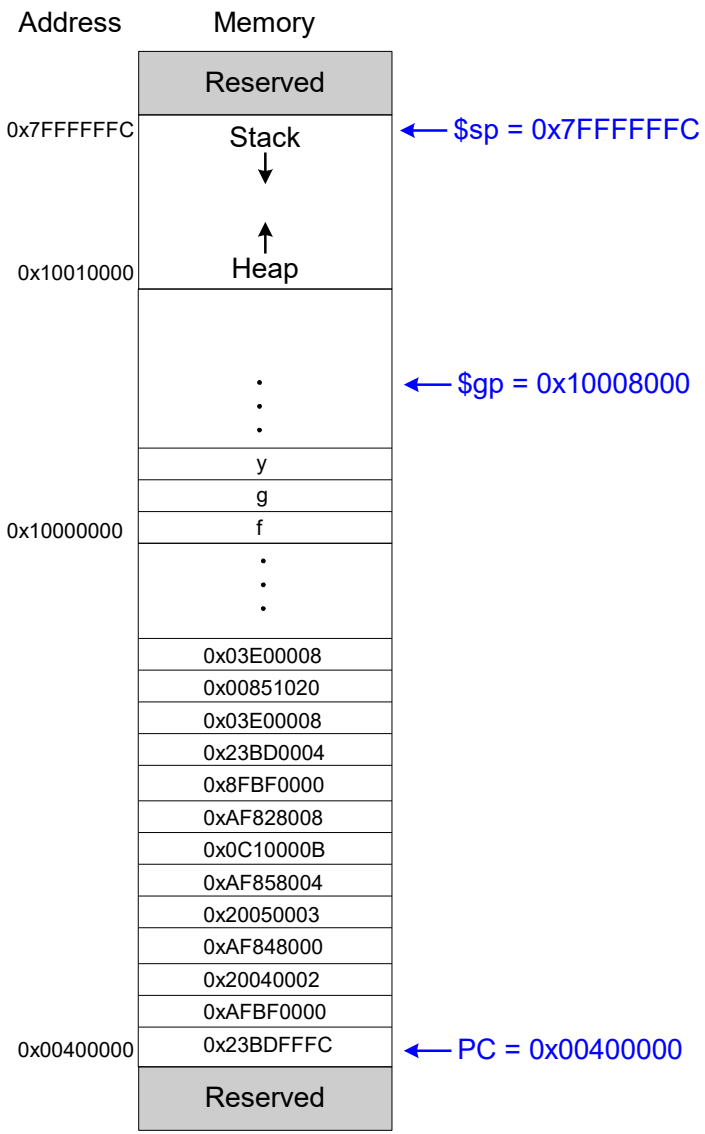
Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)

Text segment	Address	Instruction
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008

Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

```
addi $sp, $sp, -4
sw   $ra, 0 ($sp)
addi $a0, $0, 2
sw   $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw   $a1, 0x8004 ($gp)
jal  0x0040002C
sw   $v0, 0x8008 ($gp)
lw   $ra, 0 ($sp)
addi $sp, $sp, -4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

Example Program: In Memory



Odds & Ends

- Pseudoinstructions
- Exceptions
- Signed and unsigned instructions
- Floating-point instructions

Pseudoinstructions

Pseudoinstruction	MIPS Instructions
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`clear $t0`	`add $t0, $0, $0`
`move $s1, $s2`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`

Exceptions and Interrupts

- MIPS terminology:
  - **Exception** (aka *software interrupt*) = unexpected change in control flow caused internally (e.g., arithmetic overflow, undefined instruction)
  - **Trap** or **System call** = an intentional change of control yielded to the OS (e.g., request to access I/O device)
  - **Interrupt** = unexpected change in control flow caused externally (e.g., request from an I/O device)

- MIPS exceptions and interrupts are handled by a peripheral device named "**coprocessor 0**" (cp0)

Exceptions and Interrupts

- MIPS terminology:
  - **Exception** (aka *software interrupt*) = unexpected change in control flow caused internally (e.g., arithmetic overflow, undefined instruction)
  - **Trap** or **System call** = an ... ielded to the OS (e.g., request to access I/O devi...
