CS202: COMPUTER ORGANIZATION

Lecture 5

Instruction Set Architecture(3)

Today's Topic

- Recap:
  - More control instructions
  - Procedure call
- Today's topic:
  - MIPS addressing
  - Translating and starting a program
  - a C sort example
  - Other popular ISAs

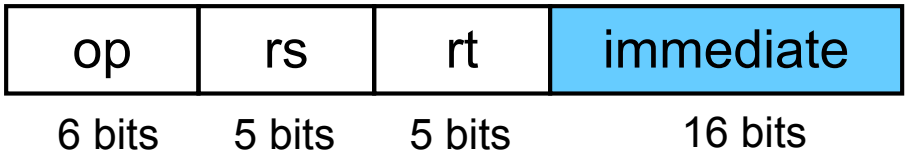
addressing mode is important for all assembly language. Not just for mips

MIPS Addressing

- Addressing: how the instructions identify the operands of the instruction.
- MIPS Addressing mode:
  - Immediate addressing — addi $s0, $s1, 5
  - Register addressing — add $s0, $s1, $s2
  - Base/Displacement addressing — lw $s0, 0($s1)
  - PC-relative addressing — bne $s0, $s1, EXIT
  - Pseudo-direct addressing — j EXIT

Immediate Addressing

- For instructions including immediate
  - E.g. addi, subi, andi, ori



- Most constants are small
  - 16-bit immediate is sufficient

$-2^{15} \geq n \geq 15$

- For the occasional 32-bit constant: y=x+4000000

  $4000000_{dec} = 11\ 1101\ 0000\ 1001\ 0000\ 0000_{bin}$

  lui rt, constant
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

lui $s0, 61

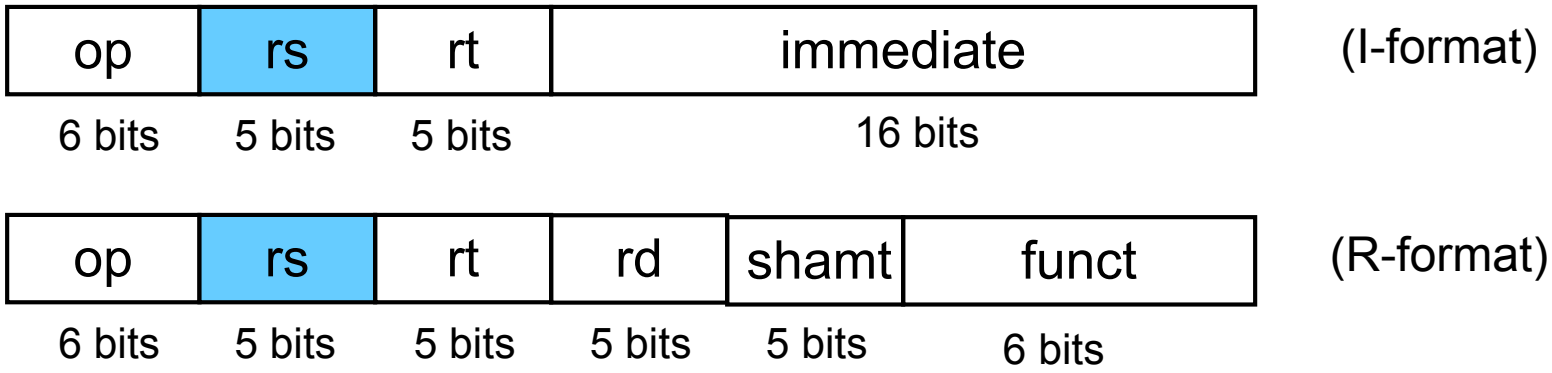
0000 0000 0011 1101	0000 0000 0000 0000

ori $s0, $s0, 2304

0000 0000 0011 1101	0000 1001 0000 0000

Register Addressing

- Using register as the operand
- E.g. add, addi, sub, subi, lw, …



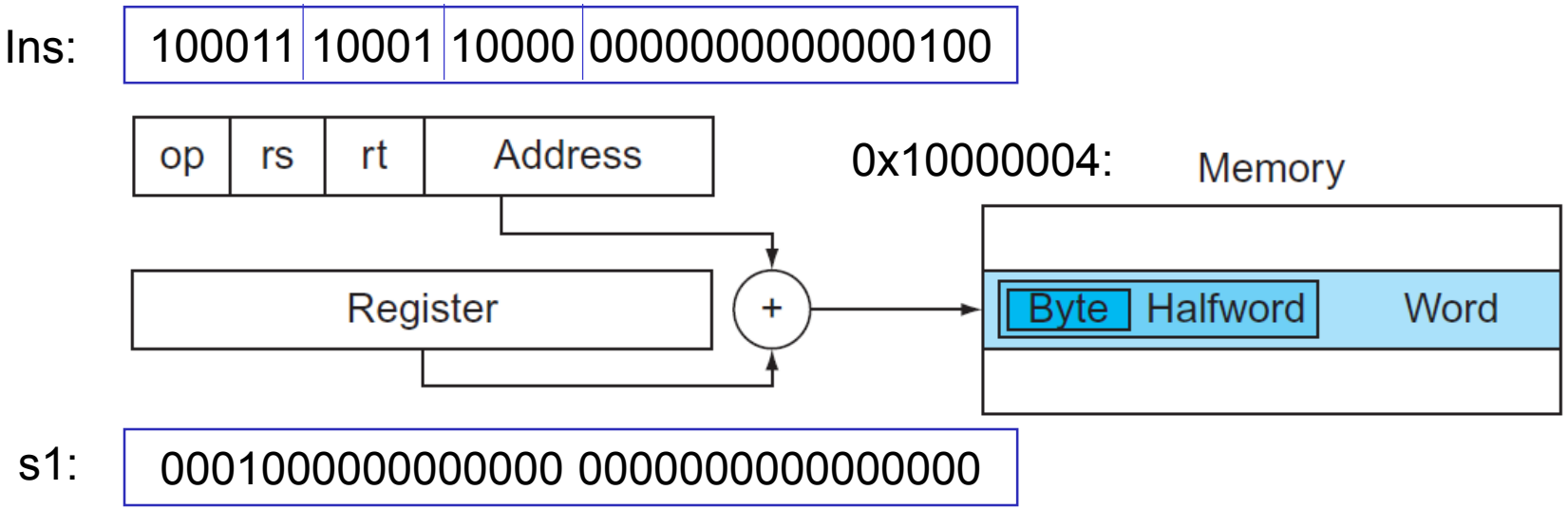
Base/displacement Addressing

- the operand is at the memory, whose address is the sum of a register and a constant lw/lh/lb/sw/sh/sb
  - e.g. lw $s0, 4($s1)

    I- format

    op of lw: 100011

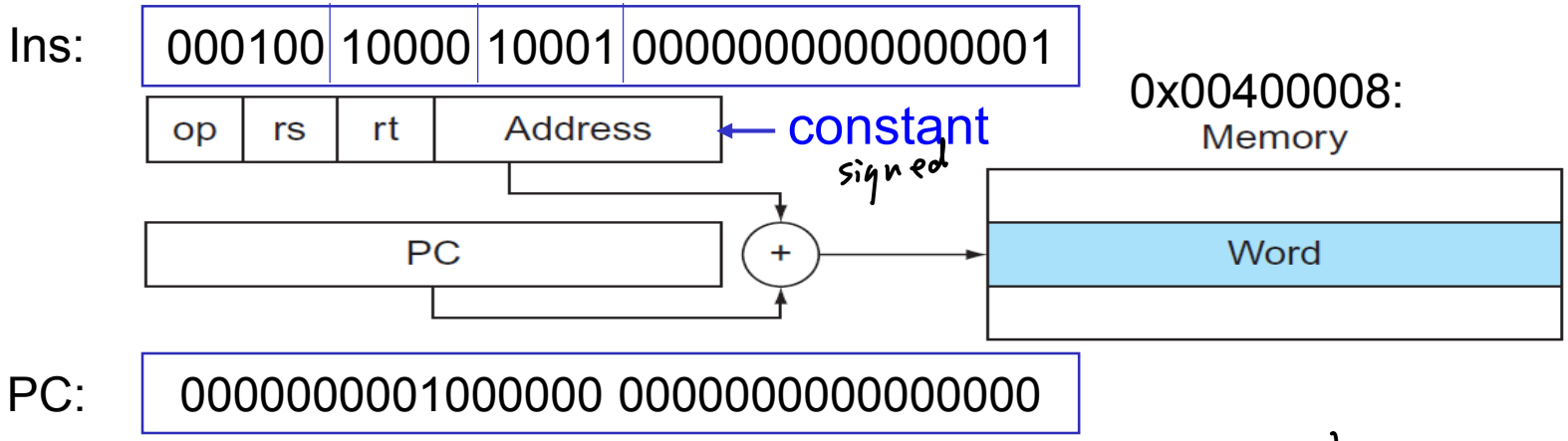
    rs: 10001 (address of s1), rt: 10000 (address of s0), address: 100 (4)



