CS202 : COMPUTER ORGANIZATION

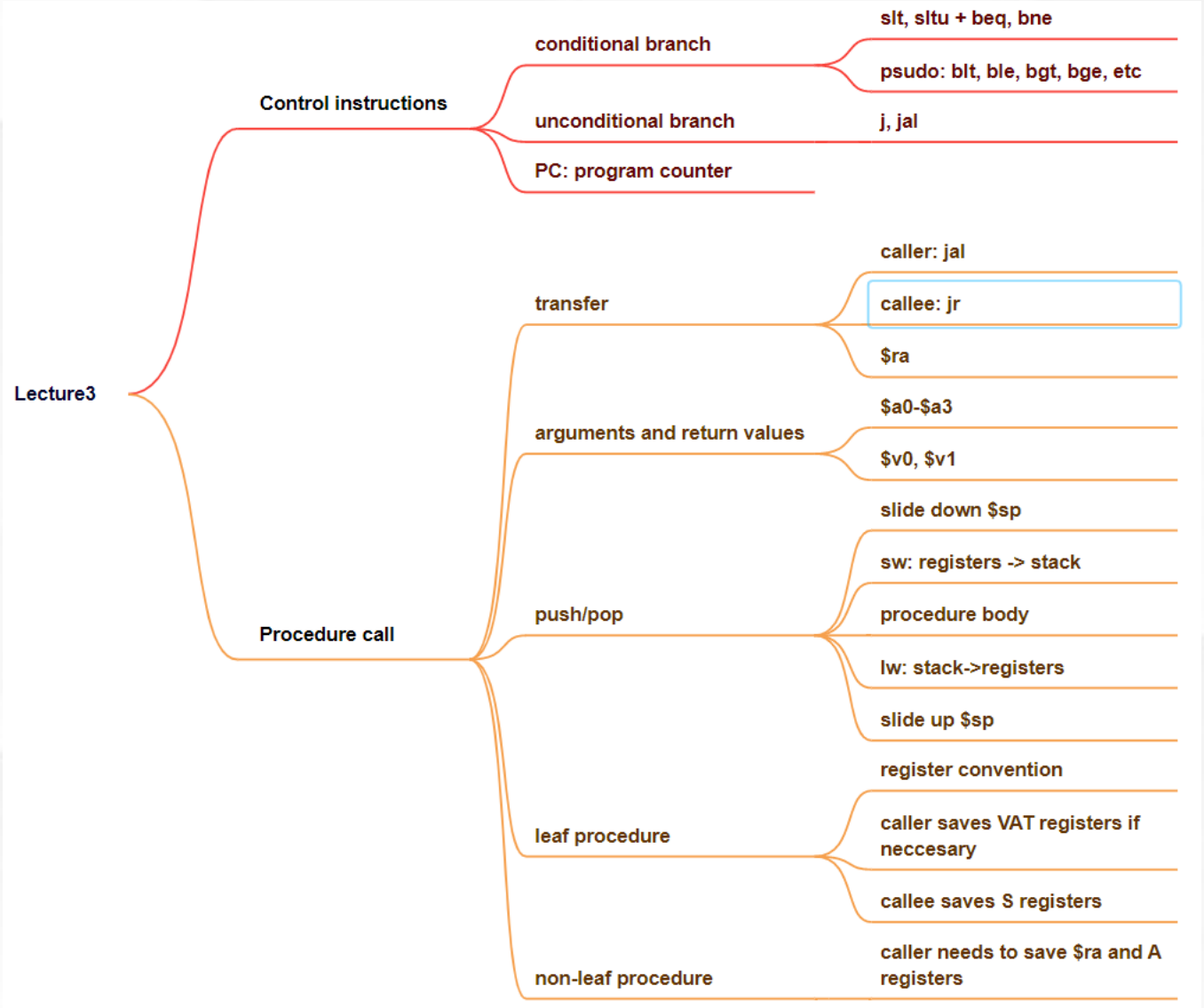
Lecture 4
Instruction Set Architecture (3)

2023 Spring

Today's Agenda

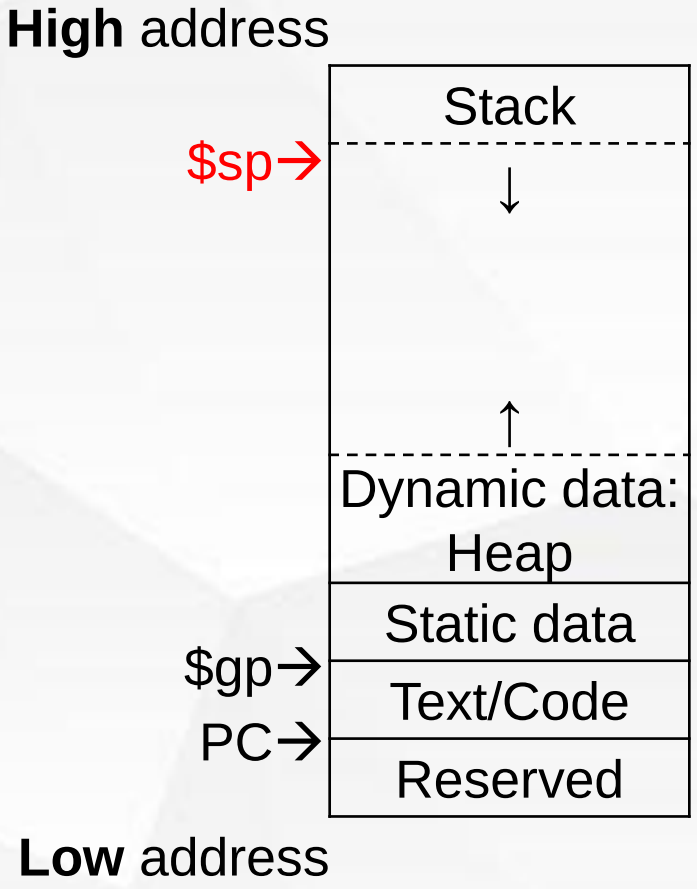
- Recap
  - More control instructions
  - Procedure Call
- Context
  - MIPS addressing
  - Translating and starting a program
  - Other popular ISAs (RISC-V)
- Reading: Textbook 2.10, 2.12-13, 2.19
- Self-study: The rest sections of Chapter 2

Recap


Lecture3
Control instructions
conditional branch
slt, sltu + beq, bne
psudo: blt, ble, bgt, bge, etc
unconditional branch
j, jal
PC: program counter
Procedure call
transfer
caller: jal
callee: jr
$ra
arguments and return values
$a0-$a3
$v0, $v1
push/pop
slide down $sp
sw: registers -> stack
procedure body
lw: stack->registers
slide up $sp
leaf procedure
register convention
caller saves VAT registers if neccesary
callee saves S registers
non-leaf procedure
caller needs to save $ra and A registers

Memory Layout summary

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



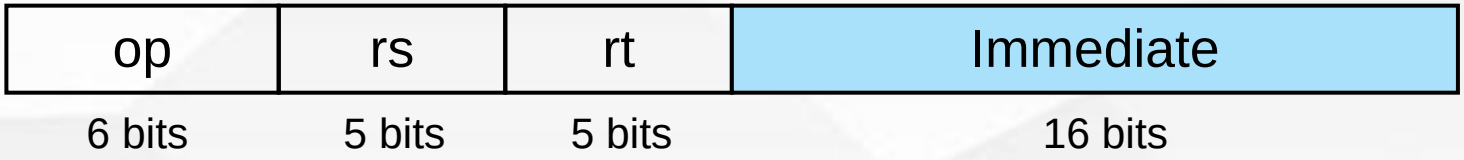
MIPS Addressing

- Addressing: how the instructions identify the operands of the instruction.
- MIPS Addressing mode:
  - Register addressing
  - Immediate addressing
  - Base/Displacement addressing
  - PC-relative addressing
  - Pseudo-direct addressing

