Instructions: Language of the Computer

涂哲誠、南政佑、楊志璿

Team 13

Introduction

- The words of computer's language are called "instructions".
- Its vocabulary is called an "instruction set".
- Similarity / simplicity of the equipment
- The goal : makes it easy to build the hardware and the compiler while maximizing performance, minimizing cost and energy.

Content

- Operation & Operands of the computer hardware
- Representing Instrutions In the computer
- Instructions for making decisions
- Supporting Procedures in computer hardware
- Parallelism and instrcutions : synchronization – code
- Sorting – code and kernel issue
- Lazy binding – code
- Other instruction

Operation & Operands of the computer hardware

- Simple add instruction (C code / Compiled MIPS code)
- Add and subtract, three operands(Two sources and one destination)

  a = b + c

  add a, b, c

- If you has 4 variables
- f = (g + h) - (i + j) : f, g, h, i, j correspond to $s0, $s1, $s2, $s3, $s4

  add $t0, $s1, $s2

  add $t1, $s3, $s4

  sub $s0, $t0, $t1

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits

Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

Operation & **Operands** of the computer hardware

Unlike programs in high-level languages, the operands of arithmetic instructions are restricted; they must be from a limited number of special locations built directly in hardware called registers.

- Memory Operands (MIPs)

A = B + C[8]

lw $t0 32($s3)

add $t0 $s2 $t0

sw $s1 $t0

- Constant / Immediate operands

D = D + 4

addi x22, x22, 4

Content

- Operation & Operands of the computer hardware
- **Representing Instrutions In the computer**
- Instructions for making decisions
- Supporting Procedures in computer hardware
- Palrallelism and instrcutions : synchronization – code
- Sorting – code and kernel issue
- Lazy binding – code
- Other instruction

Representing Instrutions In the computer

- Instructions are encoded in binary, called machine code.
- MIPS Instruction
  - It is encoded as an exact 32-bit instruction word.
  - When encoding operations or registers, it has a small number of instruction formats.
- Register numbers
  - $t0 - $t7 : Registers 8 –15
  - $s0 - $s7 : Registers 16-23
  - $t8 - $t9 : Registers 24-25

Ex) add $t0, $s1, $s2

The decimal representation is

0	17	18	8	0	32

The binary representation is

000000	10001	10010	01000	00000	100000

Representing Instrutions In the computer

✓ **R**-type and **I**-type : what's different?

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

- op : Basic operation of the instruction(opcode)
- rs : The first register source operand.
- rt : The second register source operand.
- rd : The register destination operand. It gets the result of the operation.
- shamt : Shift amount.(學到後面再說而先放著"0")
- funct : Function code (extends opcode)

- **R-type** (for Register)

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

- **I-type** (for immediate)

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

✓ Add $t0, $s1, $s2

0	17	18	8	0	32
000000	**10001**	**10010**	**01000**	**00000**	**100000**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

[예제] 다음과 같은 MIPS 명령어가 있다.

lw $t0, 1200($t1)

add $t0, $s2, $t0

sw $t0, 12000(t1)

이를 기계어로 변환하라.

```
lw   $t0,   1200($t1)      # Temporart reg $t0 gets A[300]
add  $t0,   $s2,     $t0   # Temporary reg $t0 gets h + A[300]
sw   $t0,   12000(t1)      # Stores h + A[300] back into A[300]
```

op	rs	rt	rd	Address/shamt	Funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

op	rs	rt	rd	Address/shamt	Funct
100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	1000000
101011	01001	01000	0000 0100 1011 0000		

MIPS OPERANDS

Name	Example	Comment
32 registers (General-purpose)	$s0 ~ $s7, $t0 ~ $t9	$t0 ~ $t7 = 8~15 $s0 ~ $s7 = 16~23 $t8, $t9 = 24, 25
2^{30} memory words (2^{32} memory bytes)	Memory[0], Memory[4], …	lw/sw 에 의해서만 access Sequential word addresses 는 4씩 차이

MIPS MACHINE LANGUAGE

R	OP code	rs	rt	rd	shamt	funct	Comment
	6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	
	OP Code	source1	source2	destination	Shift amount	function	
	000000	10010	10011	10001	00000	100000	add $s1, $s2, $s3
	000000	10010	10011	10001	00000	100010	sub $s1, $s2, $s3

I	OP code	rs	rt	immediate	Comment
	6 bit	5 bit	5 bit	16 bit	
	OP Code	source		constant	
	001000	10010	10011	0000 0000 0110 0100	addi $s1, $s2, 100
	100011	10010	10011	0000 0000 0110 0100	lw $s1, 100($s2)
	101011	10010	10011	0000 0000 0110 0100	sw $s1, 100($s2)

Content

- Operation & Operands of the computer hardware
- Representing Instrutions In the computer
- **Instructions for making decisions**
- Supporting Procedures in computer hardware
- Palrallelism and instrcutions : synchronization – code
- Sorting – code and kernel issue
- Lazy binding – code
- Other instruction

Instructions for making decisions

- Decision making is commonly represented in programming languages using the if statement.
- MIPS assembly language includes two decision-making instructions. For example :
- beq rs1, rs2, L1 //means go to the statement labeled L1 if the value in rs1 equals the value in rs2.
- bne rs1, rs2, L1 //means go to the statement labeled L1 if the value in rs1 does not equals the value in rs2.
- These two instructions are called "conditional branches".

- Use decision-making instructions to make a loop :
  - An example of loop in C :

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

  - Loop in assembly(MIPS)

```
Loop: sll   $t1, $s3, 2      //temp reg $t1 = $s3(i) * 4
      add   $t1, $t1, $t6    //$t1 = address of a[i]
      lw    $t0, 0($t1)      //temp reg $t0 = a[i]
      bne   $t0, $s5, Exit   //go to Exit if $t0(a[i]) != $s5(k)
      addi  $s3, $s3, 1      //i = i + 1
      j     Loop             //go to Loop
Exit:
```

