Name ：洪理川
ID　　: B113040056

Chapter 2:Instructions: Language of the Compiler

2.1 Introduction

We need computer language to command a computer. The word used is called instruction, and the vocabulary used is called instruction set. In this book, we will be using the RISC-V instruction set.

2.2 Operations of the Computer Hardware

Multiply is one of the easiest operations in the computer Hardware. RISC-V has 32 registers for general-purpose integer arithmetic. The registers are a storage that is fast to store and change the data. Below is the RISC-V assembly language notation.

Instruction	Example	Meaning
Add	Add x11, x12, x13	x11 = x12 + x13

Design Principle 1: Simplicity favors regularity

Ex: f = (g + h) − (i + j);

The compiler will produce one RISC-V instruction for every operation.

```
add r0 g h // store g + h result in r0 temporary variable
add r1 i j // store i + j result in r1 temporary variable
sub a r0 r1 // store r0 – r1 result in f variable
```

Double slashes (//) are comments. These comments will be ignored by computers, their purpose is to help humans to understand the meaning of the code.

2.3 Operands of the Computer Hardware

The operands in computers are limited, and it is built-in hardware called registers.

Word -> A group of 32 bits which is the size of a register.

Doubleword -> A group of 64 bits.

Design Principle 2: Smaller is faster

This statement is not always true, since 10 register is not always faster than 32. To maximize the speed, we must balance the need for more registers in programs to maintain a fast clock cycle.

Now, we try to compile a C program using registers

f = (g + h) − (i + j);

The f, g, h, i, and j variables are assigned to x19, x20, x21, x22, and x23 registers.

add x5, x20, x21 // x5 register = x20 + x21

add x6, x22, x23 // x6 register = x22 + x23

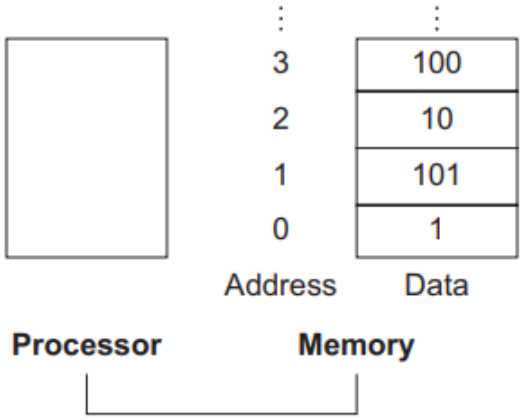
sub x19, x5, x6 // x19(f) = x5 – x6

Memory Operands

Computers have limited register space, but data structures like arrays and structures can be much larger. So, data is stored in computer memory, which is much larger than registers.

Data Transfer Instructions -> instructions that move data between memory and registers.

To access specific data in memory, instructions use addresses as an index. Addresses start at 0 and act like an array index to locate the desired data element in memory.



load -> instruction that copies data from memory to register

Instruction	Example	Meaning
Load Word	Lw x5, 40(x6)	X5 = Memory[x6 + 40]

store -> instruction that copies data from register to memory

Instruction	Example	Meaning
Store Word	sw x5, 40(x6)	Memory[x6 + 40] = x5

Compiling Using Load and Store

A[12] = h + A[8];

Assume h is in register x21 and A[0] is in the x22

lw x9, 32(x22) // load Memory[x22 + 32] from register and assign it to x9

add x9, x21, x9 // assign x9 with x9 + x21

sw x9, 48(x22) // Stores the result into A[12] which is Memory[x22 + 48]

Hardware/Software Interface

Computers have limited fast "working desk" space called registers for storing frequently used data during program execution. While programs often require more data than available registers, compilers prioritize the most crucial ones. Accessing data directly from the larger,

slower "storage room" called memory is significantly less efficient, both in terms of speed and energy consumption.