R-format → Register Addressing

- Using register as the operand
- E.g.,
  - Arithmetic: add, sub…
  - Logical: and, or, sll, srl…
  - Control: slt…

I-format → Immediate Addressing



- For instructions including immediate
  - E.g. addi, andi, ori
- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
  - lui rt, constant (Load upper immediate)
    - Copies 16-bit constant to left 16 bits of rt
    - Clears right 16 bits of rt to 0

```
lui $s0, 61
```

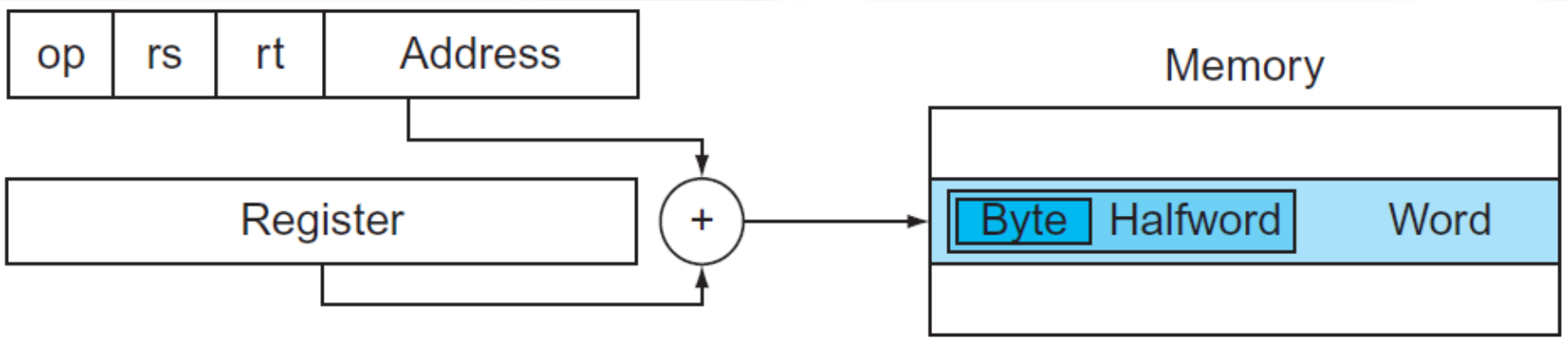
0000 0000 0011 1101 0000 0000 0000 0000

```
ori $s0, $s0, 2304
```

0000 0000 0011 1101 0000 1001 0000 0000

I-format → Base Addressing

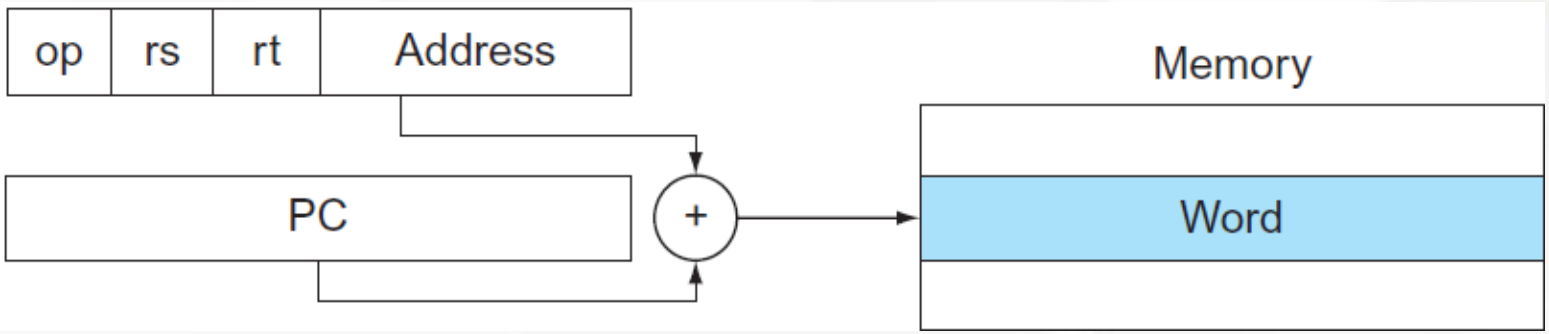
- Also called Displacement Addressing
- The operand is at the memory, whose address is the sum of a register and a constant lw/lh/lb/sw/sh/sb
  - e.g. lw $s0, 4($s1)
  - op: 100011, rs: 10001 (address of s1), rt: 10000 (address of s0), offset: 100 (4)
  - Instruction:



I-format → PC-relative addressing(1)

- Branch instructions specify
  - Opcode, two registers, target address, beq/bne
  - e.g. beq $s0, $s1, label
- Most branch targets are near branch
  - Forward or backward
- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

```
Addr. Inst.
 0    beq $t0,$t1,L1
 4    lw …
 8    add…
 12   L1:…
```

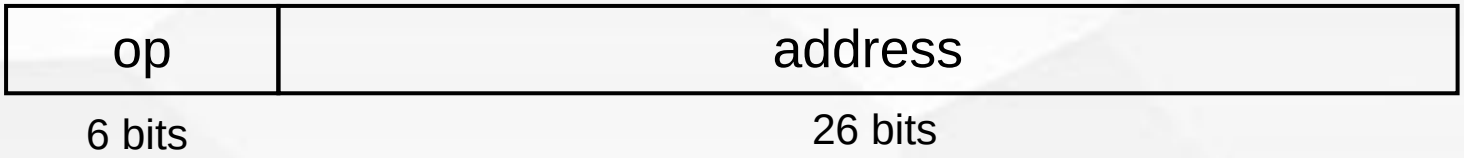


I-format → PC-relative addressing(2)

- Branch Addressing: Use the address field as a two's complement word offset to PC
  - Branches generally change the PC by a small amount
  - Can specify $\pm 2^{15}$ aligned word addresses from the PC

- Why not use byte address offset from PC?
  - MIPS Instructions are "word-aligned": Address is always a multiple of 4 (in bytes)
  - PC always points to an instruction
  - Instead of specifying $\pm 2^{15}$ bytes from the PC, we will now specify $\pm 2^{15}$ words = $\pm 2^{17}$ byte addresses around PC

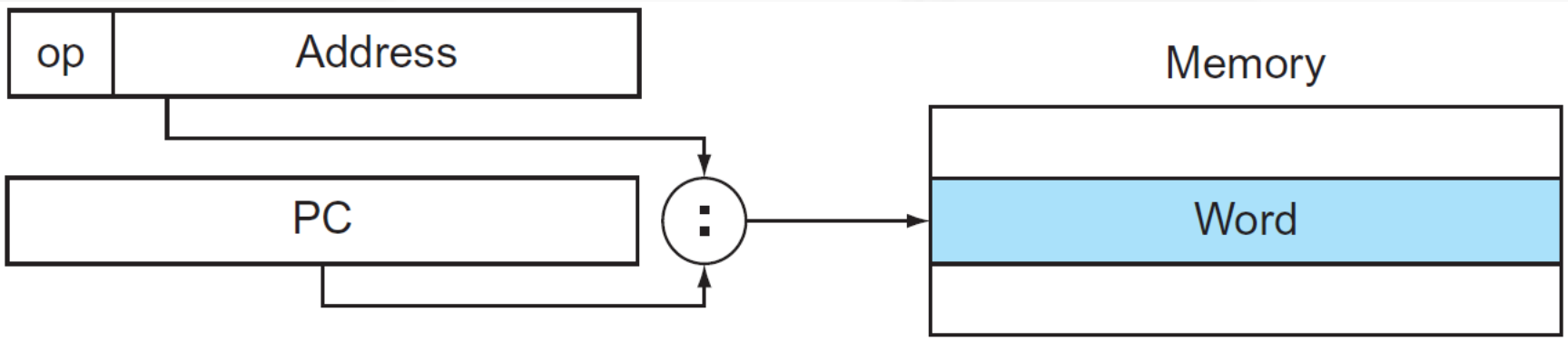
J-format → Pseudodirect addressing(1)



- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction
- Addressing in jumps
  - why absolute addressing fails?
  - solution #1: count in words
  - solution #2: invent a new format (J-type)
  - solution #3: borrow 4-bits from PC

J-format → Pseudodirect addressing(2)

- Jump Addressing
  - the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC
  - Target address = PC[31…28] : (address × 4)



- limit on code size (address boundary)
  - 26 bits word address = 28 bits byte address
    = 64 MWords = 256 MB

Target Addressing Example

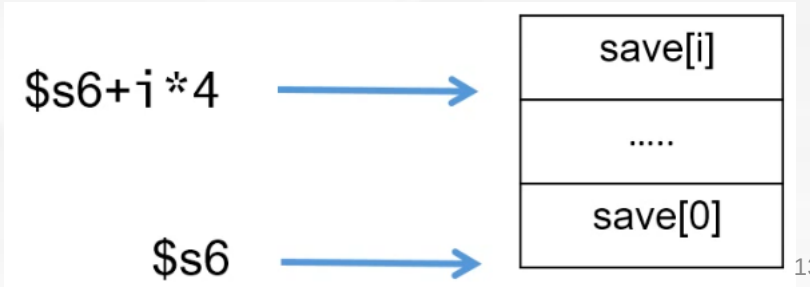
- C code

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

Register assignment:
i($s3), k ($s5), address of save ($s6)

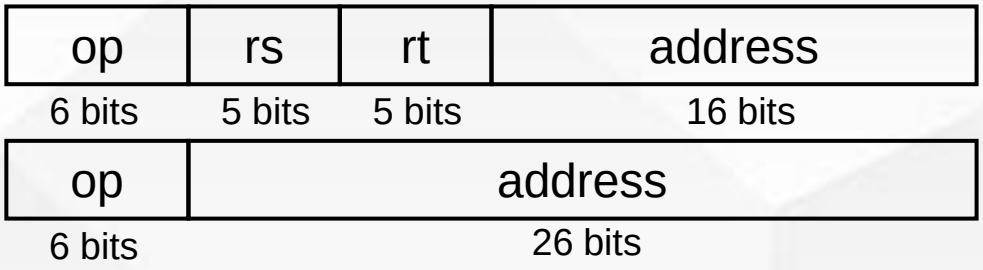
- MIPS code

```
Loop: sll  $t1, $s3, 2      # Temp reg $t1 = i*4
      add  $t1, $t1, $s6    # $tl = address of save[i]
      lw   $t0, 0($t1)      # Temp reg $t0 = save[i]
      bne  $t0, $s5, Exit   # go to Exit if save[i]!=k
      addi $s3, $s3, 1      # i=i+1
      j Loop                # go to Loop
Exit: …
```



Target Addressing Example

- bne
- j



bne instr. target addr.
= PC+ address x 4

j instr. target addr.
= PC[31...28] :(address x 4)

