Chapter 6 :: Architecture

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris

Lectured by Jeong-Gun Lee @ Hallym

Chapter 6 :: Topics

- **Introduction**
- **Assembly Language (어셈블리 언어)**
- **Machine Language (기계어)**
- **Programming (프로그래밍)**
- **MIPS Memory Map (메모리 구조)**

Introduction

- Jumping up a few levels of **abstraction** (추상화).
- **Architecture**: the programmer's view of the computer (프로그래머 관점)
  - Defined by instructions (operations) and operand locations
- **Microarchitecture**: how to implement an architecture in hardware (covered in Chapter 7) : 어떻게 하드웨어로 구현할 것인가!

Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons

Assembly Language

- To command a computer, you must understand its language.
  - Instructions(명령어): words in a computer's language
  - Instruction set(명령어집합): the vocabulary of a computer's language

- Instructions indicate the **operation** to perform and the **operands** to use.(명령어는 수행할 명령어와 사용할 데이터를 나타낸다)
  - Assembly language: human-readable format of instructions
    어셈블리 언어: 사람이 읽을 수 있는 명령어 포맷/형식
  - Machine language: computer-readable format (1's and 0's)
    기계어: 기계가 읽을 수 있는 명령어 포맷/형식

Assembly Language

- We introduce the MIPS architecture.
  - Developed by John Hennessy and his colleagues at Stanford and 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.

What ? Assembly ?

```
# a program that prints "hello world"

        .text
        .globl __start
__start:
        la $a0, str
        li $v0, 4
        syscall
        li $v0, 10
        syscall



        .data
str:     .asciiz "hello world₩n"
```

System Call Service

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	
print float	2	$f12 = float to print	
print double	3	$f12 = double to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read float	6		$f0 contains float read
read double	7		$f0 contains double read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	*See note below table*
sbrk (allocate heap memory)	9	$a0 = number of bytes to allocate	$v0 contains address of allocated memory
exit (terminate execution)	10		

Instructions: Addition (덧셈)

- The most common operation computers perform is addition. (컴퓨터가 수행하는 가장 일반적인 연산! – 덧셈!)

High-level code

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

MIPS assembly code

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

연산명령

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

Instructions: Subtraction

- Subtraction is similar to addition. Only the mnemonic changes.

High-level code

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

MIPS assembly code

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

- sub: the mnemonic indicates what operation to perform
- b, c: source operands on which the operation is performed
- a: destination operand to which the result is written

Instructions: More Complex Code

- More complex code is handled by multiple MIPS instructions.

High-level code

```
a = b + c - d;
// single line comment
/* multiple line
   comment */
```

MIPS assembly code

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

Operands

- A computer needs a physical location from which to retrieve binary operands.
- A computer retrieves operands from:
  - **Registers**
  - **Memory**
  - **Constants** (also called *immediates (즉치)*)

Operands: Registers

- Memory is slow.
- Most architectures have a small set of (**fast**) registers.
- MIPS has **thirty-two 32-bit registers**.
- MIPS is called a 32-bit architecture because it operates on 32-bit data.

(A 64-bit version of MIPS also exists, but we will consider only the 32-bit version.)

The MIPS Register Set

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

Operands: Registers

- Registers:
  - Written with a dollar sign (`$`) before their name
  - For example, register 0 is written "`$0`". It can be pronounced "register zero" or "dollar zero".

- Certain registers are used for specific purposes.
  - For example,
    - `$0` always holds the constant value 0.
    - the temporary registers, `$t0` - `$t9` are used to hold temporary values.

- For now, we only use the temporary registers (`$t0` - `$t9`) and the saved registers (`$s0` - `$s7`).
- We will use the other registers in later slides.

Instructions with registers

- We revisit previous code using registers to hold the variables.