Content

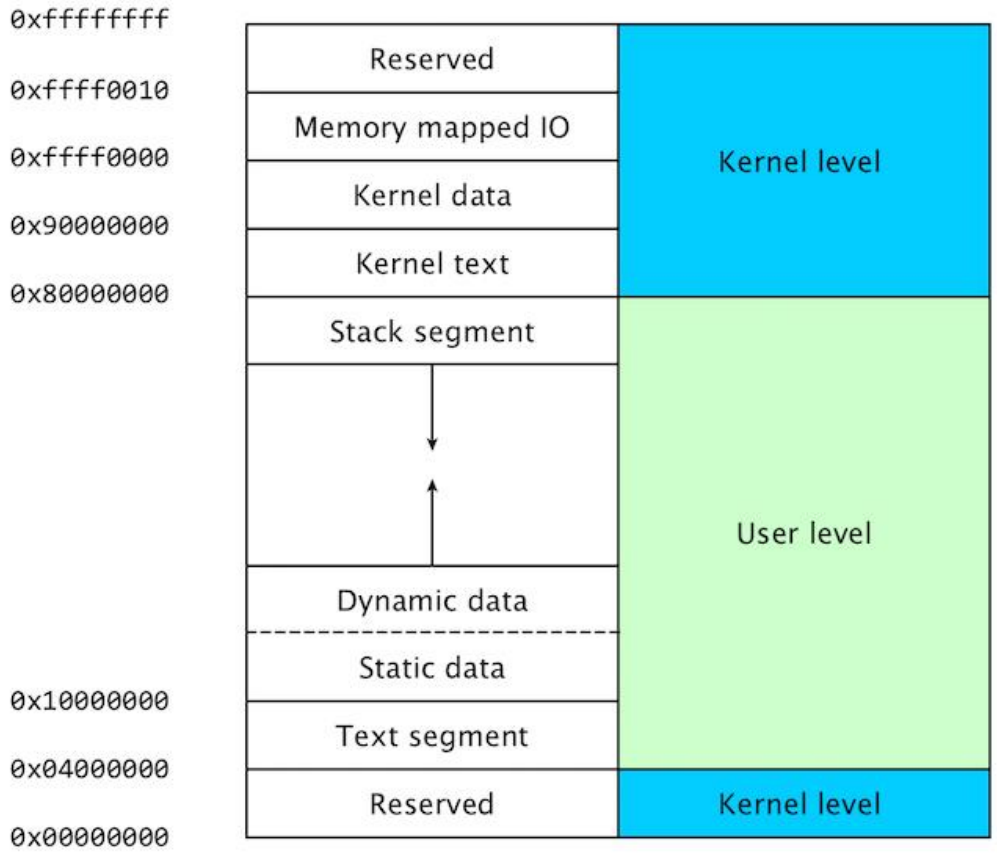
- Operation & Operands of the computer hardware
- Representing Instrutions In the computer
- Instructions for making decisions
- **Supporting Procedures in computer hardware**
- Palrallelism and instrcutions : synchronization – code
- Sorting – code and kernel issue
- Lazy binding – code
- Other instruction

Supporting Procedures in computer hardware

Supporting Procedures in computer hardware

So,
how to have a function call?

MIPS memory layout(32 bits)



http://www.it.uu.se/education/course/homepage/os/vt18/module-0/mips-and-mars/mips-memory-layout/

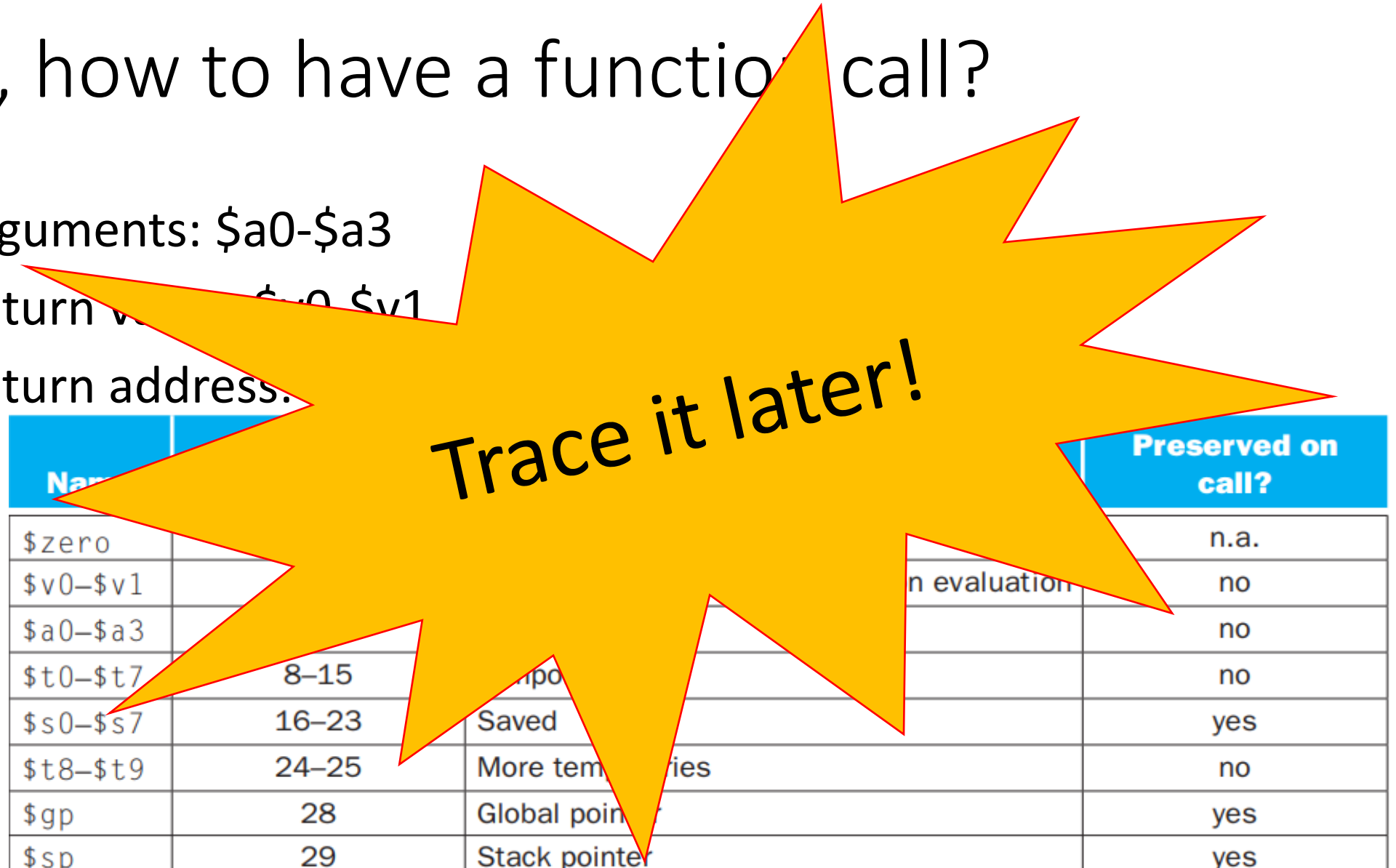
So, how to have a function call?