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: …
```

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	?		
80016	8	19	19	1		
80020	2	?				
80024						

- 80024 = PC+4 +address x 4=80012 +4+2x4
- 80000 = PC[31…28]:(address x 4) =80020[31...28]:(address x 4)=0:20000x4

Branching Far Away

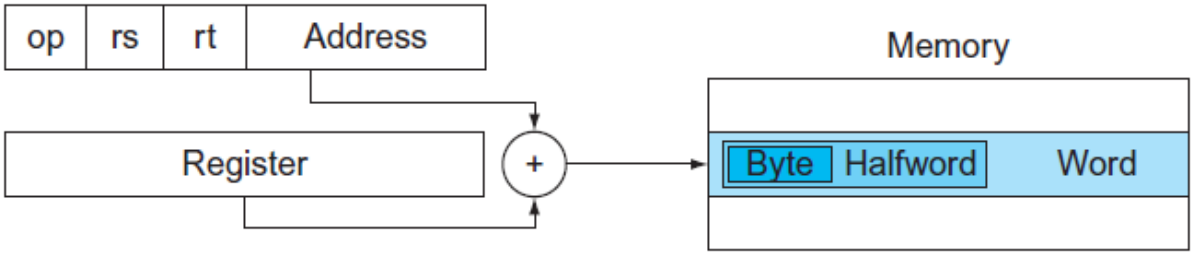
- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

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

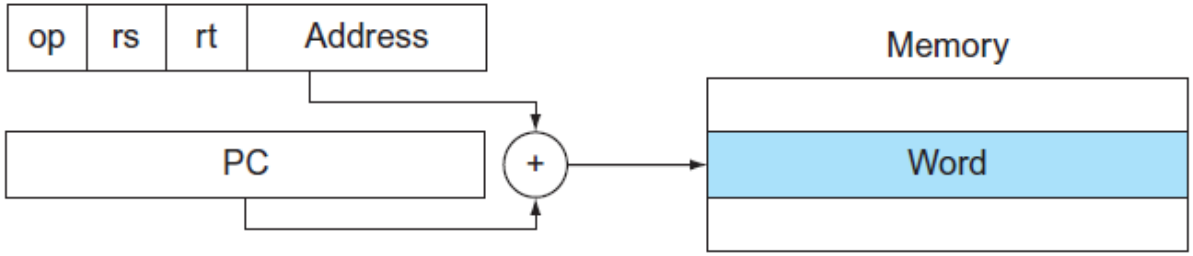
1. Immediate addressing

2. Register addressing

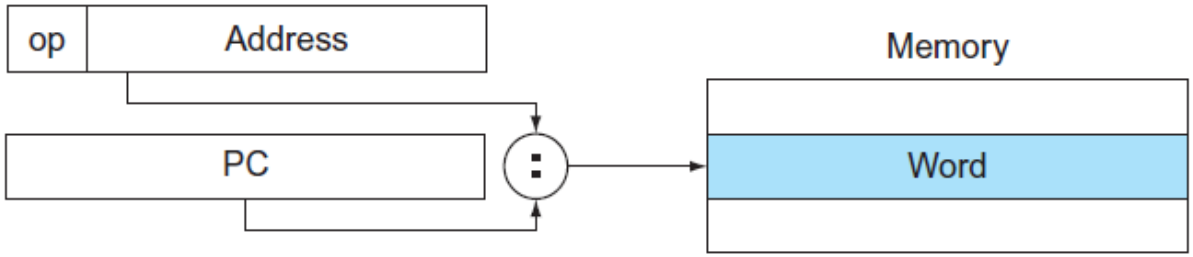
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



Decoding Machine Code

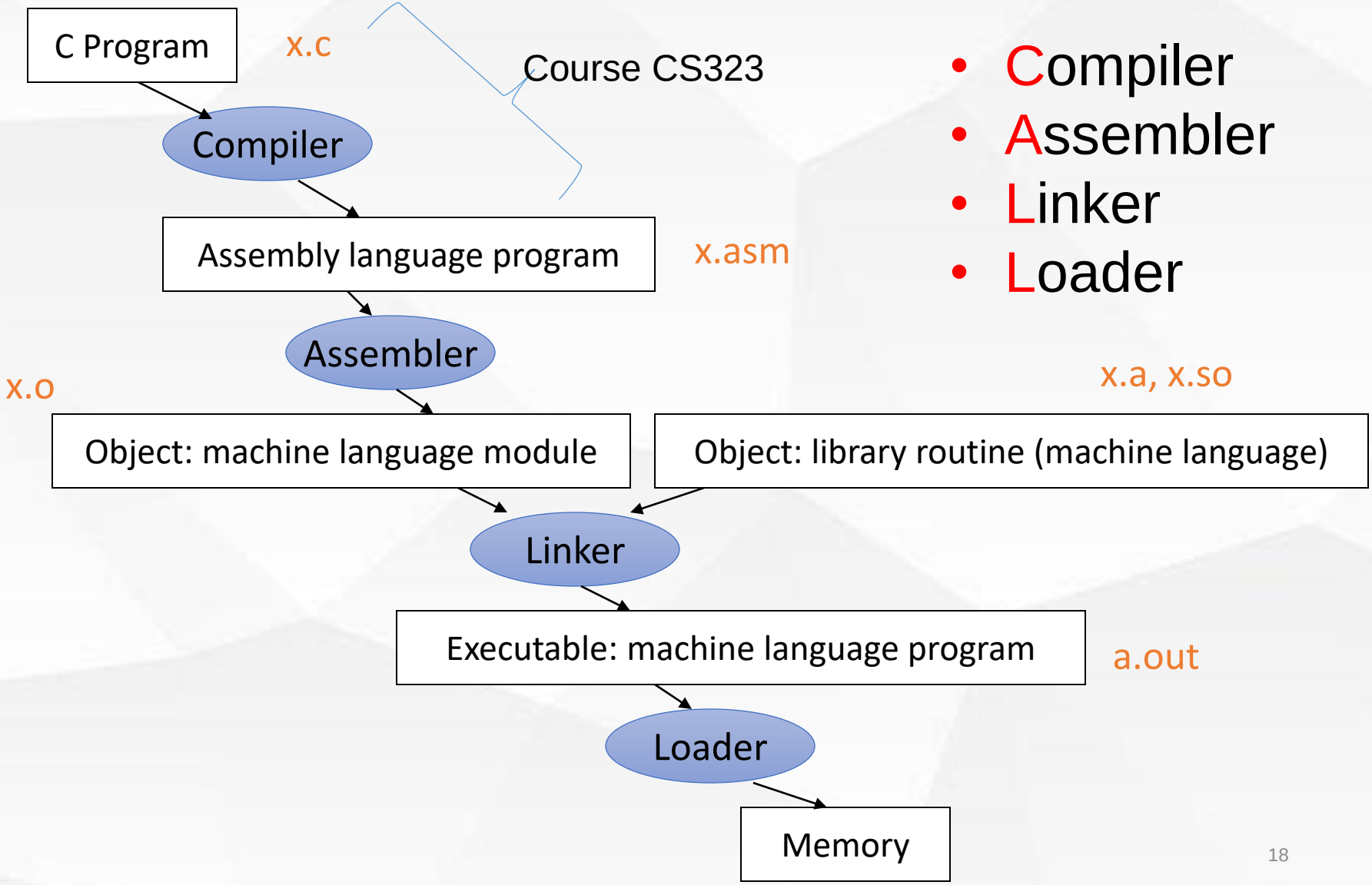
- What is the assembly language of the following machine instruction?

  0x00af8020

- Hex to bin: 0000 0000 1010 1111 1000 0000 0010 0000
  - op 000000 → R-format
  - rs 00101 → 5_{10}
  - rt 01111 → 15_{10}
  - rd 10000 → 16_{10}
  - shamt 00000
  - func 100000
- Get the instruction: add $s0, $a1, $t7

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

Running a Program: *CALL*



- Compiler
- Assembler
- Linker
- Loader

Translation vs. Interpretation

- How do we run a program written in a source language?
  - Interpreter: Directly executes a program in the source language, e.g. Python
    - closer to high-level, but slower
  - Translator: Converts a program from the source language to an equivalent program in another language, e.g. C
    - more efficient

- Translation/compilation helps “hide” the program “source” from the users
  - One model for creating value in the marketplace (e.g. Microsoft keeps all their source code secret)
  - Alternative model, “open source”, creates value by publishing the source code and fostering a community of developers

Compiler

- Input: Higher-level language code
  - e.g. C, Java in files such as foo.c
- Output: Assembly Language Code
  - e.g. foo.asm
- The output may contain pseudo-instructions
- Optimizing compilers today can produce assembly language programs nearly as well as an assembly language expert, and sometimes even better for large programs.

Assembler

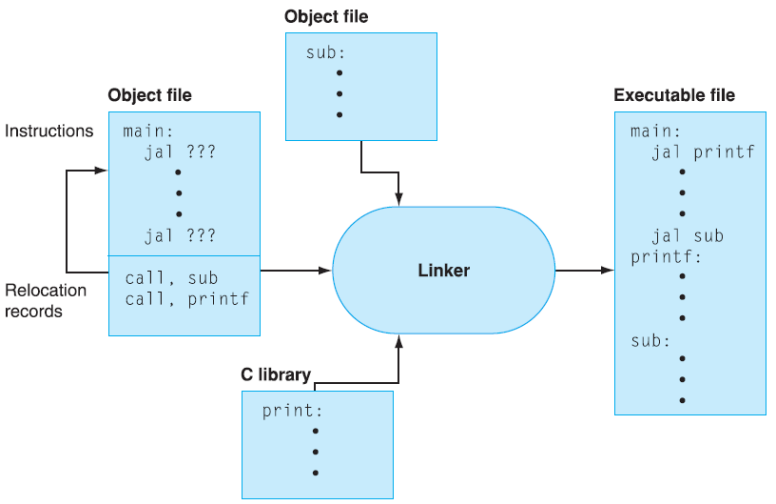
- Input: Assembly language code
  - e.g. foo.asm
- Output: Object code (True Assembly), information tables
  - e.g. foo.o – Object file
- Convert pseudo-instructions into actual hardware instructions
  - e.g. move $t0, $t1 → add $t0, $zero, $t1
    blt $t0, $t1, L → slt $at, $t0, $t1
    bne $at, $zero, L

  $at (register1) reserved for assembler

- Convert assembly instructions into machine instructions
  - compute the actual values for instruction labels
  - maintain info on external references and debugging information

Linker

- Input: Object Code files, information tables
  - e.g. foo.o, lib.o
- Output: Executable Code
  - e.g. a.out
- 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



Loader

- Input: Executable Code
  - e.g. a.out
- Output: <program is run>
- Executable files are stored on disk
- When one is run, loader's job is to load it into memory and start it running
- In reality, loader is the operating system (OS)
  - loading is one of the OS tasks

Full Example: Sort in C

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

```
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;
}
```

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 Procedure Body

