최근CPU 아키텍쳐

RISC-V 분석

First Edition

사진: 커널연구회에서 제작한 RISC-V (E31) 임베디드 보드

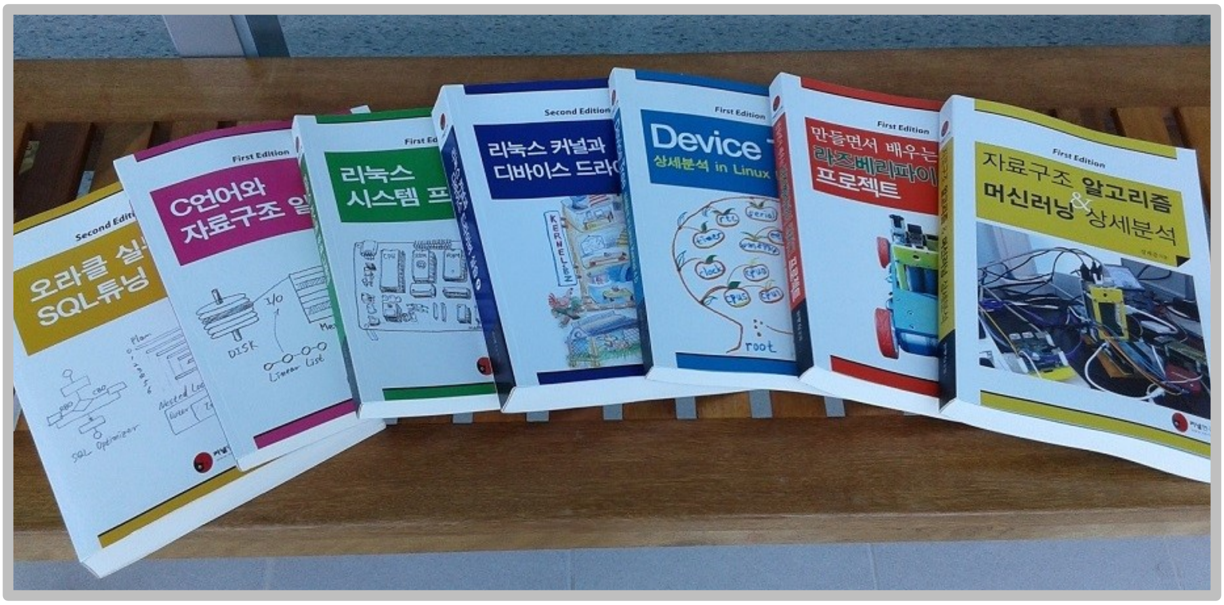
커널연구회(www.kernel.bz)

정재준(rgbi3307@nate.com)

저자 소개

정재준 (rgbi3307@nate.com) / 커널연구회(www.kernel.bz)

저자는 학창시절 마이크로프로세서 제어 기술을 배웠으며 리눅스 커널을 연구하고 있다. 15년 이상 쌓아온 실무 경험을 바탕으로 "C언어와 자료구조 알고리즘", "리눅스 시스템 프로그래밍2", "리눅스 커널과 디바이스드라이버 실습2", "자료구조 알고리즘 & 머신러닝 상세분석"등의 책을 집필하고, 월간임베디드월드 및 전자과학 잡지에 다수의 글을 기고 하였다. 또한 "맞춤형 문장 자동 번역 시스템 및 이를 위한 데이터베이스 구축방법 (The System for the customized automatic sentence translation and database construction method)" 라는 내용으로 프로그래밍을 하여 특허 등록 하였고, 서울시 버스와 지하철 교통카드 요금결재 단말기에 들어가는 리눅스 커널과 디바이스드라이버 개발 프로젝트를 성공적으로 수행했으며 여러가지 임베디드 제품을 개발했다. 저자는 Patterson(California 대학), Hennessy(Stanford 대학) 교수의 저서 "Computer Organization and Design" 책을 읽고 깊은 감명을 받았으며, 컴퓨터구조와 자료구조 알고리즘 효율성 연구를 주제로한 저술활동을 활발히 하고 있다. 아울러 커널연구회(www.kernel.bz) 웹사이트를 운영하면서 연구개발, 교육, 관련기술 공유 등을 위해 노력하고 있다.



