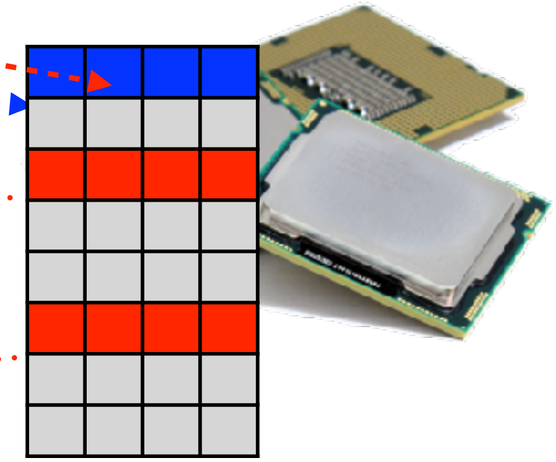
Instruction Set Architecture

Hung-Wei Tseng

Recap: von Neumann model



memory

8

2

3

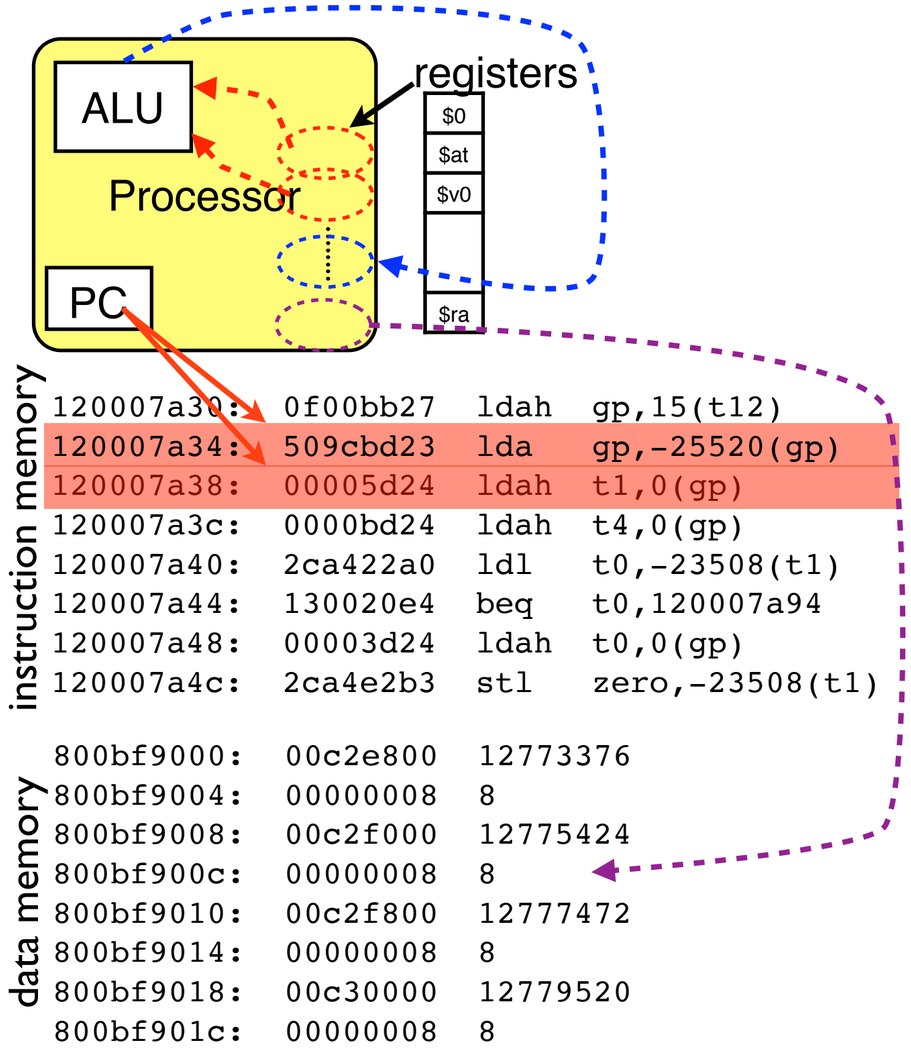
CPU is a dominant factor of performance since we heavily rely on it to execute programs

By pointing "PC" to different part of your memory, we can perform different functions!

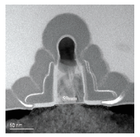
Modern computers

How CPU handle instructions

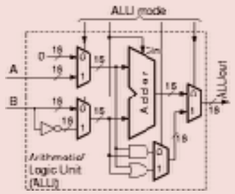
- Instruction fetch: where?
  instruction memory
- Decode:
  - What's the instruction? **registers**
  - Where are the operands? **ALUs**
- Execute
- Memory access **data memory**
  - Where is my data?
- Write back **registers**
  - Where to put the result
- Determine the next PC

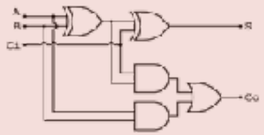


Where is “computer architecture”

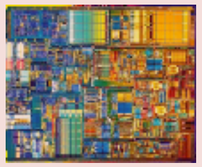


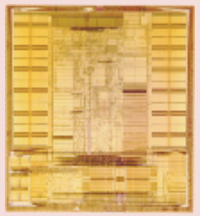
Devices

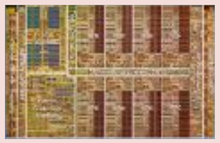




Micro-architecture

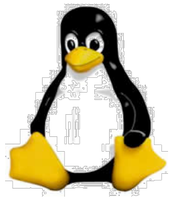






Processors

Instruction Set Architectures



Operating Systems

Compilers
Runtime
Virtual Machines

Programming Languages

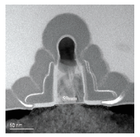
Prog

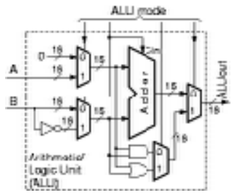
Outline

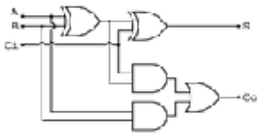
- What is an ISA (Instruction Set Architecture)?
- How source code becomes a running program
- Overview of MIPS ISA

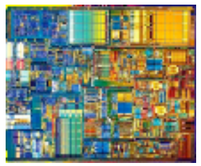
What’s an Instruction Set Architecture (ISA)?

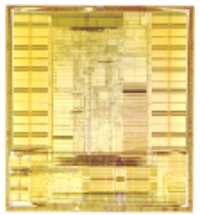
Where is “ISA”?