```
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); }}}
```

```
          move $s2, $a0           # save $a0 into $s2
          move $s3, $a1           # save $a1 into $s3
          move $s0, $zero         # i = 0
for1tst:  slt  $t0, $s0, $s3      # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
          beq  $t0, $zero, exit1  # go to exit1 if $s0 ≥ $s3 (i ≥ n)
          addi $s1, $s0, -1       # j = i - 1
for2tst:  slti $t0, $s1, 0        # $t0 = 1 if $s1 < 0 (j < 0)
          bne  $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
          sll  $t1, $s1, 2        # $t1 = j * 4
          add  $t2, $s2, $t1      # $t2 = v + (j * 4)
          lw   $t3, 0($t2)        # $t3 = v[j]
          lw   $t4, 4($t2)        # $t4 = v[j + 1]
          slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
          beq  $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3
          move $a0, $s2           # 1st param of swap is v (old $a0)
          move $a1, $s1           # 2nd param of swap is j
          jal  swap               # call swap procedure
          addi $s1, $s1, -1       # j -= 1
          j    for2tst            # jump to test of inner loop
exit2:    addi $s0, $s0, 1        # i += 1
          j    for1tst            # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

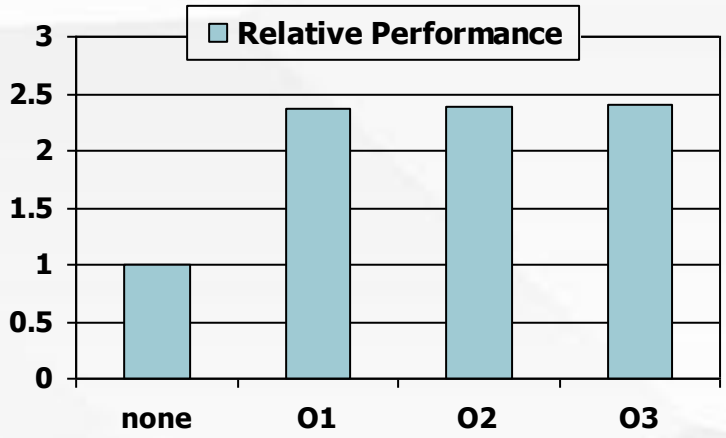
Outer loop

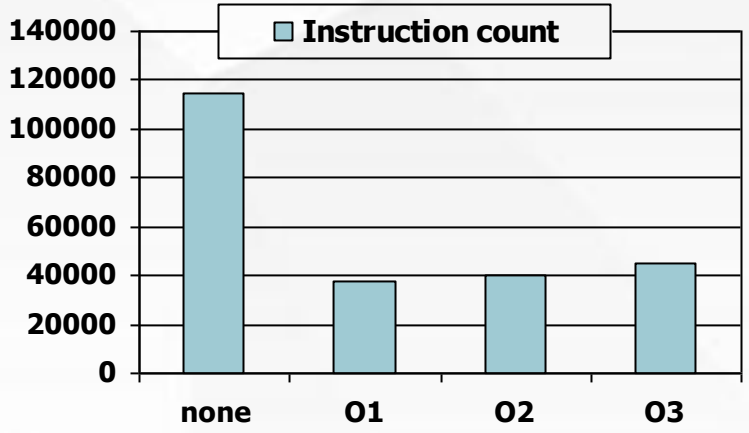
The Full Procedure

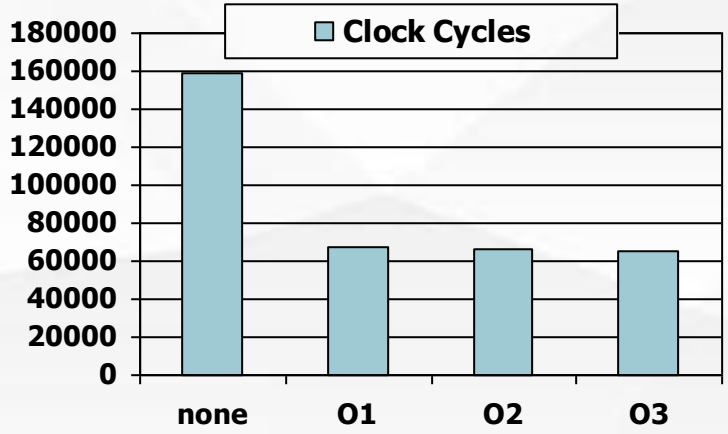
```
sort:    addi $sp,$sp, -20       # make room for 5 registers
         sw $ra, 16($sp)         # save $ra on stack
         sw $s3,12($sp)          # save $s3 on stack
         sw $s2, 8($sp)          # save $s2 on stack
         sw $s1, 4($sp)          # save $s1 on stack
         sw $s0, 0($sp)          # save $s0 on stack
         …                       # procedure body
         …
         exit1: lw $s0, 0($sp)   # restore $s0 from stack
         lw $s1, 4($sp)          # restore $s1 from stack
         lw $s2, 8($sp)          # restore $s2 from stack
         lw $s3,12($sp)          # restore $s3 from stack
         lw $ra,16($sp)          # restore $ra from stack
         addi $sp,$sp, 20        # restore stack pointer
         jr $ra                  # return to calling routine
```