- Arguments: $a0-$a3
- Return value: $v0-$v1
- Return address: $ra

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

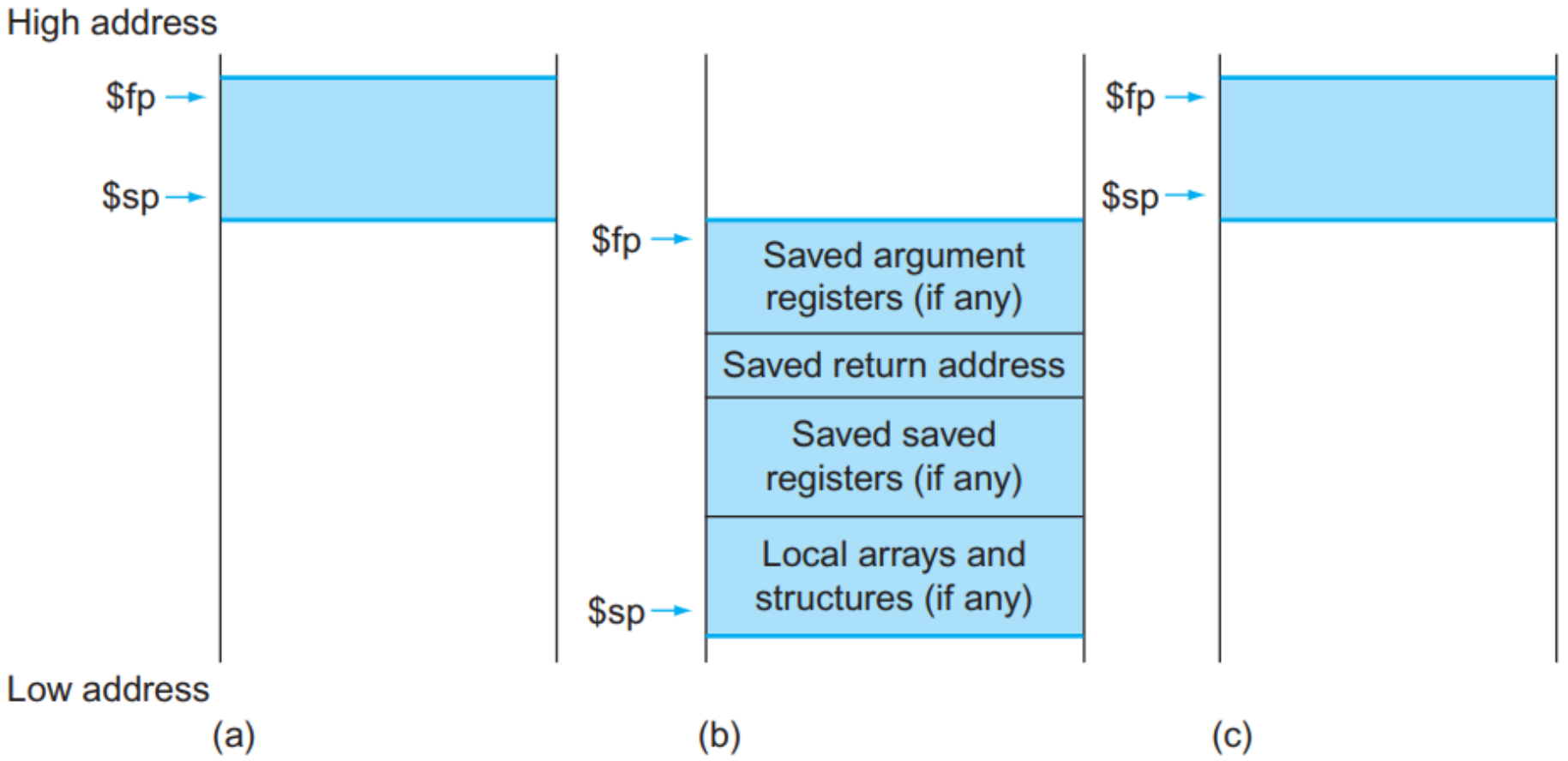
So, how to have a functio call?

- Arguments: $a0-$a3
- Return v $v0 $v1
- Return address:

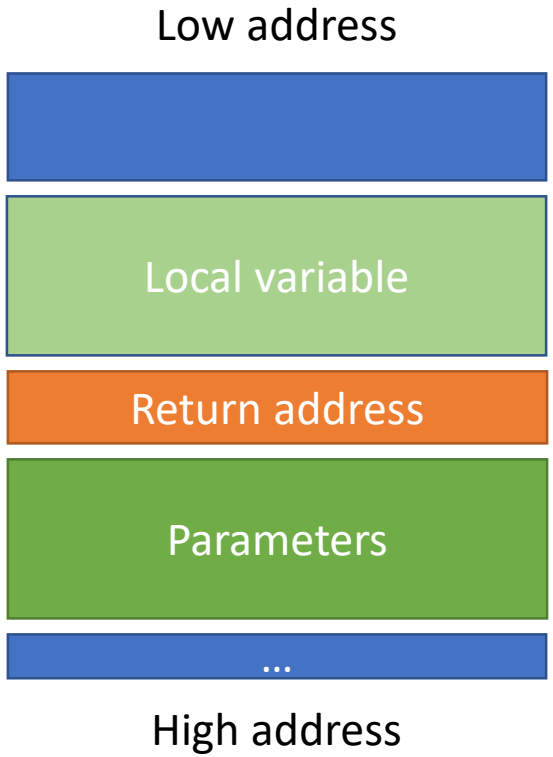


Nam			Preserved on call?
$zero			n.a.
$v0–$v1		n evaluation	no
$a0–$a3			no
$t0–$t7	8–15	po	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More tem ies	no
$gp	28	Global poin r	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Book used, but I'm not used to it.



I'm used to this.



I'm used to this.

```
int sum(int a, int b)
{
    int c = a + b;
    return c;
}
int main()
{
    int k = sum(1, 2);
    return 0;
}
```

sum:

```
addiu   $sp,$sp,-24
sw      $fp,20($sp)
move    $fp,$sp
sw      $4,24($fp)
sw      $5,28($fp)
movz    $31,$31,$0
lw      $3,24($fp)
lw      $2,28($fp)
nop
addu    $2,$3,$2
sw      $2,8($fp)
lw      $2,8($fp)
move    $sp,$fp
lw      $fp,20($sp)
addiu   $sp,$sp,24
j       $31
nop
```

main:

```
addiu   $sp,$sp,-40
sw      $31,36($sp)
sw      $fp,32($sp)
move    $fp,$sp
.cprestore      16
movz    $31,$31,$0
li      $5,2
li      $4,1
lw      $2,%got(sum)($28)
nop
move    $25,$2
```

…

I'm used to this.

```
int sum(int a, int b)
{
    int c = a + b;
    return c;
}
int main()
{
    int k = sum(1, 2);
    return 0;
}
```