High-level code

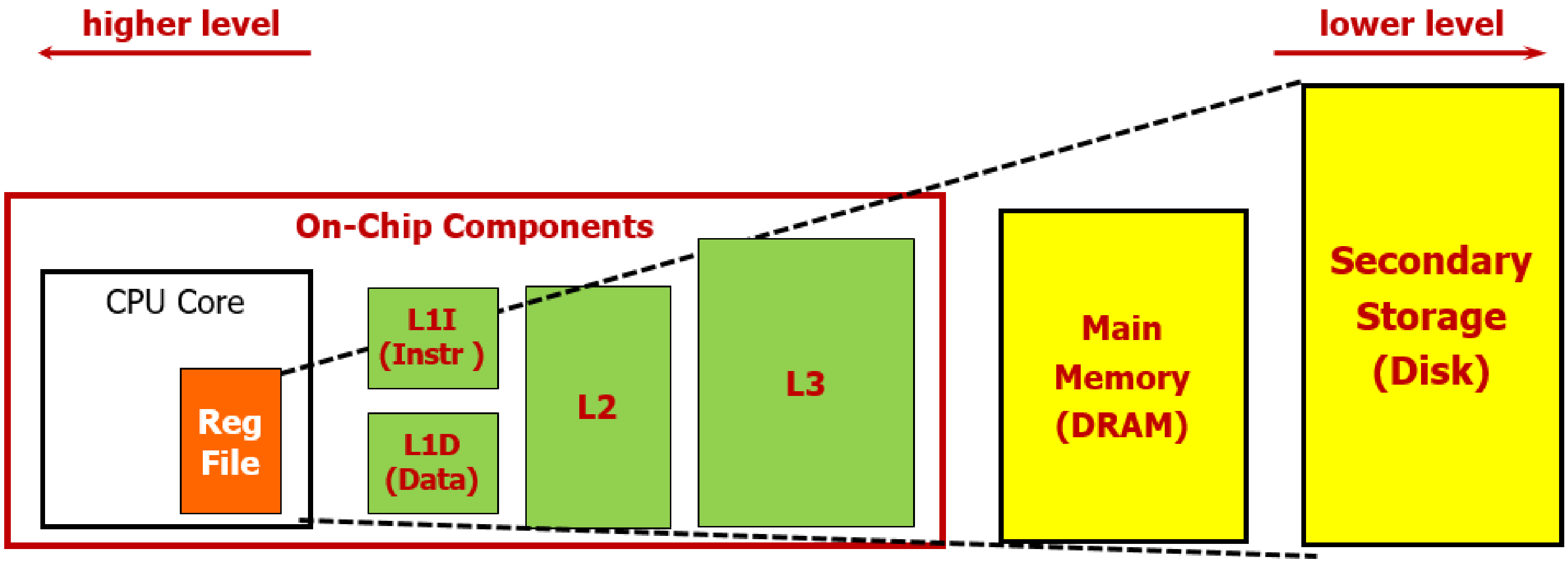
```
a = b + c
```

MIPS assembly code

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

Operands: Memory

- All the data doesn't fit in 32 registers
- Also store operands in memory
- **Memory is large**, so it can hold a lot of data.
- But it's also **slow**.
- **Commonly used variables are kept in registers**.



Word-Addressable Memory

- For explanation purposes, we begin by describing a word-addressable memory, where each 32-bit data word has a unique address.

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Reading Word-Addressable Memory

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

Assembly code

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

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Writing Word-Addressable Memory

- Memory writes are called ***stores***.
- The mnemonic is ***store word*** (`sw`).
- Write (store) the value held in `$t4` into memory address 7.
- The offset can be written in decimal (default) or hexadecimal.
- Memory address calculation:
  - Add the base address (`$0`) to the offset (0x7).
  - In this case, the memory address is ($0 + 0x7) = 7.
- Any register may be used to store the base address.

Assembly code

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

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Byte-Addressable Memory

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

Word Address	Data				
⋮		⋮			⋮
0000000C	4 0	F 3	0 7	8 8	Word 3
00000008	0 1	E E	2 8	4 2	Word 2
00000004	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0

width = 4 bytes

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 assembly code

```
lw $s3, 4($0)   # read memory word 1 into $s3
```

Word Address	Data				
⋮	⋮				⋮
0000000C	4 0	F 3	0 7	8 8	Word 3
00000008	0 1	E E	2 8	4 2	Word 2
00000004	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0

width = 4 bytes

Writing Byte-Addressable Memory

- The assembly code below stores the value held in `$t7` into memory address 0x2C (44).

MIPS assembly code

```
sw $t7, 44($0)  # write $t7 into memory word 11
```

Word Address	Data				
⋮	⋮				⋮
0000000C	40	F3	07	88	Word 3
00000008	01	EE	28	42	Word 2
00000004	F2	F1	AC	07	Word 1
00000000	AB	CD	EF	78	Word 0

width = 4 bytes

Big-Endian and Little-Endian Memory

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

Big-Endian Byte Address				Word Address	Little-Endian Byte Address			
⋮				⋮	⋮			
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB

Big-Endian and Little-Endian Memory

- From Jonathan Swift's *Gulliver's Travels* where the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end.
- As indicated by the farcical name, it doesn't really matter which addressing type is used – except when the two systems need to share data!

