Instruction Set Architecture

Hung-Wei Tseng

Setup your i-clicker

- Register your i-clicker through TritonEd
- Set your channel to “CA”
    - Press on/off button for 2 seconds
    - Press C and then press A

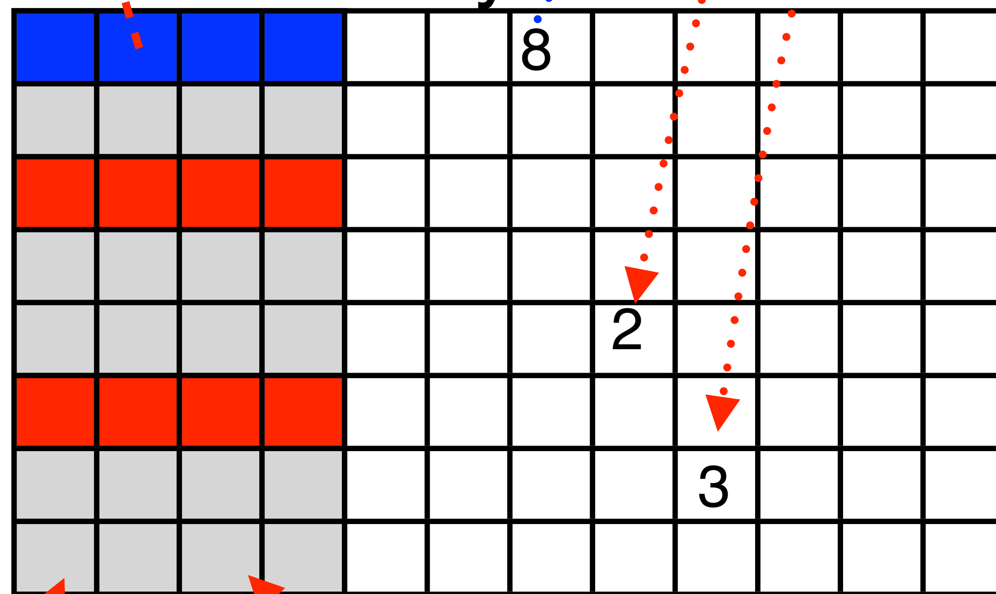
Outline

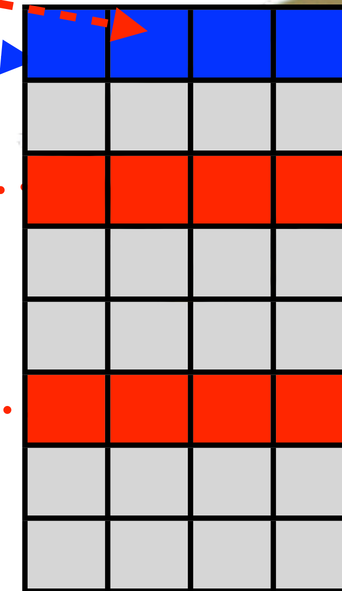
- How we talk to computers
- What is an ISA (instruction set architecture)
- MIPS ISA

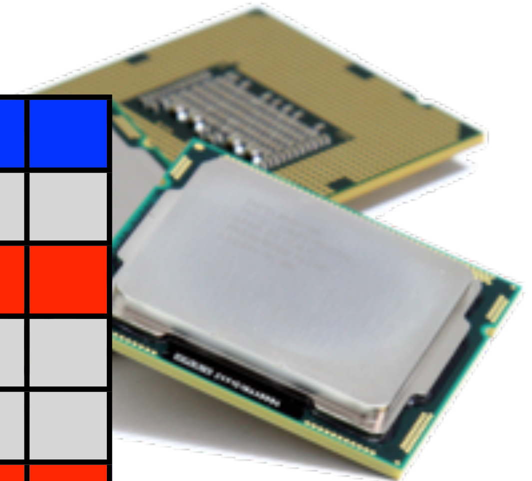
How we talk to computers

Von Neumann architecture

memory



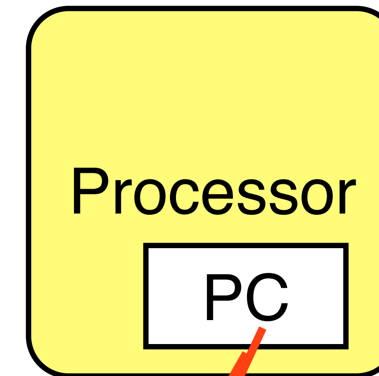




What are in these colored boxes?

The stored program computer

- The program is data
  - a series of bits
    - these bits are “instructions”!
  - lives in memory
- Program counter
  - points to the current instruction
  - processor “fetches” instructions from where PC points.
  - advances/changes after instruction execution



instruction memory

```
120007a30:  0f00bb27  ldah   gp,15(t12)
120007a34:  509cbd23  lda    gp,-25520(gp)
120007a38:  00005d24  ldah   t1,0(gp)
120007a3c:  0000bd24  ldah   t4,0(gp)
120007a40:  2ca422a0  ldl    t0,-23508(t1)
120007a44:  130020e4  beq    t0,120007a94
120007a48:  00003d24  ldah   t0,0(gp)
120007a4c:  2ca4e2b3  stl    zero,-23508(t1)
120007a50:  0004ff47  clr    v0
120007a54:  28a4e5b3  stl    zero,-23512(t4)
120007a58:  20a421a4  ldq    t0,-23520(t0)
120007a5c:  0e0020e4  beq    t0,120007a98
120007a60:  0204e147  mov    t0,t1
120007a64:  0304ff47  clr    t2
120007a68:  0500e0c3  br     120007a80
```