Spilling Register -> putting less used variables into memory

Constant or Immediate Operands

Constants are often used in operations. However, with the instructions we've covered so far, directly using a constant would require loading it from memory.

lw x9, AddrConstant4(x3) // x3 + AddrConstant4 is the address of the constant

add x22, x22, x9 // x22 = x22 + x9

This is not efficient, addi operand is a better way to do this. With this operand, the operations will be much more efficient and faster than the previous way.

addi x22, x22, 4

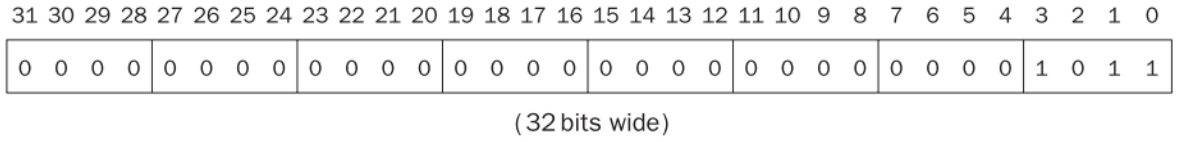
2.4 Signed and Unsigned Numbers

In computers, numbers are represented in base 2 using high and low electronic signals, which are called binary numbers.

Ex: 11 (base 10) = 1011 (base 2) = $(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$ (base 10)

Unsigned Number

Positive numbers are called unsigned numbers.



The least significant bit = the rightmost bit or the bit 0

The most significant bit = the leftmost bit or the bit 31

The 32 bits long can create number up to $2^{32} - 1$ (4, 294, 967, 295) (base 10)

if the number is too big to be represented it is called **overflow**

Signed Number

We needed a sign to represent a negative number, the leading 0s mean positive, and the leading 1s mean negative. This rule is called **two's complement** representation. Since 1 bit is used to be a sign of the number the range of the number shifts from -2^{31} to $-2^{31} - 1$.

01111111 11111111 11111111 11111111 (base 2) = 2,147,483,647 (base 10)

10000000 00000000 00000000 00000000 (base 2) = − 2,147,483,648 (base 10)

Converting Binary to Decimal

11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111000 (base 2)

We will do the two complements by reversing 1 and 0, then add 1

00000000 0000000 00000000 00000000 00000000 00000000 00000000 00001000 (base 2)

this will become -8 for signed numbers, 18, 446, 744, 073, 709, 551, and 608 for unsigned numbers.

2.5 Representing Instructions in the Computer

Design Principle 3: Good design demands good compromises

The instructions can be represented as a series of binary numbers:

0000000	10101	10100	000	01001	0110011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Each segment is called a field, and there are two types of fields: R-type and I-type.

RISC-V Fields

1. **R-type**

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

- *opcode*: Basic operation of the instruction
- *rd*: The register destination operand. It gets the result of the operation.
- *funct3*: An additional opcode field.
- *rs1*: The first register source operand.
- *rs2*: The second register source operand.
- *funct7*: An additional opcode field.

2. **I-type**

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

This type is created to address the absence of an R-type field in the 5-bit field, which is crucial for providing sufficient addressing capability in Load word instructions or operations involving a single constant.

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011
Instruction	**Format**	**immediate**		**rs1**	**funct3**	**rd**	**opcode**
addi (add immediate)	I	constant		reg	000	reg	0010011
lw (load word)	I	address		reg	010	reg	0000011
Instruction	**Format**	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**
sw (store word)	S	address	reg	reg	010	address	0100011

Hexadecimal

Hexadecimal is often used since it is a multiple of 4, making it easier to convert into binary.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

2.6 Logical Operations

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

There is a lot of Logical Operation, but we will focus on Shift, AND, OR, NOT and XOR

Shift -> A logical operation that Moves the bits to the left/right.

AND -> A logical operation with two operands that calculate a 1 only if there is a 1 in both operands.

OR -> A logical operation with two operands that calculate a 1 if there is a 1 in either operand.

NOT -> A logical operation with one operand that inverts the bits; that will replace every 1 with a 0 and every 0 with a 1

XOR -> A logical operation with two operands that will result in 0 if the bits are the same otherwise it will be 1.