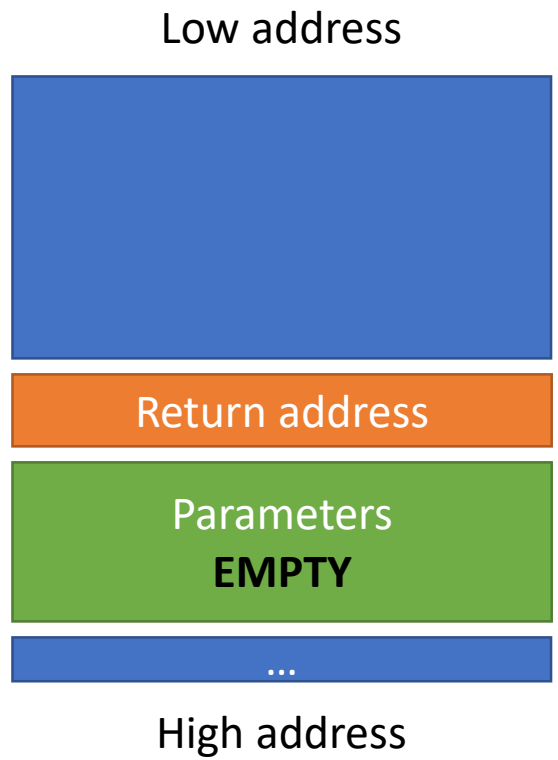
sum:

```
addiu   $sp,$sp,-24
sw      $fp,20($sp)
move    $fp,$sp
sw      $4,24($fp)
sw      $5,28($fp)
movz    $31,$31,$0
lw      $3,24($fp)
lw      $2,28($fp)
nop
addu    $2,$3,$2
sw      $2,8($fp)
lw      $2,8($fp)
move    $sp,$fp
lw      $fp,20($sp)
addiu   $sp,$sp,24
j       $31
nop
```

main:

```
addiu   $sp,$sp,-40
sw      $31,36($sp)
sw      $fp,32($sp)
move    $fp,$sp
.cprestore      16
movz    $31,$31,$0
li      $5,2
li      $4,1
lw      $2,%got(sum)($28)
nop
move    $25,$2
```

...



I'm used to this.

```
int sum(int a, int b)
{
    int c = a + b;
    return c;
}
int main()
{
    int k = sum(1, 2);
    return 0;
}
```

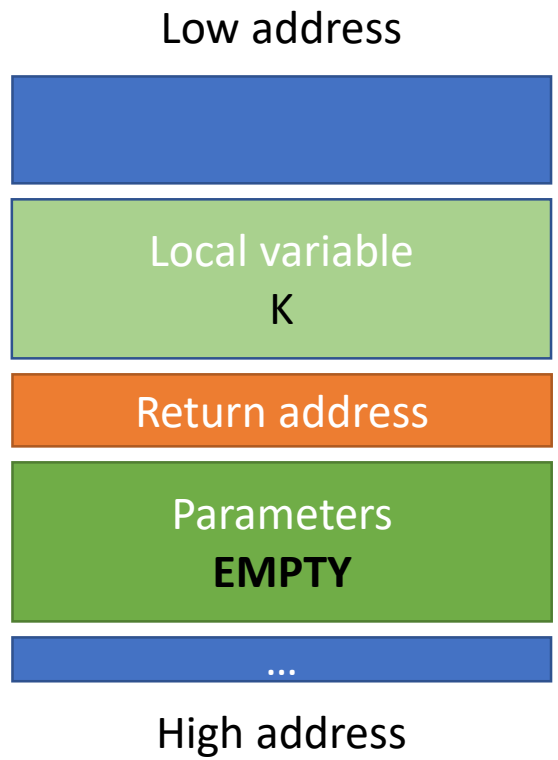
sum:

```
addiu   $sp,$sp,-24
sw      $fp,20($sp)
move    $fp,$sp
sw      $4,24($fp)
sw      $5,28($fp)
movz    $31,$31,$0
lw      $3,24($fp)
lw      $2,28($fp)
nop
addu    $2,$3,$2
sw      $2,8($fp)
lw      $2,8($fp)
move    $sp,$fp
lw      $fp,20($sp)
addiu   $sp,$sp,24
j       $31
nop
```

main:

```
addiu   $sp,$sp,-40
sw      $31,36($sp)
sw      $fp,32($sp)
move    $fp,$sp
.cprestore      16
movz    $31,$31,$0
li      $5,2
li      $4,1
lw      $2,%got(sum)($28)
nop
move    $25,$2
```

...



I'm used to this.

```
int sum(int a, int b)
{
    int c = a + b;
    return c;
}
int main()
{
    int k = sum(1, 2);
    return 0;
}
```

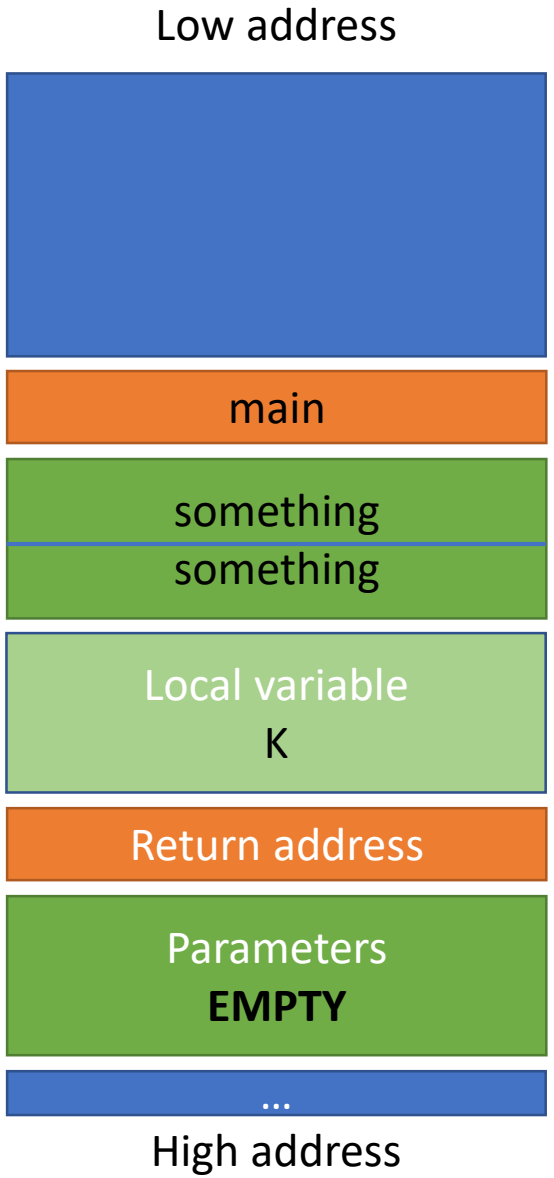
sum:

```
addiu   $sp,$sp,-24
sw      $fp,20($sp)
move    $fp,$sp
sw      $4,24($fp)
sw      $5,28($fp)
movz    $31,$31,$0
lw      $3,24($fp)
lw      $2,28($fp)
nop
addu    $2,$3,$2
sw      $2,8($fp)
lw      $2,8($fp)
move    $sp,$fp
lw      $fp,20($sp)
addiu   $sp,$sp,24
j       $31
nop
```

main:

```
addiu   $sp,$sp,-40
sw      $31,36($sp)
sw      $fp,32($sp)
move    $fp,$sp
.cprestore      16
movz    $31,$31,$0
li      $5,2
li      $4,1
lw      $2,%got(sum)($28)
nop
move    $25,$2
```

...