♣ 저자가 집필한 책들 (시중서점에서 "커널연구회"로 검색하여 구매 가능)

목차

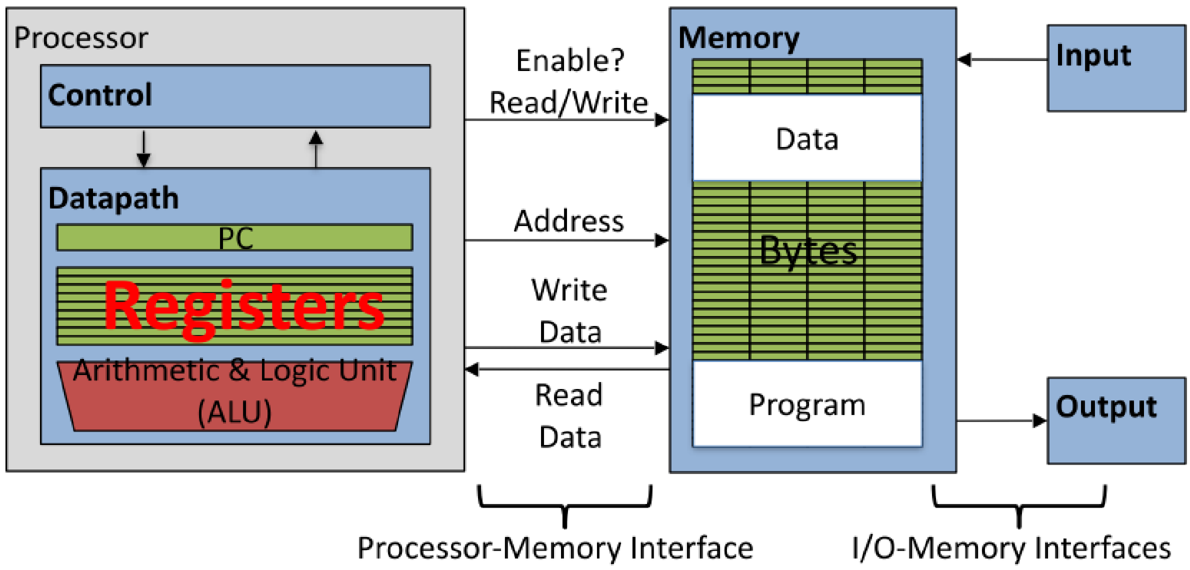
내용

1. 개요

1.1 CPU 구조

RISC: Reduced instruction Set Computer

CISC: Complex Instruction Set Computer



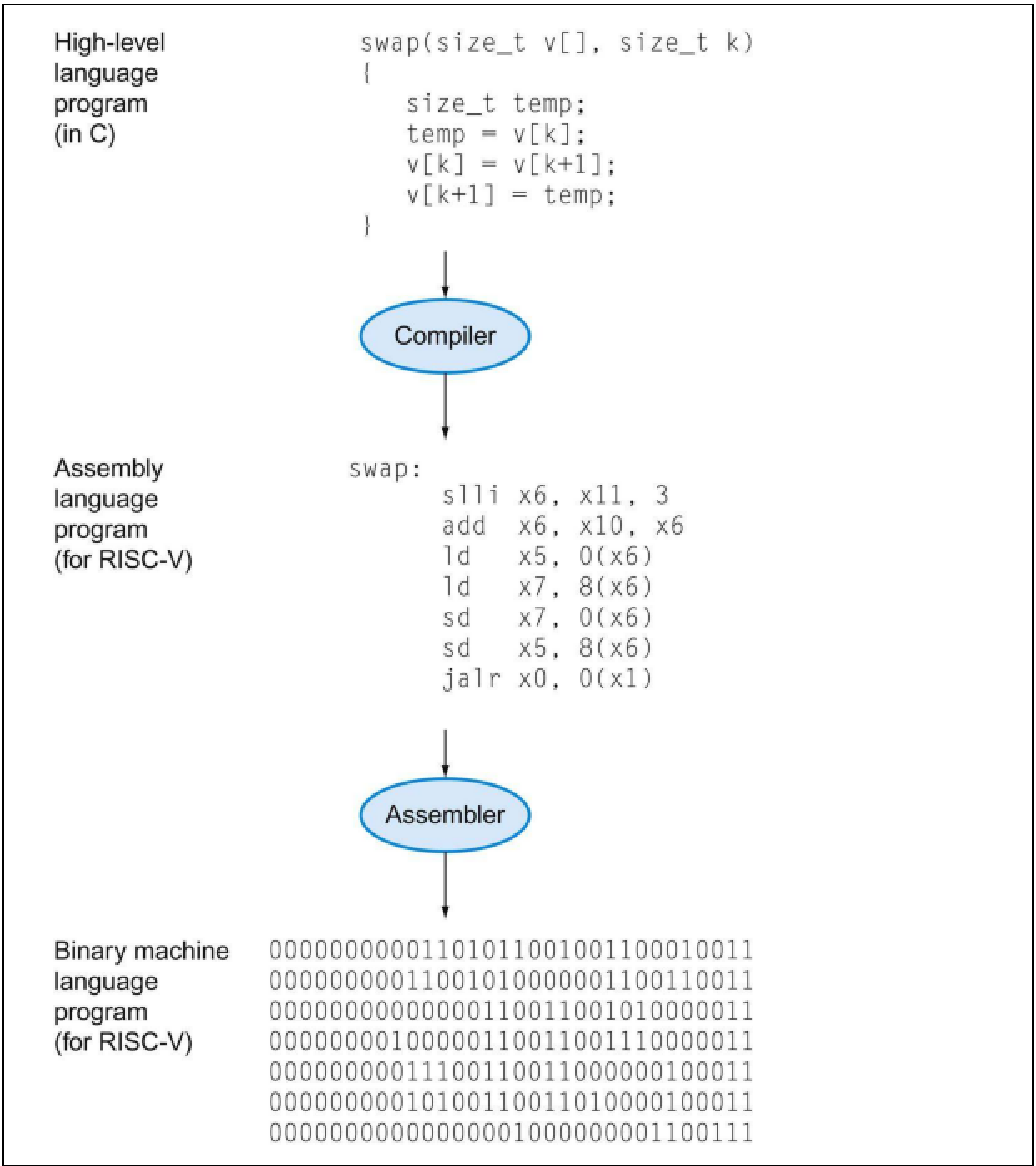
그림출처: 버클리 대학교 인터넷 강의 자료

Read(Load)

Operation(연산)

Write(Store)

1.2 Instruction Set 빌드 과정



출처: Computer Organization and Design RISC-V Edition (David A. Patterson, John L. Hennessy)

1.3 ISA (Instruction Set Architecture)

CISC(Complex Instruction Set Computer)

Computer Architecture: A Quantitative Approach (Hennessy and Patterson)

RISC(Reduced Instruction Set Computer)

MIPS(Microprocessor without Interlocked Pipeline Stages), ARM(Advanced RISC Machine)

Instruction Set

RISC-V

동일한 명령 셋으로 32 비트와 64 비트 주소 명령어 모두 지원

Open Source Instruction Set

RISC-V foundation (https://riscv.org/)

ARM, Intel 제외

AMD, Google, Hewlett Packard, IBM, Microsoft, NVIDIA, Oracle, Qualcomm 참여

1.4 CPU 성능(Performance) 평가

clock cycles per instruction (CPI)

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

전력소모

$$Power \propto 1/2 \times Capacitive\,load \times Voltage^2 \times Frequency\,switched$$