Big-Endian Byte Address				Word Address	Little-Endian Byte Address			
⋮				⋮	⋮			
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB

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

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

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

Big-Endian

Byte Address	0	1	2	3
Data Value	23	45	67	89
	MSB			LSB

Word Address: 0

Little-Endian

3	2	1	0	Byte Address
23	45	67	89	Data Value
MSB			LSB	

Operands: Constants/Immediates

- `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
```

Machine Language

- Computers only understand 1's and 0's
- Machine language: binary representation of instructions
- 32-bit instructions
  - Again, simplicity favors regularity: 32-bit data and instructions
- **Three instruction formats**:
  - R-Type: register operands
  - I-Type: immediate operand
  - J-Type: for jumping (we'll discuss later)

R-Type

- *Register-type*
- 3 register operands:
  - `rs`, `rt`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode*
  - `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

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

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:
i.e., `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 the opcode

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

I-Type Examples

Assembly Code / Field Values

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

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	

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

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

Machine Language: J-Type

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

J-Type

op	addr
6 bits	26 bits

Review: Instruction Formats

R-Type

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

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J-Type

op	addr
6 bits	26 bits

The Power of the Stored Program

- 32-bit instructions and data stored in memory
- Sequence of instructions: only difference between two applications (for example, a text editor and a video game)
- To run a new program:
  - No tedious rewiring required
  - Simply store new program in memory
- The processor hardware executes the program:
  - *fetches* (reads) the instructions from memory in sequence
  - performs the specified operation
- A program counter (PC) keeps track of the current instruction.
- In MIPS, programs typically start at memory address 0x00400000.

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

Address	Instructions	
⋮	⋮	
0040000C	0 1 6 D 4 0 2 2	
00400008	2 2 6 8 F F F 4	
00400004	0 2 3 2 8 0 2 0	
00400000	8 C 0 A 0 0 2 0	← PC
⋮	⋮	

Main Memory

Interpreting Machine Language Code

- Start with opcode
- Opcode tells how to parse the remaining bits
- If opcode is all 0's
  - R-type instruction
  - Function bits tell what instruction it is
- Otherwise
  - opcode tells what instruction it is

Machine Code

	op	rs	rt	imm
(0x2237FFF1)	001000	10001	10111	1111 1111 1111 0001
	2	2	3 7	F F F 1

	op	rs	rt	rd	shamt	funct
(0x02F34022)	000000	10111	10011	01000	00000	100010
	0	2	F 3	4	0	2 2

Field Values / **Assembly Code**

op	rs	rt	imm	Assembly Code
8	17	23	-15	`addi $s7, $s1, -15`

op	rs	rt	rd	shamt	funct	Assembly Code
0	23	19	8	0	34	`sub $t0, $s7, $s3`

Programming

- High-level languages:
  - e.g., C, Java
  - Written at more abstract level
- Common high-level software constructs:
  - if/else statements
  - for loops
  - while loops
  - array accesses
  - procedure calls
- Other useful instructions:
  - Arithmetic/logical instructions
  - Branching

Logical Instructions

- `and`, `or`, `xor`, `nor`
  - `and`: useful for *masking* bits
    - Masking all but the least significant byte of a value:

      0xF234012F AND 0xFF = 0x0000002F
  - `or`: useful for combining bits
    - 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

Logical Instruction Examples

Source Registers

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

Result

Assembly Code									
`and $s3, $s1, $s2`	$s3	0100	0110	1010	0001	0000	0000	0000	0000
`or  $s4, $s1, $s2`	$s4	1111	1111	1111	1111	1111	0000	1011	0111
`xor $s5, $s1, $s2`	$s5	1011	1001	0101	1110	1111	0000	1011	0111
`nor $s6, $s1, $s2`	$s6	0000	0000	0000	0000	0000	1111	0100	1000

Logical Instruction Examples

Source Values

$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

zero-extended

Assembly Code

```
andi $s2, $s1, 0xFA34
ori  $s3, $s1, 0xFA34
xori $s4, $s1, 0xFA34
```

Result

$s2	0000	0000	0000	0000	0000	0000	0011	0100
$s3	0000	0000	0000	0000	1111	1010	1111	1111
$s4	0000	0000	0000	0000	1111	1010	1100	1011

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`

Variable shift instructions:

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

Shift Instructions

Assembly Code / Field Values

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

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`:

High-level code

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

MIPS assembly code

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

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

  (`lui` loads the 16-bit immediate into the upper half of the register and sets the lower half to 0.)

High-level code