From C/C++ to Machine Code

one time cost

compiler frontend
(e.g. gcc/llvm)

Intermediate Representation

compiler backend
assembler/optimizer

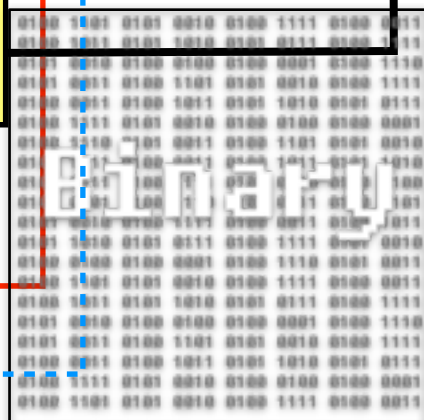
machine code/binary

Object

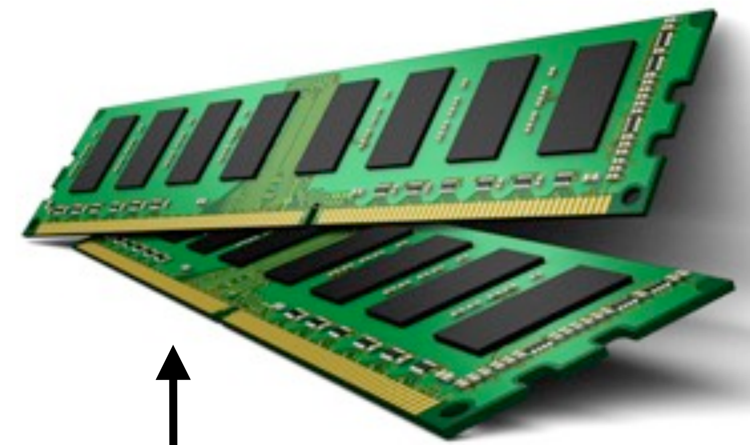
linker (e.g. ld)