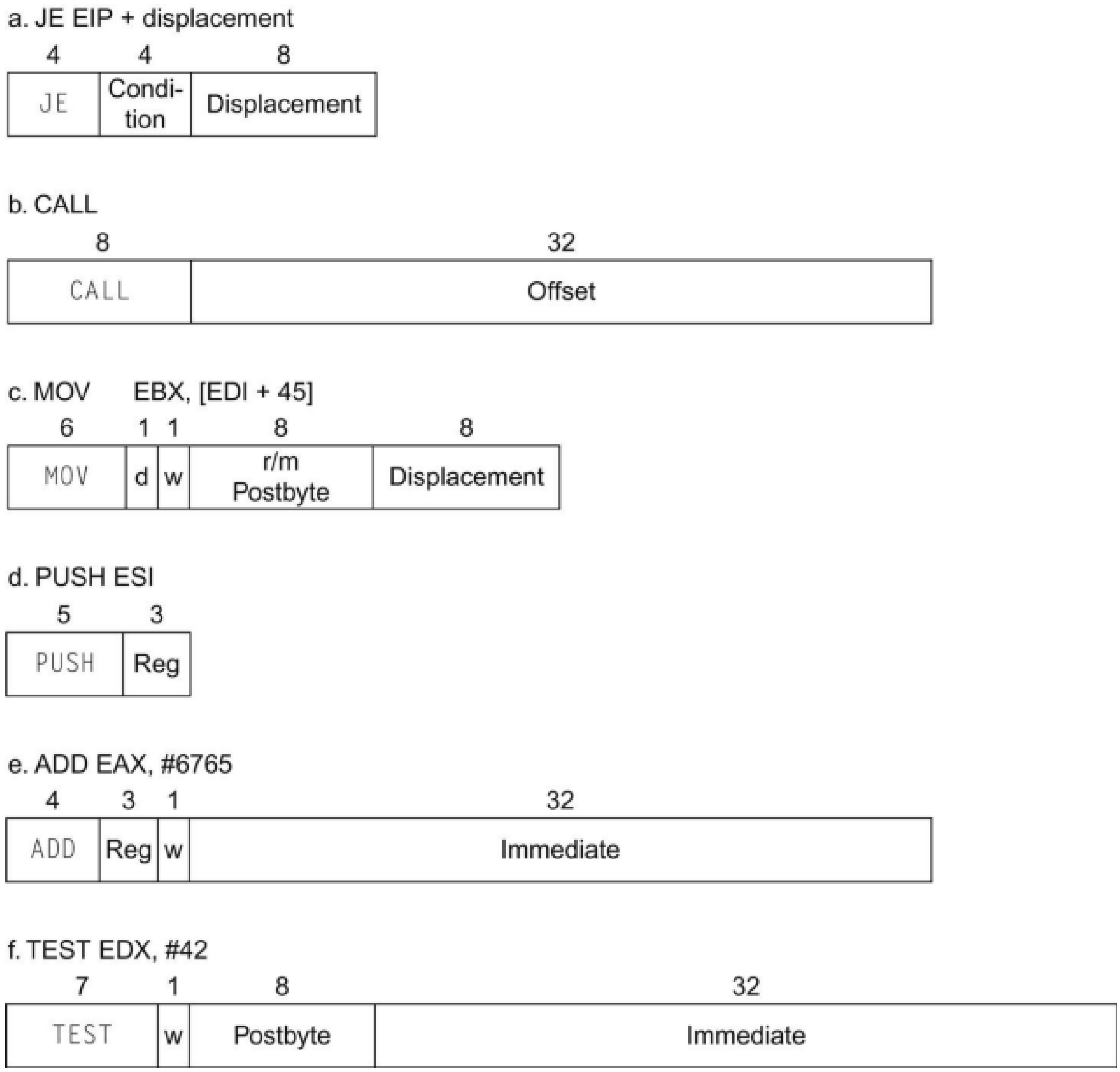
MIPS (million instructions per second)