I'm used to this.

```
int sum(int a, int b)
{
    int c = a + b;
    return c;
}
int main()
{
    int k = sum(1, 2);
    return 0;
}
```

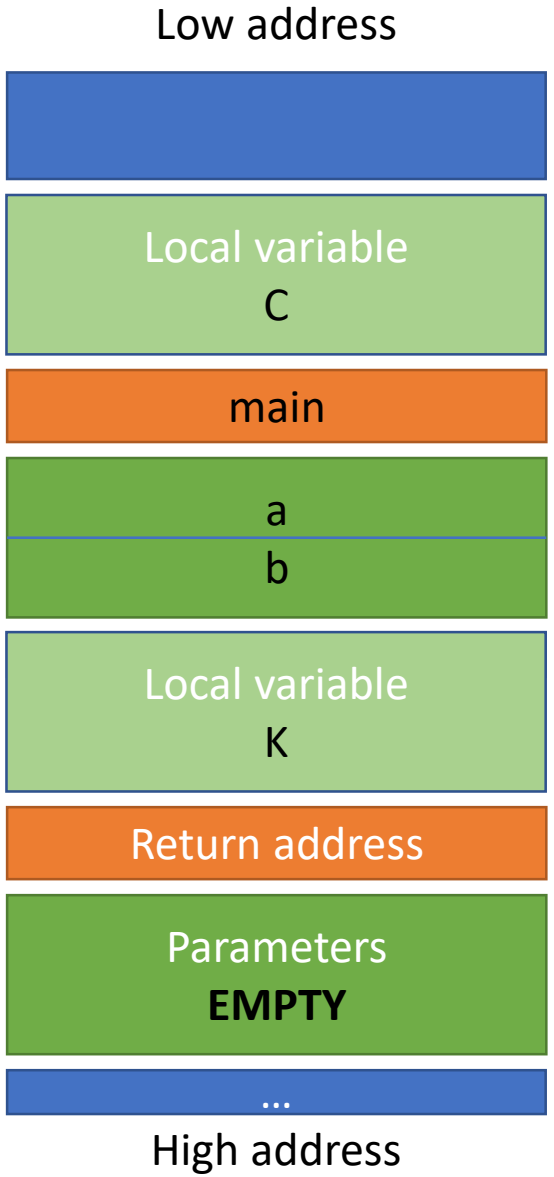
sum:

```
addiu   $sp,$sp,-24
sw      $fp,20($sp)
move    $fp,$sp
sw      $4,24($fp)
sw      $5,28($fp)
movz    $31,$31,$0
lw      $3,24($fp)
lw      $2,28($fp)
nop
addu    $2,$3,$2
sw      $2,8($fp)
lw      $2,8($fp)
move    $sp,$fp
lw      $fp,20($sp)
addiu   $sp,$sp,24
j       $31
nop
```

main:

```
addiu   $sp,$sp,-40
sw      $31,36($sp)
sw      $fp,32($sp)
move    $fp,$sp
.cprestore      16
movz    $31,$31,$0
li      $5,2
li      $4,1
lw      $2,%got(sum)($28)
nop
move    $25,$2
```

...



I'm used to this.

```
int sum(int a, int b)
{
    int c = a + b;
    return c;
}
int main()
{
    int k = sum(1, 2);
    return 0;
}
```

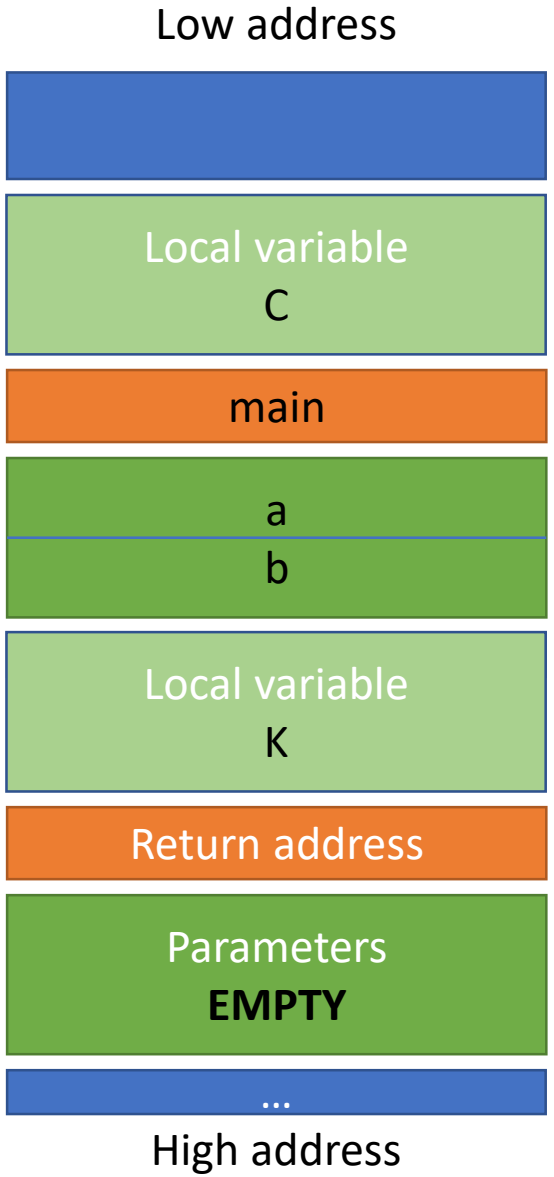
sum:

```
addiu   $sp,$sp,-24
sw      $fp,20($sp)
move    $fp,$sp
sw      $4,24($fp)
sw      $5,28($fp)
movz    $31,$31,$0
lw      $3,24($fp)
lw      $2,28($fp)
nop
addu    $2,$3,$2
sw      $2,8($fp)
lw      $2,8($fp)
move    $sp,$fp
lw      $fp,20($sp)
addiu   $sp,$sp,24
j       $31
nop
```

main:

```
addiu   $sp,$sp,-40
sw      $31,36($sp)
sw      $fp,32($sp)
move    $fp,$sp
.cprestore      16
movz    $31,$31,$0
li      $5,2
li      $4,1
lw      $2,%got(sum)($28)
nop
move    $25,$2
```

...



