

Chapter 2

Instructions: Language of the Computer

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
 - But with many aspects in common
- Early computers had very simple instruction sets
 - Simplified implementation
- Many modern computers also have simple instruction sets

The RISC-V Instruction Set

- Used as the example throughout the book
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
 - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market
 - Applications in consumer electronics, network/storage equipment, cameras, printers, ...

Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination
add a, b, c // a gets $b + c$
- All arithmetic operations have this form
- *Design Principle 1*: Simplicity favors regularity
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled RISC-V code:

```
add t0, g, h    // temp t0 = g + h
```

```
add t1, i, j    // temp t1 = i + j
```

```
add f, t0, t1   // f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- RISC-V has a 32×64 -bit register file
 - Use for frequently accessed data
 - 64-bit data is called a “doubleword”
 - 32 x 64-bit general purpose registers x0 to x31
 - 32-bit data is called a “word”
- *Design Principle 2: Smaller is faster*
 - c.f. main memory: millions of locations

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

Register Operand Example

- C code:

$f = (g + h) - (i + j);$

- f, \dots, j in $x19, x20, \dots, x23$

- Compiled RISC-V code:

add x5, x20, x21

add x6, x22, x23

sub x19, x5, x6

Memory Operands

- Main memory used for composite data
 - Arrays, structures, dynamic data
- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- Memory is byte addressed
 - Each address identifies an 8-bit byte
- RISC-V is Little Endian
 - Least-significant byte at least address of a word
 - *c.f.* Big Endian: most-significant byte at least address
- RISC-V does not require words to be aligned in memory
 - Unlike some other ISAs

Memory Operand Example

- C code:
 $A[12] = h + A[8];$
 - h in $x21$, base address of A in $x22$
 - Compiled RISC-V code:
 - Index 8 requires offset of 64
 - 8 bytes per doubleword
- ld $x9, 64(x22)$
add $x9, x21, x9$
sd $x9, 96(x22)$

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction
`addi x22, x22, 4`
- Make the common case fast
 - Small constants are common
 - Immediate operand avoids a load instruction

Sign Extension

- Representing a number using more bits
 - Preserve the numeric value
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110
- In RISC-V instruction set
 - lb: sign-extend loaded byte
 - lbu: zero-extend loaded byte

Representing Instructions

- Instructions are encoded in binary
 - Called machine code
- RISC-V instructions
 - Encoded as 32-bit instruction words
 - Small number of formats encoding operation code (opcode), register numbers, ...
 - Regularity!

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)

R-format Example

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

add x9,x20,x21

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

0000 0001 0101 1010 0000 0100 1011 0011_{two} =
015A04B3₁₆

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
- *Design Principle 3: Good design demands good compromises*
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

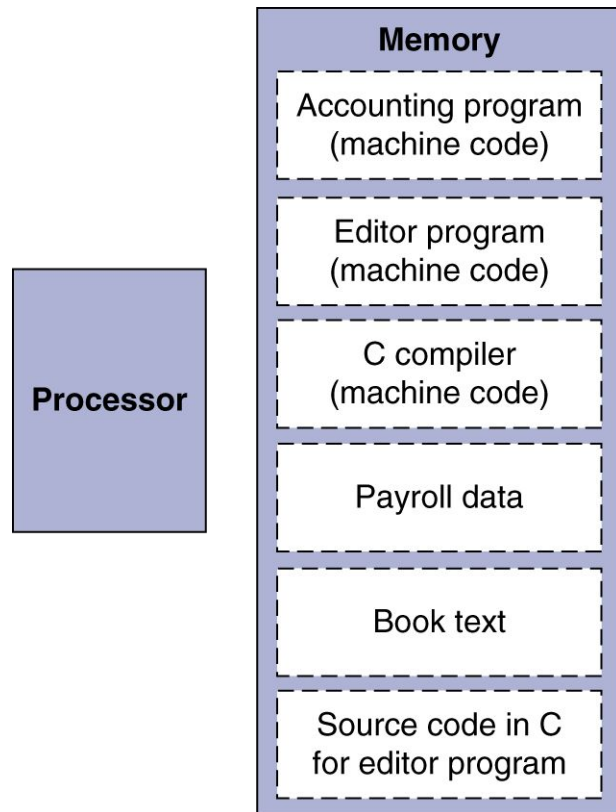
RISC-V S-format Instructions



- 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

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
 - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
 - Standardized ISAs

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	slli
Shift right	>>	>>>	srli
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	

- Useful for extracting and inserting groups of bits in a word

Shift Operations

funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

- immed: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - slli by i bits multiplies by 2^i
- Shift right logical
 - Shift right and fill with 0 bits
 - srli by i bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
 - Select some bits, clear others to 0
- and x9,x10,x11

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x1	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
1	
x9	00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

OR Operations

- Useful to include bits in a word
 - Set some bits to 1, leave others unchanged
- or x9,x10,x11

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x1	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
1	
x9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

XOR Operations

- Differencing operation
 - Set some bits to 1, leave others unchanged

xor x9,x10,x12 // NOT operation

x10	00000000	00000000	00000000	00000000	00000000	00000000	00001101	11000000
x12	11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111111
x9	11111111	11111111	11111111	11111111	11111111	11111111	11110010	00111111

Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- `beq rs1, rs2, L1`
 - if (`rs1 == rs2`) branch to instruction labeled L1
- `bne rs1, rs2, L1`
 - if (`rs1 != rs2`) branch to instruction labeled L1

Compiling If Statements

- C code:

```
if (i==j) f = g+h;  
else f = g-h;
```

- f, g, ... in x19, x20, ...

- Compiled RISC-V code:

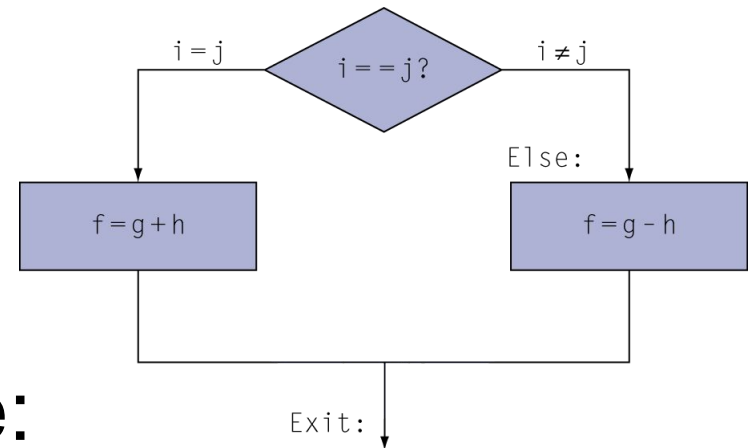
```
bne x22, x23, Else
```

```
add x19, x20, x21
```

```
beq x0,x0,Exit // unconditional
```

```
Else: sub x19, x20, x21
```

```
Exit: ...
```



Assembler calculates addresses

Compiling Loop Statements

- C code:

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

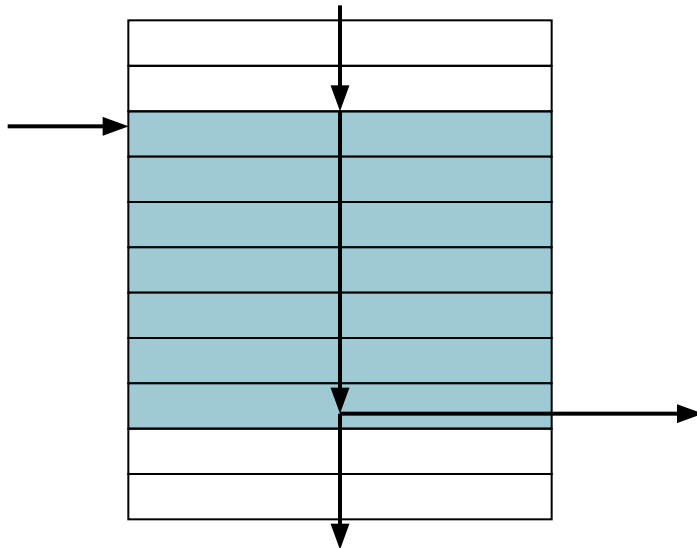
- i in x22, k in x24, address of save in x25

- Compiled RISC-V code:

```
Loop: slli x10, x22, 3  
      add x10, x10, x25  
      ld  x9, 0(x10)  
      bne x9, x24, Exit  
      addi x22, x22, 1  
      beq x0, x0, Loop  
Exit: ...
```

Basic Blocks

- A basic block is a sequence of instructions with
 - No embedded branches (except at end)
 - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- `blt rs1, rs2, L1`
 - if ($rs1 < rs2$) branch to instruction labeled L1
- `bge rs1, rs2, L1`
 - if ($rs1 \geq rs2$) branch to instruction labeled L1
- Example
 - if ($a > b$) $a += 1$;
 - a in `x22`, b in `x23`
`bge x23, x22, Exit` // branch if $b \geq a$
`addi x22, x22, 1`
Exit:

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
 - $x_{22} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$
 - $x_{23} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$
 - $x_{22} < x_{23}$ // signed
 - $-1 < +1$
 - $x_{22} > x_{23}$ // unsigned
 - $+4,294,967,295 > +1$

Procedure Calling

- Steps required
 1. Place parameters in registers x10 to x17
 2. Transfer control to procedure
 3. Acquire storage for procedure
 4. Perform procedure's operations
 5. Place result in register for caller
 6. Return to place of call (address in x1)

Procedure Call Instructions

- Procedure call: jump and link
`jal x1, ProcedureLabel`
 - Address of following instruction put in x1
 - Jumps to target address
- Procedure return: jump and link register
`jalr x0, 0(x1)`
 - Like `jal`, but jumps to `0 + address in x1`
 - Use x0 as rd (x0 cannot be changed)
 - Can also be used for computed jumps
 - e.g., for case/switch statements

Leaf Procedure Example

- C code:

```
long long int leaf_example (  
    long long int g, long long int h,  
    long long int i, long long int j) {  
    long long int f;  
    f = (g + h) - (i + j);  
    return f;  
}
```

- Arguments g, ..., j in x10, ..., x13
- f in x20
- temporaries x5, x6
- Need to save x5, x6, x20 on stack

Leaf Procedure Example

■ RISC-V code:

leaf_example:

addi sp,sp,-24

Save x5, x6, x20 on stack

sd x5,16(sp)

sd x6,8(sp)

sd x20,0(sp)

add x5,x10,x11

$x5 = g + h$

add x6,x12,x1

$x6 = i + j$

sub x20,x5,x6

$f = x5 - x6$

addi x10,x20,0

copy f to return register

ld x20,0(sp)

Restore x5, x6, x20 from stack

ld x6,8(sp)

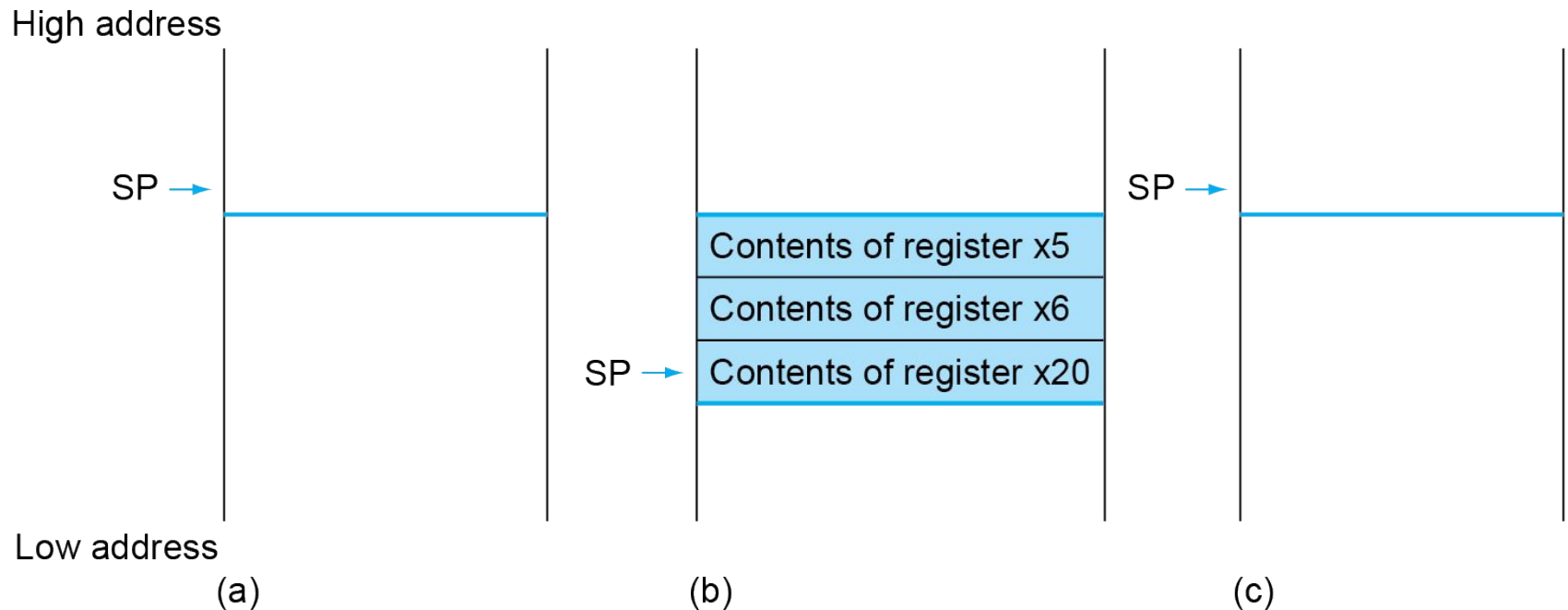
ld x5,16(sp)

addi sp,sp,24

jalr x0,0(x1)

Return to caller

Local Data on the Stack



Register Usage

- $x5 - x7, x28 - x31$: temporary registers
 - Not preserved by the callee
- $x8 - x9, x18 - x27$: saved registers
 - If used, the callee saves and restores them

Memory Layout

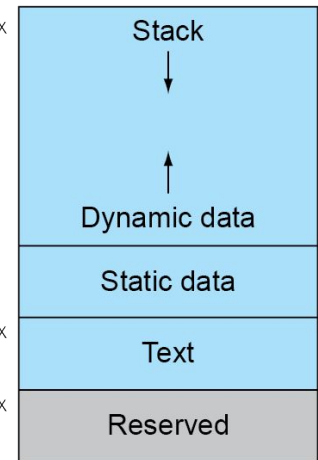
- Text: program code
- Static data: global variables
 - e.g., static variables in C, constant arrays and strings
 - x3 (global pointer) initialized to address allowing \pm offsets into this segment
- Dynamic data: heap
 - E.g., malloc in C, new in Java
- Stack: automatic storage

SP → 0000 003f ffff fff0_{hex}

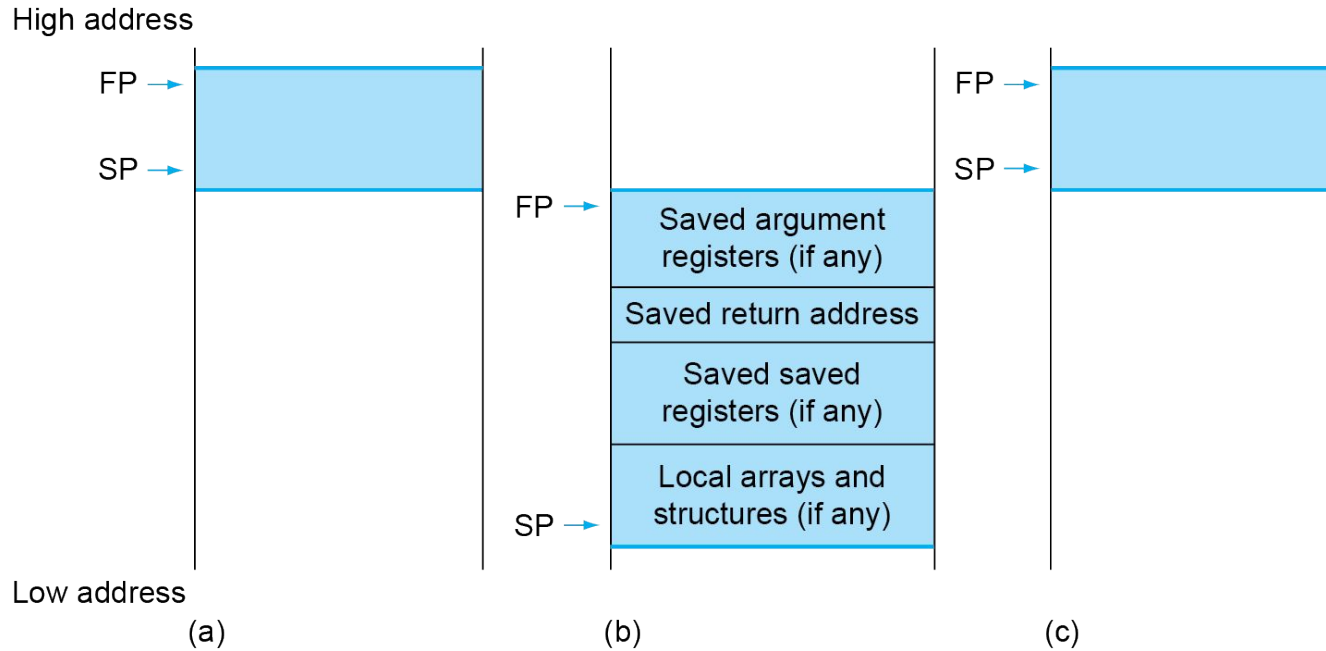
0000 0000 1000 0000_{hex}

PC → 0000 0000 0040 0000_{hex}

0



Local Data on the Stack



- Local data allocated by callee
 - e.g., C automatic variables
- Procedure frame (activation record)
 - Used by some compilers to manage stack storage

Character Data

- Byte-encoded character sets