Library

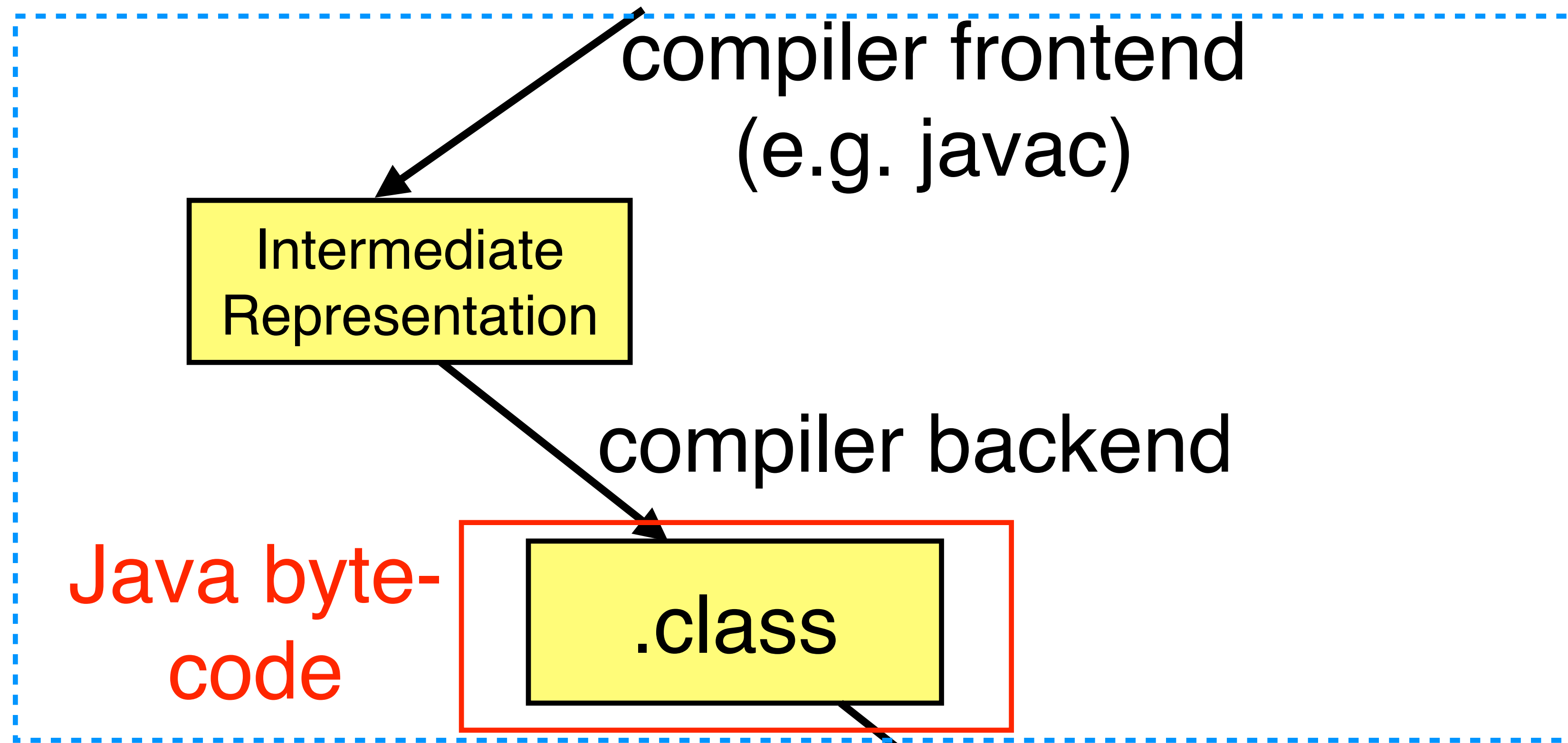
Executable

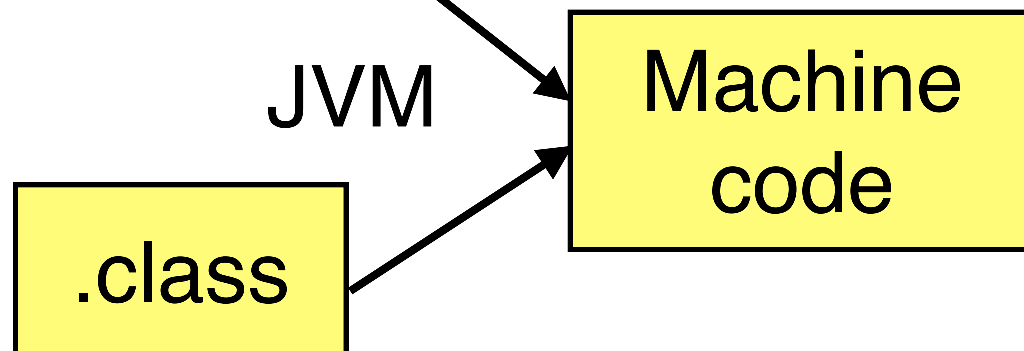


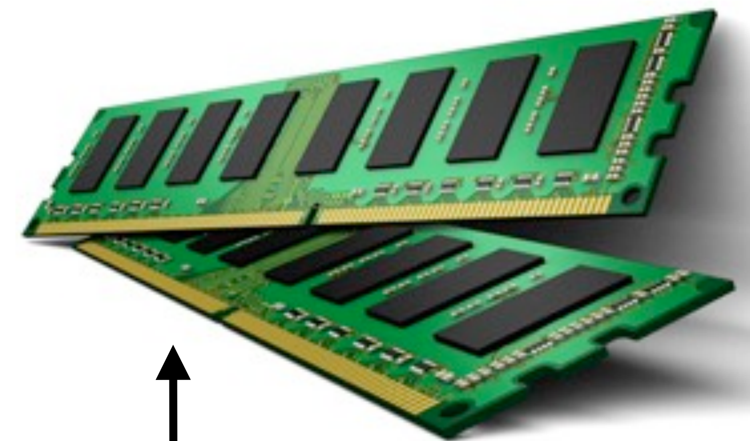
OS loader



From Java to Machine Code







From Script Languages to Machine Code

interpreter (python, perl)

Intermediate Representation

compiler

binary

compiler

binary

executable

machine code

runtime

What’s an Instruction Set Architecture (ISA)?

Instruction Set Architecture (ISA)

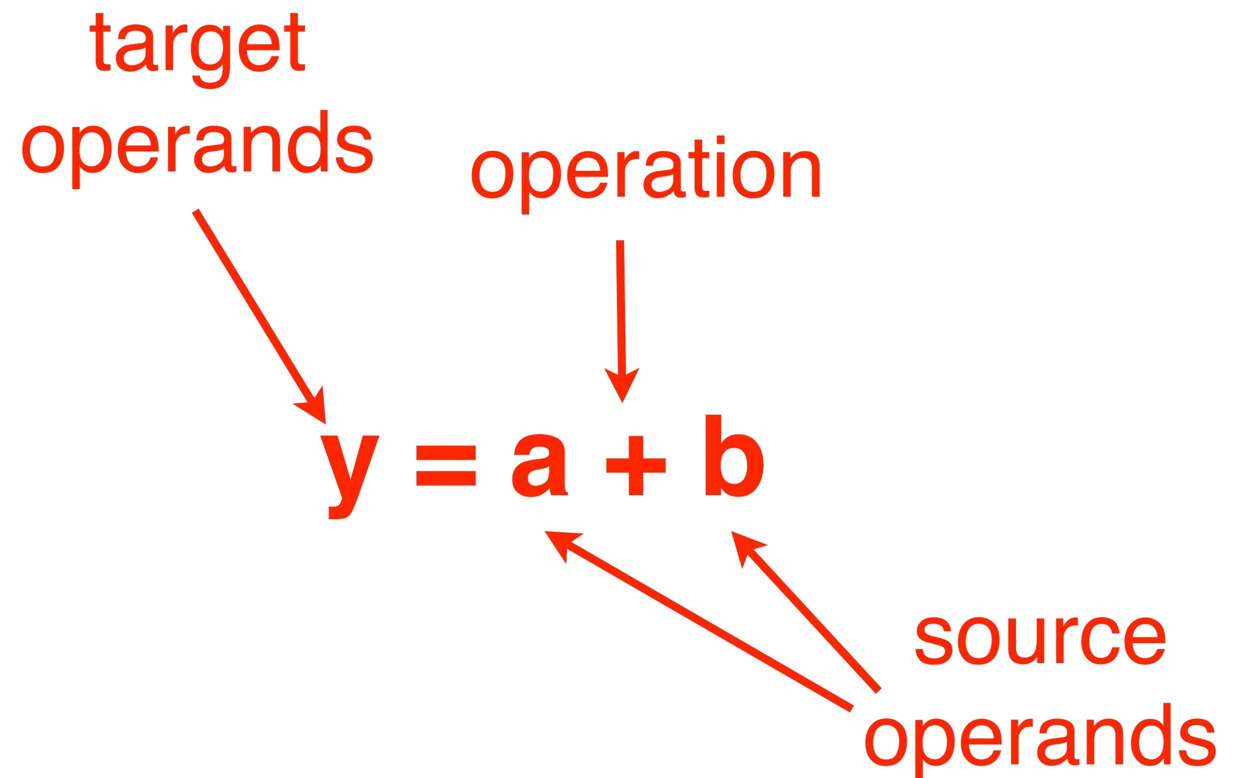
- The contract between the hardware and software
- Defines the set of operations that a computer/processor can execute
- Programs are combinations of these instructions
  - Abstraction to programmers/compilers
- The hardware implements these instructions in any way it choose.
  - Directly in hardware circuit. e.g. CPU
  - Software virtual machine. e.g. VirtualPC
  - Simulator/Emulator. e.g. DeSmuME
  - Trained monkey with pen and paper