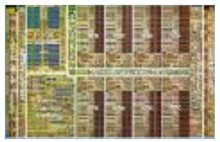












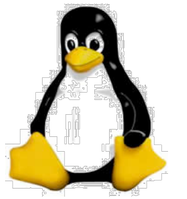
ARM

Devices

Micro-architecture

Processors

Instruction Set Architectures



Operating Systems

JVM

Compilers
Runtime
Virtual Machines

Programming Languages

Prog

Example ISAs

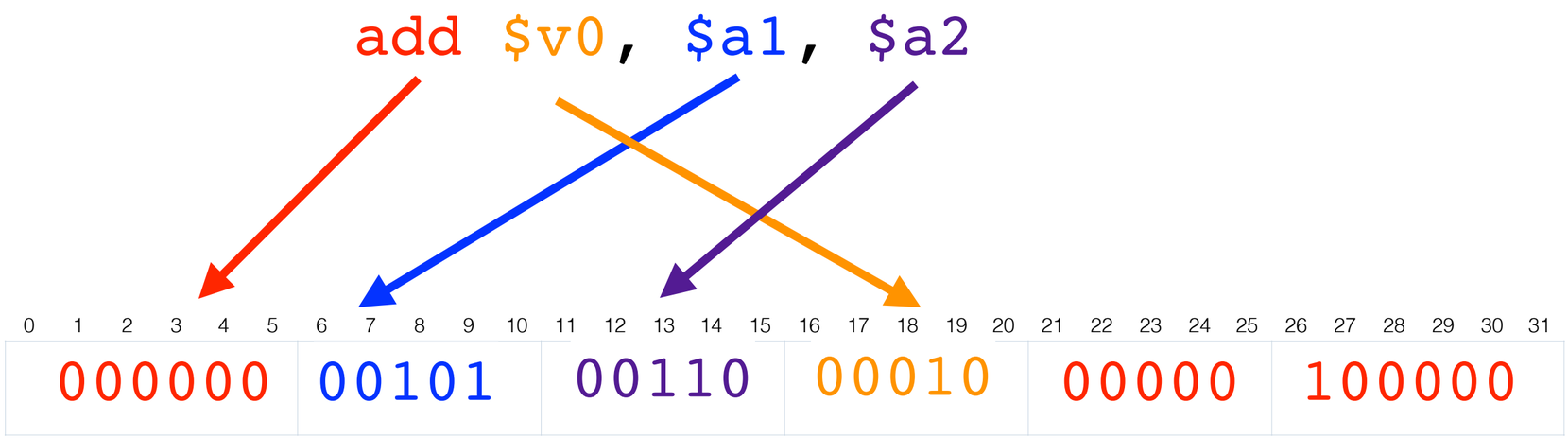
- x86: intel Xeon, intel Core i7/i5/i3, intel atom, AMD Athlon/Opteron, AMD RyZen, AMD FX, AMD A-series **Almost every desktop/laptop/server is using this**
- ARM: Apple A-Series, Qualcomm Snapdragon, TI OMAP, nVidia Tegra **Almost every mobile phone/tablet is using t**
- MIPS: Sony/Toshiba Emotion Engine, MIPS R-4000(PSP)
- DEC Alpha: 21064, 21164, 21264
- PowerPC: Motorola PowerPC G4, Power 6
- IA-64: Itanium
- SPARC and many more ...

Instruction Set Architecture (ISA)

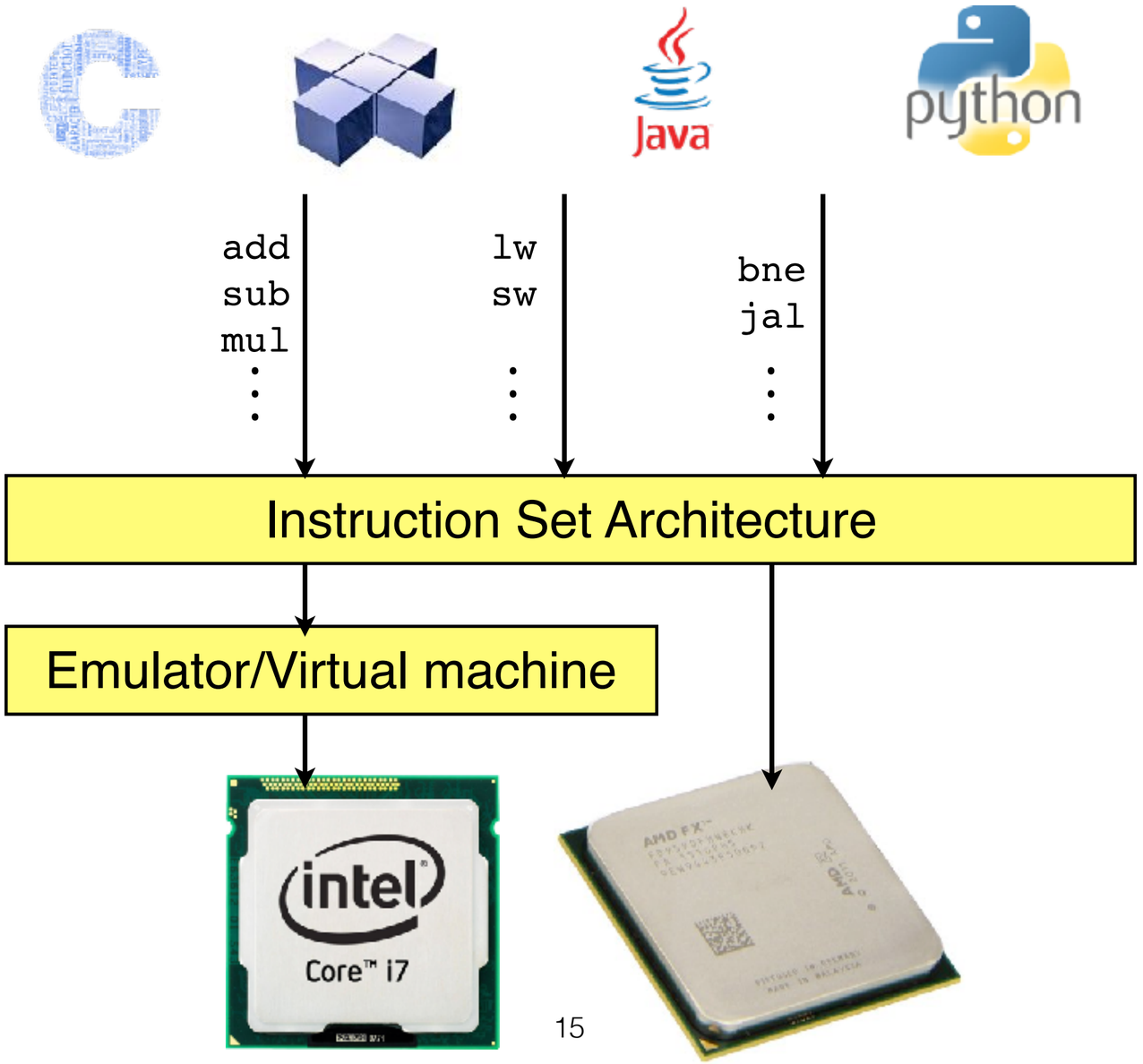
- The **contract** between the hardware and software
- Defines the set of operations that a computer/processor can execute
- Programs are combinations of these instructions
  - Abstraction to programmers/compilers
- The hardware implements these instructions in any way it choose.
  - Directly in hardware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