 - ASCII: 128 characters
 - 95 graphic, 33 control
 - Latin-1: 256 characters
 - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
 - Used in Java, C++ wide characters, ...
 - Most of the world's alphabets, plus symbols
 - UTF-8, UTF-16: variable-length encodings

Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
 - Load byte/halfword/word: Sign extend to 64 bits in rd
 - lb rd, offset(rs1)
 - lh rd, offset(rs1)
 - lw rd, offset(rs1)
 - Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
 - lbu rd, offset(rs1)
 - lhu rd, offset(rs1)
 - lwu rd, offset(rs1)
 - Store byte/halfword/word: Store rightmost 8/16/32 bits
 - sb rs2, offset(rs1)
 - sh rs2, offset(rs1)
 - sw rs2, offset(rs1)

32-bit Constants

- Most constants are small
 - 12-bit immediate is sufficient
- For the occasional 32-bit constant

lui rd, constant

 - Copies 20-bit constant to bits [31:12] of rd
 - Extends bit 31 to bits [63:32]
 - Clears bits [11:0] of rd to 0

lui x19, 976 // 0x003D0

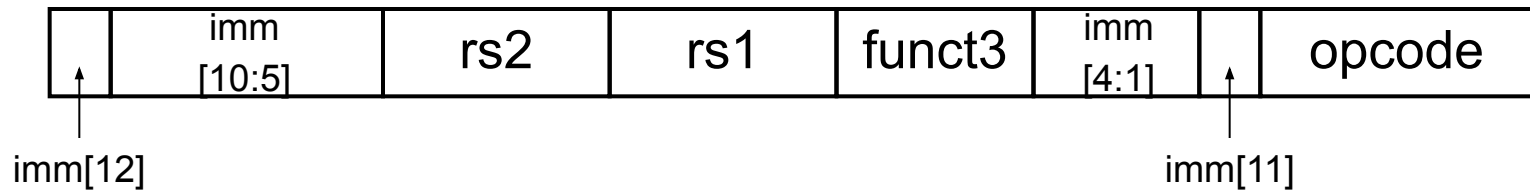
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000
---------------------	---------------------	--------------------------	----------------

addi x19,x19,1280 // 0x500

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000
---------------------	---------------------	--------------------------	----------------

Branch Addressing

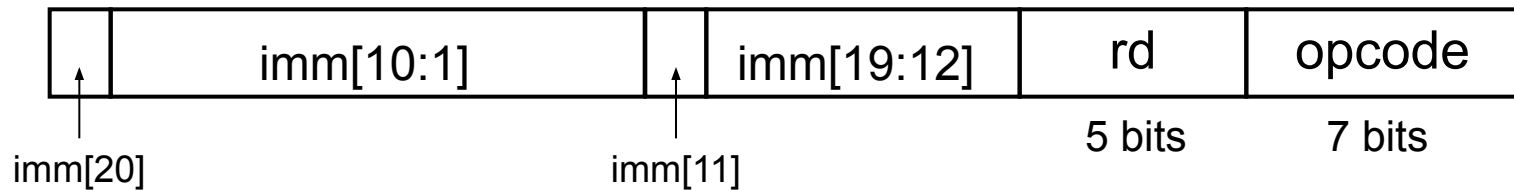
- Branch instructions specify
 - Opcode, two registers, target address
- Most branch targets are near branch
 - Forward or backward
- SB format:



- PC-relative addressing
 - Target address = PC + immediate \times 2

Jump Addressing