Branch Addressing (PC-relative addressing)

- Branch instructions specify
  - Opcode, two registers, target address
  - e.g. beq $s0 $s1 label
- Most branch targets are near branch
  - Forward or backward

beq $s0 $s1 label
add $s2 $s3 $s4
lable: sub $s2 $s3 $s4



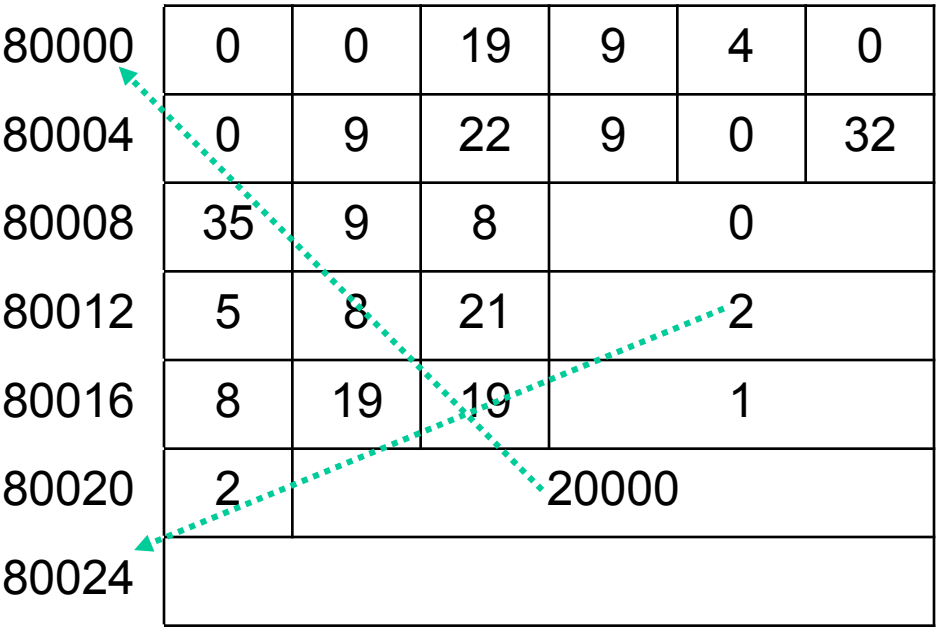
- PC-relative addressing
  - Target address = PC + 4 + constant × 4

you have one line data and one line for 4

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

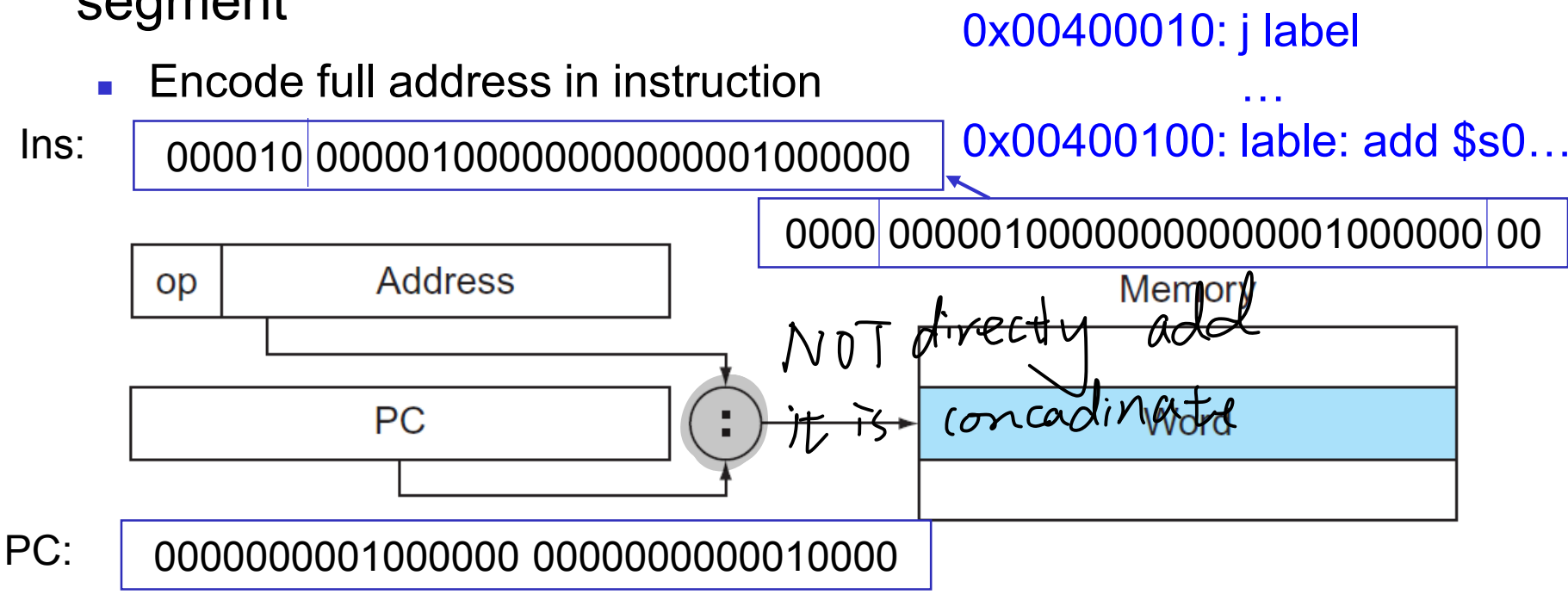
```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```



J format

Jump Addressing (Pseudo-direct addressing)

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction



- (Pseudo)Direct jump addressing
  - Target address = PC31…28 : (address × 4)

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

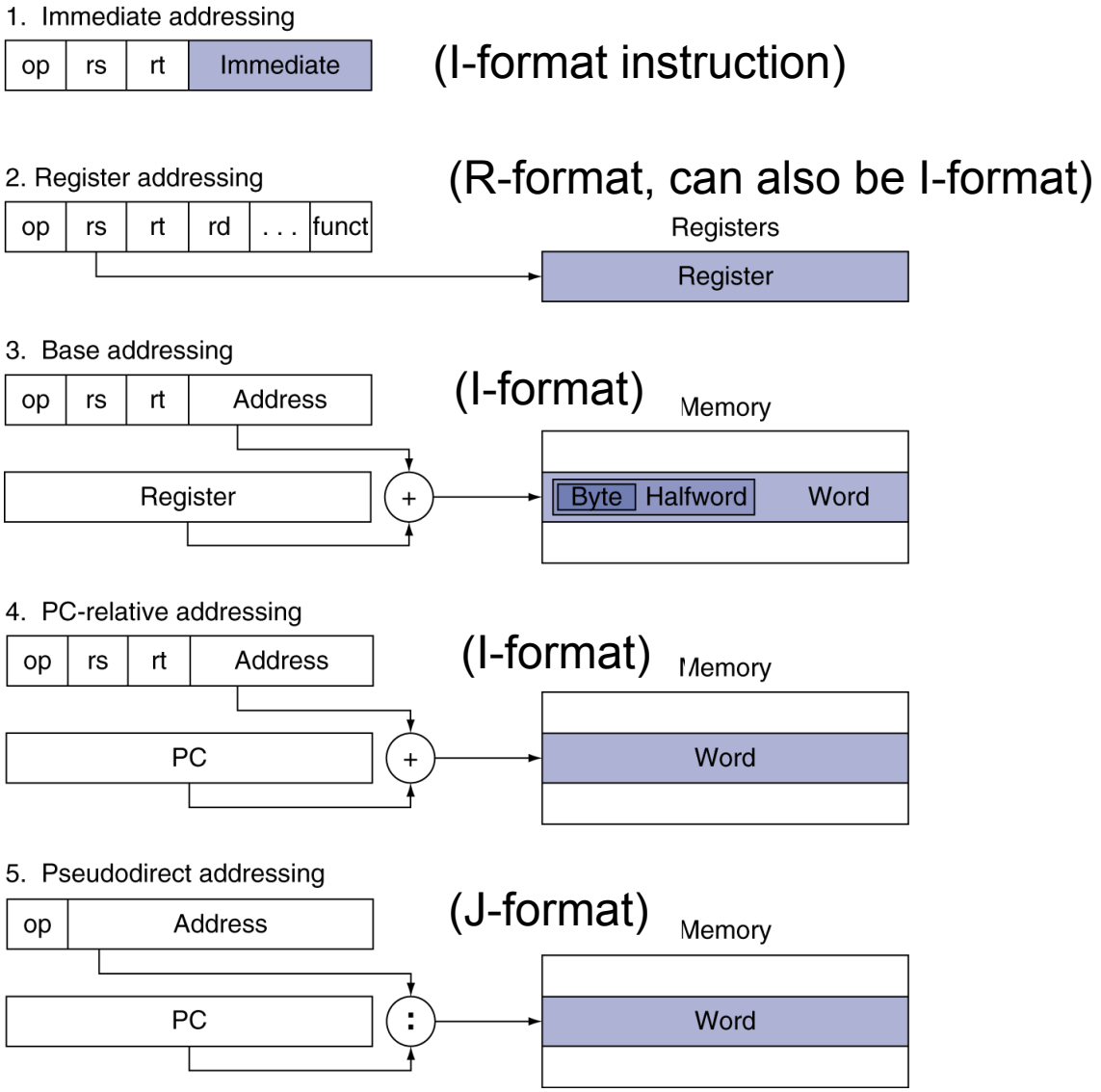
```
        beq $s0,$s1, L1
                ↓
        bne $s0,$s1, L2
        j L1
L2:     …
```

C code to assembly code

↓

translator: compiler

Addressing Mode Summary



There is no “direct addressing”