Assembly language

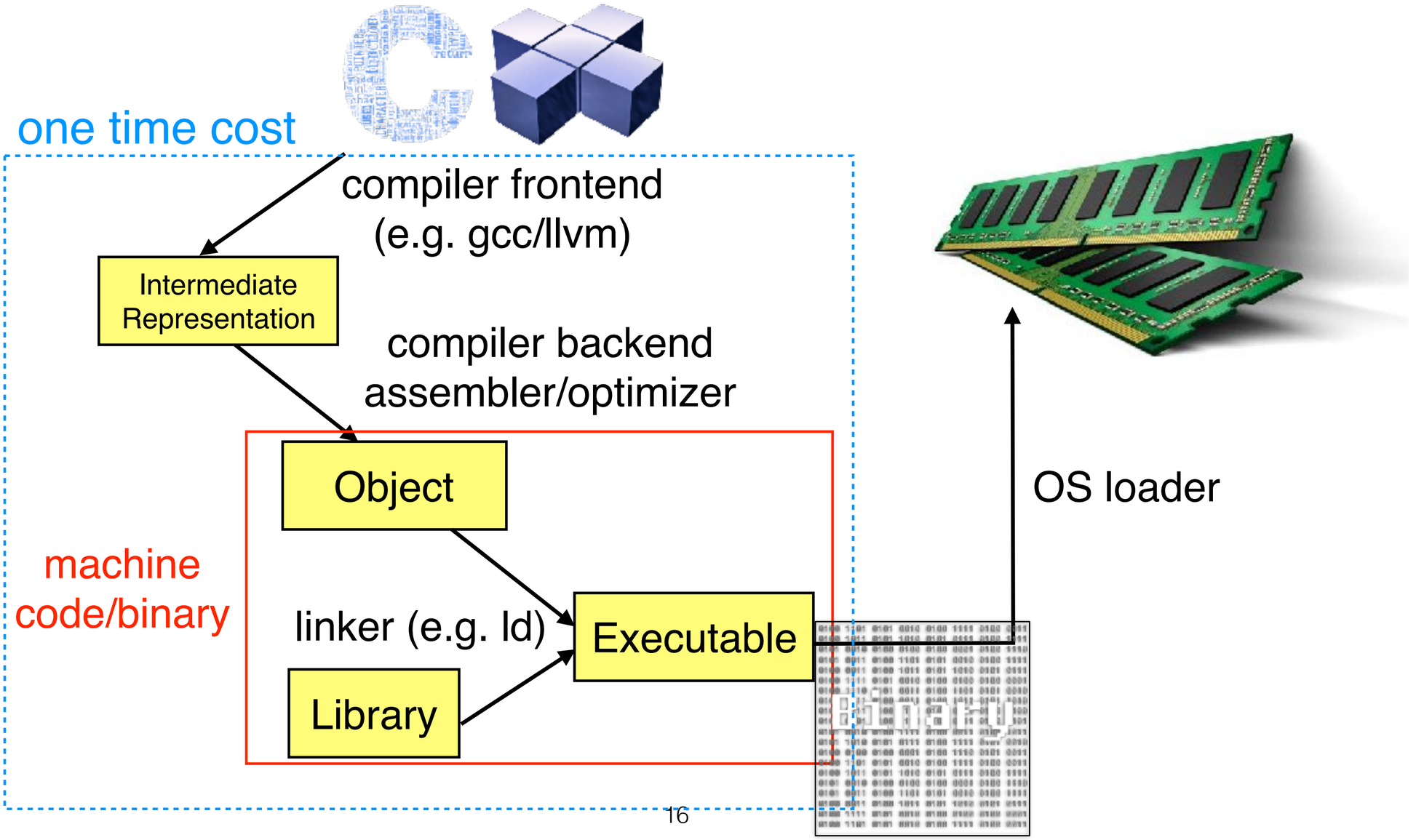
- The human-readable representation of “instructions”/“machine language”
- Has a direct mapping between assembly code and instructions
  - Assembly may contain “pseudo instructions” for programmer to use
  - Each pseudo instruction still has its own mapping to a set of real machine instructions



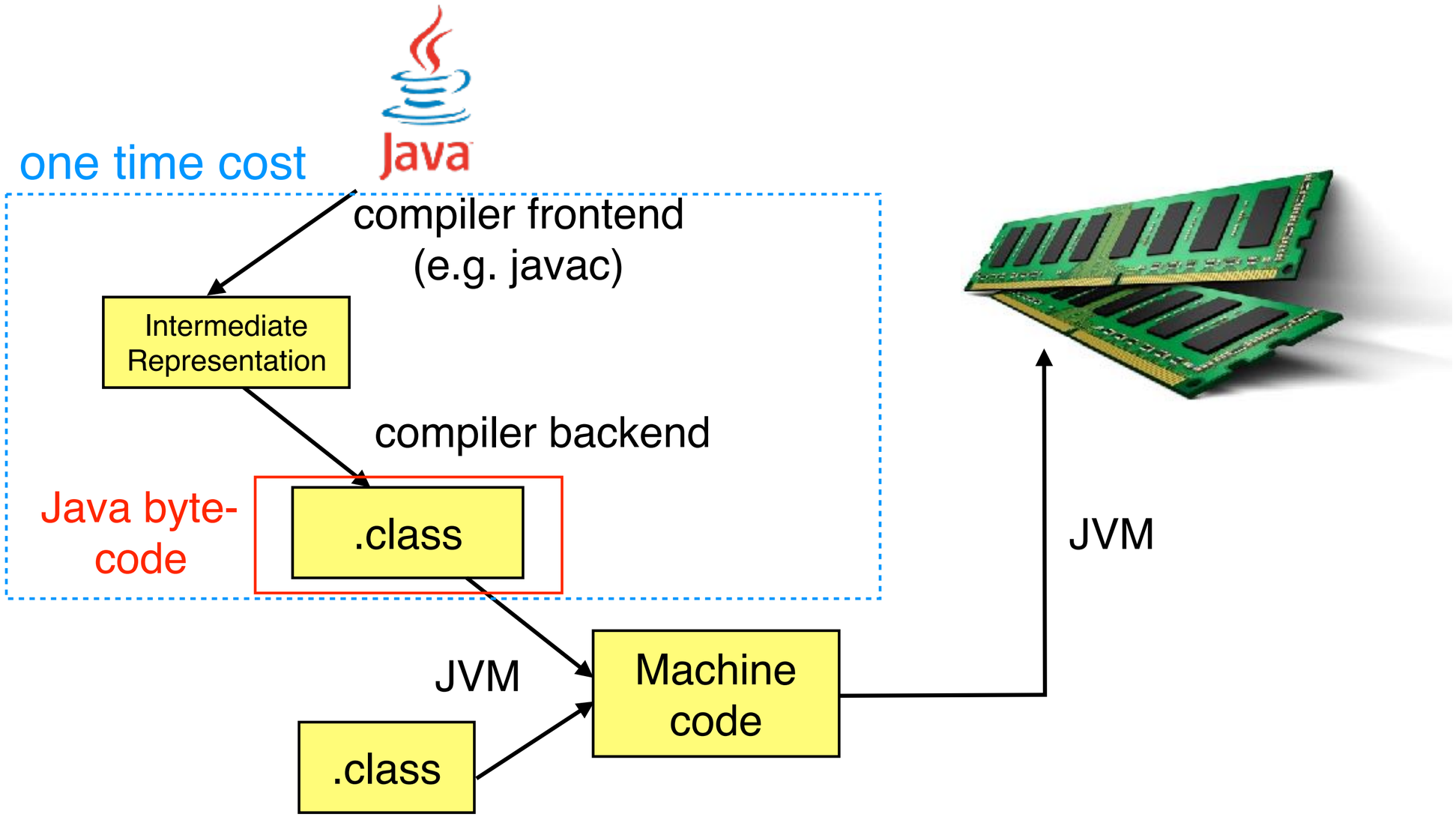
Instruction Set Architecture (ISA)