  - **Interrupt** = unexpected ch... xternally (e.g., request from an I/O device...

Automatically triggered by the occurrence of a special condition (e.g., arithmetic overflow)

- MIPS exceptions and interrupts are handled by a peripheral device named "**coprocessor 0**" (cp0)

Exceptions and Interrupts

- MIPS terminology:
  - **Exception** (aka *software interrupt*) = unexpected change in control flow caused internally (e.g., arithmetic overflow, undefined instruction)
  - **Trap** or **System call** = an intentional change of control yielded to the OS (e.g., request to access I/O device)
  - **Interrupt** = unexpected ch[illegible]ternally (e.g., request from an I/O device)

Triggered by the execution of `syscall` or `break` instructions

- MIPS exceptions and interrupts a[illegible]evice named "**coprocessor 0**" (cp0)

Exceptions and Interrupts

- MIPS terminology:
  - **Exception** (aka *software interrupt*) = unexpected change in control flow caused internally (e.g., arithmetic overflow, undefined instruction)
  - **Trap** or **System call** = an intentional change of control yielded to the OS (e.g., request to access I/O device)
  - **Interrupt** = unexpected change in control flow caused externally (e.g., request from an I/O device)

- MIPS exceptions and interrupts a vice named "**coprocessor 0**" (cp0)

Triggered by the occurrence of an external event detected by a peripheral, which raised one of the 8 interrupt input signals

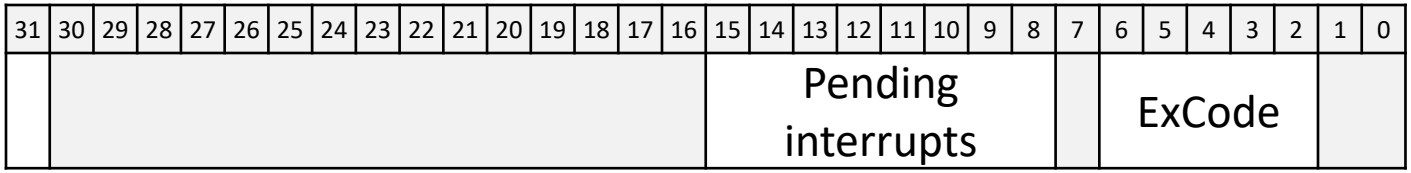
Coprocessor 0

- MIPS exceptions and interrupts are handled by “coprocessor 0” (cp0)
- cp0 has several 32-bit registers
  - **EPC** (cp0 register $14) – exception program counter
  - **Cause** register (cp0 register $13) – contains bits identifying exception cause
  - **Status** register (cp0 register $12) – is used to configure interrupts
- cp0 registers cannot be directly accessed by MIPS instructions
- Special instructions transfer information from/to cp0 like load/store:
  - `mfc0 $c0rt, $rd` *Move from coprocessor 0*

    Semantics: rd ← c0rt
  - `mtc0 $c0rd, $rt` *Move to coprocessor 0*

    Semantics: $c0rd ← rt

MIPS exception handling: Cause



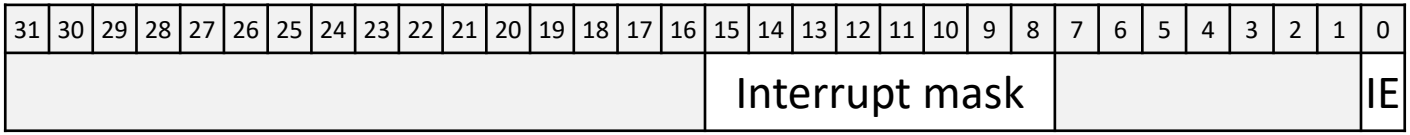
- **Cause** register (alias $cr):
  - Exception code (bits 0-6): describes the cause of the last exception

0	INT	External hardware interrupt
4	AdEL	Address Error (Load)
5	AdES	Address Error (Store)
6	IBUS	Invalid instruction
8	SYSCALL	Program-initiated system call
12	OVF	Arithmetic overflow

  - Pending interrupts (bits 8-15): when an interrupt is raised, the corresponding bit is asserted

MIPS exception handling: Status

In order to receive interrupts, the software has to enable them.