```
.data
        str: .asciiz "the answer = "
.text
main:
        li $v0, 4
        la $a0, str
        syscall
        lb $t0, ($a0)

        li $v0, 10
        syscall
```

We can only use lw/sw to visit the memory

ori/lui is immediate addressing
lb is base/displacement addressing

Address	Code	Basic		
0x00400000	0x24020004	addiu $2, $0, 0x00000004	5:	li $v0, 4
0x00400004	0x3c011001	lui $1, 0x00001001	6:	la $a0, str
0x00400008	0x34240000	ori $4, $1, 0x00000000		
0x0040000c	0x0000000c	syscall	7:	syscall
0x00400010	0x80880000	lb $8, 0x00000000($4)	8:	lb $t0, ($a0)
0x00400014	0x2402000a	addiu $2, $0, 0x0000000a	10:	li $v0, 10
0x00400018	0x0000000c	syscall	11:	syscall

N...	Nu...
$...	0
$at	1
$v0	2
$v1	3
$a0	4

Decoding Machine Language

- What is the assembly language of the following machine instruction?

  0x00af8020

- Hex to bin: 0000 0000 1010 1111 1000 0000 0010 0000

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

- Get the instruction: add $s0,$a1,$t7

Decoding Machine Language

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

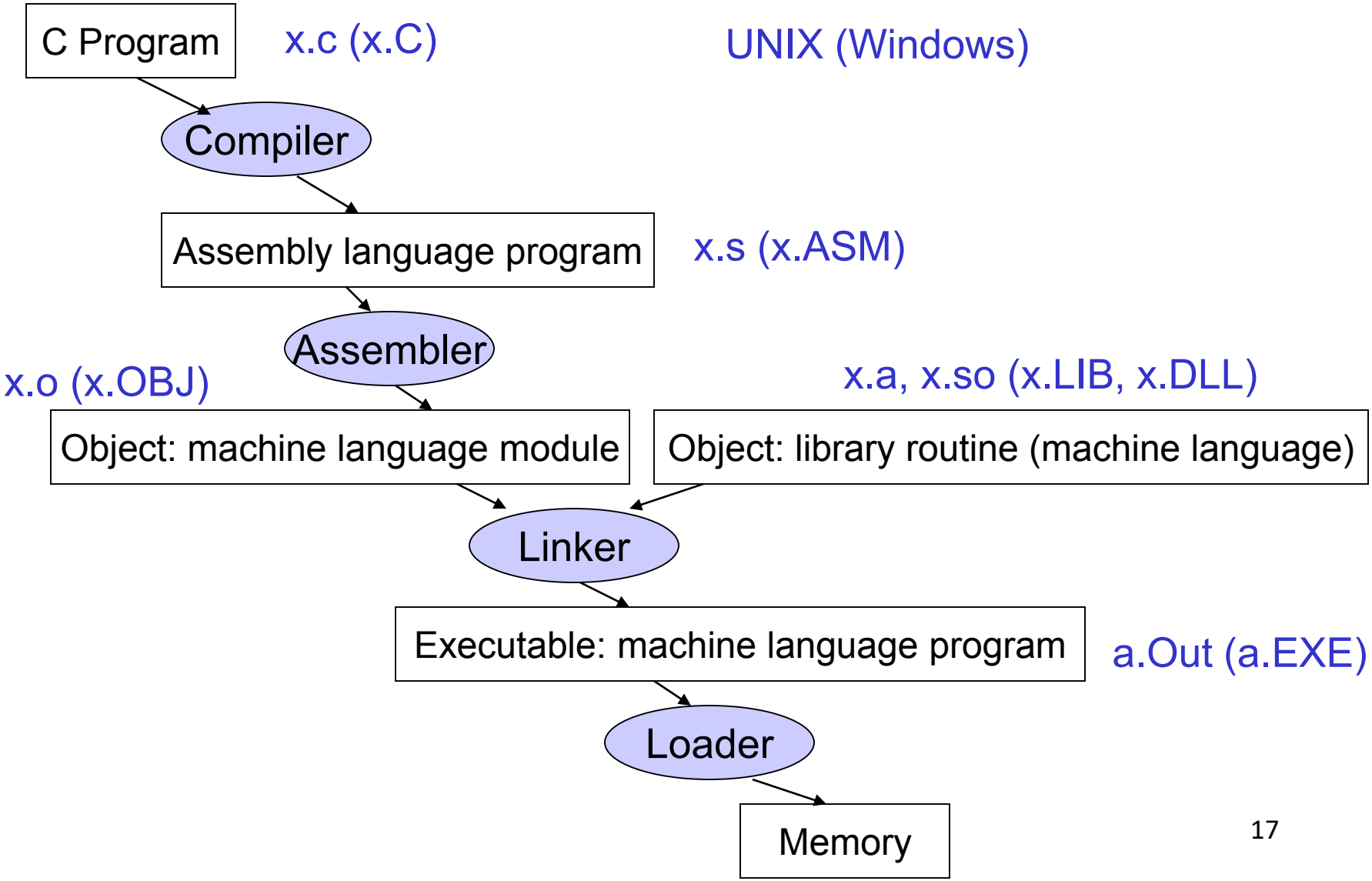
Decoding Machine Language

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

MIPS Instruction Formats

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Starting a C Program



Role of Assembler

- Convert pseudo-instructions into actual hardware instructions – pseudo-instrs make it easier to program in assembly – examples: “move”, “blt”, 32-bit immediate operands, etc.

- Convert assembly instrs into machine instrs – a separate object file (x.o) is created for each C file (x.c) – compute the actual values for instruction labels – maintain info on external references and debugging information

Role of Linker

- Stitches different object files into a single executable
  - patch internal and external references
  - determine addresses of data and instruction labels
  - organize code and data modules in memory
- Some libraries (DLLs) are dynamically linked – the executable points to dummy routines – these dummy routines call the dynamic linker-loader so they can update the executable to jump to the correct routine

Role of Linker

Object file 1:

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	

Role of Linker

Object file 2:

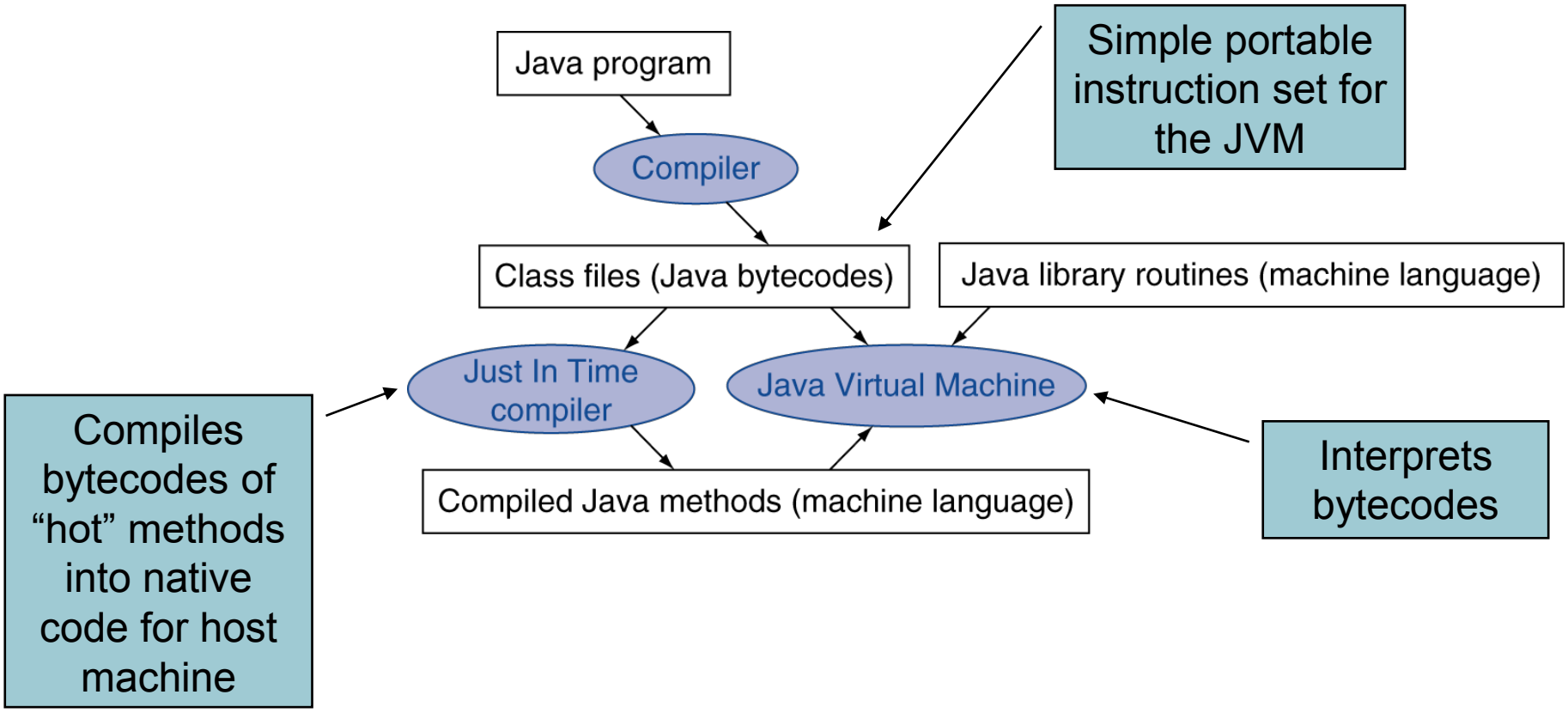
Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Role of Linker

Executable file: $gp+8000_{hex}=10008000_{hex}+ffff8000_{hex}=10000000_{hex}$

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
A	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
	$0040\ 0004_{hex}$	jal $40\ 0100_{hex}$
	...	...
B	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
	$0040\ 0104_{hex}$	jal $40\ 0000_{hex}$
	...	...
Data segment	Address	
X	$1000\ 0000_{hex}$	(X)
	...	...
Y	$1000\ 0020_{hex}$	(Y)
	...	...

Starting Java Applications



Full Example – Sort in C (pg. 133)

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

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

- Allocate registers to program variables
- Produce code for the program body
- Preserve registers across procedure invocations

The swap Procedure

- Register allocation: $a0 and $a1 for the two arguments, $t0 for the temp variable – no need for saves and restores as we're not using $s0-$s7 and this is a leaf procedure (won't need to re-use $a0 and $a1)