I'm used to this.

```
int sum(int a, int b)
{
    int c = a + b;
    return c;
}
int main()
{
    int k = sum(1, 2);
    return 0;
}
```

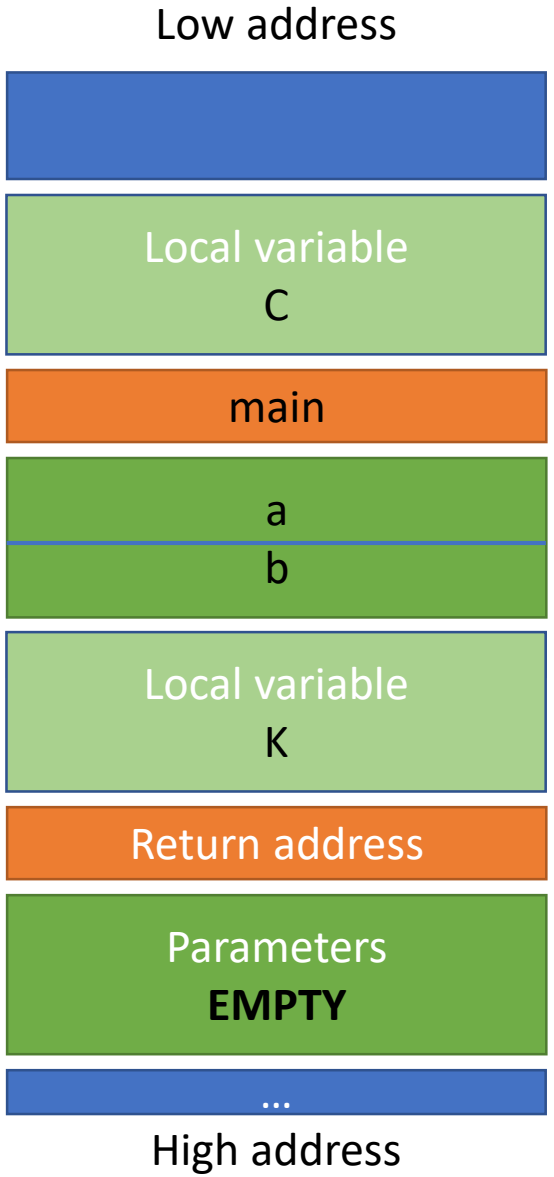
sum:

```
addiu   $sp,$sp,-24
sw      $fp,20($sp)
move    $fp,$sp
sw      $4,24($fp)
sw      $5,28($fp)
movz    $31,$31,$0
lw      $3,24($fp)
lw      $2,28($fp)
nop
addu    $2,$3,$2
sw      $2,8($fp)
lw      $2,8($fp)
move    $sp,$fp
lw      $fp,20($sp)
addiu   $sp,$sp,24
j       $31
nop
```

main:

```
addiu   $sp,$sp,-40
sw      $31,36($sp)
sw      $fp,32($sp)
move    $fp,$sp
.cprestore      16
movz    $31,$31,$0
li      $5,2
li      $4,1
lw      $2,%got(sum)($28)
nop
move    $25,$2
```

...



I'm used to this.

```
int sum(int a, int b)
{
    int c = a + b;
    return c;
}
int main()
{
    int k = sum(1, 2);
    return 0;
}
```

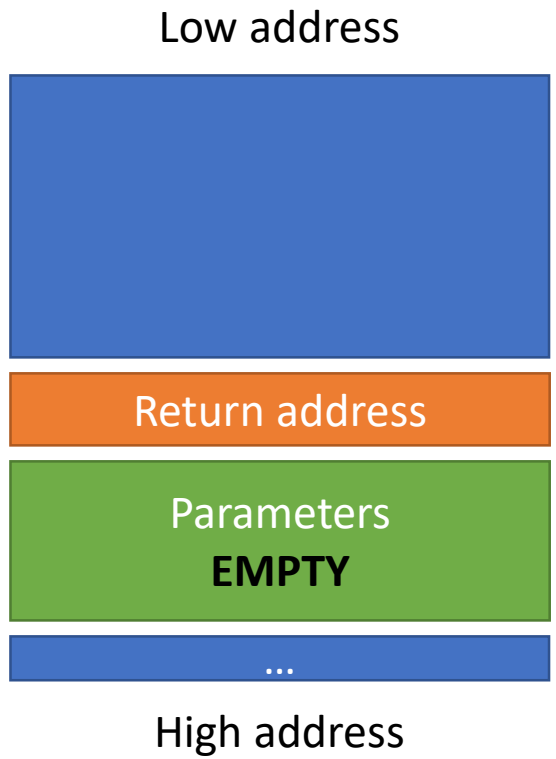
sum:

```
addiu   $sp,$sp,-24
sw      $fp,20($sp)
move    $fp,$sp
sw      $4,24($fp)
sw      $5,28($fp)
movz    $31,$31,$0
lw      $3,24($fp)
lw      $2,28($fp)
nop
addu    $2,$3,$2
sw      $2,8($fp)
lw      $2,8($fp)
move    $sp,$fp
lw      $fp,20($sp)
addiu   $sp,$sp,24
j       $31
nop
```

main:

```
addiu   $sp,$sp,-40
sw      $31,36($sp)
sw      $fp,32($sp)
move    $fp,$sp
.cprestore      16
movz    $31,$31,$0
li      $5,2
li      $4,1
lw      $2,%got(sum)($28)
nop
move    $25,$2
```

...



“Talk is cheap. Show me the code.”