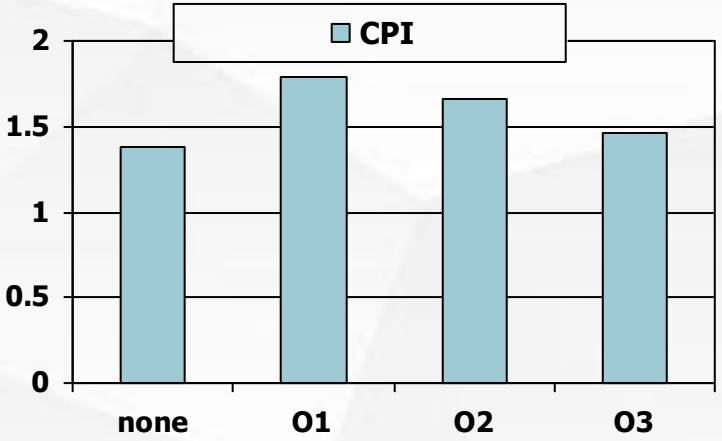
Effect of Compiler Optimization

- Compiled with gcc for Pentium 4 under Linux

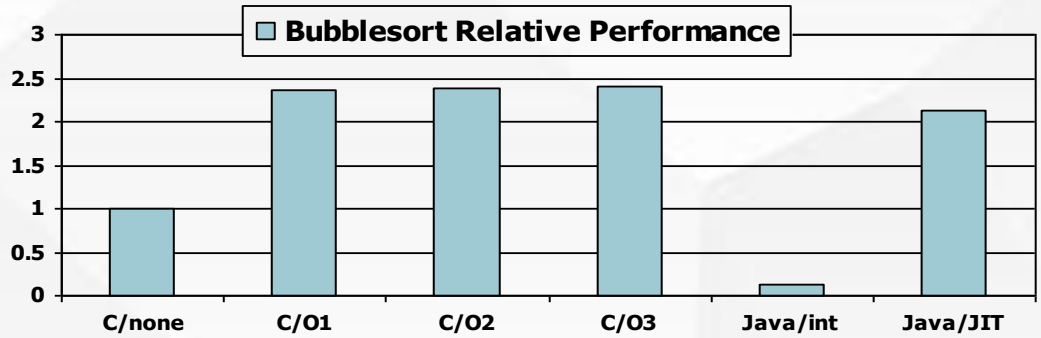


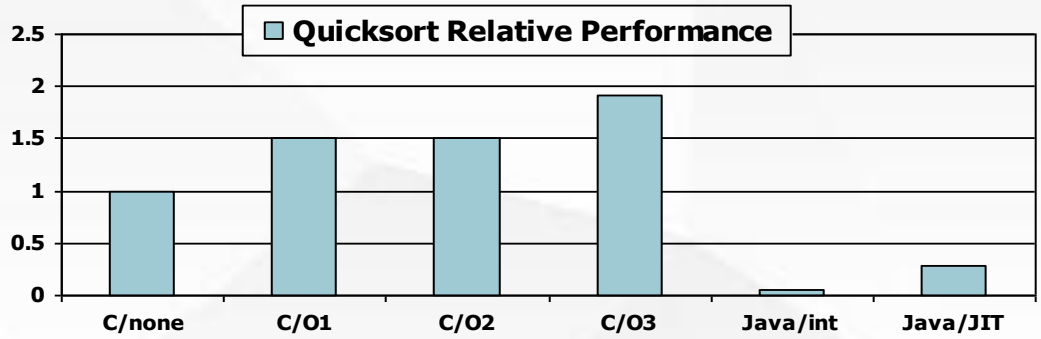


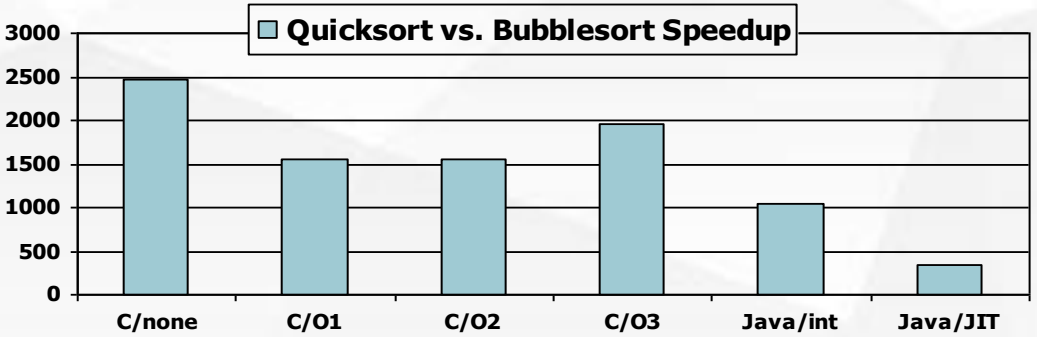




Effect of Language and Algorithm







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!

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)

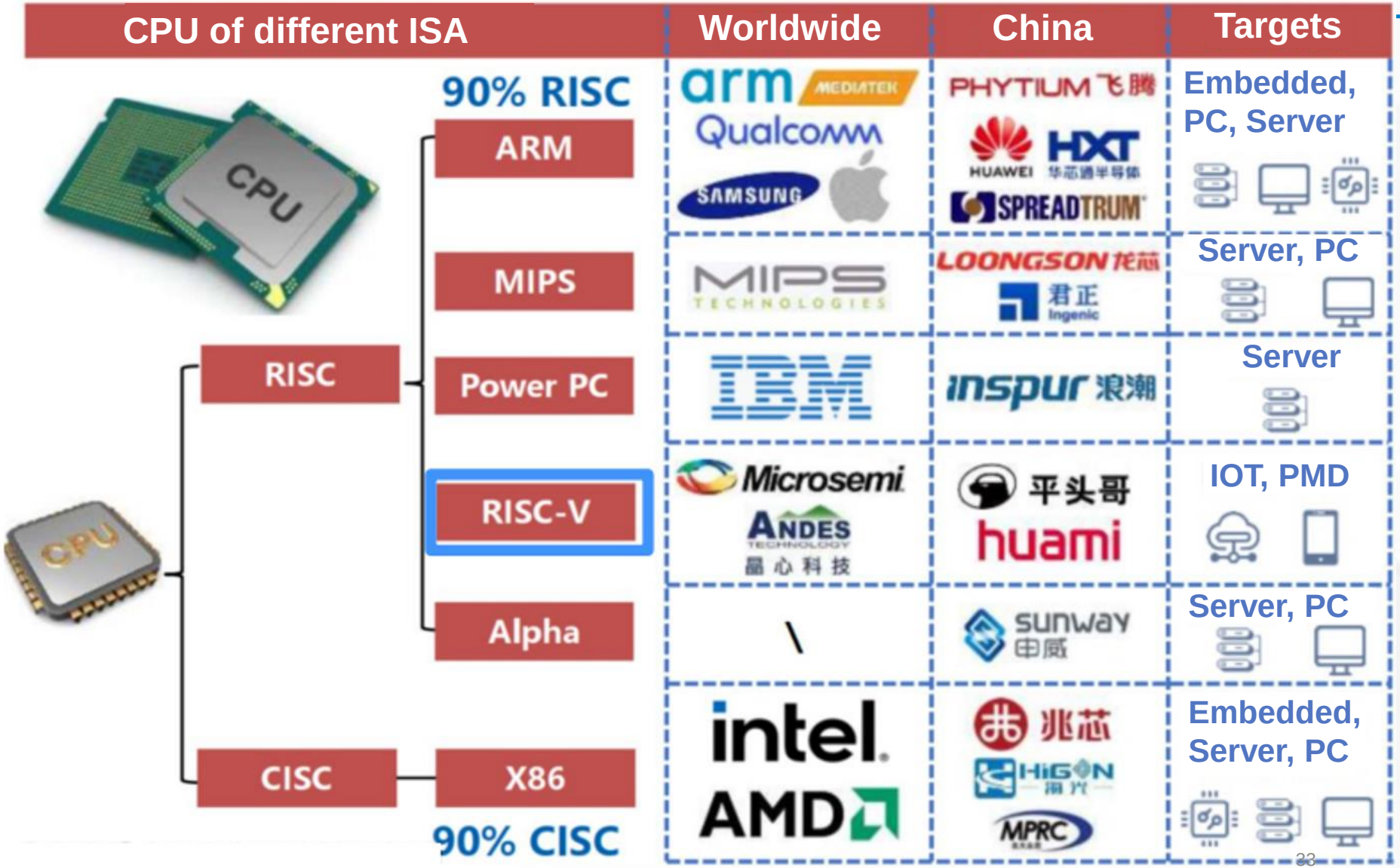
Smartphone-like devices (iPhone, Android), Raspberry Pi, Embedded systems
Apple M series