On a MIPS processor, this is done by writing to the **Status** register.



- **Status** register (alias $sr):
  - **Interrupt enable** (bit 0): interrupts are not handled when 0
  - **Interrupt mask** (bits 8-15): defining masks for the eight interrupt levels
    - If an interrupt/exception occurs when its mask is set to 0, it is ignored though pending interrupt bit is raised
    - Pending interrupt is handled as soon as its mask is re-enabled

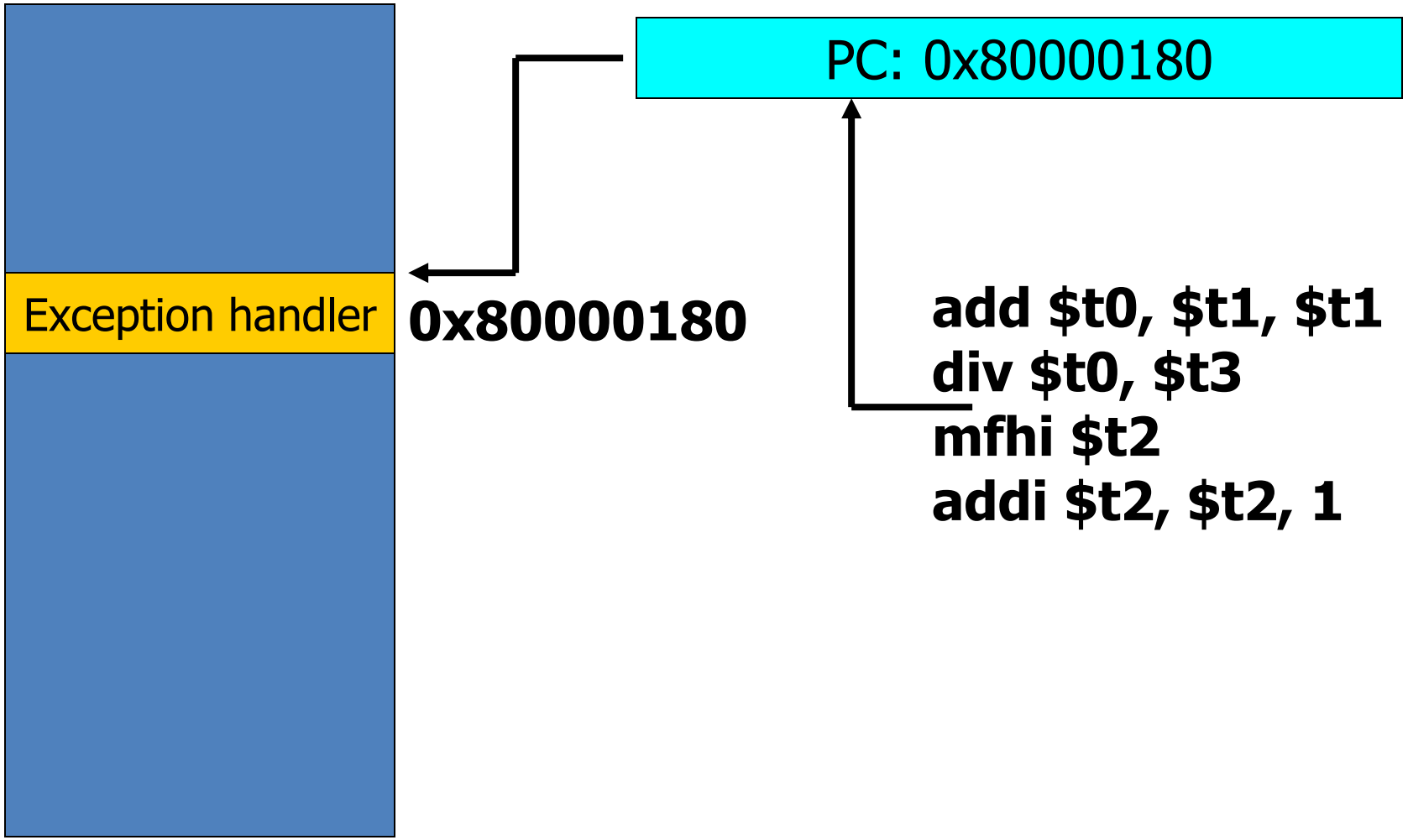
Exceptions

- Unscheduled function call to *exception handler*
- Caused by:
  - Hardware, also called an *interrupt*, e.g., keyboard
  - Software, also called *traps*, e.g., undefined instruction
- When exception occurs, the processor:
  - Records the cause of the exception
  - Jumps to exception handler (at instruction address 0x80000180)
  - Returns to program

Exception vs. function call

- When a function is called via `jal`:
  - Caller saves state prior to issuing `jal` so callee can safely modify registers
  - Control is transferred by setting PC to the address provided by the instruction
  - Return address to next instruction (PC+4) is saved to $ra
  - Function returns via `jr $ra`

Example


PC: 0x80000180
Exception handler
0x80000180
add $t0, $t1, $t1
div $t0, $t3
mfhi $t2
addi $t2, $t2, 1

Exception Flow

- Processor saves cause and exception PC in `Cause` and `EPC`
- Processor jumps to exception handler (0x80000180)
- Exception handler:
  - Saves registers on stack
  - Reads `Cause` register

    ```
    mfc0 $k0, Cause
    ```
  - Handles exception
  - Restores registers
  - Returns to program

    ```
    mfc0 $k0, EPC
    jr $k0
    ```

Exception Flow

- Exceptions/interrupts have no explicit call:
  - Control is transferred to a fixed location (0x80000180) where the **exception handler** must reside
  - EPC stores the return address (PC+4)
  - Status register IE bit is masked to disable interrupts
  - If control is returned to the program, special instruction `rfe` (return from exception) restores the status register reenabling interrupts