Example ISAs

- x86: intel Xeon, intel Core i7/i5/i3, intel atom, AMD Athlon/Opteron, AMD FX, AMD A-series
- ARM: Apple A-Series, Qualcomm Snapdragon, TI OMAP, nVidia Tegra
- MIPS: Sony/Toshiba Emotion Engine, MIPS R-4000(PSP)
- DEC Alpha: 21064, 21164, 21264
- PowerPC: Motorola PowerPC G4, Power 6
- IA-64: Itanium
- SPARC and many more ...

What should an instruction look like?

- Operations
  - What operations?
    e.g. add, sub, mul, and etc.
  - How many operations?
- Operands
  - How many operand?
  - What type of operands?
    - Memory/register/label/number(immediate value)
- Format
  - Length? How many bits? Equal length?
  - Formats?



add r1, r2, r3
add r1, r2, 64

What ISA includes?

- Instructions: what programmers want processors to do?
  - Math: add, subtract, multiply, divide, bitwise operations
  - Control: if, jump, function call
  - Data access: load and store
- Architectural states: the current execution result of a program
  - Registers: a few named data storage that instructions can work on
  - Memory: a much larger data storage array that is available for storing data
  - Program Counter (PC): the number/address of the current instruction

We will study two ISAs

- MIPS
  - Simple, elegant, easy to implement
    - That's why we want to implement it in CSE141L
  - Designed with many-year ISA design experience
  - The prototype of a lot of modern ISAs
    - MIPS itself is not widely used, though
- x86
  - Ugly, messy, inelegant, hard to implement, ...
  - Designed for 1970s technology
  - The dominant ISA in modern computer systems

You should know how to **write** MIPS code after this class

You should know how to **read** x86 code after this class

MIPS

MIPS ISA

- All instructions are 32 bits
- 32 32-bit registers
  - All registers are the same
  - $zero is always 0
- 50 opcodes
  - Arithmetic/Logic operations
  - Load/store operations
  - Branch/jump operations
- 3 instruction formats
  - R-type: all operands are registers
  - I-type: one of the operands is an immediate value
  - J-type: non-conditional, non-relative branches

name	number	usage	saved?
$zero	0	zero	N/A
$at	1	assembler temporary	no
$v0-$v1	2-3	return value	no
$a0-$a3	4-7	arguments	no
$t0-$t7	8-15	temporaries	no
$s0-$s7	16-23	saved	yes
$t8-$t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

MIPS ISA (cont.)

- Only load and store instructions can access memory
- Memory is "byte addressable"
  - Most modern ISAs are byte addressable, too
  - byte, half words, words are aligned

Byte addresses

Address	Data
0x0000	0xAA
0x0001	0x15
0x0002	0x13
0x0003	0xFF
0x0004	0x76
...	.
0xFFFE	.
0xFFFF	.

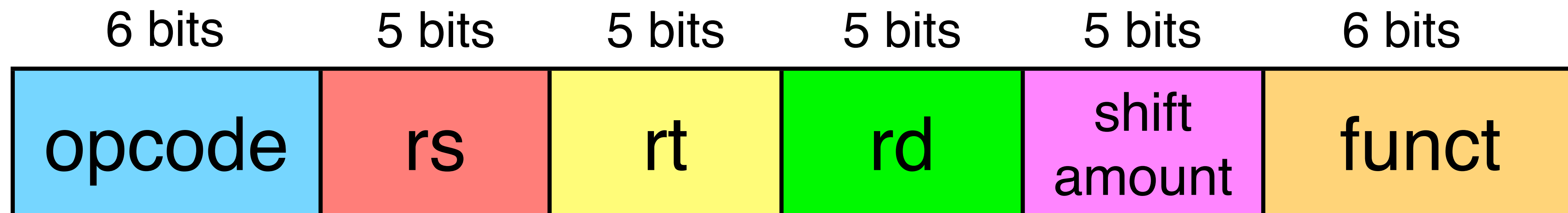
Half Word Addresses

Address	Data
0x0000	0xAA15
0x0002	0x13FF
0x0004	.
0x0006	.
...	.
...	.
...	.
0xFFFC	.

Word Addresses

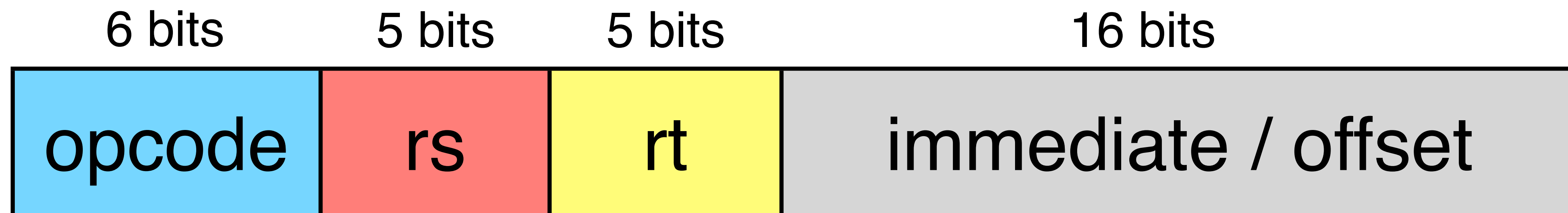
Address	Data
0x0000	0xAA1513FF
0x0004	.
0x0008	.
0x000C	.
...	.
...	.
...	.
0xFFFC	.

