FU05 Computer Architecture

4.Assembly Language 2
(アセンブリ言語2)

Ben Abdallah Abderazek
E-mail: benab@u-aizu.ac.jp

Procedure calls

```
main() {
 if (a == 0)
  b = (g+h)-(g*16);
 else
  c = (k+m)-(k*16);
}
```

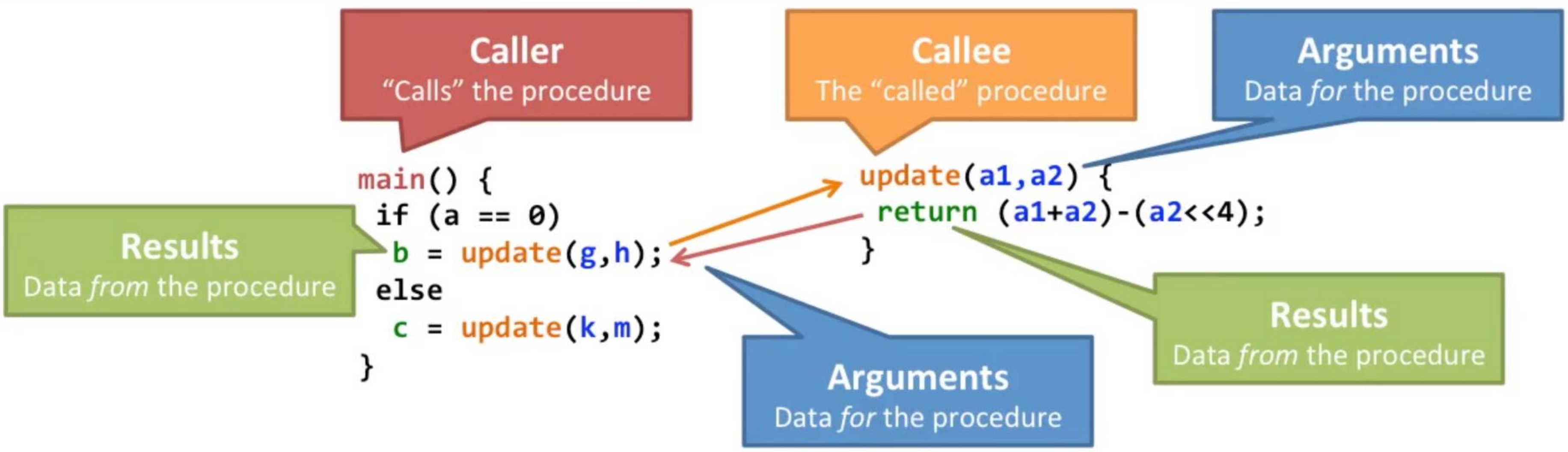
```
main() {
 if (a == 0)
  b = update(g,h);
 else
  c = update(k,m);
}
```

```
update(a1,a2) {
 return (a1+a2)-(a2<<4);
}
```

- **Procedures (functions/subroutines/) are needed for structured programming**
  - Avoid repeated code
  - Call functions you didn't write (libraries)

- **What needs to happen:**
  - Put data where the procedure can access it
  - Start the procedure
    - Calculate
    - Put the results where the caller can access them

Procedure call terminology



- The Caller calls the procedure
- The procedure is the Callee
- The Caller gives the Callee Arguments (data)
- The Callee returns Results (data) to the Caller

How to do a procedure call

- Transfer control to the callee to start the procedure:

```
jal ProcedureAddress      ; jump-and-link to the procedure
```

  - Keeps track of the instruction after the **jal** so we can continue in the right place when we are done with the procedure
  - **Stores the return address (PC+4) in $ra (R31)**

- **Return** control to the caller when the procedure is done:

```
jr $ra                    ; jump-return to the address in $ra
```

  - **Jumps back to the address stored in $ra (R31)**
  - This is why you need to store the return address so you know to go back to!