```
int a = 0xFEDC8765;
```

MIPS assembly code

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

Multiplication, Subtraction

- Special registers: `lo`, `hi`
- 32 × 32 multiplication, 64 bit result
  - `mult $s0, $s1`
  - Result in `hi`, `lo`
- 32-bit division, 32-bit quotient, 32-bit remainder
  - `div $s0, $s1`
  - Quotient in `lo`
  - Remainder in `hi`

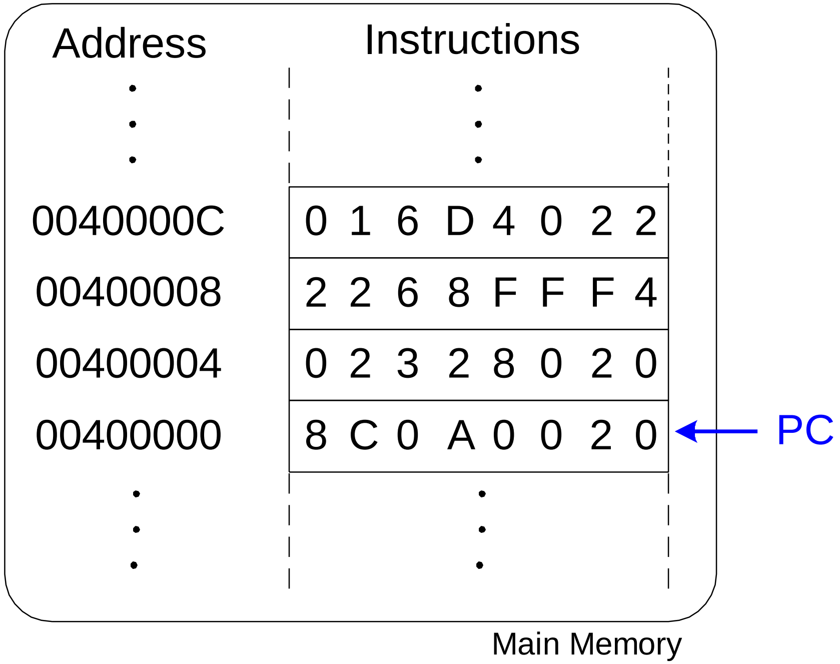
Branching

- Allows a program to execute instructions out of sequence.
- Types of branches:
  - **Conditional branches:**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
  - **Unconditional branches:**
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)

Review: 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



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 locations in a program. They cannot use reserve words and must be followed by a 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)
```

High-Level Code Constructs

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

If Statement

High-level 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
```

Notice that the assembly tests for the opposite case (`i != j`) than the test in the high-level code (`i == j`).

If / Else Statement

High-level 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:
```

While Loops

High-level 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:
```

Notice that the assembly tests for the opposite case (`pow == 128`) than the test in the high-level code (`pow != 128`).

For Loops

The general form of a for loop is:

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

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

For Loops

High-level 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
       addi $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:
```

Notice that the assembly tests for the opposite case (`i == 128`) than the test in the high-level code (`i != 10`).

For Loops: Using `slt`

High-level 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.

Arrays

- Useful for accessing large amounts of similar data
- Array element: accessed by *index*
- Array *size:* number of elements in the array

Arrays

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

0x12340010	array[4]
0x1234800C	array[3]
0x12348008	array[2]
0x12348004	array[1]
0x12348000	array[0]

Arrays

```
// high-level code
    int array[5];
    array[0] = array[0] * 2;
    array[1] = array[1] * 2;

# MIPS assembly code
# array base address = $s0
 lui  $s0, 0x1234            # put 0x1234 in upper half of $S0
 ori  $s0, $s0, 0x8000       # put 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

```
// high-level code
   int array[1000];
   int i;

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

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:
```

ASCII Codes

- *American Standard Code for Information Interchange* (*ASCII*): assigns each text character a unique byte value
- For example, S = 0x53, a = 0x61, A = 0x41
- Lower-case and upper-case letters differ by 0x20 (32).
- See Table 6.2 in *Digital Design and Computer Architecture*, Harris and Harris, for a complete list of ASCII codes

Procedure Calls

Definitions

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

High-level code

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

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

Procedure Calls

Procedure calling conventions:

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

MIPS conventions:

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

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.

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
```