```
swap:   sll   $t1, $a1, 2
        add   $t1, $a0, $t1
        lw    $t0, 0($t1)
        lw    $t2, 4($t1)
        sw    $t2, 0($t1)
        sw    $t0, 4($t1)
        jr    $ra
```

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

The sort Procedure

- Register allocation: arguments v and n use $a0 and $a1, i and j use $s0 and $s1; must save $a0 and $a1 before calling the leaf procedure

- The outer for loop looks like this: (note the use of pseudo-instrs)

```
                move   $s0, $zero          # initialize the loop
loopbody1: bge     $s0, $a1, exit1     # will eventually use slt and beq
                … body of inner loop …
                addi    $s0, $s0, 1
                j          loopbody1
exit1:
```

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

The sort Procedure

- The inner for loop looks like this:

```
                addi    $s1, $s0, -1          # initialize the loop
loopbody2:      blt     $s1, $zero, exit2     # will eventually use slt and beq
                sll     $t1,  $s1, 2
                add     $t2, $a0, $t1
                lw      $t3, 0($t2)
                lw      $t4, 4($t2)
                bgt     $t3, $t4, exit2
                … body of inner loop …
                addi    $s1, $s1, -1
                j        loopbody2
exit2:
```

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

Saves and Restores

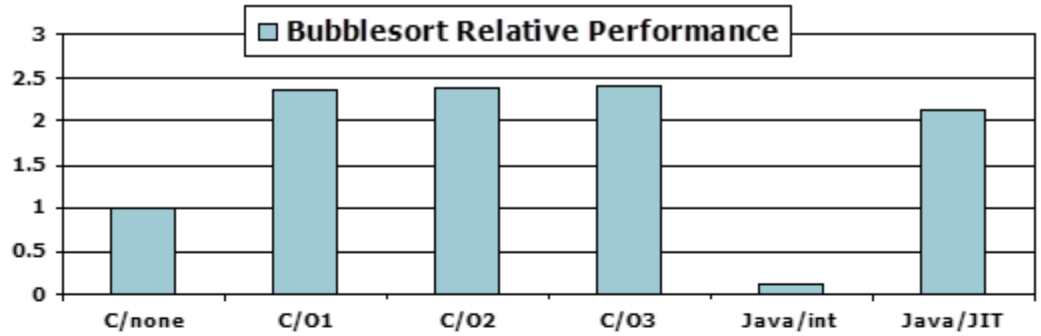
- Since we repeatedly call “swap” with $a0 and $a1, we begin “sort” by copying its arguments into $s2 and $s3 – must update the rest of the code in “sort” to use $s2 and $s3 instead of $a0 and $a1
- Must save $ra at the start of “sort” because it will get over-written when we call “swap”
- Must also save $s0-$s3 so we don’t overwrite something that belongs to the procedure that called “sort”