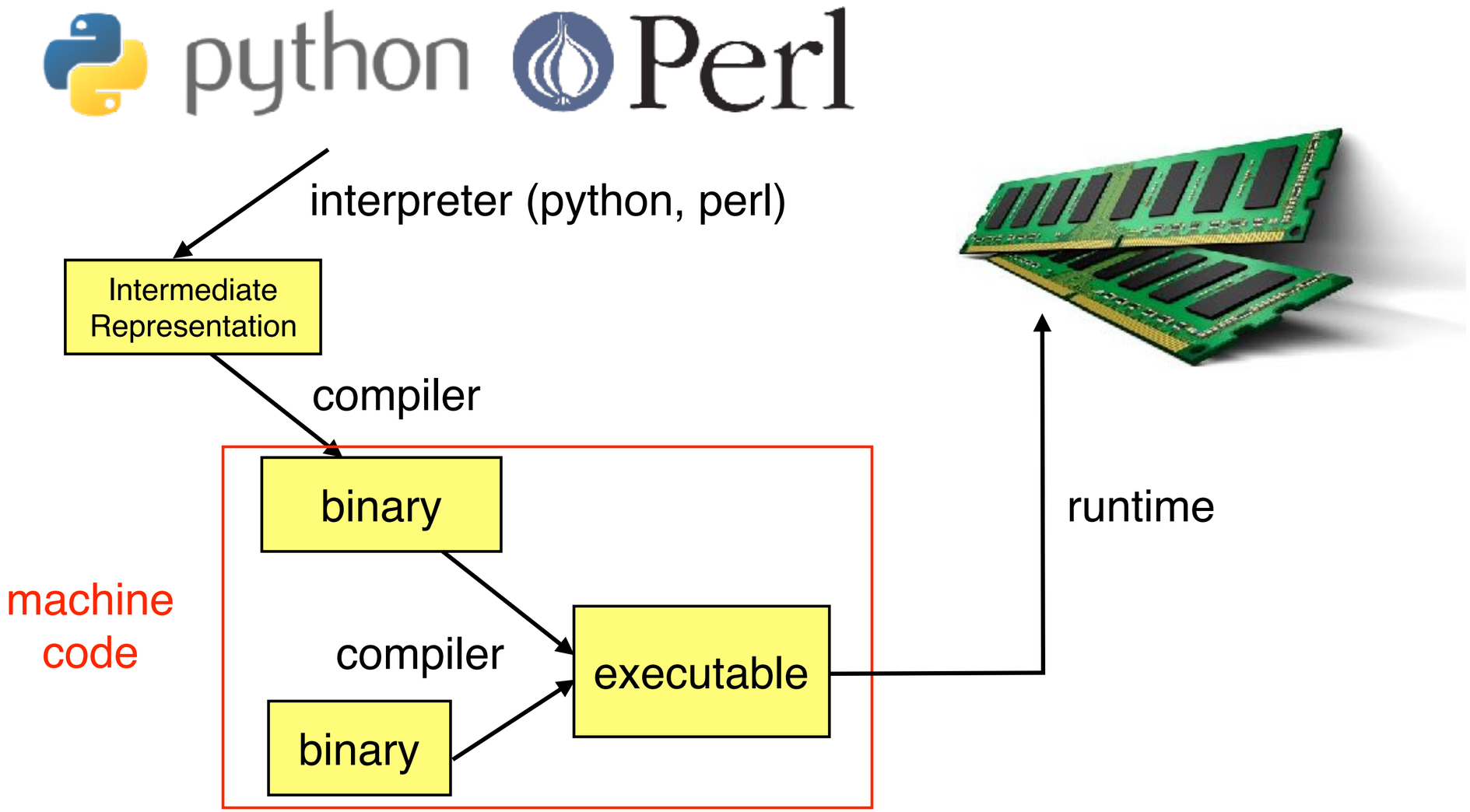
From C/C++ to Machine Code



From Java to Machine Code



From Script Languages to Machine Code

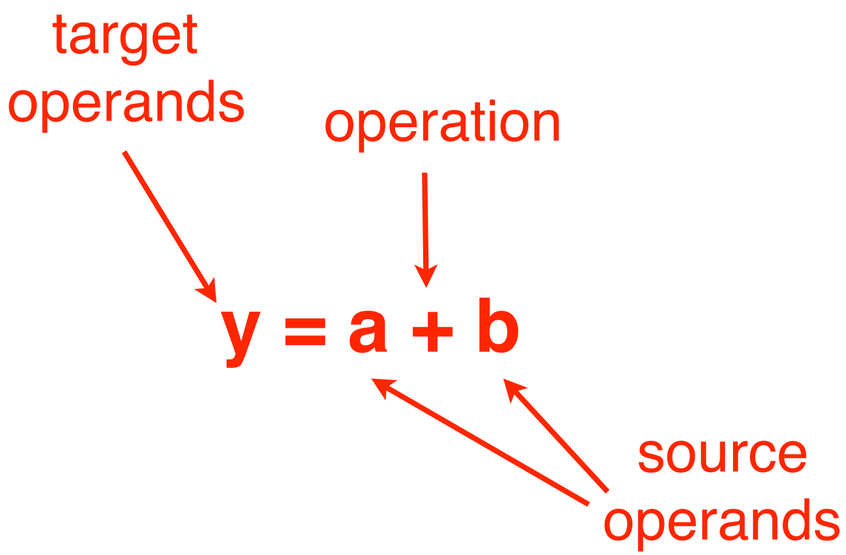


What ISA includes?