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

$$\text{MIPS} = \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

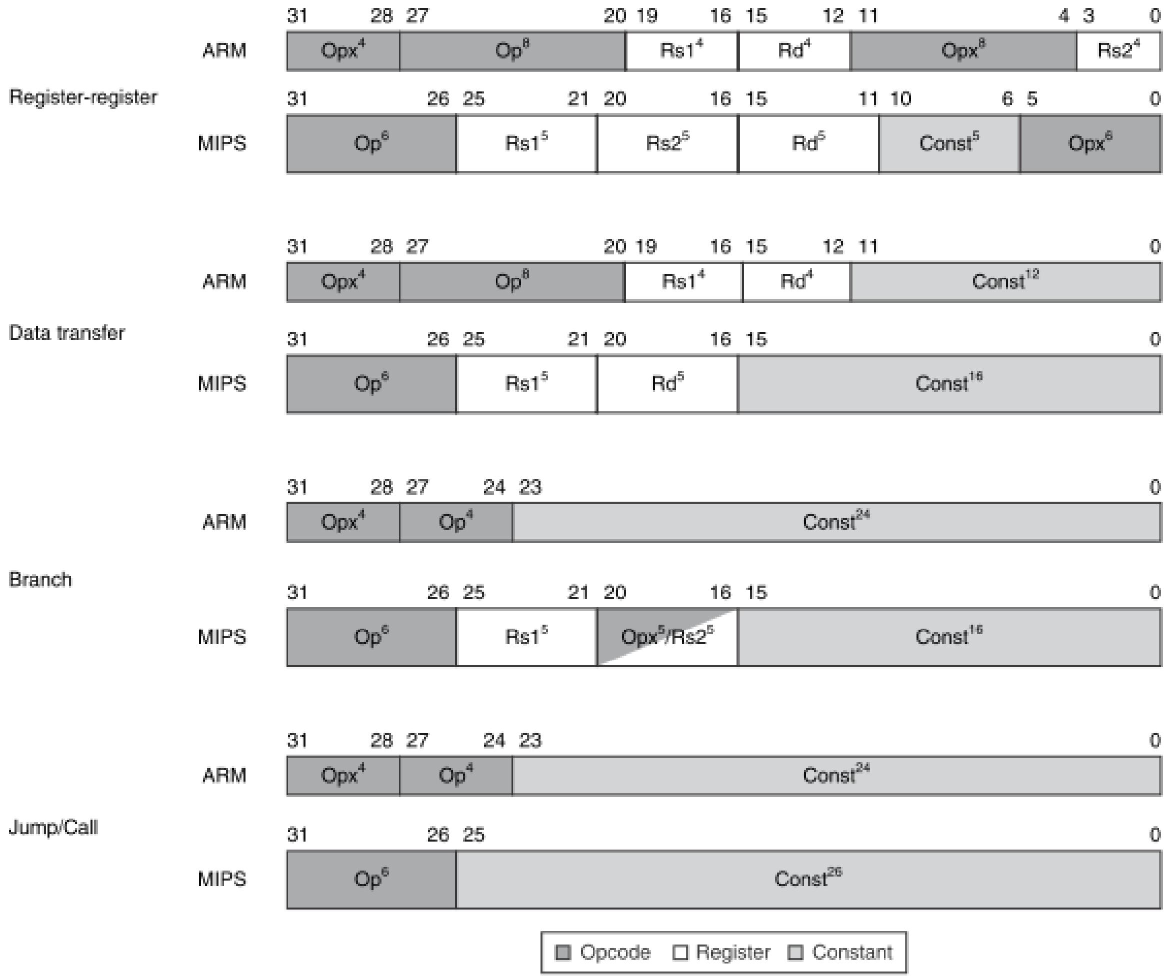
1.5. Instruction Set 비교

Intel x86 Instruction Set



출처: Computer Organization and Design RISC-V Edition (David A. Patterson, John L. Hennessy)

ARM과 MIPS Instruction Set 비교



출처: Computer Organization and Design RISC-V Edition (David A. Patterson, John L. Hennessy)

RISC-V와 MIPS Instruction Set 비교

Register-register

RISC-V	31–25 funct7(7)	24–20 rs2(5)	19–15 rs1(5)	14–12 funct3(3)	11–7 rd(5)	6–0 opcode(7)
MIPS	31–26 Op(6)	25–21 Rs1(5)	20–16 Rs2(5)	15–11 Rd(5)	10–6 Const(5)	5–0 Opx(6)