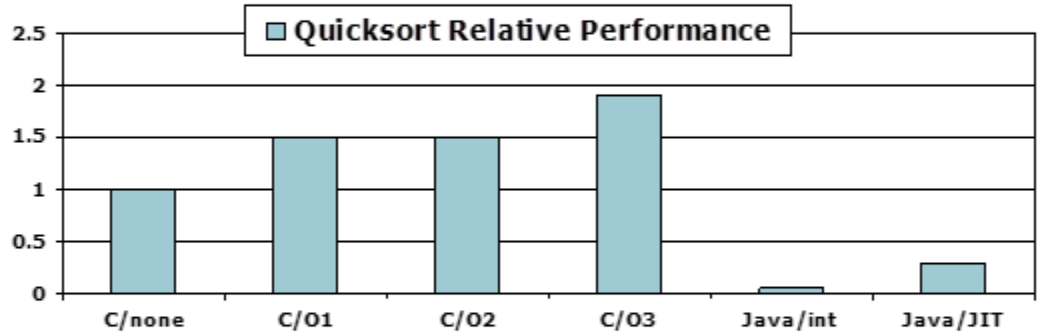
Saves and Restores

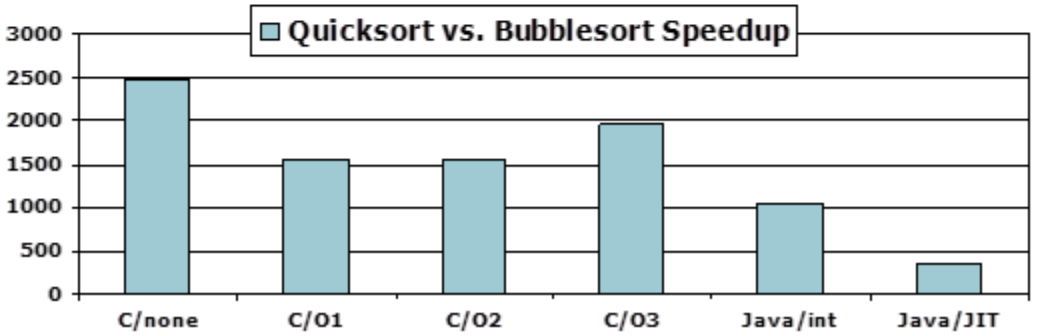
```
sort:   addi    $sp, $sp, -20
        sw      $ra, 16($sp)
        sw      $s3, 12($sp)
        sw      $s2, 8($sp)
        sw      $s1, 4($sp)
        sw      $s0, 0($sp)
        move    $s2, $a0
        move    $s3, $a1
          …
        move    $a0, $s2        # the inner loop body starts here
        move    $a1, $s1
        jal     swap
          …
exit1:  lw      $s0, 0($sp)
          …
        addi    $sp, $sp, 20
        jr      $ra
```

9 lines of C code → 35 lines of assembly

Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

Other ISAs

- RISC-V
- ARM
- x86

Mainstream ISAs

Full RISC-V Architecture：
https://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2016-1.pdf

x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

Core i3, i5, i7…

ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)

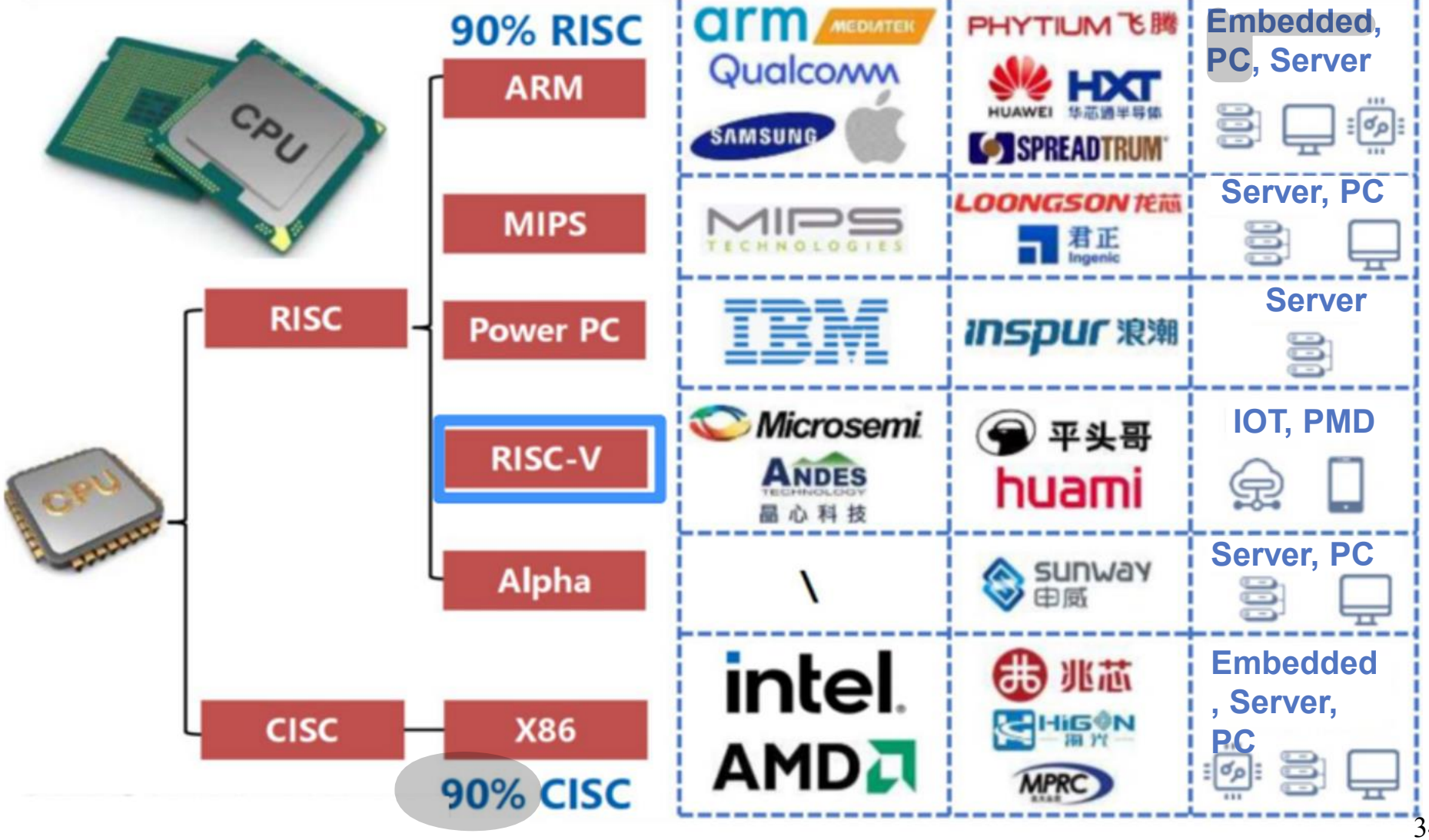
Smartphone-like devices (iPhone, Android), Raspberry Pi, Embedded systems
Apple M series

RISC-V

Designer	University of California, Berkeley
Bits	32, 64, 128
Introduced	2010
Version	2.2
Design	RISC
Type	Load-store
Encoding	Variable
Branching	Compare-and-branch
Endianness	Little

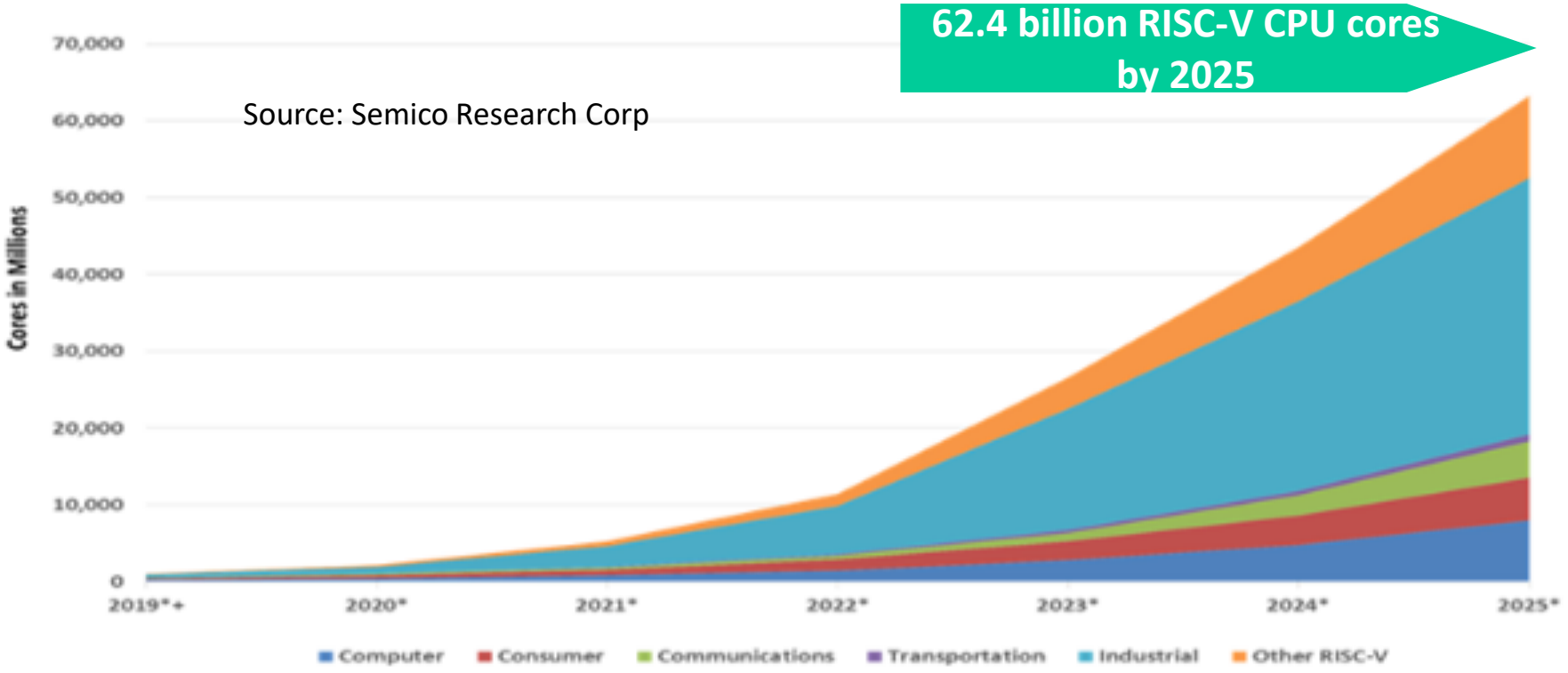
Versatile and open-source
Relatively new, designed for cloud computing, embedded systems, academic use

RISC-V among ISAs



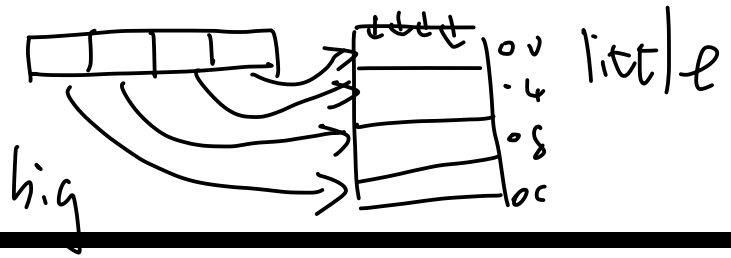
Rapid RISC-V growth led by industrial

- Semico Research predicts the market will consume 62.4 billion RISC-V CPU cores by 2025, a 146.2% CAGR 2018-2025. The industrial sector to lead with 16.7 billion cores.
- Custom ICs Based on RISC-V Will Enable Cost-Effective IoT Product Differentiation



RISC-V foundation

MIPS vs. RISC-V



- Similar basic set of instructions