- 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

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 $s1, $s2, $s3
add $s1, $s2, 64
```

Overview of MIPS ISA

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
$k0-$k1	26-27	OS kernel	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	.

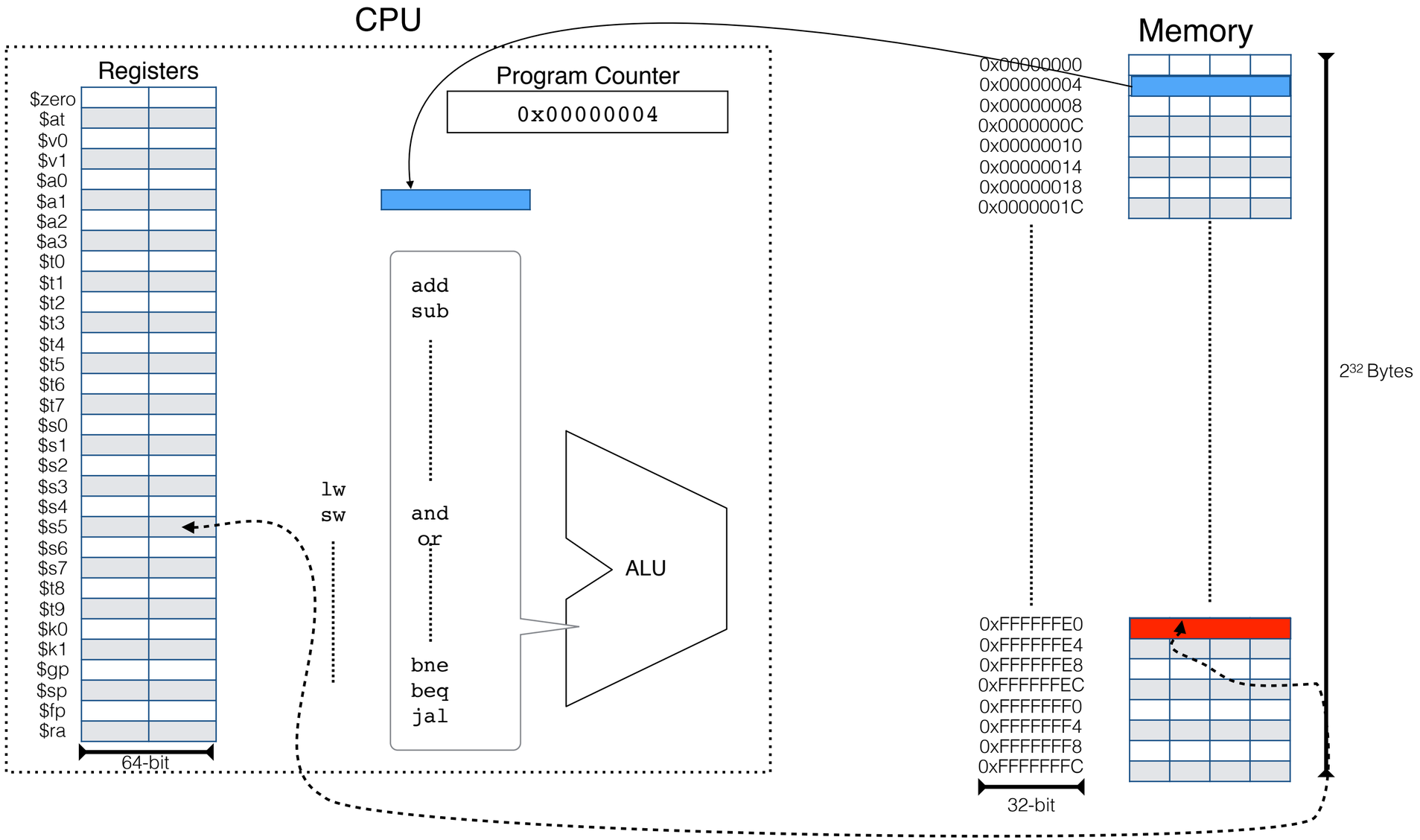
Half Word Addresses

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

Word Addresses

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

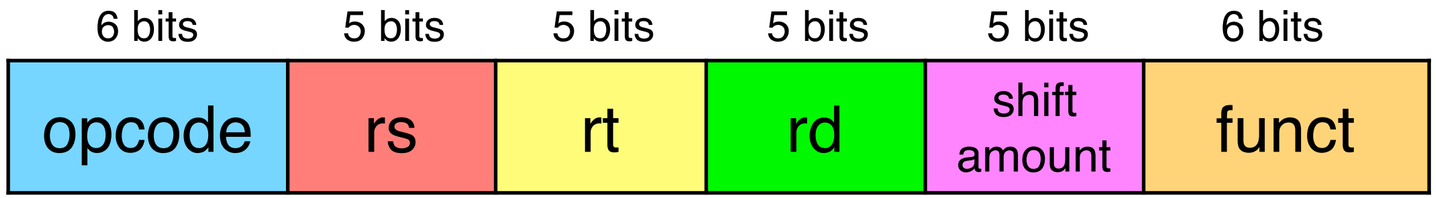
“Abstracted” MIPS Architecture



Frequently used MIPS instructions

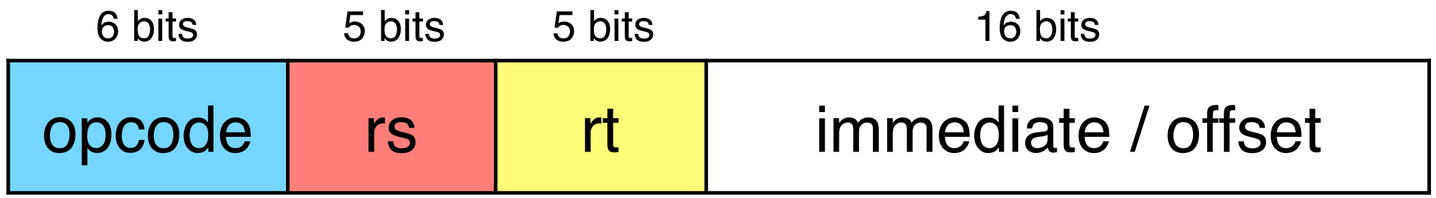
Category	Instruction	Usage	Meaning
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
	addi	`addi $s1,$s2, 20`	`$s1 = $s2 + 20`
	sub	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`
Logical	and	`and $s1, $s2, $s3`	`$s1 = $s2 & $s3`
	or	`or $s1, $s2, $s3`	`$s1 = $s2 | $s3`
	andi	`andi $s1, $s2, 20`	`$s1 = $s2 & 20`
	sll	`sll $s1, $s2, 10`	`$s1 = $s2 * 2^10`
	srl	`srl $s1, $s2, 10`	`$s1 = $s2 / 2^10`
Data Transfer	lw	`lw $s1, 4($s2)`	`$s1 = mem[$s2+4]`
	sw	`lw $s1, 4($s2)`	`mem[$s2+4] = $s1`
Branch	beq	`beq $s1, $s2, 25`	`if($s1 == $s2), PC = PC + 100`
	bne	`bne $s1, $s2, 25`	`if($s1 != $s2), PC = PC + 100`
Jump	jal	`jal 25`	`$ra = PC + 4, PC = 100`
	jr	`jr $ra`	`PC = $ra`

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

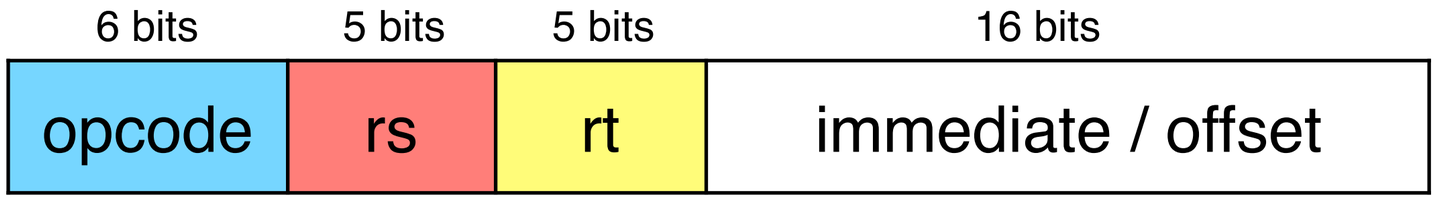
~~`lw $s0, $s2($s1)`~~

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

Data transfer instructions

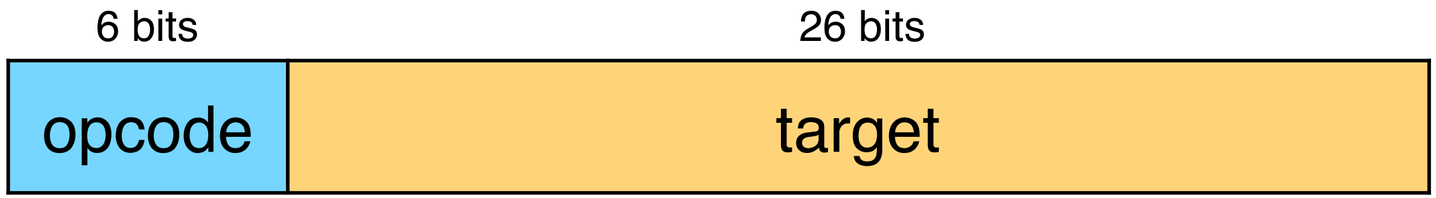
- The ONLY type of instructions that can interact with memory in MIPS
- Two big categories
  - Load (e.g, lw): copy data from memory to a register
  - Store (e.g., sw): copy data from a register to memory
- Two parts of operands
  - A source or destination register
  - Target memory address = base address + offset
    - Register contains the “base address”
    - Constant as the “offset”
    - `8($s0)` = (the content in `$s0`) + 8

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:

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

label

```
        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 (if i = 0, it must < 100)
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