Jump-and-link

Program

```
4:   addi R1, R0, 12
8:   jal my_procedure
12:  add R1, R2, R2

my_procedure:
80:  addi R2, R1, 8
84:  jr
```

Register File

R0:	0
R1:	40
R2:	20
...	
R31:	12

Register Usage

Name	Register Number	Usage	Preserved by callee?
$zero	0	hardwired 0	N/A
$v0-$v1	2-3	return value and expression evaluation	no
$a0-$a3	4-7	arguments	no
$t0-$t7	8-15	temporary values	no
$s0-$s7	16-23	saved values	YES
$t8-$t9	24-25	more temporary values	no
$gp	28	global pointer	YES
$sp	29	stack pointer	YES
$fp	30	frame pointer	YES
$ra	31	return address	YES

Procedure Call Instructions

- Procedure call: **jump and link**

  `jal ProcedureLabel`

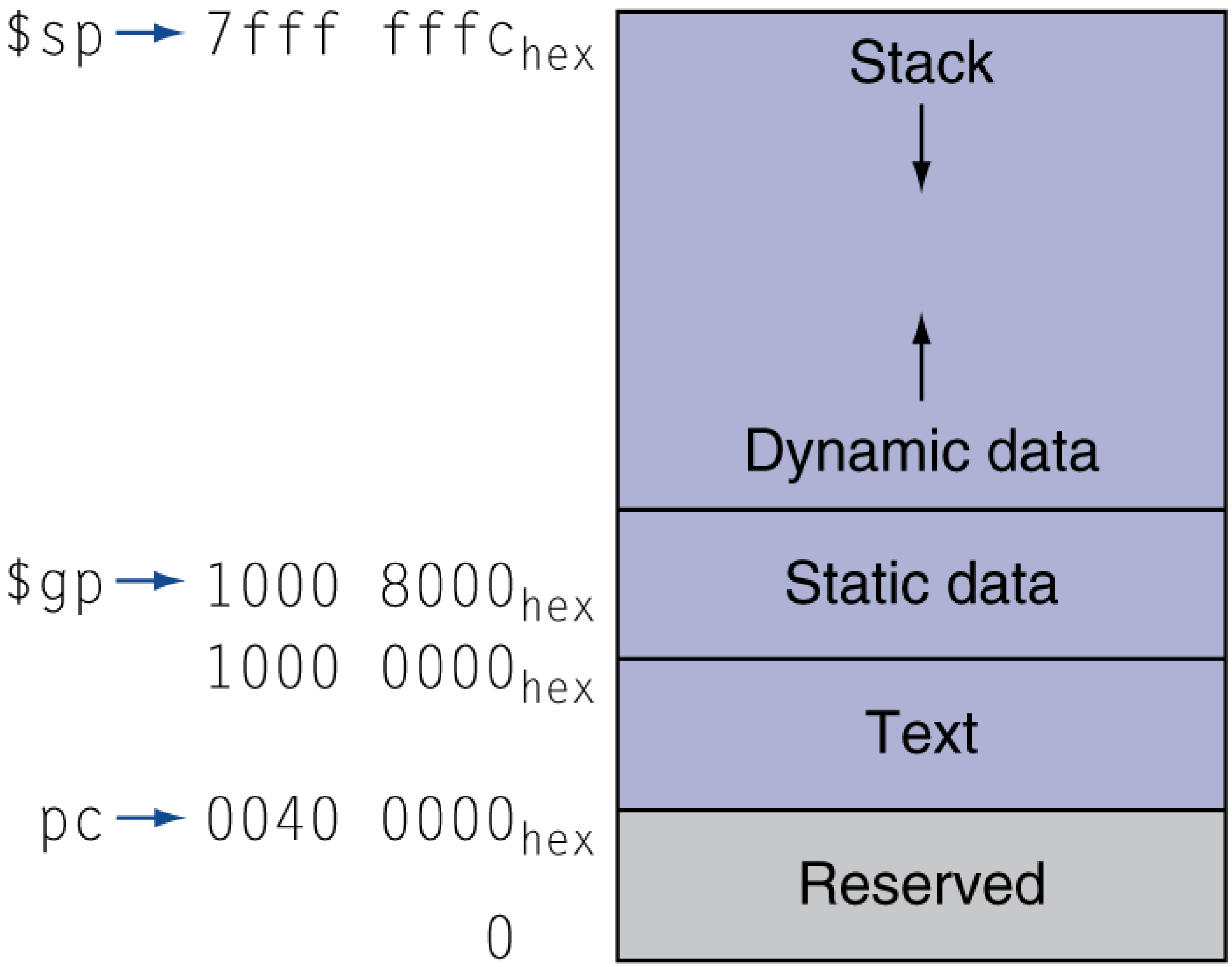
  - Address of following instruction (PC+4) put in `$ra`
  - Jumps to target address