R-type



- op $rd, $rs, $rt
  - 3 regs.: add, addu, and, nor, or, sltu, sub, subu
  - 2 regs.:sll, srl
  - 1 reg.: jr
- 1 arithmetic operation, 1 I-memory access
- Example:
  - `add $v0, $a1, $a2: R[2] = R[5] + R[6]`
    opcode = 0x0, funct = 0x20
  - `sll $t0, $t1, 8: R[8] = R[9] << 8`
    opcode = 0x0, shamt = 0x8, funct = 0x0

I-type



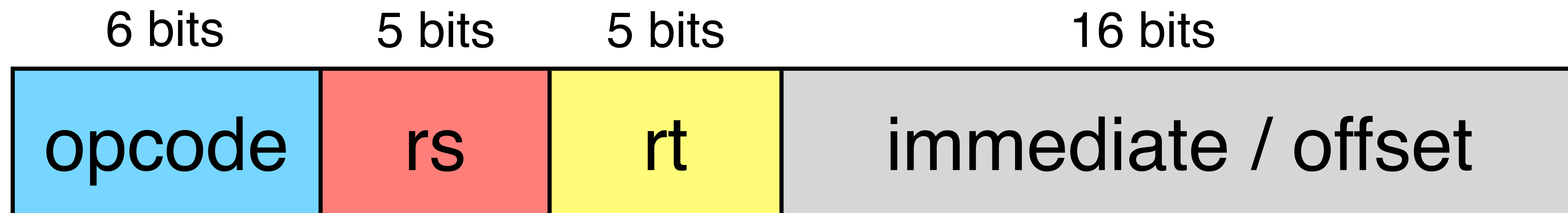
- op $rt, $rs, immediate
  - addi, addiu, andi, beq, bne, ori, slti, sltiu
- op $rt, offset($rs)
  - lw, lbu, lhu, ll, lui, sw, sb, sc, sh
- 1 arithmetic op, 1 I-memory and 1 D-memory access
- Example:
  - `lw $s0, 4($s2):`
    `R[16] = mem[R[18]+4]`

only two addressing modes

```
lw $s0, $s2($s1)
```

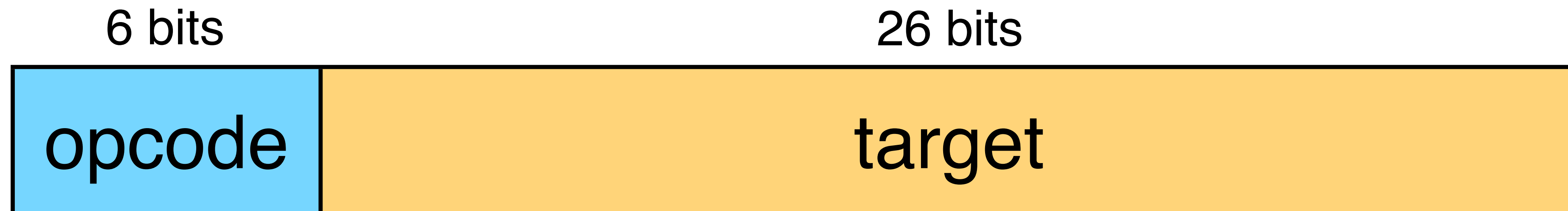
```
add $s2, $s2, $s1
lw  $s0, 0($s2)
```

I-type (cont.)



- op $rt, $rs, immediate
  - addi, addiu, andi, beq, bne, ori, slti, sltiu
- op $rt, offset($rs)
  - lw, lbu, lhu, ll, lui, sw, sb, sc, sh
- 1 arithmetic op, 1 I-memory and 1 D-memory access
- Example:
  - ```
    beq $t0, $t1, -40
    if (R[8] == R[9]) PC = PC + 4 + 4*(-40)
    ```
    ```

J-type



- op immediate
  - j, jal
- 1 instruction memory access, 1 arithmetic op
- Example:
  - ```
    jal quicksort
    R[31] = PC + 4
    PC = quicksort
    ```
    ```

Practice

- Translate the C code into assembly:

label

```
for(i = 0; i < 100; i++)
{
  sum+=A[i];
}
```

```
       and  $t0, $t0, $zero #let i = 0
       addi $t1, $zero, 100 #temp = 100
LOOP:  lw   $t3, 0($s0)     #temp1 = A[i]
       add  $v0, $v0, $t3   #sum += temp1
       addi $s0, $s0, 4     #addr of A[i+1]
       addi $t0, $t0, 1     #i = i+1
       bne  $t1, $t0, LOOP  #if i < 100
```

1. Initialization
2. Load A[i] from memory to register
3. Add the value of A[i] to sum
4. Increase by 1
5. Check if i still < 100

Assume
int is 32 bits
$s0 = &A[0]
$v0 = sum;
$t0 = i;

There are many ways to translate the C code.
But efficiency may be differ among translations

Tower of Hanoi

```
int hanoi(int n)
{
  if(n==1)
    return 1;
  else
    return 2*hanoi(n-1)+1;
}

int main(int argc, char **argv)
{
  int n, result;
  n = atoi(argv[0]);
  result = hanoi(n);
  printf("%d\n", result);
}
```

Recursive Function call

Function call

Function calls

- Passing arguments
  - $a0-$a3
  - more to go using the memory stack
- Invoking the function
  - jal <label>
  - store the PC of jal +4 in $ra
- Return value in $v0
- Return to caller
  - jr $ra