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

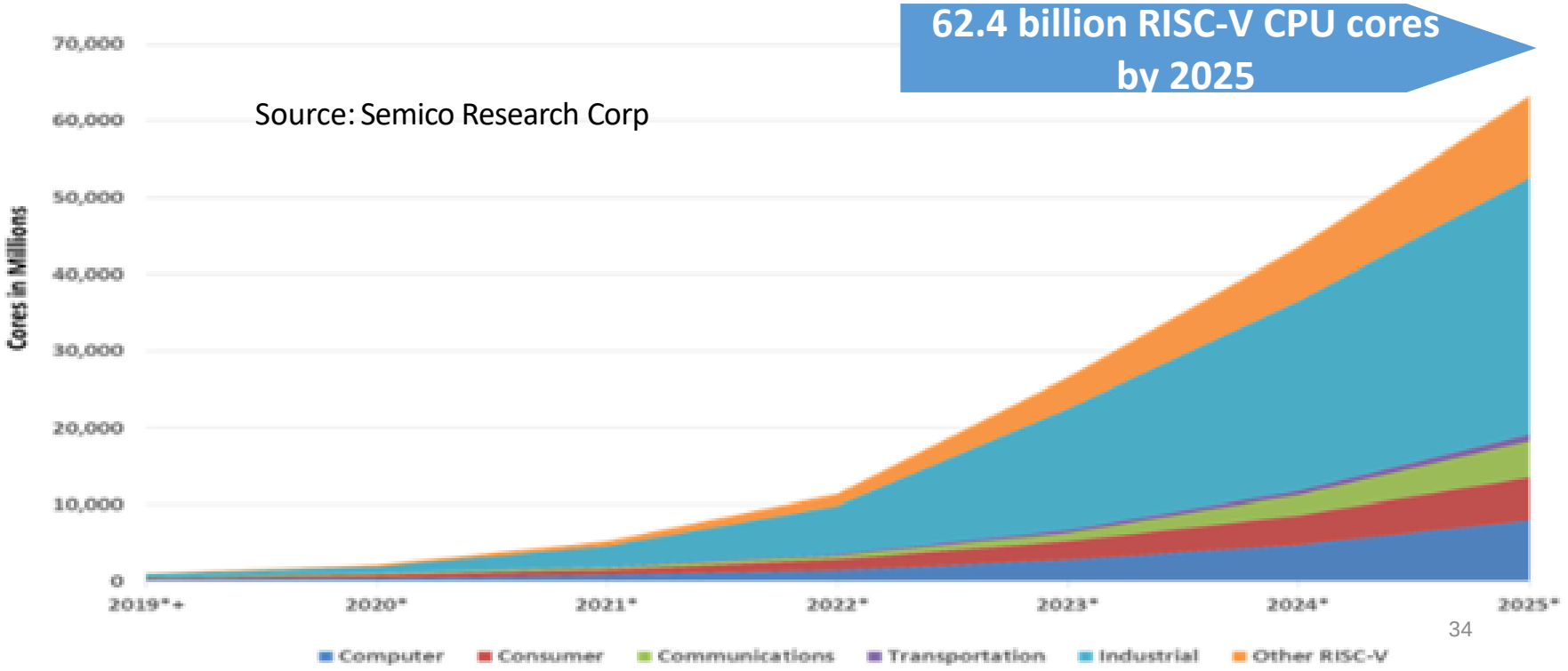
RISC-V among ISAs



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

--- Gartner, June 2020



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

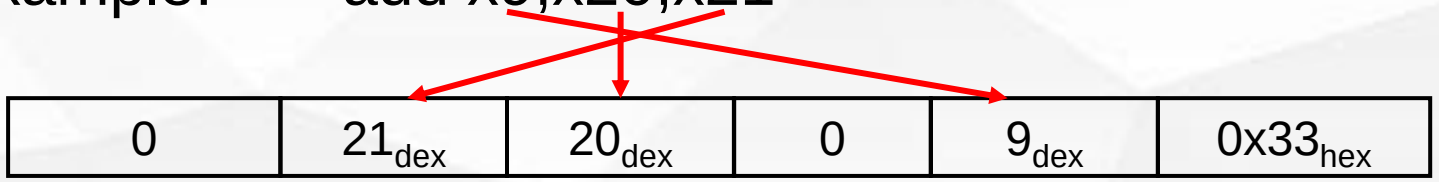
- R-Format: instructions using 3 register inputs
  - add, xor — arithmetic/logical ops
- I-Format: instructions with immediates, loads
  - addi, lw
- S-Format: store instructions: sw, sb
- SB-Format: branch instructions: beq, bge
- U-Format: instructions with upper immediates
  - lui — upper immediate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

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
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

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- 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



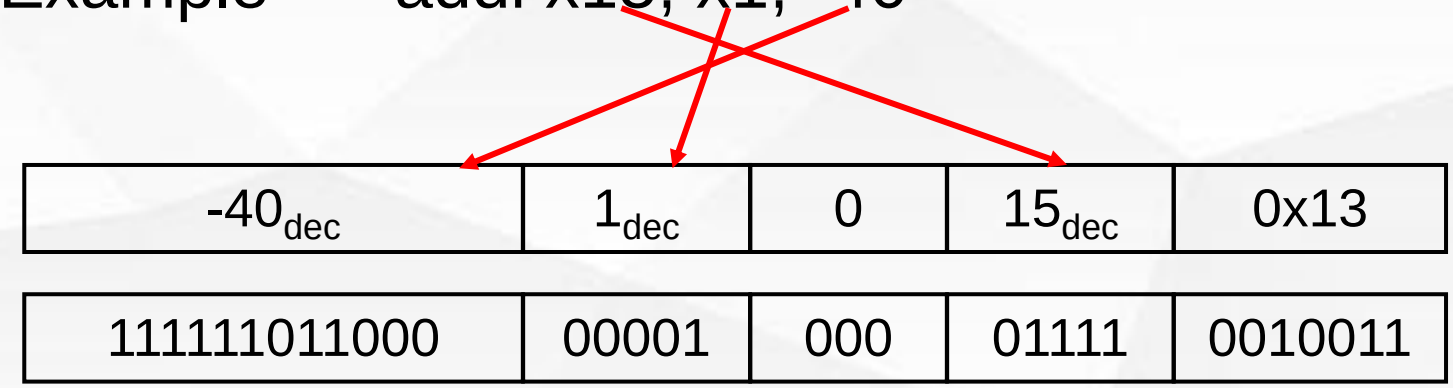
0	21_{dex}	20_{dex}	0	9_{dex}	$0x33_{hex}$
0000000	10101	10100	000	01001	0110011

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

RISC-V I-format Instructions

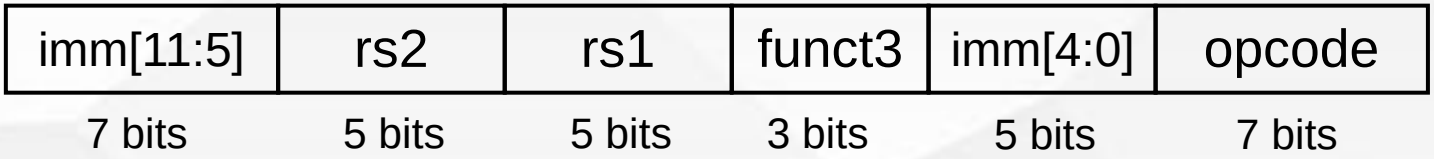
immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- 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

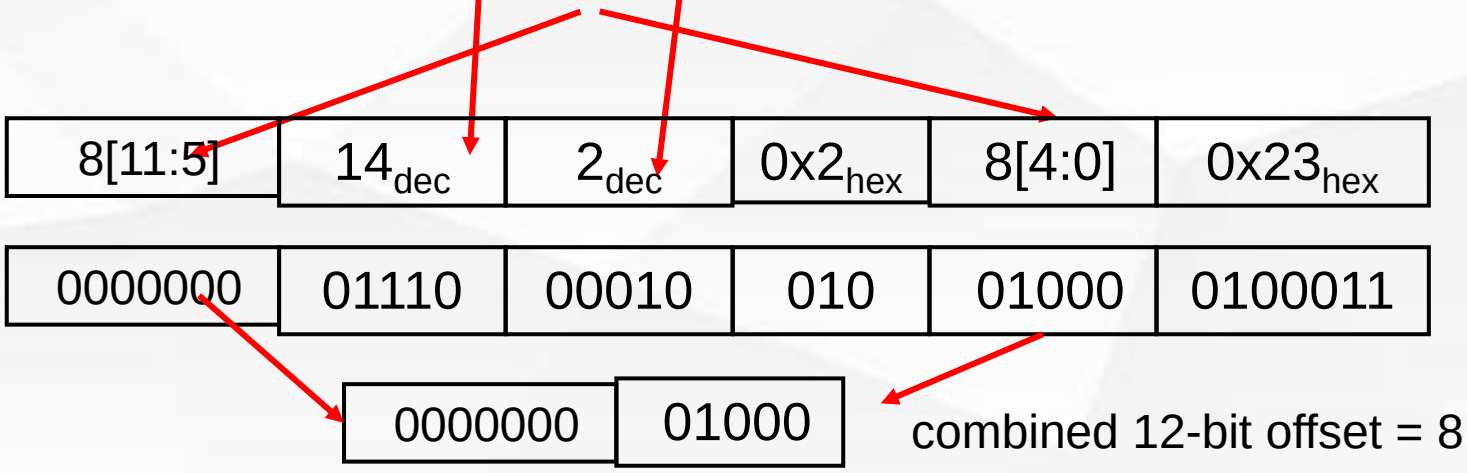


0xFD808793

RISC-V S-format Instructions

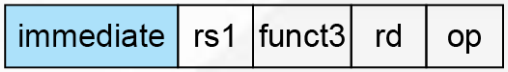


- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place
- Example sw x14, 8(x2)