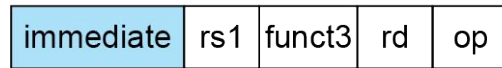
- Jump and link (jal) target uses 20-bit immediate for larger range
- UJ format:



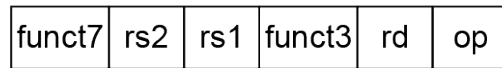
- For long jumps, eg, to 32-bit absolute address
 - lui: load address[31:12] to temp register
 - jalr: add address[11:0] and jump to target

RISC-V Addressing Summary

1. Immediate addressing



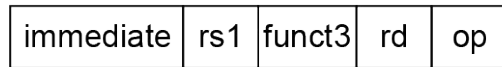
2. Register addressing



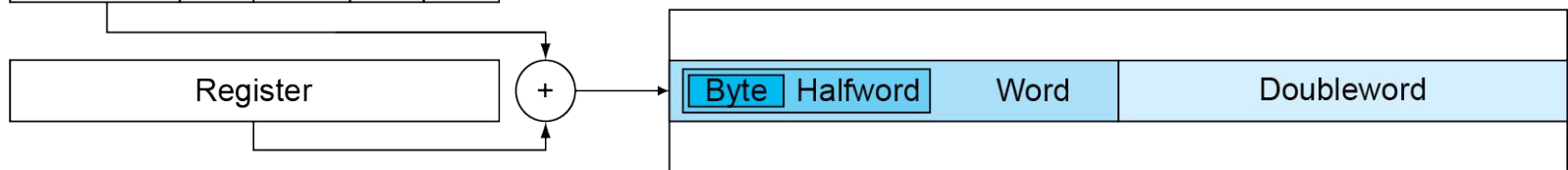
Registers

Register

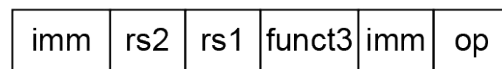
3. Base addressing



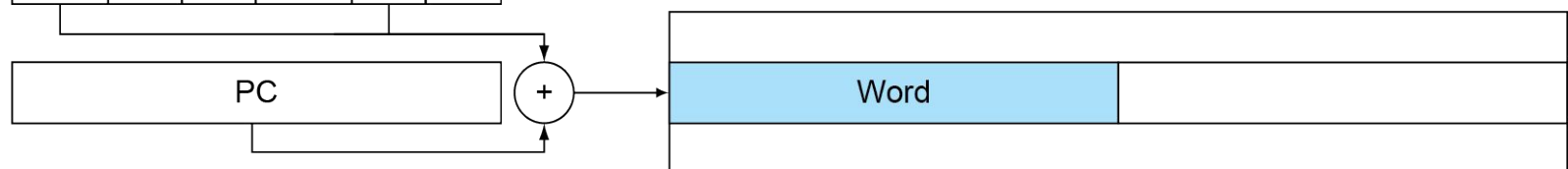
Memory



4. PC-relative addressing



Memory



RISC-V Encoding Summary

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

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!

MIPS Instructions

- MIPS: commercial predecessor to RISC-V
- Similar basic set of instructions
 - 32-bit instructions
 - 32 general purpose registers, register 0 is always 0
 - 32 floating-point registers
 - Memory accessed only by load/store instructions
 - Consistent use of addressing modes for all data sizes
- Different conditional branches
 - For <, <=, >, >=
 - RISC-V: blt, bge, bltu, bgeu
 - MIPS: slt, sltu (set less than, result is 0 or 1)
 - Then use beq, bne to complete the branch

Instruction Encoding

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)									

Concluding Remarks

- Design principles
 1. Simplicity favors regularity
 2. Smaller is faster
 3. Good design demands good compromises
- Make the common case fast
- Layers of software/hardware
 - Compiler, assembler, hardware
- RISC-V: typical of RISC ISAs
 - c.f. x86