- Linus Torvalds

https://github.com/torvalds

C to MIPS compiler

- http://reliant.colab.duke.edu/c2mips/

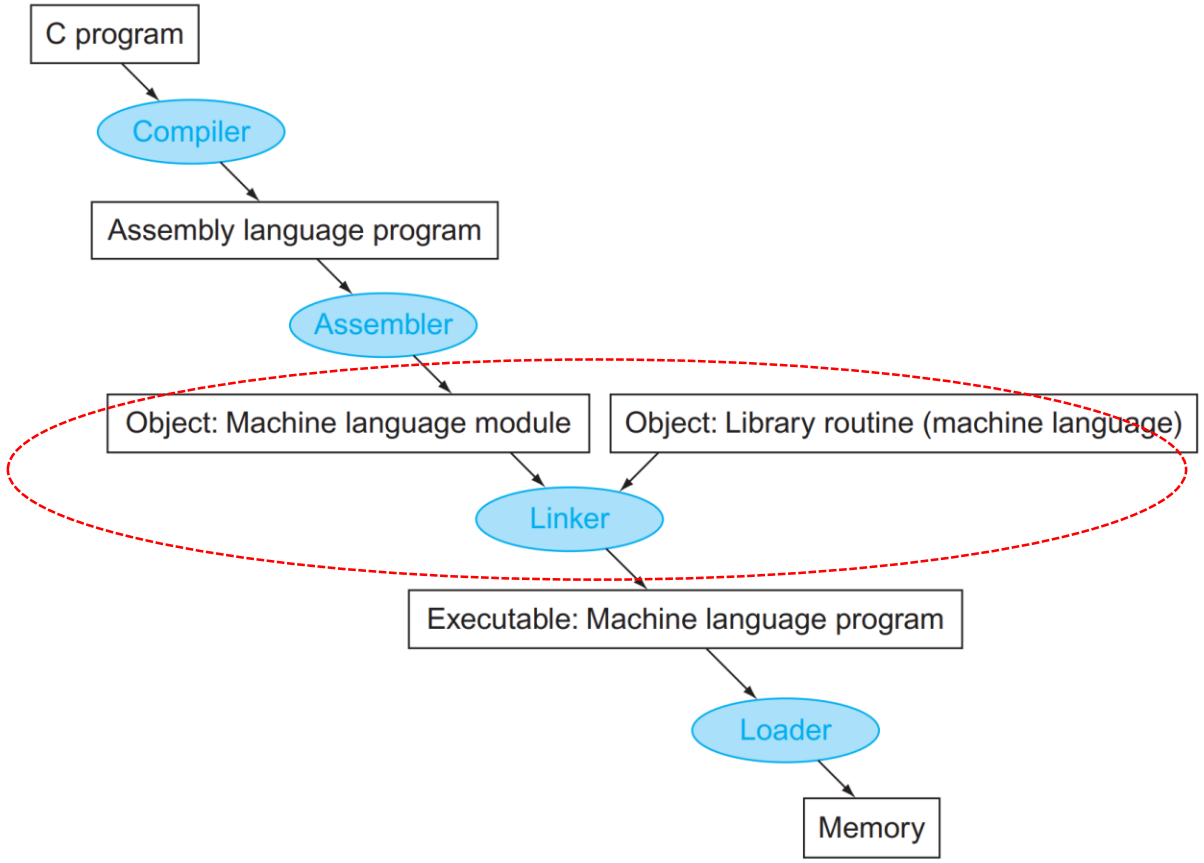
Parallelism and instrcutions : synchronization

- Store condition(Check yourself):
  - When cooperating threads of a parallel program need to synchronize to get proper behavior for reading and writing shared data.
  - When cooperating processes on a uniprocessor need to synchronize to get proper behavior for reading and writing shared data.

Demo

- pthread: using strace to see symbol
- Fork: just in stdout

GCC tool chain



Linker

- Dynamic linking(Lazy Binding)
- Static linking(Immediate Binding)
- Let’s trace!
  - File size
  - Speed test
  - $ objdump -S

Sort

swap:

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

sort

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

But wait!!!

- The book said…

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (procedure integration)	2.41	65,747	44,993	1.46

FIGURE 2.28 Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort. The programs sorted 100,000 words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20.

But wait!!!

- The book said…

gcc optimization	Relative performance	Clock cycles (millions)	Instru (r
None	1.00	158,615	
01 (medium)	2.37	66,990	
02 (full)	2.38	66,521	
03 (procedure integration)	2.41	65,747	

FIGURE 2.28 Comparing performance, instruction count, and optimization for Bubble Sort. The programs sorted 100,000 words with the values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20.

Pentium 4

採用LGA 775插座的Pentium 4

產品化	從 2000 至 2006
生產商	Intel
微架構	NetBurst
指令集架構	x86（i386）,x86-64, SSE, SSE2, SSE3, 虛擬化技術
製作製程/製程	180nm 至 65nm
CPU主頻範圍	1.3 GHz 至 3.8 GHz
前端匯流排速率	400 MHz 至 1066 MHz
CPU插座	Socket 423 Socket 478 LGA 775
核心代號	Willamette Northwood Prescott（單核心）, Smithfield（雙核心） Cedar Mill（單核心）, Presler（雙核心）

But wait!!!

- The book said…

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (procedure integration)	2.41	65,747	44,993	1.46

FIGURE 2.28 Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort. The programs sorted 100,000 words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20.

Linux scheduler

- O(N) scheduler: circular queue
- O(1) scheduler: array
- O(logN) scheduler: CFS
- Reference

Knowing your kernel feature is important!

Linked list merge sort (trace)

- Demo

Other instructions

- ARM v7
- x86
- ARM v8
- Trace too!

Other instructions

- ARM v7
- x86
- ARM v8
- Trace too!