2.7 Instructions for Making Decisions

Conditional branches

- If (rs1 == rs2) can be turned into beq rs1, rs2 L1 in RISC-V assembly language

beq = branch if equal

- If (rs1 != rs2) can be turned into bne rs1, rs2 L1 in RISC-V assembly language

beq = branch if not equal

Loops

Usually, we combine the if statement in the iteration. There are a lot of comparisons, less than, less than or equal, greater than, greater than or equal, equal, and not equal.

Case/Switch Statement

The statement chooses 1 value from other values. Case/Switch Statement can be encoded into 2 types.

1. Into a chain of if-then-else statements
2. Branch address tables -> Table of addresses of alternative instruction sequences

2.8 Supporting Procedures in Computer Hardware

Procedure/function -> A stored subroutine that performs a specific task, making it easier to structure programs and reuse code.

Since it is a stored subroutine, it needed an address and we called the procedure by jump-and-link-instruction (jal)

jal x1, ProdecureAddress // go to the address and write return address to x1 (origin)

Using More Register

When the compiler needs more registers for a procedure, we need to spill registers to memory. The ideal data structure for this mission is **stack.** Stack works with the LIFO (last-in-first-out) principle.

2.9 Communicating with People

These days Computer uses the American Standard Code for Information Interchange (ASCII) to represent characters.

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

2.10 RISC-V Addressing for Wide Immediate and Addresses

Wide Immediate Operands

In RISC-V, commands have a fixed length, which means that the immediate values (constants) that can be directly put into a command are limited in size.

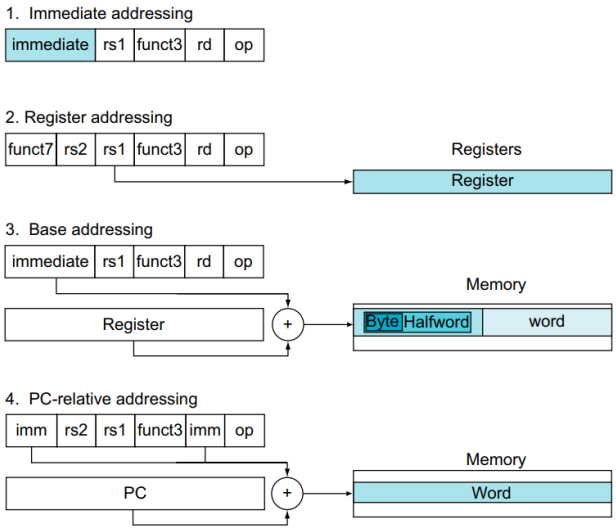
Addressing in Branches

Program counter = Register + Branch address

PC-relative addressing -> an addressing regime in which the address is the sum of the program counter(PC) and a constant in the instruction.

RISC-V Addressing Mode

1. Immediate addressing, where the operand is a constant within the instruction itself.
2. Register addressing, where the operand is a register.
3. Base or displacement addressing, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction.
4. PC-relative addressing, where the branch address is the sum of the PC and a constant in the instruction.



2.11 Parallelism and Instructions: Synchronization

Synchronization is crucial when tasks need to cooperate and share data. Without proper synchronization, there's a risk of data races and unpredictable program behavior.

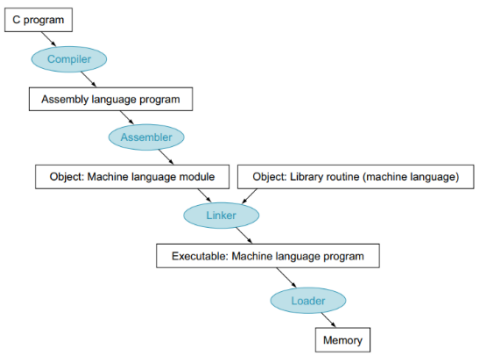
Data Race -> Two memory accesses form a data race if they are from different threads to the same location, at least one is a write, and they occur one after another.