	MIPS32	RISC-V (RV32)
Date announced	1985	2010
License	Proprietary	Open-Source
Instruction size	32 bits	32 bits
Endianness	Big-endian	Little-endian
Addressing modes	5	4
Registers	32 × 32-bit	32 × 32-bit
Pipeline Stages	5 stages	5 stages
ISA type	Load-store	Load-store
Conditional branches	slt, sltu + beq, bnq	+blt,bge,bltu,bgeu

RISC-V Registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

RISC-V Assembly

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
Data transfer	Load doubleword	ld x5,40(x6)	x5 = Memory[x6 + 40]	Doubleword from memory to register
	Store doubleword	sd x5,40(x6)	Memory[x6 + 40] = x5	Doubleword from register to memory
	Load word	lw x5,40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	lwu x5,40(x6)	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	sw x5,40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5,40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5,40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5,40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5,40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5,40(x6)	x5 = Memory[x6 + 40]	Byte halfword from memory to register
	Store byte	sb x5,40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20	Bit-by-bit OR reg. with consta
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant

RISC-V Assembly

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Shift	Shift left logical	`sll x5, x6, x7`	x5 = x6 << x7	Shift left by register
	Shift right logical	`srl x5, x6, x7`	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	x5 = x6 >> 3	Arithmetic shift right by immediate
Conditional branch	Branch if equal	`beq x5, x6, 100`	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greatr/eq, unsigned	`bgeu x5, x6, 100`	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