`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

MIPS conventions:

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

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

```
# $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 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
```

Input Arguments and Return Values

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 when more space is needed
- *Contracts*: 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

Address	Data	
7FFFFFFC	12345678	⟵ $sp
7FFFFFF8		
7FFFFFF4		
7FFFFFF0		
⋮	⋮	

Address	Data	
7FFFFFFC	12345678	
7FFFFFF8	AABBCCDD	
7FFFFFF4	11223344	⟵ $sp
7FFFFFF0		
⋮	⋮	

How Procedures use the Stack

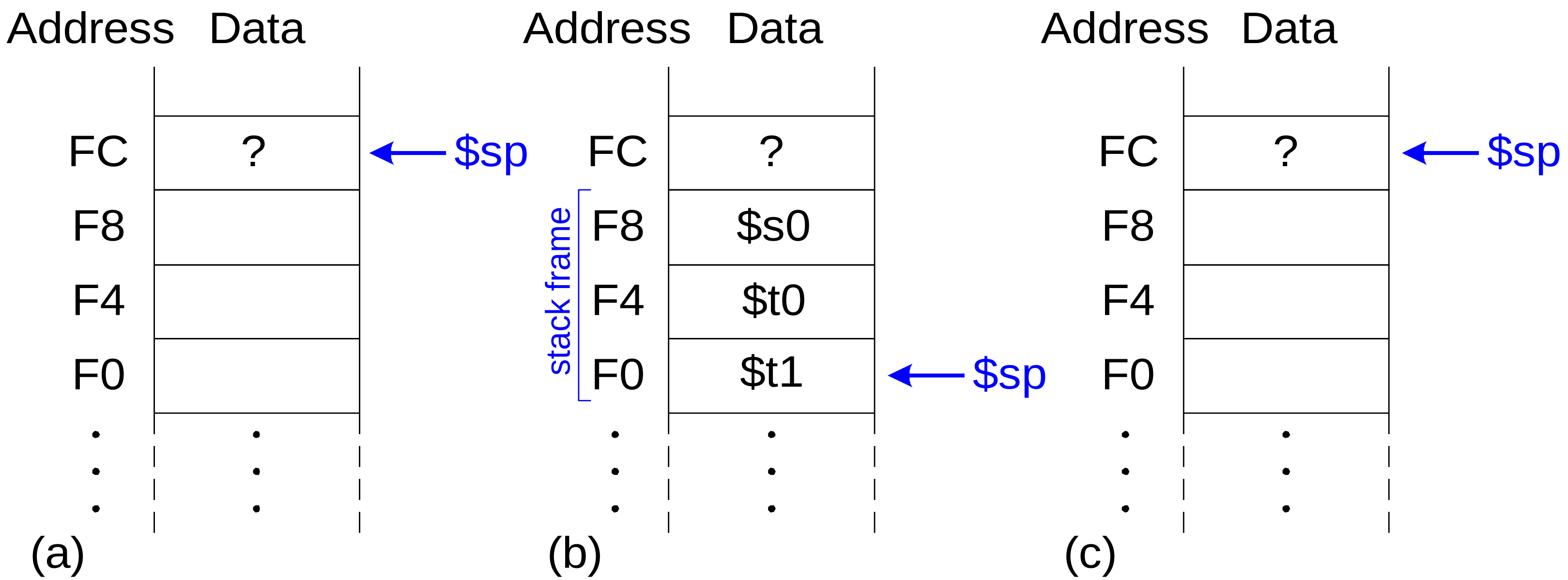
- Called procedures must have no other unintended side effects.
- But `diffofsums` overwrites 3 registers: `$t0`, `$t1`, `$s0`

```
# 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
```