Atomic Exchange/ Atomic Swap

Changes a value in register with a value in memory.

For example: When 2 processors want to do an exchange at the same time. It will one side will finish first leading another one failed. The alternative solution will use a pair of instructions to show if the other pair was executed.

2.12 Translating and Starting a Program



Compiler

Compilers turn high-level language into a low-level language, which is an assembly language program.

Assembler

Assemblers turn assembly language into machine language code

pseudoinstruction -> A common variation of assembly language instructions often treated as if it were an instruction in its own right
Assemblers must know all of the addresses on each label and store them in the symbol table.

Linker
Linker will compile and assemble the changed code intro new procedure, without compiling and assembling the whole program.
3 steps for the linker:

1. Place code and data modules symbolically in memory.
2. Determine the address of data and instruction labels.
3. Patch both the internal and external references.

Loader
In this step, there will be an executable file on the disk waiting for the operating system to start it.
Executable file -> A functional program in the format of an object file that contains no unresolved references.

The loader steps in the Unix systems:

1. Reads the executable file header to obtain file information.

2. Create an address space for the text and data.

3. Copies data from the executable file into memory.

4. Copies the parameters of the main program onto the stack.

5. Initializes the processor registers and sets the stack pointer to the first free

location.

6. Branches to a start-up routine that copies the parameters into the argument

registers and calls the main routine of the program.

2.14 Arrays versus Pointer

```
       addi x5, x0, 0       // i = 0
loop1: slli x6, x5, 2       // x6 = i * 4
       add  x7, x10, x6     // x7 = address of array[i]
       sw   x0, 0(x7)       // array[i] = 0
       addi x5, x5, 1       // i = i + 1
       blt  x5, x11, loop1  // if (i < size) go to loop1
```

```
       addi   x5, 0         // p = address of array[0]
       slli x6, x11, 2      // x6 = size * 4
       add  x7, x10, x6     // x7 = address of array[size]
loop2: sw   x0, 0(x5)       // Memory[p] = 0
       addi x5, x5, 4       // p = p + 4
       bltu x5, x7, loop2   // if (p < &array[size]) go to loop2
```

(Left code is Array Version, Right code is Pointer Version)

From what we can see from the code, the array version needed to recalculate the address of the array. On the other hand, the pointer version just increments the pointer directly. This difference makes the pointer version more efficient.

2.15 Advanced Material: Compiling C and Interpreting Java

Anatomy of a Compiler

A. The Front End

This part is to read and check the program syntax and semantics.
4 separate functions:
- *Scanning*: This step converts the characters in the source code into tokens, which are the basic units of syntax, such as keywords, names, operators, and symbols.
- *Parsing*: This step checks the syntax of the token sequence and builds an abstract syntax tree.
- *Semantic analysis*: This step checks the meaning of the program, such as the types and declarations of variables and functions.
- *Intermediate representation generation*: This step transforms the abstract syntax tree and the symbol table into a lower-level representation that is closer to the target machine language.

B. High-level Optimizations

High-level optimizations are transformations applied close to the source code level. One common example is **procedure inlining**, where a function call is replaced by the function's body, using the caller's arguments for the function's parameters. Other high-level optimizations include loop transformations that reduce loop overhead, improve memory access, and enhance hardware utilization.

C. Global optimizer

Global optimizers have the same purpose as the local one, including common subexpression elimination, constant propagation, copy propagation, and dead store and dead code elimination.

2 important global optimizations:
1. Code Motion: Identifying loop-invariant code, which computes the same result in each iteration, and moving it outside the loop for efficiency.

2. Induction Variable Elimination: This optimization aims to reduce overhead on array indexing by replacing it with pointer accesses. It's particularly useful in loops. However, the text suggests referring to Section 2.14 for a more detailed explanation of this process.

D. Code Generator

The last steps of a compiler are code generation and assembly. In modern compilers, these steps are combined to improve efficiency. Code generation is simpler for processors with less complex architectures, but more complex for architectures like x86. Modern compilers use pattern matching to generate code and perform final optimizations.