Unconditional branch	Jump and link	`jal x1, 100`	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	x1 = PC+4; go to x5+100	Procedure return; indirect call

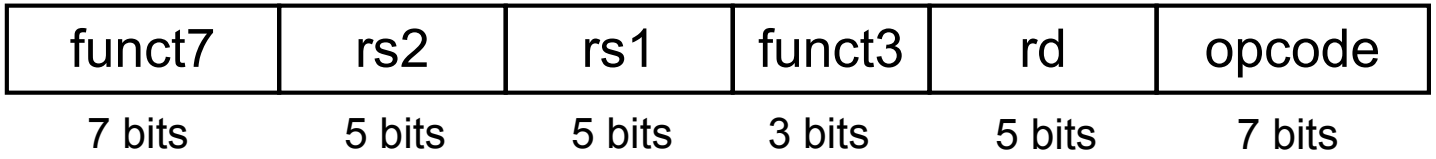
RISC-V: 6 instruction formats

- R-Format: instructions using 3 register inputs
  - add, xor — arithmetic/logical ops
- I-Format: instructions with immediates, loads
  - addi, lw
- S-Format: store instructions: sw, sb
- SB-Format: branch instructions: beq, bge
- U-Format: instructions with upper immediates
  - lui — upper immediate is 20-bits
- UJ-Format: the jump instruction: jal

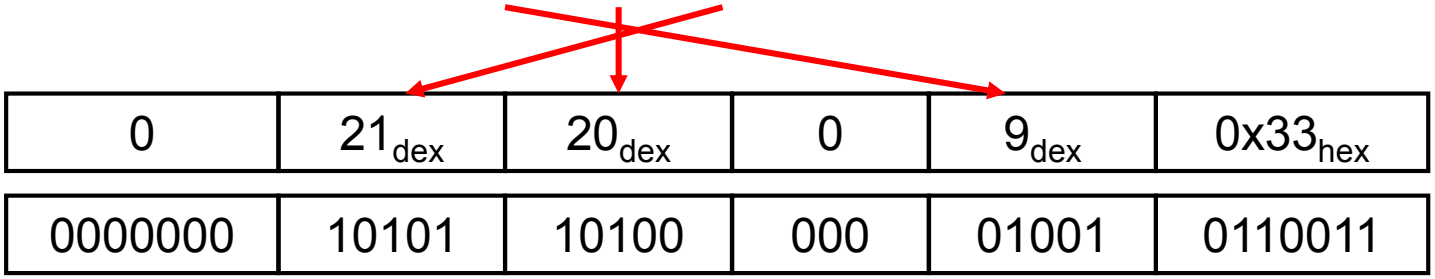
very similar to mips

Name (Field Size)	Field						Comments
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

RISC-V R-format Instructions

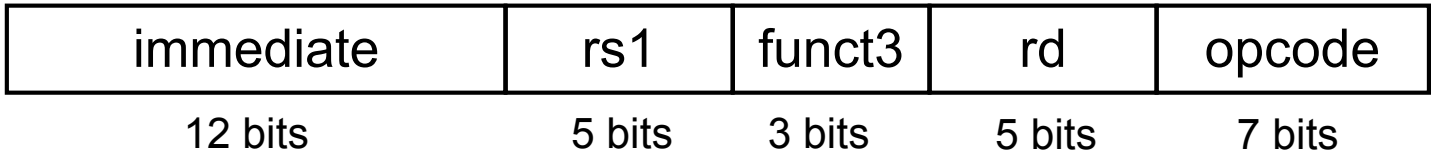


- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)
- Example: add x9,x20,x21

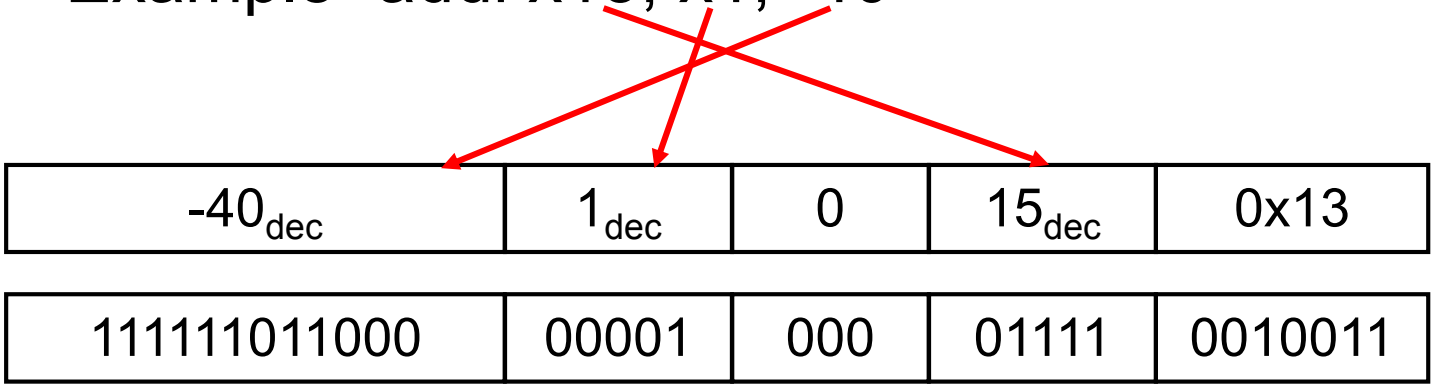


$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

RISC-V I-format Instructions

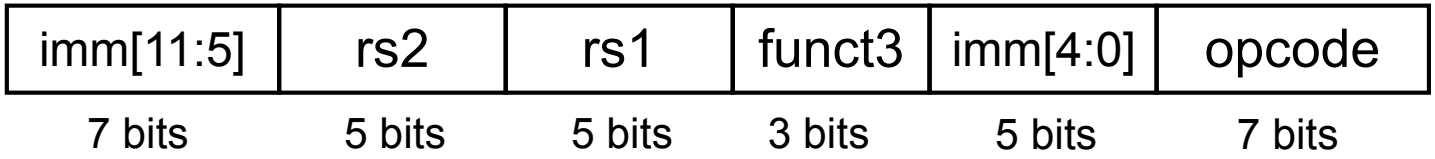


- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended
- Example addi x15, x1, -40

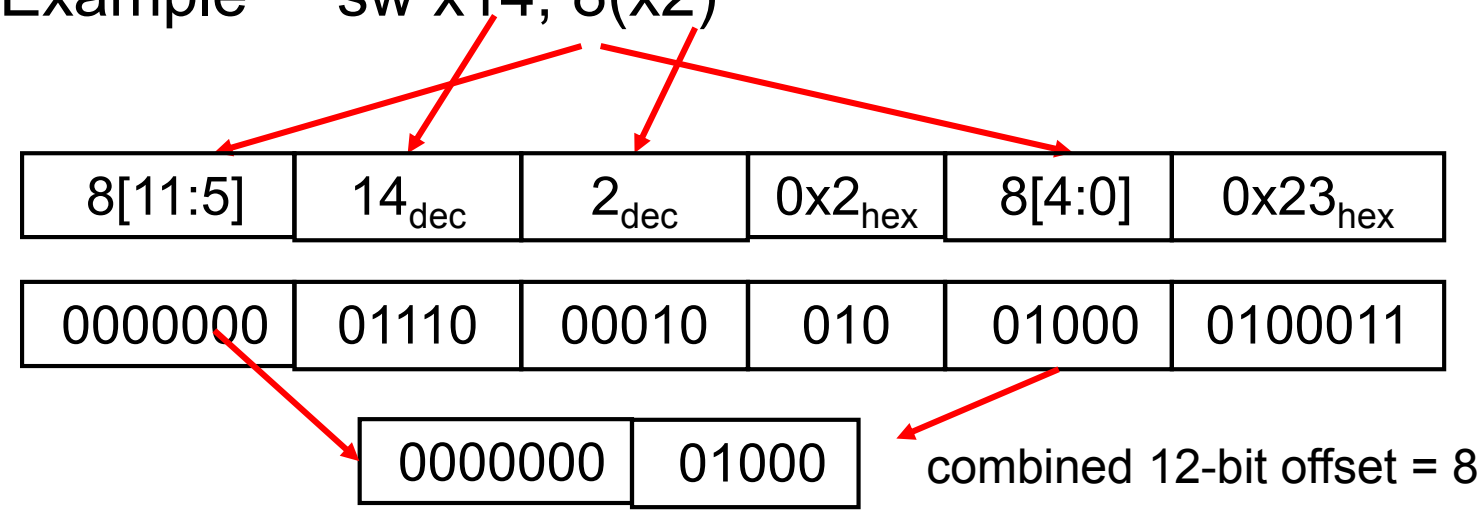


0xFD808793

RISC-V S-format Instructions

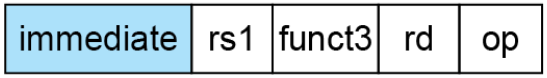


- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place
- Example sw x14, 8(x2)



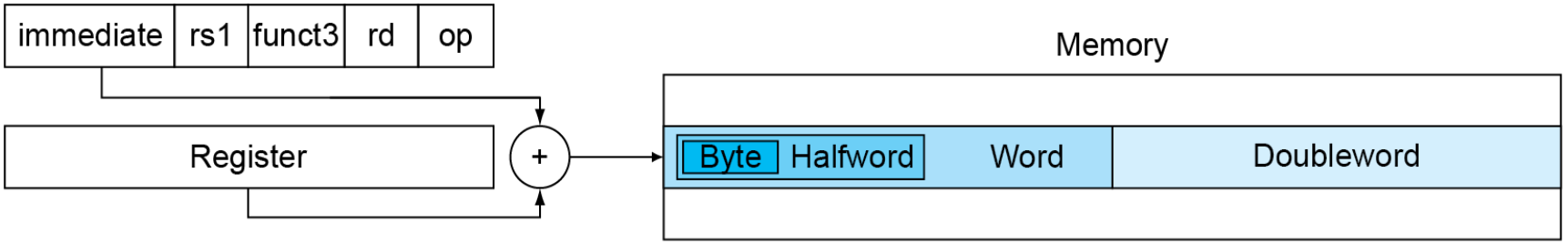
RISC-V Addressing Summary

1. Immediate addressing

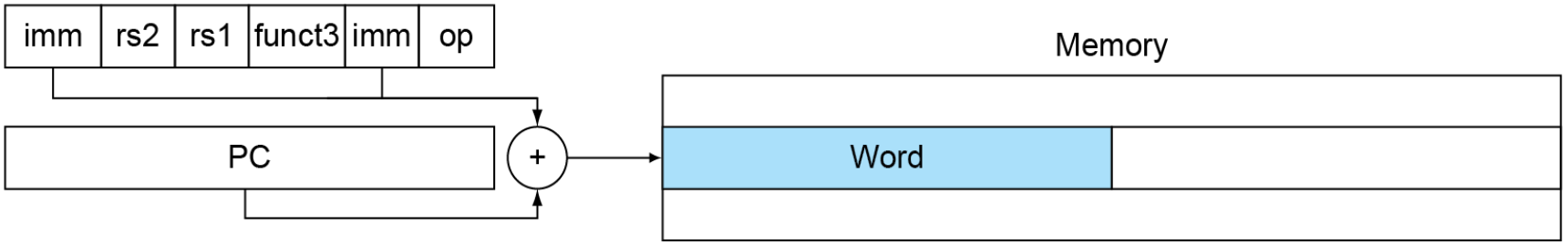


2. Register addressing

3. Base addressing



4. PC-relative addressing



MIPS & RISC-V Instruction Formats

Register-register

	31–25	24–20	19–15	14–12	11–7	6–0
RISC-V	funct7(7)	rs2(5)	rs1(5)	funct3(3)	rd(5)	opcode(7)

	31–26	25–21	20–16	15–11	10–6	5–0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Rd(5)	Const(5)	Opx(6)