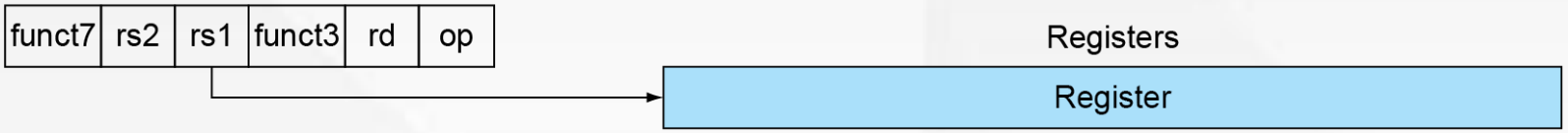


RISC-V Addressing Summary

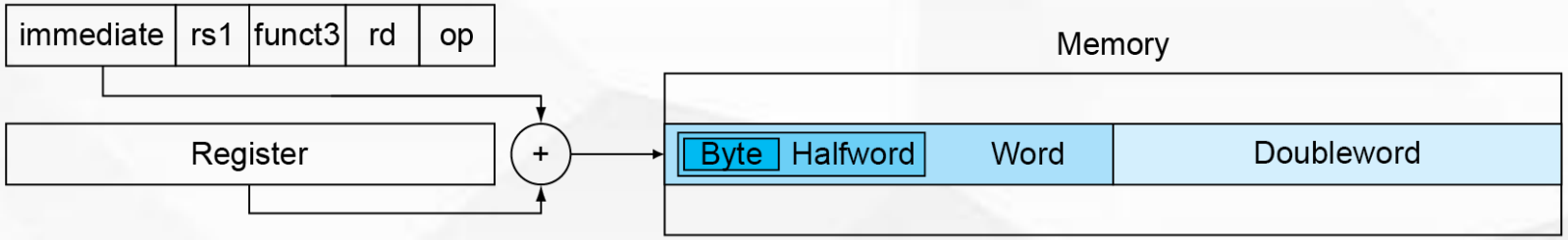
1. Immediate addressing



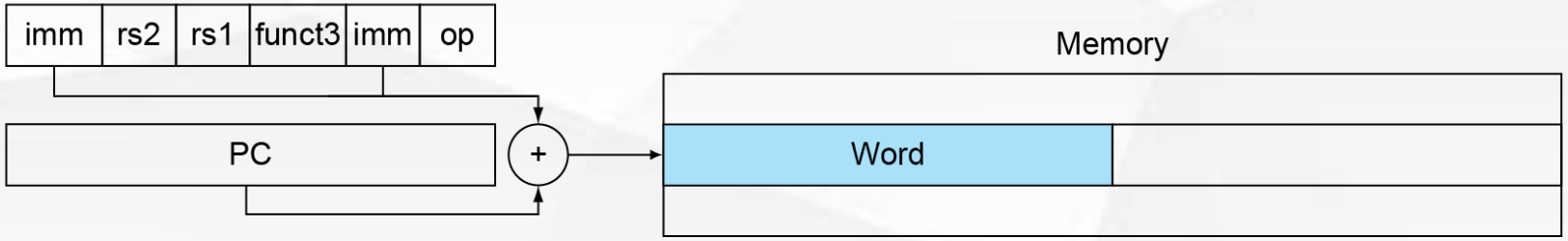
2. Register addressing



3. Base addressing

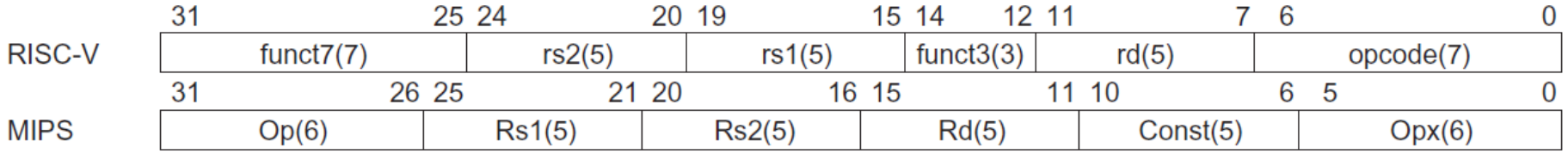


4. PC-relative addressing

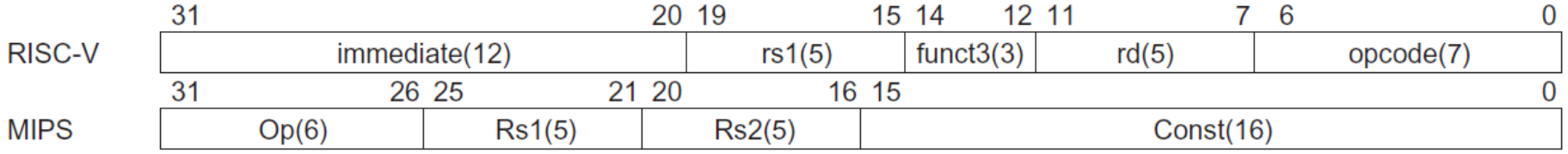


MIPS & RISC-V Instruction Formats

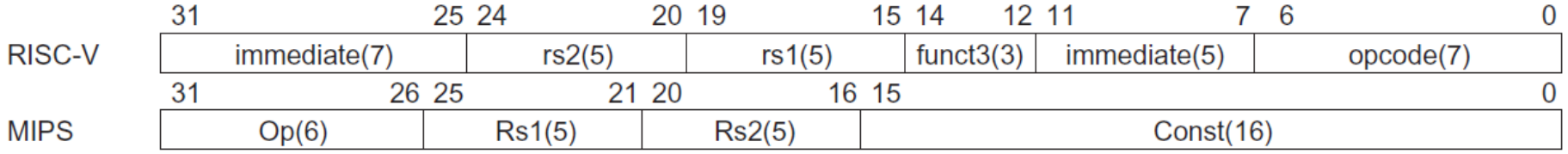
Register-register



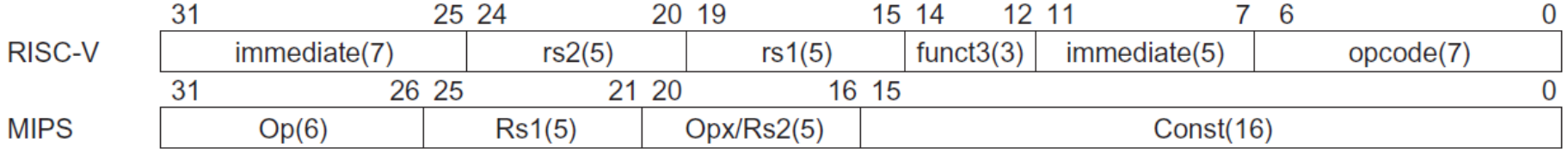
Load



Store



Branch



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