The Stack during `diffofsums` Call



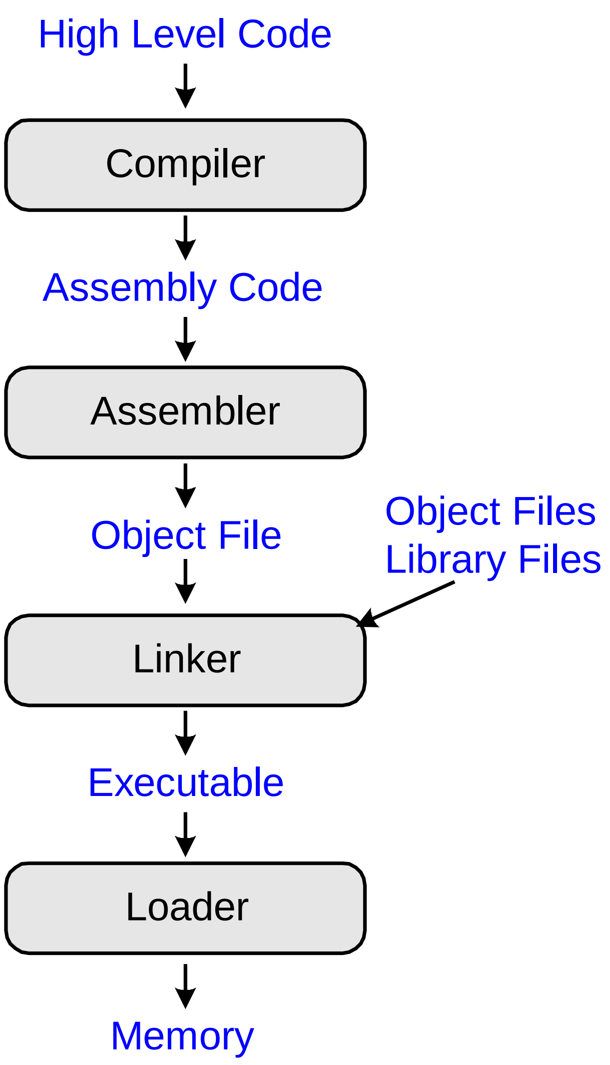
Multiple Procedure Calls

```
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
  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
```

Running a Program



Example Program: C Code

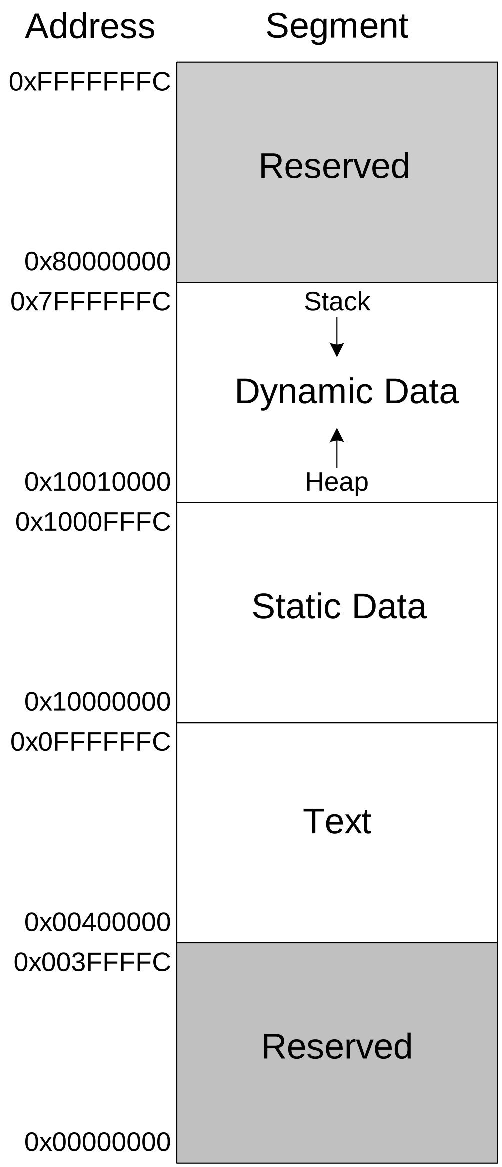
```
int f, g, y;  // global variables


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

  return y;
}


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

Example Program: Assembly Code

```c
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
```

The MIPS Memory Map



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	addi $sp, $sp, -4
	0x00400004	0xAFBF0000	sw $ra, 0 ($sp)
	0x00400008	0x20040002	addi $a0, $0, 2
	0x0040000C	0xAF848000	sw $a0, 0x8000 ($gp)
	0x00400010	0x20050003	addi $a1, $0, 3
	0x00400014	0xAF858004	sw $a1, 0x8004 ($gp)
	0x00400018	0x0C10000B	jal 0x0040002C
	0x0040001C	0xAF828008	sw $v0, 0x8008 ($gp)
	0x00400020	0x8FBF0000	lw $ra, 0 ($sp)
	0x00400024	0x23BD0004	addi $sp, $sp, -4
	0x00400028	0x03E00008	jr $ra
	0x0040002C	0x00851020	add $v0, $a0, $a1
	0x00400030	0x03E0008	jr $ra

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