Load

RISC-V	31–20 immediate(12)	19–15 rs1(5)	14–12 funct3(3)	11–7 rd(5)	6–0 opcode(7)
MIPS	31–26 Op(6)	25–21 Rs1(5)	20–16 Rs2(5)	15–0 Const(16)	

Store

RISC-V	31–25 immediate(7)	24–20 rs2(5)	19–15 rs1(5)	14–12 funct3(3)	11–7 immediate(5)	6–0 opcode(7)
MIPS	31–26 Op(6)	25–21 Rs1(5)	20–16 Rs2(5)	15–0 Const(16)		

Branch

RISC-V	31–25 immediate(7)	24–20 rs2(5)	19–15 rs1(5)	14–12 funct3(3)	11–7 immediate(5)	6–0 opcode(7)
MIPS	31–26 Op(6)	25–21 Rs1(5)	20–16 Opx/Rs2(5)	15–0 Const(16)		

출처: Computer Organization and Design RISC-V Edition (David A. Patterson, John L. Hennessy)

Subset Naming Convention

Subset	Name
Standard General-Purpose ISA	
Integer	I
Integer Multiplication and Division	M
Atomics	A
Single-Precision Floating-Point	F
Double-Precision Floating-Point	D
General	G = IMAFD
Standard User-Level Extensions	
Quad-Precision Floating-Point	Q
Decimal Floating-Point	L
16-bit Compressed Instructions	C
Bit Manipulation	B
Dynamic Languages	J
Transactional Memory	T
Packed-SIMD Extensions	P
Vector Extensions	V
User-Level Interrupts	N
Non-Standard User-Level Extensions	
Non-standard extension “abc”	Xabc
Standard Supervisor-Level ISA	
Supervisor extension “def”	Sdef
Non-Standard Supervisor-Level Extensions	
Supervisor extension “ghi”	SXghi

2. RISC-V Instruction Set

Instruction Length Encoding

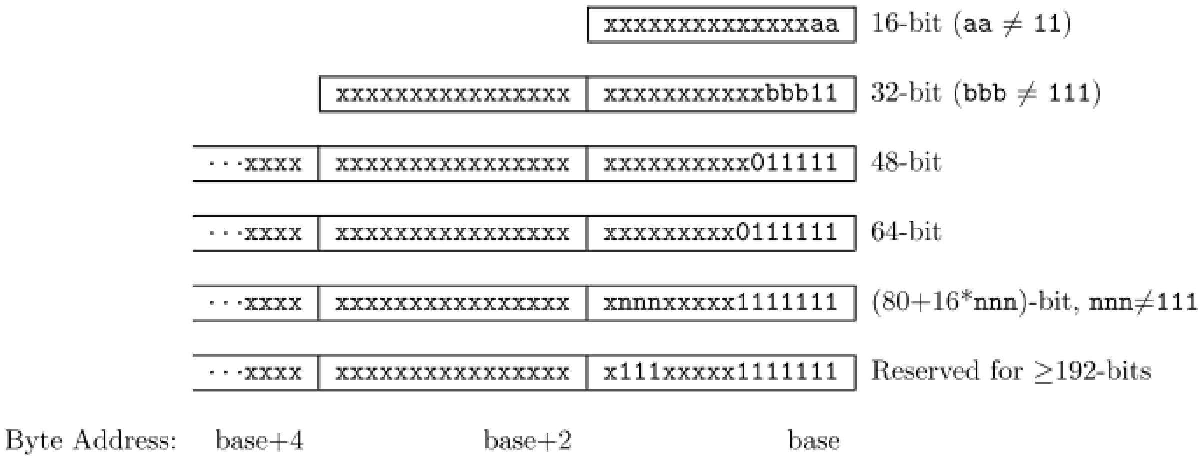


Figure 1.1: RISC-V instruction length encoding.