Let's write the hanoi()

```
int hanoi(int n)
{
  if(n==1)
    return 1;
  else
    return 2*hanoi(n-1)+1;
}
```

```
hanoi:    addi $a0, $a0, -1          // n = n-1
          bne  $a0, $zero, hanoi_1 // if(n == 0) goto: hanoi_1
          addi $v0, $zero, 1         // return_value = 0 + 1 = 1
          j    return                // return
hanoi_1: jal  hanoi                  // call honai
          sll  $v0, $v0, 1           // return_value=return_value*2
          addi $v0, $v0, 1           // return_value = return_value+1
return:   jr   $ra                   // return to caller
```

Function calls

registers

zero	
at	
v0	1
v1	
a0	0
a1	
a2	
a3	
t0	0
t1	2
⋮	
ra	hanoi_1+4

Caller (main)

Callee (hanoi)

Prepare argument for hanoi
$a0 - $a3 for passing arguments

Point to PC1+4

```
     addi $a0, $t1, $t0
PC1: jal  hanoi
     sll  $v0, $v0, 1
     addi $v0, $v0, 1
     add  $t0, $zero, $a0
     li   $v0, 4
     syscall
```

```
hanoi:   addi $a0, $a0, -1
         bne  $a0, $zero, hanoi_1
         addi $v0, $zero, 1
         j    return
hanoi_1: jal  hanoi
         sll  $v0, $v0, 1
         addi $v0, $v0, 1
return:  jr   $ra
```

```
hanoi:   addi $a0, $a0, -1
         bne  $a0, $zero, hanoi_1
         addi $v0, $zero, 1
         j    return
hanoi_1: jal  hanoi
         sll  $v0, $v0, 1
         addi $v0, $v0, 1
return:  jr   $ra
```

Overwrite!
$a0 != $t1+$t0

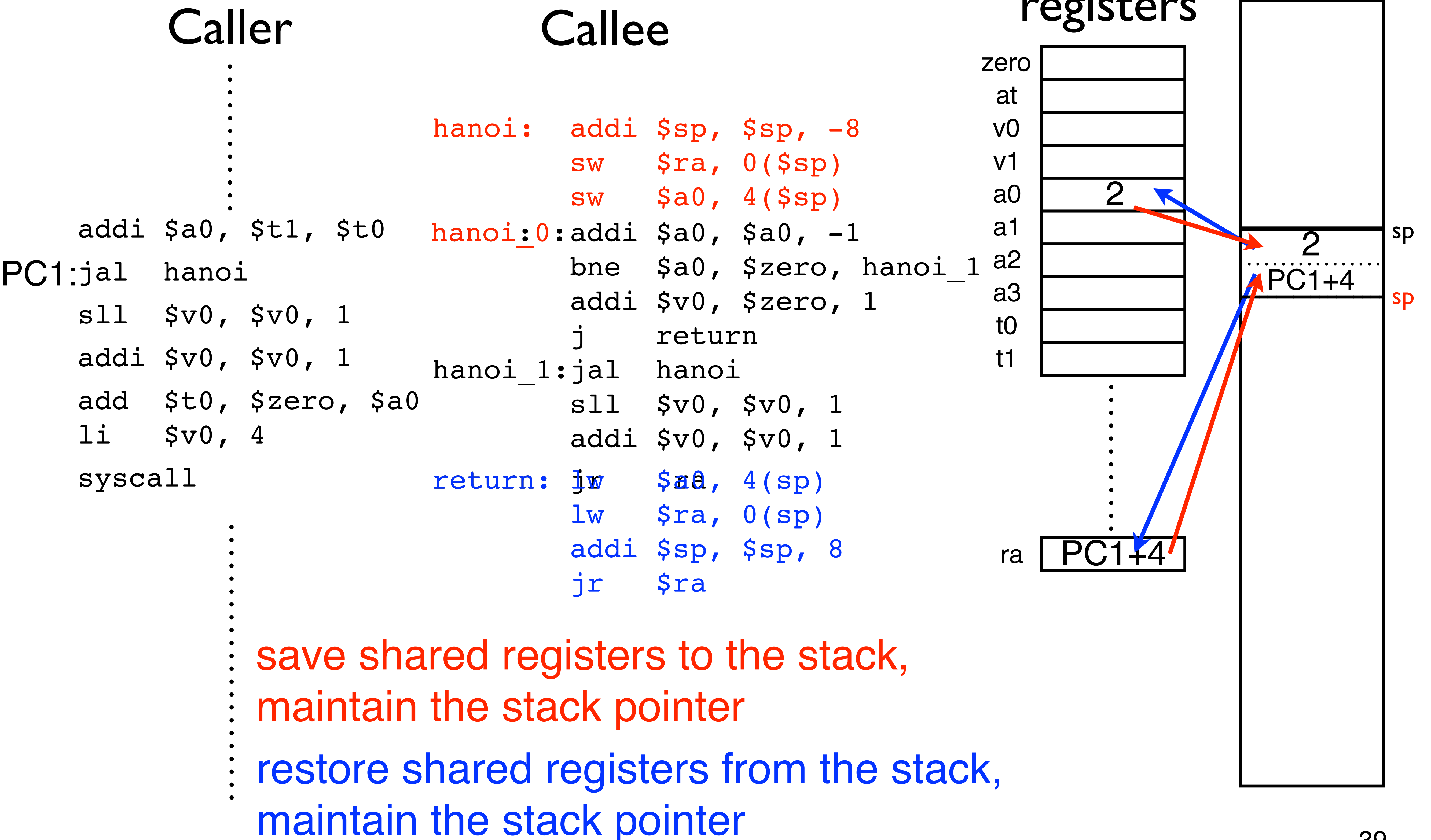
Where are we going now?
We are supposed to go to PC1+4 not hanoi_1+4!