- Procedure return: **jump register**

  `jr $ra`

  - Copies `$ra` to PC

Memory Layout



Stack (スタック)
Automatic storage

Dynamic data (動的データ)
Heap
E.g., malloc in C, new in Java

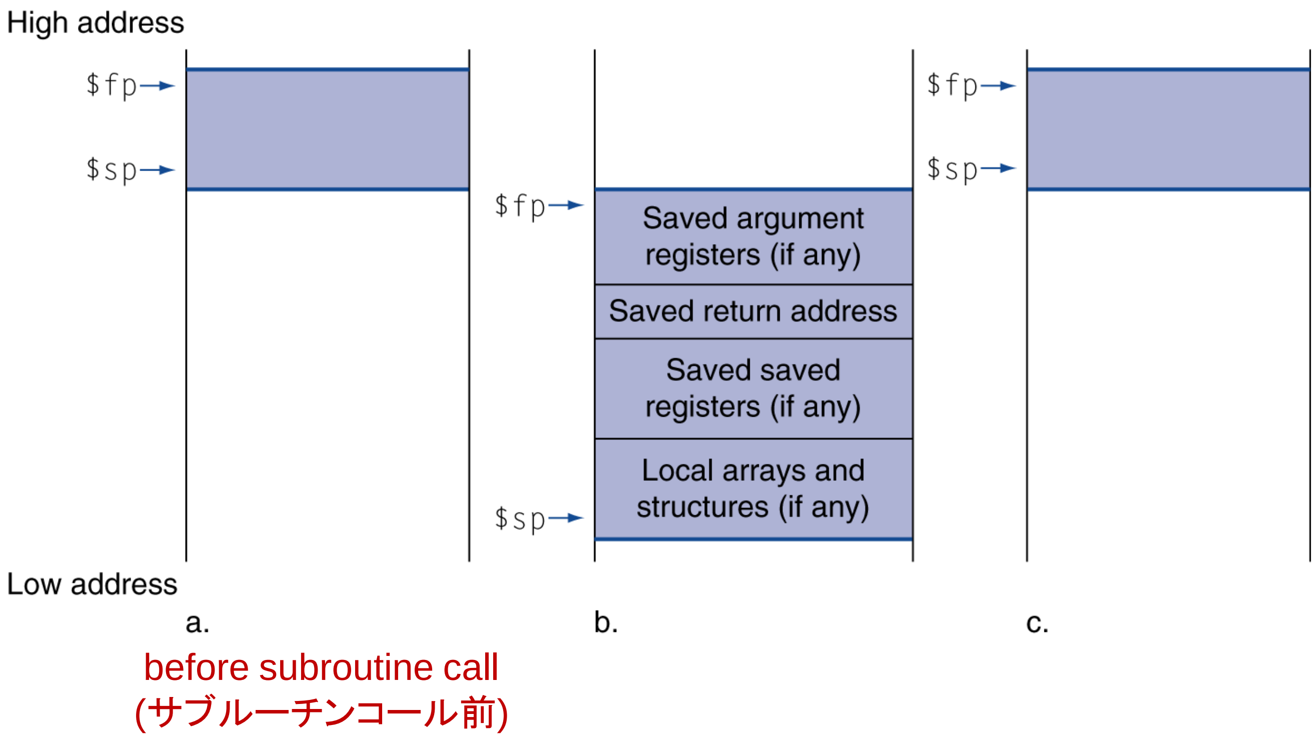
Static data (静的データ)
Global variables
e.g., static variables in C,
constant arrays and strings