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32*b*)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64*b*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48*b*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq$ 80*b*

Table 19.1: RISC-V base opcode map, inst[1:0]=11

2.1 RISC-V 레지스터 정의

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

출처: The RISC-V Instruction Set Manual

Computer Organization and Design RISC-V Edition (David A. Patterson, John L. Hennessy)

For RV32, the x registers are 32 bits wide, and for RV64, they are 64 bits wide. This document uses the term XLEN to refer to the current width of an x register in bits (either 32 or 64)

2.2 RISC-V 기본 명령 형식

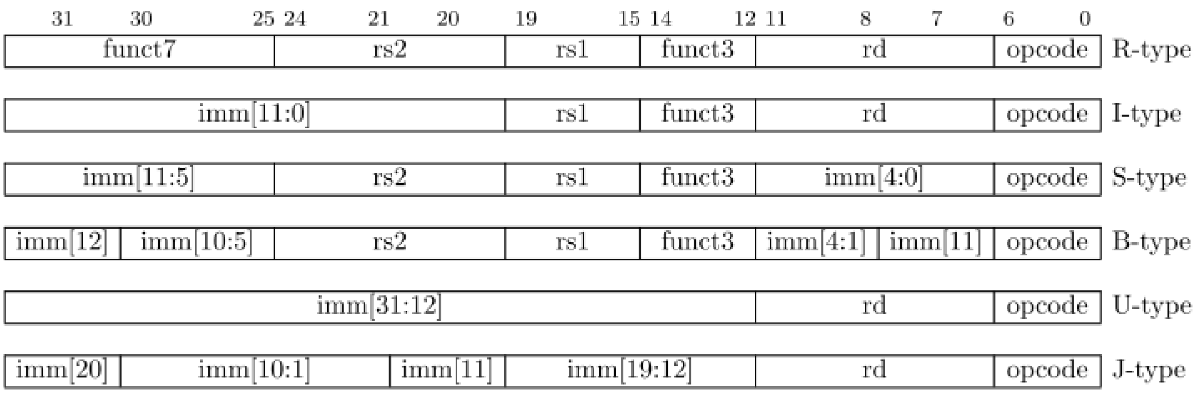


Figure 2.3: RISC-V base instruction formats showing immediate variants.

2.2.1 Integer Computational Instructions

Integer Register-Immediate Instructions

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

31 … 12	11 … 7	6 … 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

Integer Register-Register Operations

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

NOP Instruction

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

2.2.2 Control Transfer Instructions

Unconditional Jumps

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
	offset[20:1]			dest	JAL

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

Conditional Branches

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
	offset[12,10:5]	src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH
	offset[12,10:5]	src2	src1	BLT[U]	offset[11,4:1]		BRANCH
	offset[12,10:5]	src2	src1	BGE[U]	offset[11,4:1]		BRANCH

2.2.3 Load and Store Instructions

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

2.2.4 Memory Model

RISC-V harts

31 28	27	26	25	24	23	22	21	20	19 15	14 12	11 7	6 0
0	PI	PO	PR	PW	SI	SO	SR	SW	rs1	funct3	rd	opcode
4	1	1	1	1	1	1	1	1	5	3	5	7
0	predecessor				successor				0	FENCE	0	MISC-MEM

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	FENCE.I	0	MISC-MEM

2.2.5 Control and Status Register Instructions

CSR Instructions

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

Timers and Counters

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
RDCYCLE[H]	0	CSRRS	dest	SYSTEM
RDTIME[H]	0	CSRRS	dest	SYSTEM
RDINSTRET[H]	0	CSRRS	dest	SYSTEM

2.3 Multiplication and Division

2.3.1 Multiplication Operations

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	multiplier	multiplicand	MUL/MULH[[S]U]	dest	OP
MULDIV	multiplier	multiplicand	MULW	dest	OP-32

2.3.2 Division Operations

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	divisor	dividend	DIV[U]/REM[U]	dest	OP
MULDIV	divisor	dividend	DIV[U]W/REM[U]W	dest	OP-32

커널연구회(www.kernel.bz) 정재준 (rgbi3307@nate.com)

2.4 Floating-Point Instructions

31 … 8	7 … 5	4	3	2	1	0
Reserved	Rounding Mode (`frm`)	Accrued Exceptions (`fflags`)				
		NV	DZ	OF	UF	NX
24	3	1	1	1	1	1

Figure 8.2: Floating-point control and status register.