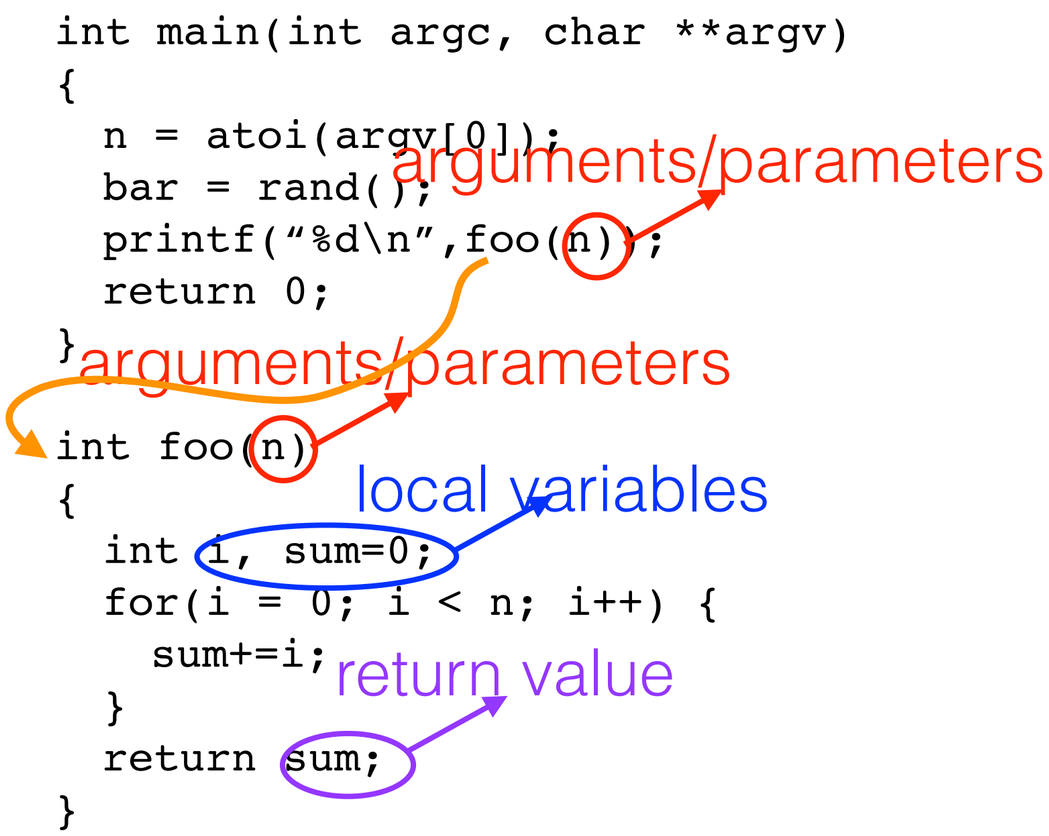
Assume
int is 32 bits
$s0 = &A[0]
$v0 = sum;
$t0 = i;

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

Function calls

Function calls

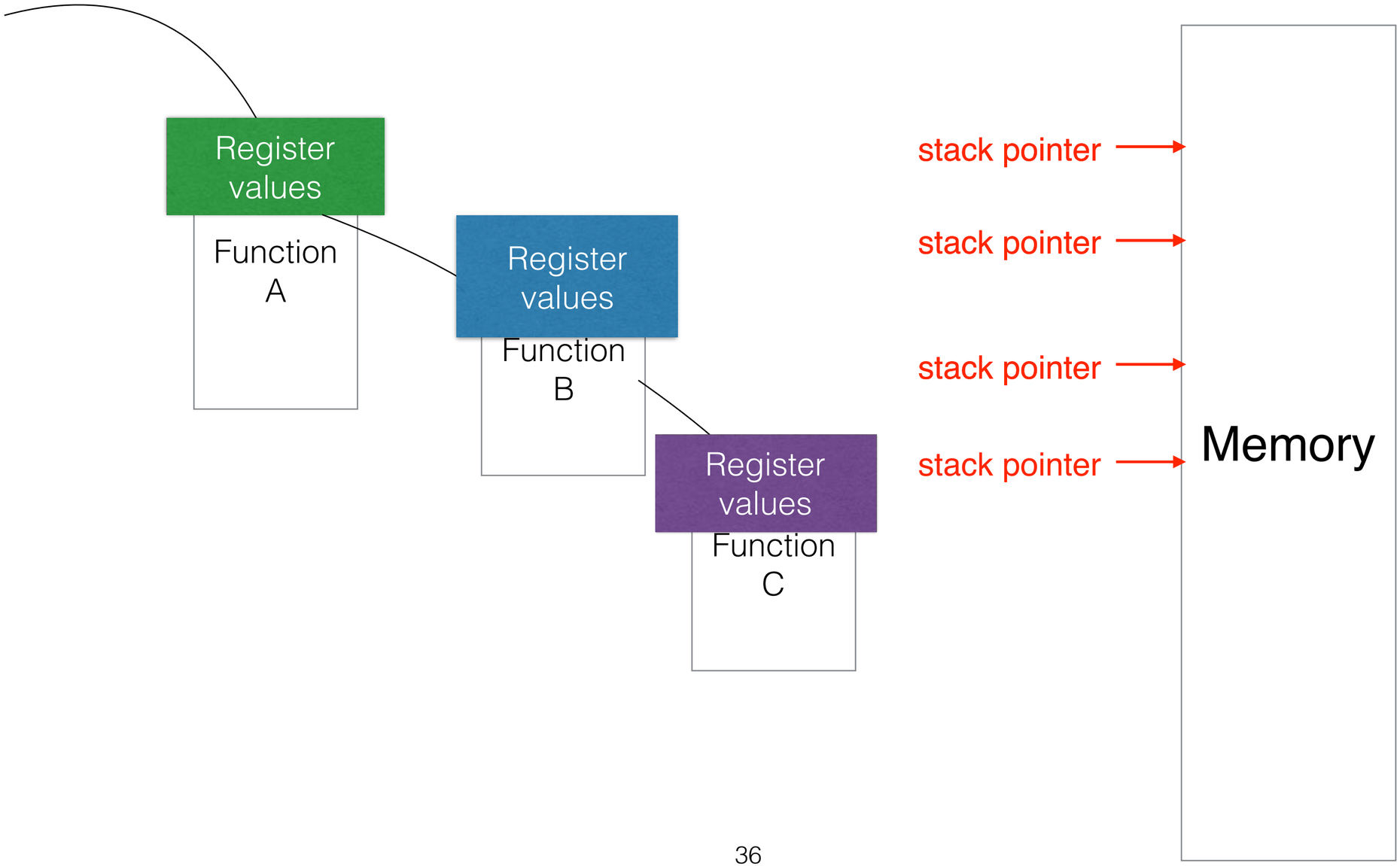
change the PC/control to the callee



Function calls

- Parameters
- Transfer the control from the caller (the code calling the function) to the callee (the function being called)
- Prepare resource (registers/memory locations) for the function call
- Compute
- Return the value
- Return the control to the caller

How to manage the memory space?