Text (テキスト＝プログラム)
Program code

Reserved (予約済み)

$gp initialized to address allowing ±offsets into this segment

Local Data on the Stack (スタック)



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Pass Parameters and Return Values With Registers

- Up to 4 words size arguments can be passed to a function through $a0 -$a3
- A function can return up to two word size values through $v0 -$v1

```
      add $a0, $zero, 100    # args in $a0-$a3
      jal sub                # Call subprogram
      add $t2, $v0, $zero    # Get return val
      ...
sub:  add $t0, $a0, $zero    # Copy param
      ...                    # Do subprogram job
      add $v0, $t1, $zero    # Set return val
      jr  $ra                # Return
```

Pass Parameters and Return Values With Stack

- If more **data** needs to be communicated across the function call or return than is available space in the registers.
- The **caller** will reserve a space on the Stack for any **arguments** or **return values** that need to use the Stack.

```
     addi $sp, $sp, -8 # Save space in stack
     sw   $s0, 0($sp)  # Store args in stack
     jal  sub          # Call subprogram
     lw   $s1, 4($sp)  # Load return vals
     addi $sp, $sp, 8  # Restore space in stack
     ...
sub: lw  $t0, 0($sp)   # Read args from stack
     ...               # Do the subprogram job
     sw  $t1, 4($sp)   # Put return val in stack
     jr  $ra           # Return
```

Leaf Procedure Example

- A Procedure that does not make any additional call is known as a **Leaf Procedure**

- C code:

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

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Return value (result) in $v0

Leaf Procedure Example

- MIPS code:

```
leaf_example:
  addi $sp, $sp, -4
  sw   $s0, 0($sp)
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero
  lw   $s0, 0($sp)
  addi $sp, $sp, 4
  jr   $ra
```

- Save $s0 on stack
- Procedure body
- Result
- Restore $s0
- Return

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

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example 1

- C code:

```
int non_leaf (int g, h, i, j)
{ int f;
  f = leaf (g + h, i + j);
  return f;
}
int leaf (int m, n)
{ int f;
  f = m - n;
  return f;
}
```

Argument n in $a0

- Result in $v0

Non-Leaf Procedure Example 1

non_leaf:

```
add $t0, $a0, $a1
add $t1, $a2, $a3
addi $sp, $sp, -4
sw   $ra, 0($sp)
add $a0, $t0, $zero
add $a1, $t1, $zero
jal leaf
add $t2, $v0, $zero
add $v0, $t2, $zero
lw   $ra, 0($sp)
addi $sp, $sp, 4
jr $ra
```

leaf:

```
sub $t0, $a0, $a1
add $v0, $t0, $zero
jr  $ra
```

```c
int non_leaf (int g, h, i, j)
{ int f;
  f = leaf (g + h, i + j);
  return f;
}
int leaf (int m, n)
{ int f;
  f = m - n;
  return f;
}
```