2.16 Real Stuff: MIPS Instructions

MIPS is also an instruction set like RISC-V. These two instruction set share the same design so here are the common features between RISC-V and MIPS.

Common features between RISC-V and MIPS

- All instructions are 32 bits wide for both architectures.
- Both have 32 general-purpose registers, with one register being hardwired to 0.
- The only way to access memory is via load and store instructions on both architectures.

- Unlike some architectures, there are no instructions that can load or store many registers in MIPS or RISC-V.
- Both have instructions that branch if a register is equal to zero and branch if a register is not equal to zero.
- Both sets of addressing modes work for all word sizes

The Difference

MIPS uses comparison instruction that sets a register to 0 or 1 depending on whether the comparison is true for a conditional branch. On the other hand, RISC-V provides branch instructions to compare between two registers.

2.17 Real Stuff: ARMv7 (32-bit) Instructions

The principal difference is that RISC-V has more registers and ARM has more addressing modes.

Addressing Modes
- ARM does not require a register to contain 0, so it has separate opcodes to perform operations.

Compare and Conditional Branch
- ARM uses four condition code bits (negative, zero, carry, and overflow) stored in the program status word to evaluate conditional branches. These condition codes can be set on any arithmetic or logical instruction, with the setting being optional on each instruction.

2.18 Real Stuff: ARMv8 (64-bit) Instructions

What changed in v8:

Dropped Features:

- There is no conditional execution field as there was in nearly every instruction in v7.
- The immediate field is simply a 12-bit constant rather than essentially an input to a function that produces a constant as in v7.
- ARM dropped Load Multiple and Store Multiple instructions.
- The PC is no longer one of the registers, which resulted in unexpected branches if you wrote to it.

Added Features:

- V8 has 32 general-purpose registers, which compiler writers surely love. Like MIPS, one register is hardwired to 0, although in load and store instructions it instead refers to the stack pointer.
- Its addressing modes work for all word sizes in ARMv8, which was not the case in ARMv7.
- It includes a divide instruction, which was omitted from ARMv7.
- It adds the equivalent of the MIPS branch if equal and the branch if not equal.

2.19 Real Stuff: x86 Instructions

2 important differences between x86 with RISC-V and MIPS

1. **Arithmetic and Logical Instructions**

x86: Must have 1 operand act as both a source and destination.
RISC-V and MIPS: Allow different registers for the source and destination.

2. **Operand in Memory**

x86: can do it, makes any instruction may have 1 operand in the memory.
RISC-V and MIPS: Do not allow that.

2.22 Fallacies and Pitfalls

Fallacy: More powerful instructions mean higher performance.
Contrary to popular belief, simpler and standard instructions can sometimes outperform more complex ones. For instance, memory-to-memory moves using prefixes can be slower than standard load-store operations.

Fallacy: Write in assembly language to obtain the highest performance.
While writing in assembly used to be essential for performance optimization, modern compilers have become so advanced that they can generate code that rivals or even surpasses hand-written assembly in terms of performance. Furthermore, writing in assembly presents challenges such as increased coding and debugging time, loss of portability, and difficulty in maintaining the code over time.

Fallacy: The importance of commercial binary compatibility means successful instruction sets don’t change.
Instruction sets continually evolve and expand to incorporate new features and optimizations. Therefore, it is not accurate to assume that successful instruction sets remain static forever. However, backward binary compatibility is still critical.

Pitfall: Forgetting that sequential word addresses in machines with byte addresses do not differ by one.
In byte-addressable machines, sequential word addresses differ by the word size in bytes, not by one byte. Neglecting this fact can lead to memory access and manipulation errors.

Pitfall: Using a pointer to an automatic variable outside its defining procedure.
Automatic variables exist on the stack and are deallocated once the defining procedure returns. Attempting to access these variables outside their scope can result in undefined behavior and memory corruption.