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

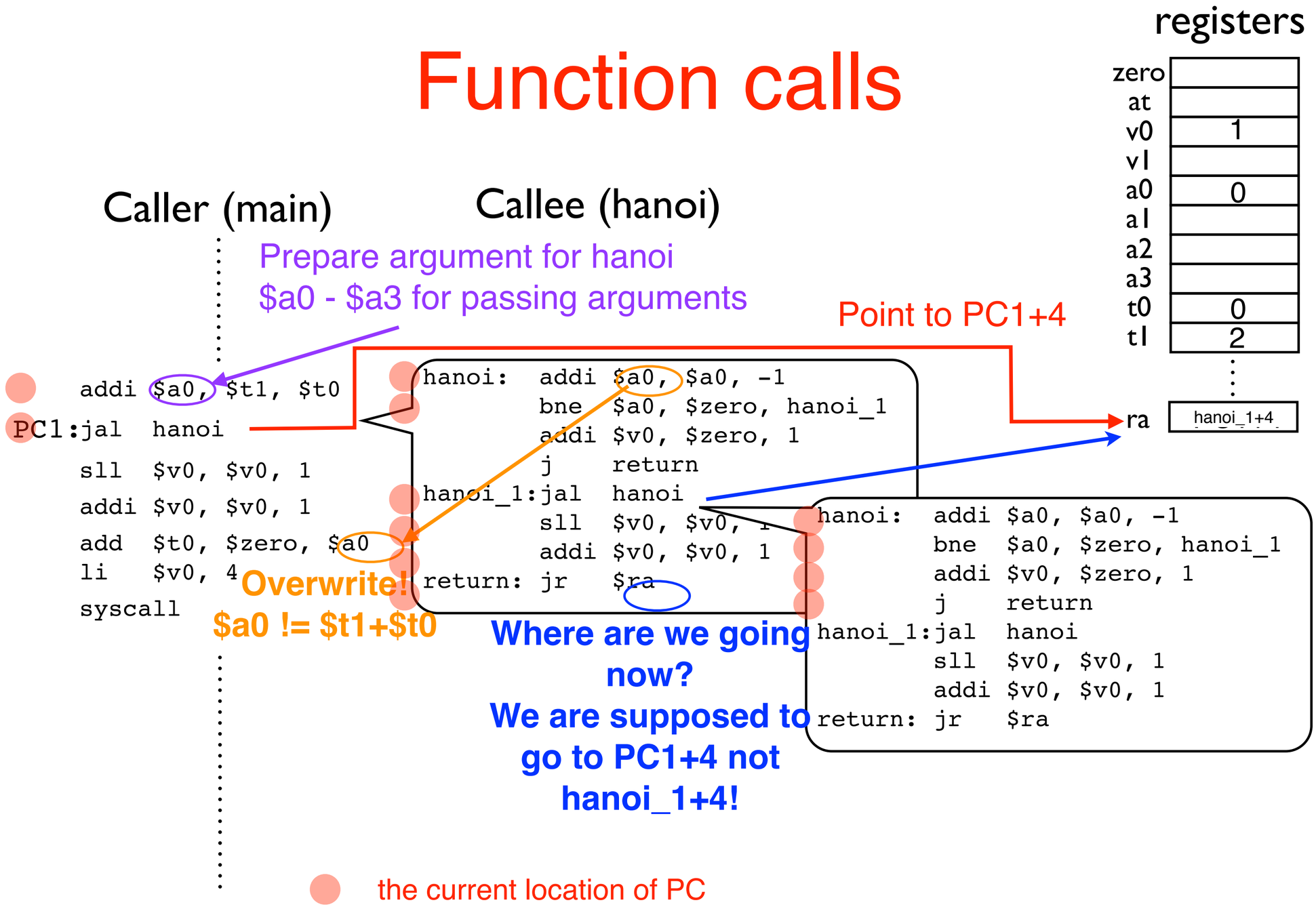
- Passing arguments
  - $a0-$a3
  - more to go using the memory stack
- Invoking the function
  - jal <label>
  - store the PC of jal +4 in $ra
- Return value in $v0
- Return to caller
  - jr $ra

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



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

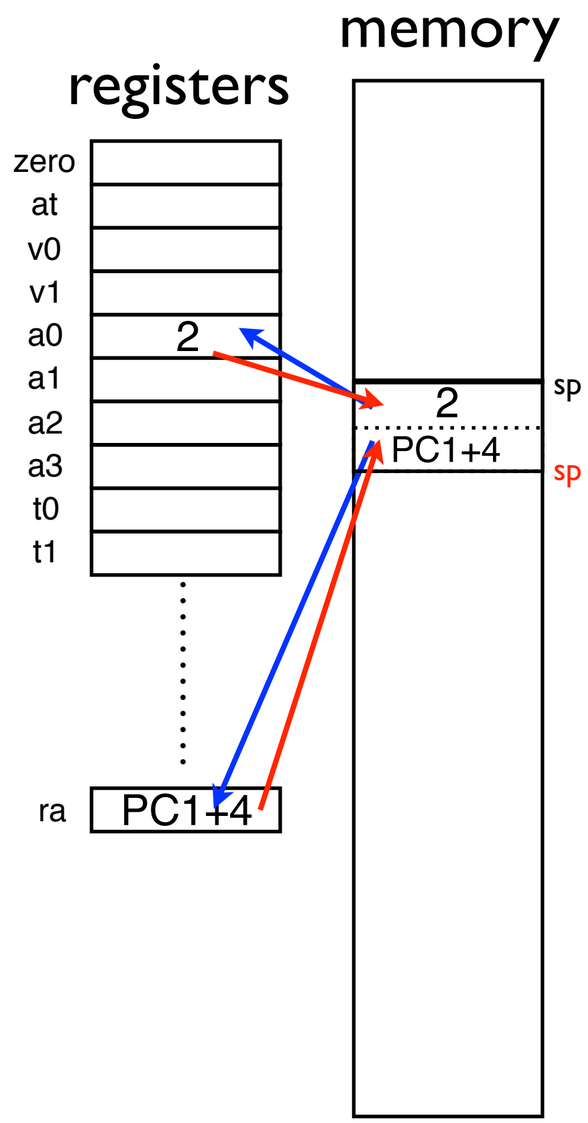
Caller

```
        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
```