the current location of PC

Manage registers

- Sharing registers
  - A called function will modified registers
  - The caller may use these values later
- Using memory stack
  - The stack provides local storage for function calls
  - FILO (first-in-last-out)
  - For historical reasons, the stack grows from high memory address to low memory address
  - The stack pointer ($sp) should point to the top of the stack

Function calls



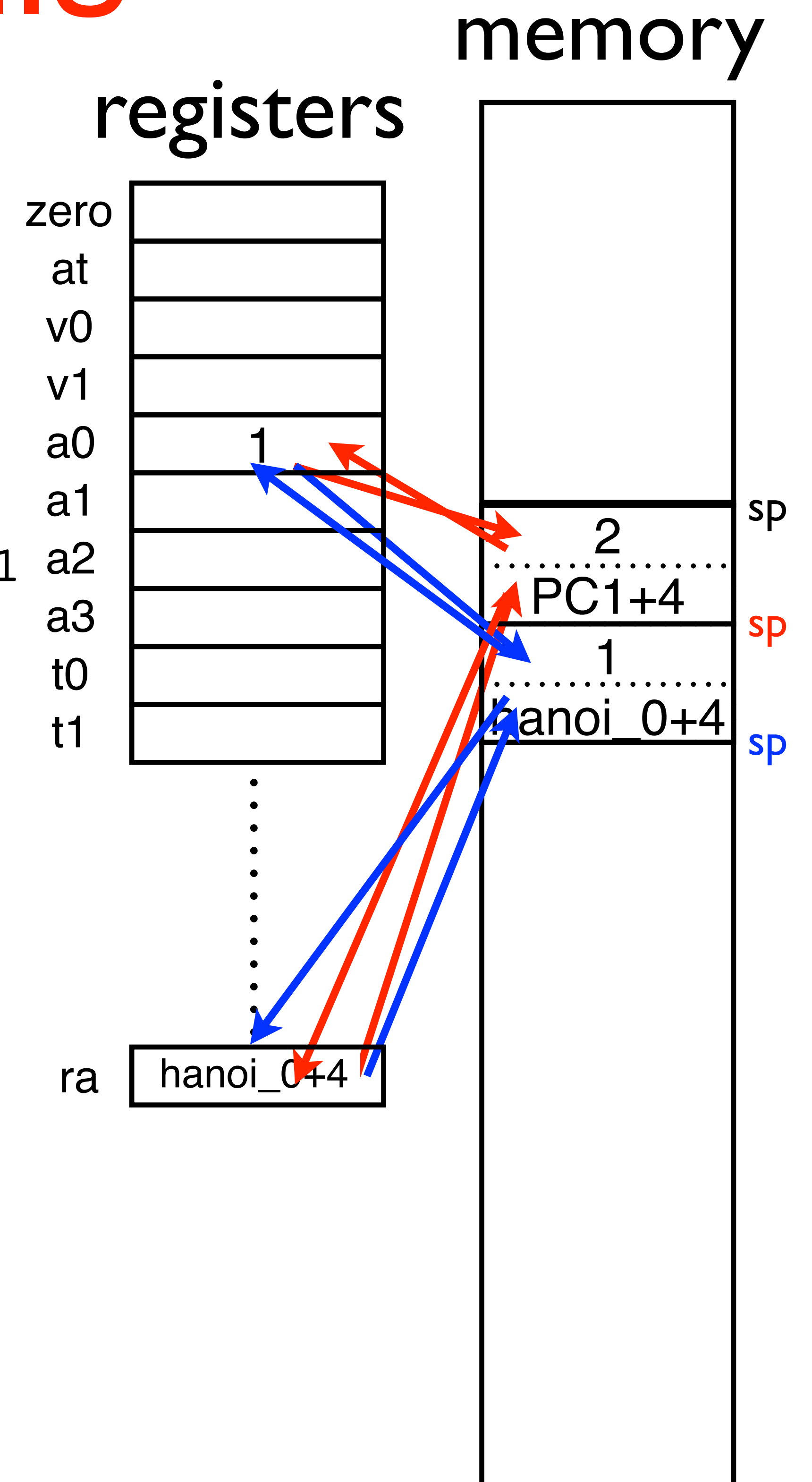
Recursive calls

Caller

```
addi $a0, $zero, 2
addi $a0, $t1, $t0
PC1:jal   hanoi
sll  $v0, $v0, 1
addi $v0, $v0, 1
add  $t0, $zero, $a0
li   $v0, 4
syscall
```

Callee

```
hanoi:   addi $sp, $sp, -8
         sw   $ra, 0($sp)
         sw   $a0, 4($sp)
hanoi_0:addi $a0, $a0, -1
         bne  $a0, $zero, hanoi_1
         addi $v0, $zero, 1
         j    return
hanoi_1:jal   hanoi
         sll  $v0, $v0, 1
         addi $v0, $v0, 1
return:  lw   $a0, 4(sp)
         lw   $ra, 0(sp)
         addi $sp, $sp, 8
         jr   $ra
```