Non-Leaf Procedure Example 2

- C code

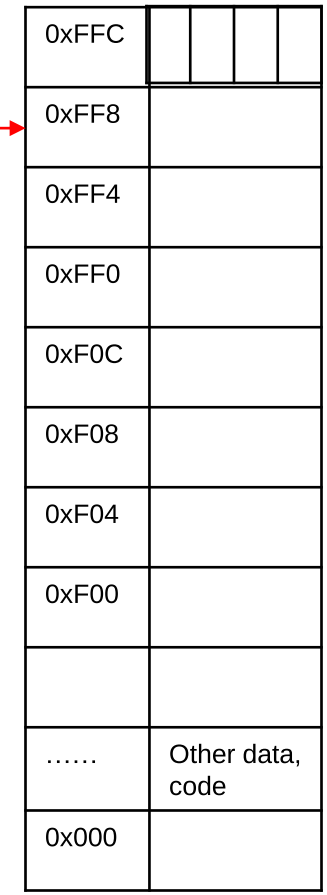
```
int  fact (int n)
{
        if (n < 1) return f;
        else return n * fact (n-1);
}
```

- Augment n in $a0
- Result in $v0

Non-Leaf Procedure Example 2

MIPS code:

```
fact:
      addi $sp, $sp, -8     # adjust stack for 2 items
      sw   $ra, 4($sp)      # save return address
      sw   $a0, 0($sp)      # save argument
      slti $t0, $a0, 1      # test for n < 1
      beq  $t0, $zero, L1   # branch to L1 if n is not< 1
      addi $v0, $zero, 1    # if so, result is 1
      addi $sp, $sp, 8      #   pop 2 items from stack
      jr   $ra              #   and return
L1: addi $a0, $a0, -1      # else decrement n
      jal  fact             # recursive call
      lw   $a0, 0($sp)      # restore original n
      lw   $ra, 4($sp)      #   and return address
      addi $sp, $sp, 8      # pop 2 items from stack
      mul  $v0, $a0, $v0    # multiply to get result
      jr   $ra              # and return
```



stack

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address

- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

```
move $t0, $t1     →  add $t0, $zero, $t1
blt $t0, $t1, L   →  slt $at, $t0, $t1
                     bne $at, $zero, L
```

  - $at (register 1): assembler temporary

Conclusions

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

Conclusions (Cont.)

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`add, sub, addi`	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	12%	4%
Cond. Branch	`beq, bne, slt, slti, sltiu`	34%	8%
Jump	`j, jr, jal`	2%	0%

Practice Problems (練習問題)

- **2.7** Show how the value 0xabcdef12 would be arranged in memory of a little-endian and a big-endian machine. Assume the data is stored starting at address 0.

Solution

2.7

Practice Problems (練習問題)

- **2.7** Show how the value 0xabcdef12 would be arranged in memory of a little-endian and a big-endian machine. Assume the data is stored starting at address 0.

Solutions

2.7

Little-Endian		Big-Endian	
Address	Data	Address	Data
12	ab	12	12
8	cd	8	ef
4	ef	4	cd
0	12	0	ab

Practice Problems (練習問題)

2.12 Assume that registers `$s0` and `$s1` hold the values `0x80000000` and `0xD0000000`, respectively.

2.12.1 [5] <§2.4> What is the value of `$t0` for the following assembly code?

```
add $t0, $s0, $s1
```

2.12.2 [5] <§2.4> Is the result in `$t0` the desired result, or has there been overflow?

2.12.3 [5] <§2.4> For the contents of registers `$s0` and `$s1` as specified above, what is the value of `$t0` for the following assembly code?

```
sub $t0, $s0, $s1
```

2.12.4 [5] <§2.4> Is the result in `$t0` the desired result, or has there been overflow?

2.12.5 [5] <§2.4> For the contents of registers `$s0` and `$s1` as specified above, what is the value of `$t0` for the following assembly code?

```
add $t0, $s0, $s1
add $t0, $t0, $s0
```

2.12.6 [5] <§2.4> Is the result in `$t0` the desired result, or has there been overflow?

Practice Problems (練習問題)

Solution

2.12

2.12.1 50000000

2.12.2 overflow

2.12.3 B0000000

2.12.4 no overflow

2.12.5 D0000000

2.12.6 overflow