save shared registers to the stack, maintain the stack pointer

restore shared registers from the stack, maintain the stack pointer

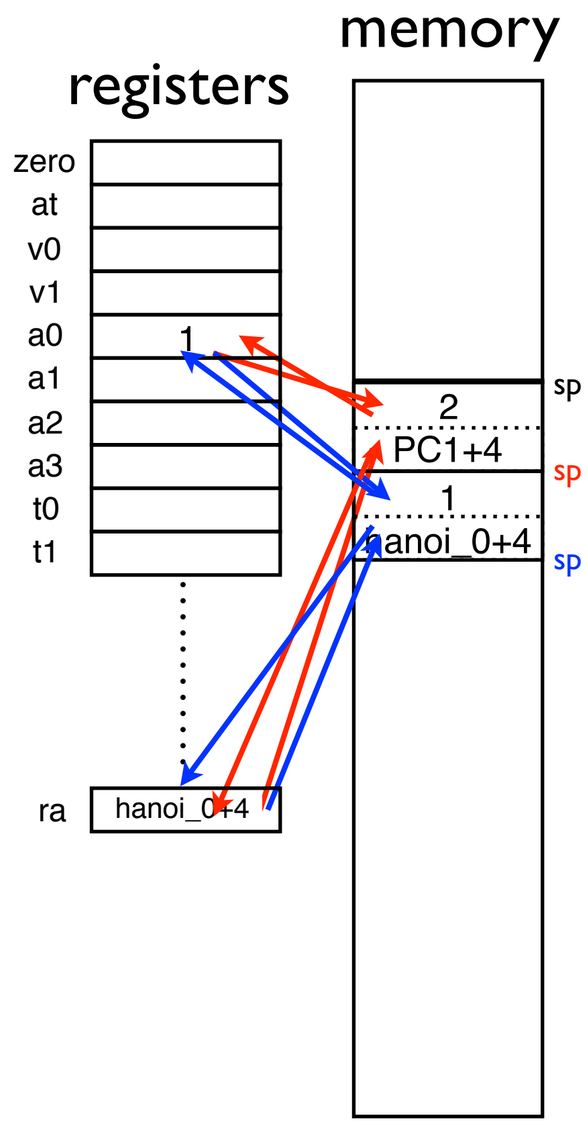
Recursive calls

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

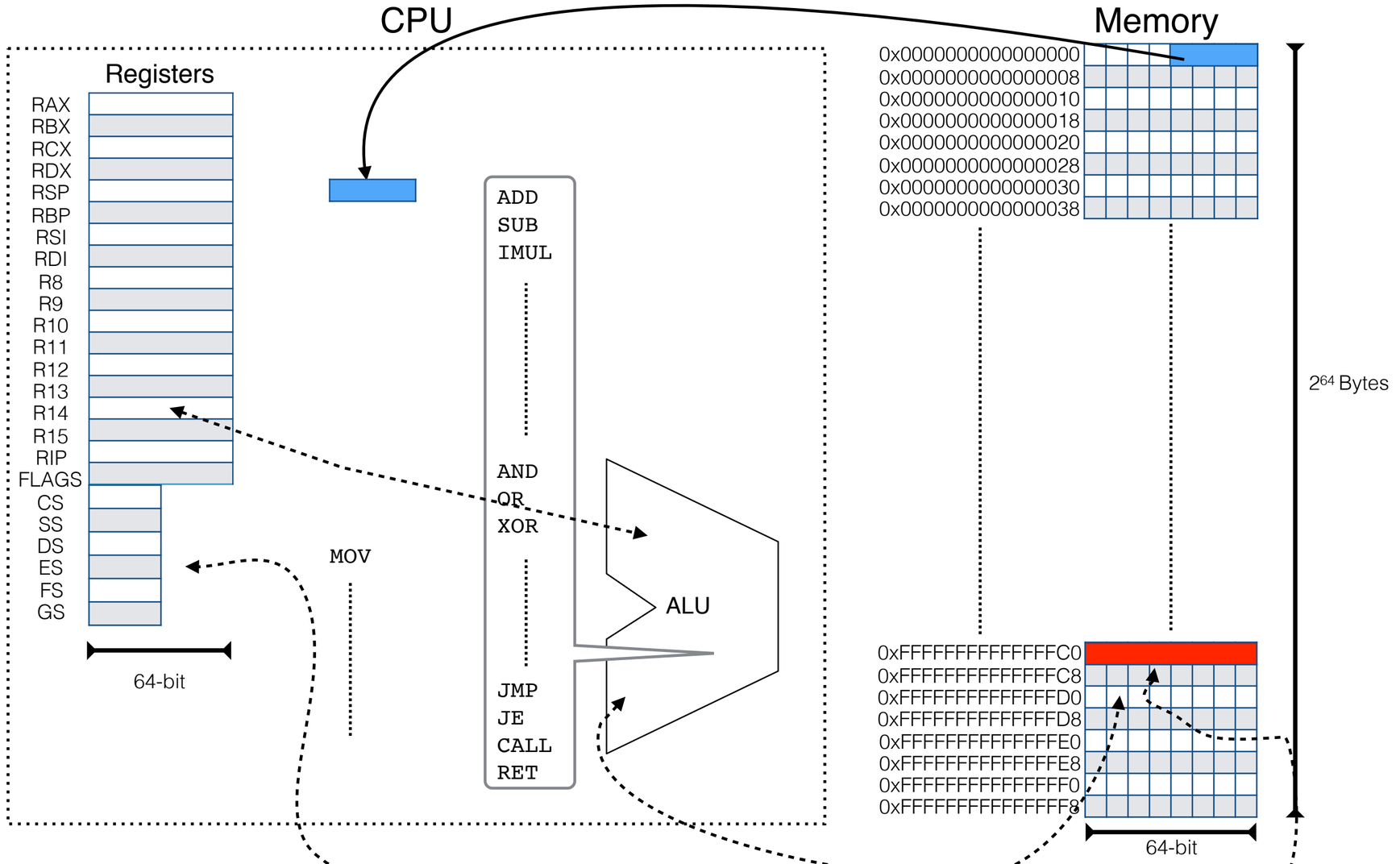
- The overhead of function calls
- The keyword `inline` in C can embed the callee code at the call site
  - Eliminates function call overhead
- Does not work if it’s called using a function pointer

Overview of x86 ISA

x86 ISA

- 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

The abstracted x86 machine architecture



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

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

User-defined data structure

- Programming languages allow user to define their own data types
- In C, programmers can use `struct` to define new data structure

```
struct node {
    int data;
    struct node *next;
};
```

How many bytes each “struct node” will occupy?

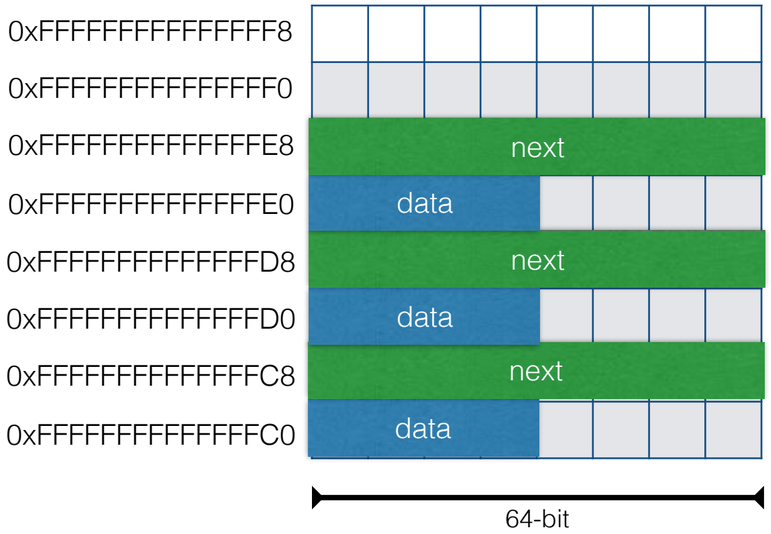
Generate an x86 assembly file on your PC!

- You can do this on your PC with Linux or MacOS
  - You need to have gcc/Xcode installed on your Linux/MacOS machine
  - You will need to complete your 3rd project under this environment — if you're using **Windows** or **MacOS**, you need to install VMWare/VirtualBox to host a linux system
- gcc -S source_file
  - Using "gcc -S hello_world.c", you can get "hello_world.s"
  - Demo

Addressing and accessing the data structure

- Memory allocation
  - Each object/instance of the data structure occupies consecutive memory locations that can accommodate all members in this object/instance
  - The starting address of each object/instance must be aligned with the multiple of 8
    - Try to have as many members aligned with address multiplied by 8 using the smallest amount of space, but also maintains the member order
    - Although ARM supports unaligned access — they are slow
- Memory access:
  - The base address register points to the beginning of the accessing object/instance
  - The offset points to the member — one of the reason why we have an offset field

Memory layout of data structures




```
struct node {
    int data;
    struct node *next;
};
```

Taxonomy of ISAs

How many operations: CISC v.s. RISC

- CISC (Complex Instruction Set Computing)
  - Examples: x86, Motorola 68K
  - Provide **many** **powerful/complex** instructions
    - Many: more than 1503 instructions since 2016
    - Powerful/complex: an instruction can perform both ALU and memory operations
    - Each instruction takes more cycles to execute
- RISC (Reduced Instruction Set Computer)
  - Examples: ARMv8, MIPS (the first RISC instruction, invented by the authors of our textbook)
  - Each instruction only performs simple tasks
  - Easy to decode
  - Each instruction takes less cycles to execute

How many operands: accumulator, stack, memory-register, load-store

	stack	accumulator	register-memory	load-store
operands	0	1	2 or 3	3
operations	work on top elements of the stack	work on one accumulator once	work on registers and memory addresses	work only on several registers, only load/store instructions can interact with memory

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`
			`mul R1,`
			`mem[X], mem[Y]`
			`mul R2,`
			`mem[B], mem[C]`
			`sub 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`

What's in your java classes?

```
public static int fibonacci(int n)  {
                if(n == 0)
                   return 0;
             else if(n == 1)
                   return 1;
                    else
 return fibonacci(n - 1) + fibonacci(n - 2);
                   }
```

```
 0: iload_0
 1: ifne          6
 4: iconst_0
 5: ireturn
 6: iload_0
 7: iconst_1
 8: if_icmpne     13
11: iconst_1
12: ireturn
13: iload_0
14: iconst_1
15: isub
16: invokestatic  #2
19: iload_0
20: iconst_2
21: isub
22: invokestatic  #2
25: iadd
26: ireturn
```

labels

Most instructions doesn't have an argument!

How many operands: accumulator, stack, memory-register, load-store

	stack	accumulator	register-memory	load-store
operands	0	1	2 or 3	3
operations	work on top elements of the stack	work on one accumulator once	work on registers and memory addresses	work only on several registers, only load/store instructions can interact with memory

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`
			`mul R1,`
			`mem[X], mem[Y]`
			`mul R2,`
			`mem[B], mem[C]`
			`sub 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		•fewest instructions	•simple hardware •fewest memory access
-	•hardware stack design •most memory accesses		•complex hardware design	•code size •instruction count