Load

	31–20	19–15	14–12	11–7	6–0
RISC-V	immediate(12)	rs1(5)	funct3(3)	rd(5)	opcode(7)

	31–26	25–21	20–16	15–0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Const(16)

Store

	31–25	24–20	19–15	14–12	11–7	6–0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)

	31–26	25–21	20–16	15–0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Const(16)

Branch

	31–25	24–20	19–15	14–12	11–7	6–0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)

	31–26	25–21	20–16	15–0
MIPS	Op(6)	Rs1(5)	Opx/Rs2(5)	Const(16)

ARM Market share

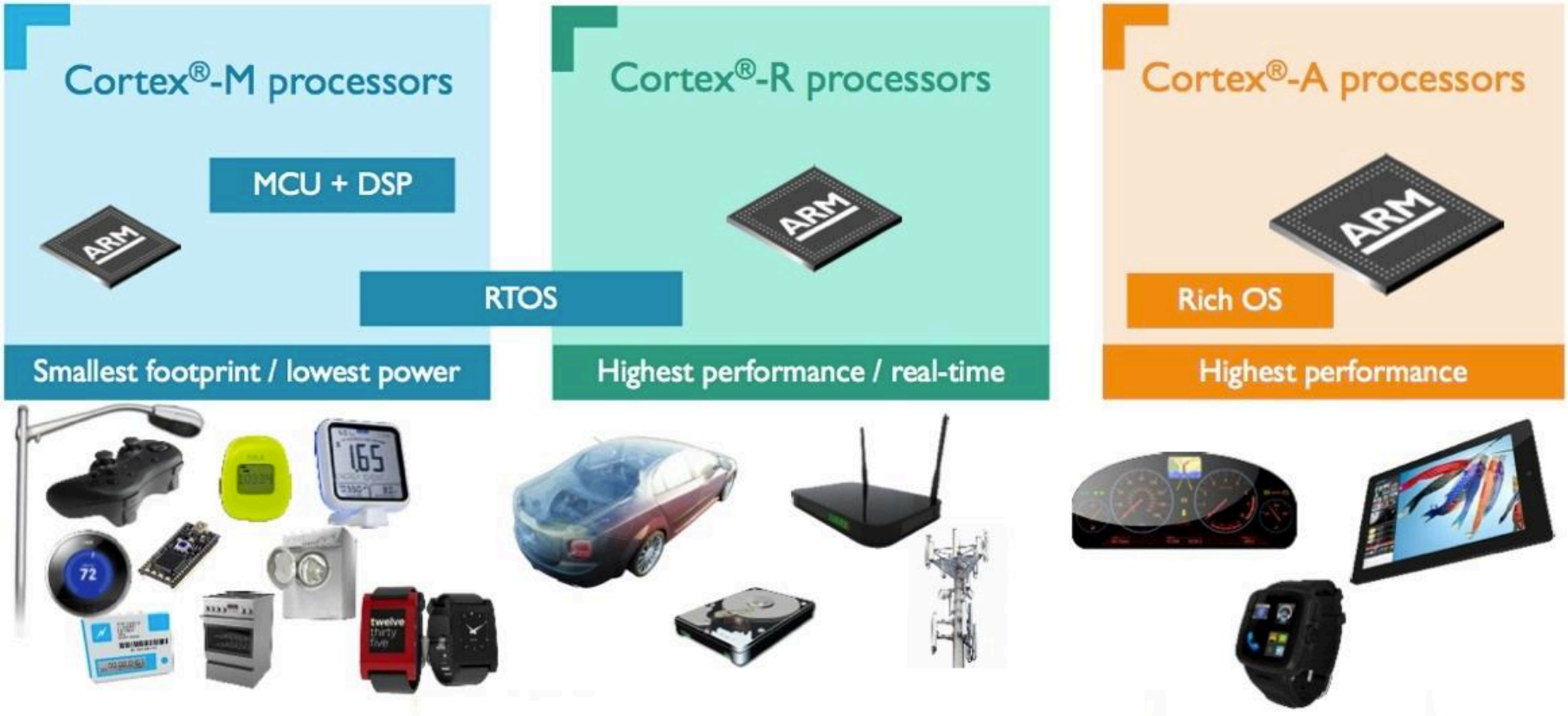
Markets for ARM in 2012

	Devices Shipped (Million of Units)	2012 Devices	Chips/ Device	TAM 2012 Chips	2012 ARM	2012 Share
Mobile	Smart Phone	730	3-5	2,500	2,200	90%
	Feature Phone	460	2-3	1,200	1,100	95%
	Low End Voice	730	1-2	730	700	95%
	Portable Media Players	130	1-3	250	220	90%
	Mobile Computing* (apps only)	400	1	400	160	40%
Home	Digital Camera	150	1-2	230	180	80%
	Digital TV & Set-top-box	420	1-2	640	290	45%
Enterprise	Desktop PCs & Servers (apps)	200	1	200	-	0%
	Networking	1,200	1-2	1,300	420	35%
	Printers	120	1	120	85	70%
	Hard Disk & Solid State Drives	700	1	700	620	90%
Embedded	Automotive	2,600	1	2,600	210	8%
	Smart Card	6,000	1	6,000	710	13%
	Microcontrollers	8,700	1	8,700	1,500	18%
	Others **	2,000	1	2,000	300	15%
	Total	**25,500**		**27,000**	**8,700**	**32%**

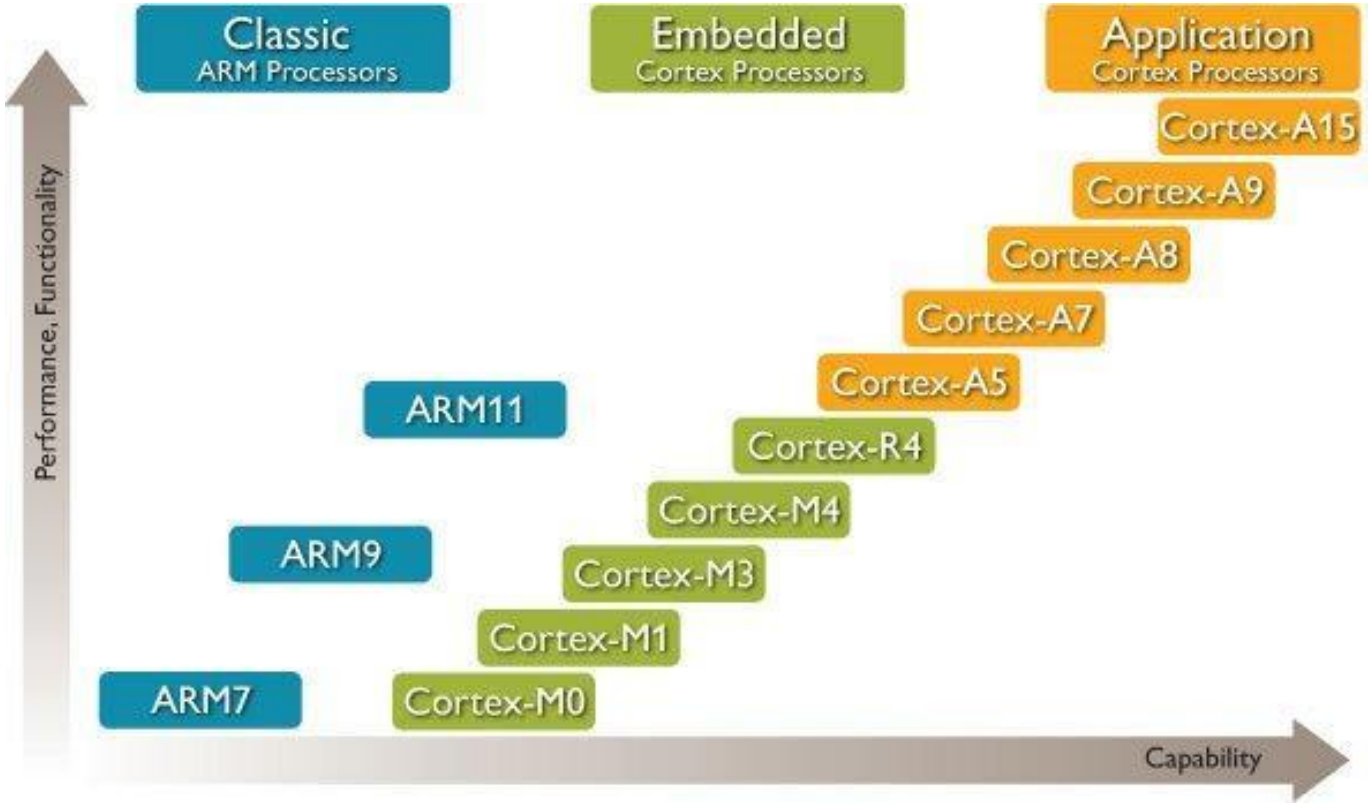
Year	Market Share
2007	17%
2008	20%
2009	22%
2010	25%
2011	29%
2012	32%

Source: Gartner, IDC, SIA, and ARM estimates

ARM Applications



ARM CPU Series



ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

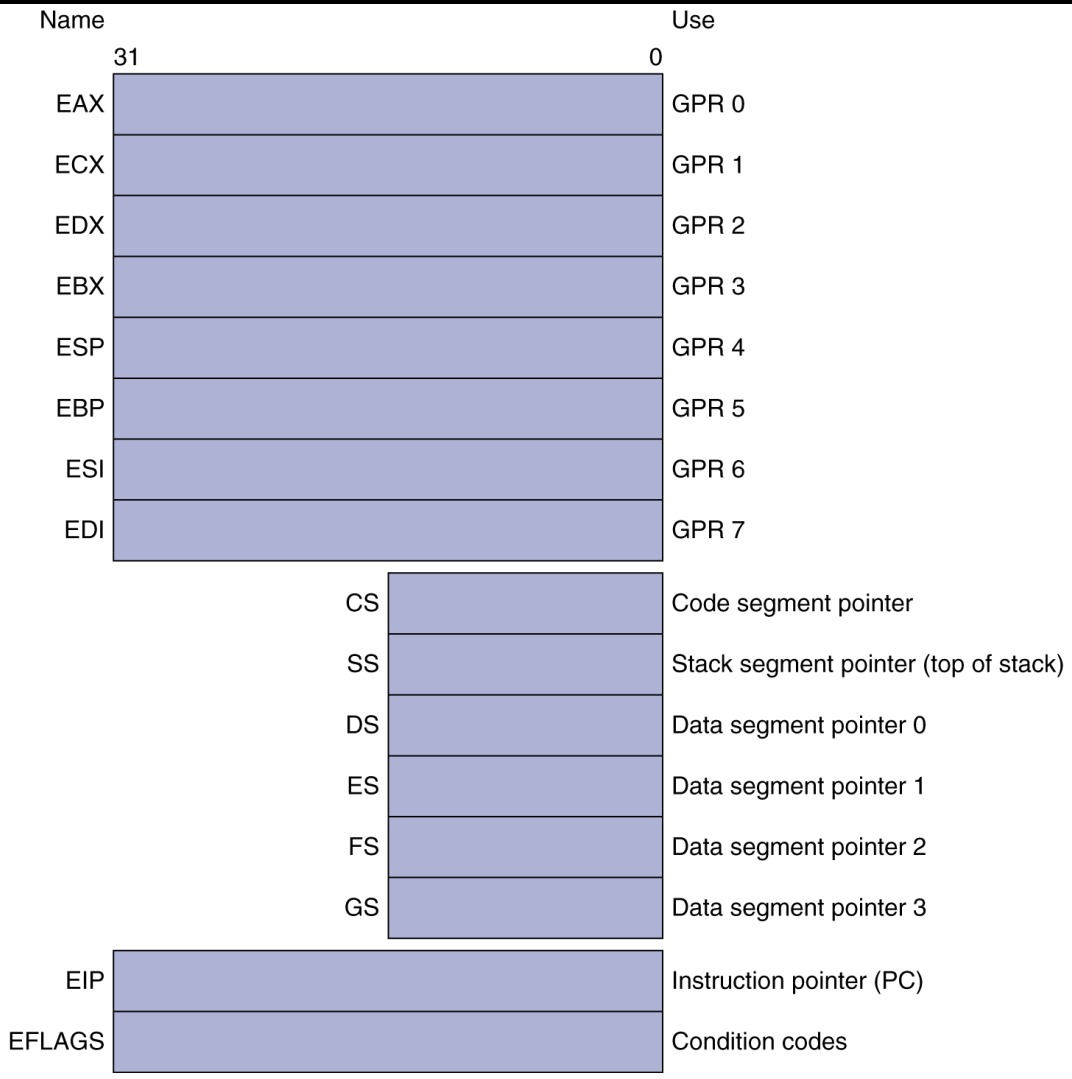
The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Basic x86 Registers



Concluding Remarks

- 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