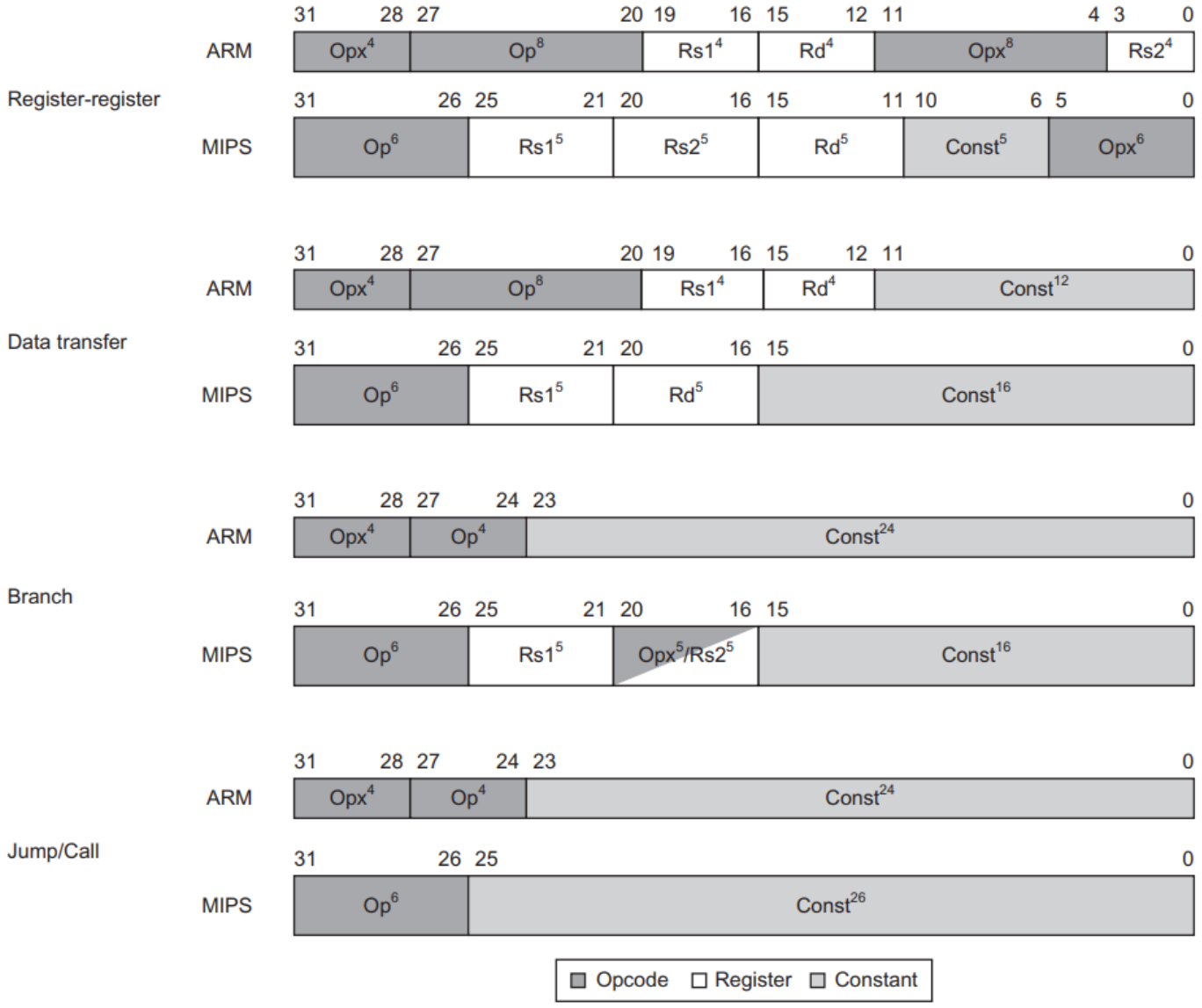


FIGURE 2.34 Instruction formats, ARM and MIPS. The differences result from whether the architecture has 16 or 32 registers.

Wait! What is Opx?

4.2 The Condition Field

In ARM state, all instructions are conditionally executed according to the state of the CPSR condition codes and the instruction's condition field. This field (bits 31:28) determines the circumstances under which an instruction is to be executed. If the state of the C, N, Z and V flags fulfils the conditions encoded by the field, the instruction is executed, otherwise it is ignored.

There are sixteen possible conditions, each represented by a two-character suffix that can be appended to the instruction's mnemonic. For example, a Branch (`B` in assembly language) becomes `BEQ` for "Branch if Equal", which means the Branch will only be taken if the Z flag is set.

In practice, fifteen different conditions may be used: these are listed in ***Table 4-2: Condition code summary***. The sixteenth (1111) is reserved, and must not be used.

In the absence of a suffix, the condition field of most instructions is set to "Always" (sufix AL). This means the instruction will always be executed regardless of the CPSR condition codes.

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Table 4-2: Condition code summary

https://iitd-plos.github.io/col718/ref/arm-instructionset.pdf

Other instructions

- ARM v7
- x86
- ARM v8
- Trace too!

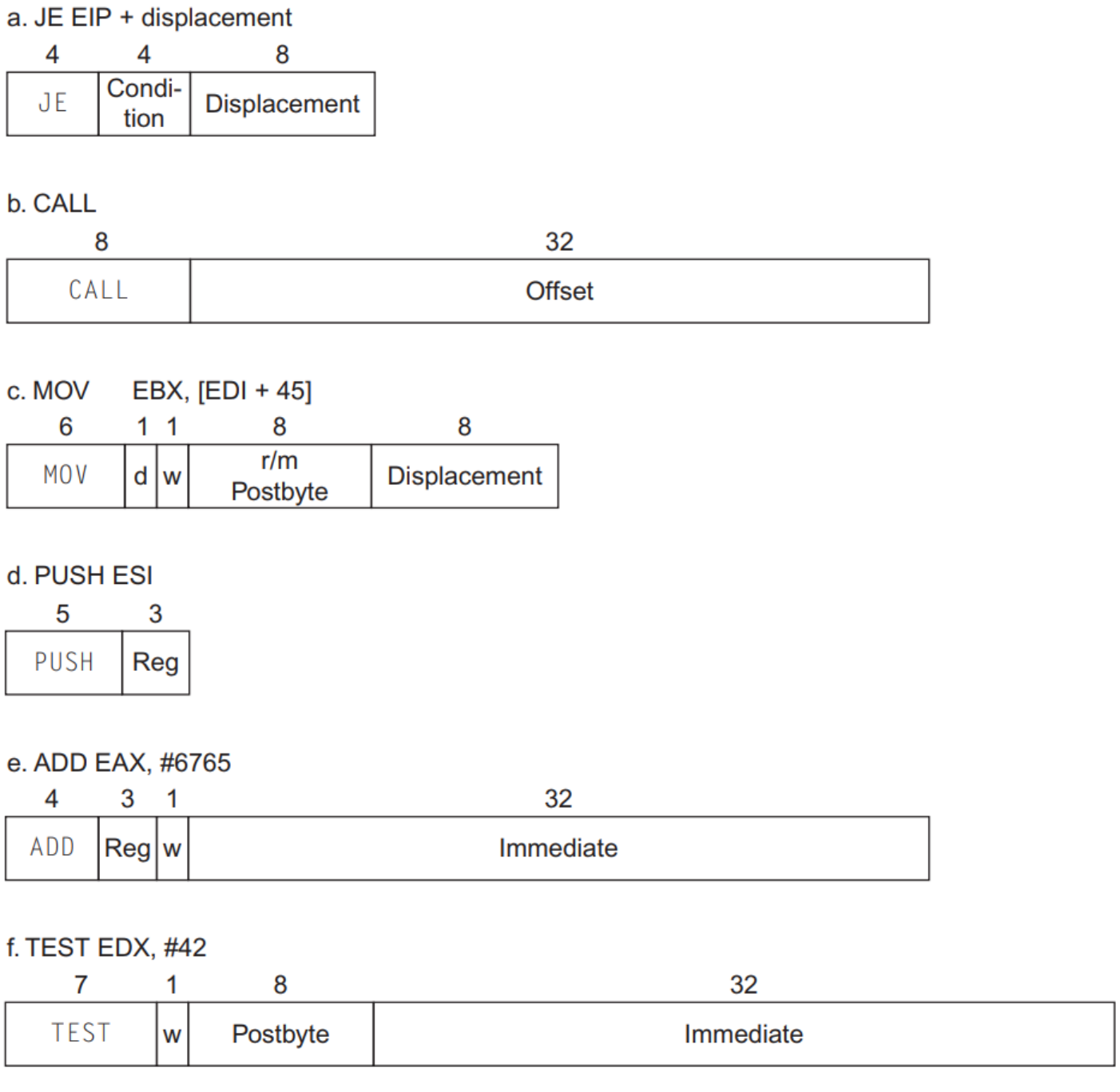


FIGURE 2.41 Typical x86 instruction formats. Figure 2.42 shows the encoding of the postbyte.

Other instructions

- ARM v7
- x86
- ARM v8
- Trace too!

As the philosophy of the v8 instruction set is much closer to MIPS than it is to v7, our conclusion is that the main similarity between ARMv7 and ARMv8 is the name.

Other instructions

- ARM v7
- x86
- ARM v8
- Last Trace!

EOF