Demo

- The overhead of function calls
- The keyword `inline` in C can embed the callee code at the call site
  - Eliminates function call overhead
- Does not work if it’s called using a function pointer

x86

x86

- The most widely used ISA
- A poorly-designed ISA
  - It breaks almost every rule of a good ISA
    - variable length of instructions
    - the work of each instruction is not equal
    - makes the hardware become very complex
  - It's popular != It's good
- You don't have to know how to write it, but you need to be able to read them and compare x86 with other ISAs
- Reference
  - http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax

x86 Registers

16bit	32bit	64bit	Description	Notes
AX	EAX	RAX	The accumulator register	These can be used more or less interchangeably
BX	EBX	RBX	The base register	
CX	ECX	RCX	The counter	
DX	EDX	RDX	The data register	
SP	ESP	RSP	Stack pointer	
BP	EBP	RBP	Pointer to the base of stack frame	
	Rn	RnD	General purpose registers (8-15)	
SI	ESI	RSI	Source index for string operations	
DI	EDI	RDI	Destination index for string operations	
IP	EIP	RIP	Instruction pointer	
FLAGS			Condition codes	

MOV and addressing modes

- MOV instruction can perform load/store as in MIPS
- MOV instruction has many address modes
  - an example of non-uniformity

instruction	meaning	arithmetic op	memory op
movl $6, %eax	R[eax] = 0x6	1	0
movl .L0, %eax	R[eax] = .L0	1	0
movl %ebx, %eax	R[ebx] = R[eax]	1	0
movl -4(%ebp), %ebx	R[ebx] = mem[R[ebp]-4]	2	1
movl (%ecx,%eax,4), %eax	R[eax] = mem[R[ebx]+R[edx]*4]	3	1
movl -4(%ecx,%eax,4), %eax	R[eax] = mem[R[ebx]+R[edx]*4-4]	4	1
movl %ebx, -4(%ebp)	mem[R[ebp]-4] = R[ebx]	2	1
movl $6, -4(%ebp)	mem[R[ebp]-4] = 0x6	2	1

Arithmetic Instructions

- Accepts memory addresses as operands
  - Register-memory ISA

instruction	meaning	arithmetic op	memory op
subl $16, %esp	R[%esp] = R[%esp] - 16	1	0
subl %eax, %esp	R[%esp] = R[%esp] - R[%eax]	1	0
subl -4(%ebx), %eax	R[eax] = R[eax] - mem[R[ebx]-4]	2	1
subl (%ebx, %edx, 4), %eax	R[eax] = R[eax] - mem[R[ebx]+R[edx]*4]	3	1
subl -4(%ebx, %edx, 4), %eax	R[eax] = R[eax] - mem[R[ebx]+R[edx]*4-4]	3	1
subl %eax, -4(%ebx)	mem[R[ebx]-4] = mem[R[ebx]-4]-R[eax]	3	2

Branch instructions

- x86 use condition codes for branches
  - Arithmetic instruction sets the flags
  - Example:
    `cmp %eax, %ebx` #computes %eax-%ebx, sets the flag
    `je <location>` #jump to location if equal flag is set
- Unconditional branches
  - Example:
    `jmp <location>` #jump to location

Summation for x86

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)
{
  sum+=A[i];
}
```

→

```
     xorl %eax, %eax
.L2: addl  (%ecx,%eax,4), %edx
     addl  $1, %eax
     cmpl  $100, %eax
     jne .L2
```

Assume
int is 32 bytes
%ecx = &A[0]
%edx = sum;
%eax = i;

MIPS v.s. x86

	MIPS	x86
ISA type	RISC	CISC
instruction width	32 bits	1 ~ 17 bytes
code size	larger	smaller
registers	32	16
addressing modes	reg+offset	base+offset base+index scaled+index scaled+index+offset
hardware	simple	complex

Uniformity of MIPS

- Only 3 instruction formats
  - opcodes, rs, rt, immediate are always at the same place
- Similar amounts of work per instruction
  - only 1 read from instruction memory
  - <= 1 arithmetic operations
  - <= 2 register reads, <= 1 register write
  - <= 1 data memory access
- Fixed instruction length
- Relatively large register file: 32 registers
- Reasonably large immediate field: 16 bits
- Wise use of opcode space: only 6 bit, R-type get another 6

Translate from C to Assembly

- gcc: gcc [options] [src_file]
  - compile to binary
    - gcc -o foo foo.c
  - compile to assembly (assembly in foo.s)
    - gcc -S foo.c
  - compile with debugging message
    - gcc -g -S foo.c
  - optimization
    - gcc -O*n* -S foo.c
      - n from 0 to 3 (0 is no optimization)

Demo

- The magic of compiler optimization!
- Without optimization

- After compiled with -O3

Other than MIPS & x86

ISA alternative

- MIPS is a 3-address ISA
- 2-address ISA
  - add $t1, $t2: R[$t1] = R[$t1] + R[$t2]
  - pros: fewer operands, shorter instructions
  - cons: lots of extra memory copies
- 1-address ISA: accumulator
  - add $t1: accu = accu + R[$t1]
- 0-address ISA: stack-based ISA
  - add: t1 = pop, t2 = pop, t3 = t1+t2, push

Different types of ISA

	stack	accumulator	register-memory	load-store
addresses	0	1	2 or 3	3

A=X*Y-B*C	`push B`
	`push C`
	`mul`
	`push X`
	`push Y`
	`mul`
	`sub`
	`pop A`
		`load B`
		`mul C`
		`store temp`
		`load X`
		`mul Y`
		`sub temp`
		`store A`
			`R1 = X*Y`
			`R2 = B*C`
			`A = R1-R2`
				`load t1, X`
				`load t2, Y`
				`mul t2, t1, t2`
				`load t3, B`
				`load t4, C`
				`mul t4, t4, t3`
				`sub t4, t3, t4`
				`store t4, A`

+	•high code density •easy to compile	•short instructions	•fewest instructions	•simple hardware •fewest memory access
-	•hardware stack design	•most memory access	•complex hardware design	•code size

Q&A