- An exception is an unexpected event and the exception handler shall take care of saving and restoring the previous state
- Exception handlers may use registers $26-$27 (alias $k0-$k1) that by convention shall not be used by user programs

Exception Handler

```
if (cause == Arithmetic Overflow)
      ArithmeticOverflowHandler();
else if (cause == DivideByZero)
      DivideByZeroHandler();
else if (cause == Illegal Instruction)
      IllegalInstructionHandler();
else if (cause == external interrupt)
      InterruptHandler();
-----
-----
```

Exception Handler

```
exception-handler()
{
  /* keep the interrupts disabled */
   (1) save EPC, cause and updated
        register on system stack
   (2) enable interrupts
   (3) decode cause and call appropriate
        ISR
 }
```

ISR = Interrupt Service Routine

```
exception-handler:
  mfc0 $k0, $cause
  mfc0 $k1, $EPC
  sw $k0, -4($sp)
  sw $k1, -8($sp)
  sw $t0, -12($sp)
  addi $sp, $sp, -12
  mfc0 $t0, $status
  ori $t0, $t0, 0x1
  mtc0 $status, $t0
  andi $k0, $k0, 0x7c
  beq $k0, $0, interrupt
  ---
  ---
```

Timer Interrupt

- Additional cp0 registers:
  - Count ($9)
  - Compare ($11)
- cp0 generates interrupt when Count==Compare
- Can be used to implement pre-emptive multitasking OS kernel

Signed & Unsigned Instructions

- Addition and subtraction
- Multiplication and division
- Set less than

Addition & Subtraction

- **Signed:** `add`, `addi`, `sub`
  - Same operation as unsigned versions
  - But processor takes exception on overflow
- **Unsigned:** `addu`, `addiu`, `subu`
  - Doesn't take exception on overflow

Note: `addiu` sign-extends the immediate

Multiplication & Division

- **Signed:** `mult`, `div`
- **Unsigned:** `multu`, `divu`

Set Less Than

- **Signed:** `slt`, `slti`
- **Unsigned:** `sltu`, `sltiu`

Note: `sltiu` sign-extends the immediate before comparing it to the register

Loads

- **Signed:**
  - Sign-extends to create 32-bit value to load into register
  - Load halfword: `lh`
  - Load byte: `lb`
- **Unsigned:**
  - Zero-extends to create 32-bit value
  - Load halfword unsigned: `lhu`
  - Load byte: `lbu`