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101		*Invalid. Reserved for future use.*
110		*Invalid. Reserved for future use.*
111		In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *Invalid*.

Table 8.1: Rounding mode encoding.

2.4.1 Single-Precision Load and Store Instructions

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
imm[11:0]	rs1	width	rd	opcode
12	5	3	5	7
offset[11:0]	base	W	dest	LOAD-FP

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
imm[11:5]	rs2	rs1	width	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	W	offset[4:0]	STORE-FP

2.4.2 Double-Precision Load and Store Instructions

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	width	rd	opcode
12	5	3	5	7
offset[11:0]	base	D	dest	LOAD-FP

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	width	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	D	offset[4:0]	STORE-FP

FLD and FSD are only guaranteed to execute atomically if the effective address is naturally aligned and XLEN≥64.

2.4.3 Quad-Precision Load and Store Instructions

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	width	rd	opcode
12	5	3	5	7
offset[11:0]	base	Q	dest	LOAD-FP

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	width	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	Q	offset[4:0]	STORE-FP

FLQ and FSQ are only guaranteed to execute atomically if the effective address is naturally aligned and XLEN=128.

2.5 Compressed Instructions

Format	Meaning	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd′	op
CL	Load	funct3	imm		rs1′	imm	rd′	op
CS	Store	funct3	imm		rs1′	imm	rs2′	op
CB	Branch	funct3	offset		rs1′	offset		op
CJ	Jump	funct3	jump target					op

Table 12.1: Compressed 16-bit RVC instruction formats.

2.6 Atomic Instructions

31–27	26	25	24–20	19–15	14–12	11–7	6–0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
LR	ordering		0	addr	width	dest	AMO
SC	ordering		src	addr	width	dest	AMO

31–27	26	25	24–20	19–15	14–12	11–7	6–0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
AMOSWAP.W/D	ordering		src	addr	width	dest	AMO
AMOADD.W/D	ordering		src	addr	width	dest	AMO
AMOAND.W/D	ordering		src	addr	width	dest	AMO
AMOOR.W/D	ordering		src	addr	width	dest	AMO
AMOXOR.W/D	ordering		src	addr	width	dest	AMO
AMOMAX[U].W/D	ordering		src	addr	width	dest	AMO
AMOMIN[U].W/D	ordering		src	addr	width	dest	AMO

3. RISC-V 어셈블리 언어

3.1 기본명령

RISC-V operands

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory words	Memory[0], Memory[8], ..., Memory[18,446,744,073,709,551,608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands; add
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands; subtract
	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 unsigned 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 constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
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, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal, unsigned
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

3.2 곱셈, 나눗셈 연산 명령

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
	Set if less than	`slt x5, x6, x7`	`x5 = 1 if x5 < x6, else 0`	Three register operands
	Set if less than, unsigned	`sltu x5, x6, x7`	`x5 = 1 if x5 < x6, else 0`	Three register operands
	Set if less than, immediate	`slti x5, x6, x7`	`x5 = 1 if x5 < x6, else 0`	Comparison with immediate
	Set if less than immediate, uns.	`sltiu x5, x6, x7`	`x5 = 1 if x5 < x6, else 0`	Comparison with immediate
	Multiply	`mul x5, x6, x7`	`x5 = x6 × x7`	Lower 64 bits of 128-bit product
	Multiply high	`mulh x5, x6, x7`	`x5 = (x6 × x7) >> 64`	Upper 64 bits of 128-bit signed product
	Multiply high, unsigned	`mulhu x5, x6, x7`	`x5 = (x6 × x7) >> 64`	Upper 64 bits of 128-bit unsigned product
	Multiply high, signed-unsigned	`mulhsu x5, x6, x7`	`x5 = (x6 × x7) >> 64`	Upper 64 bits of 128-bit signed-unsigned product
	Divide	`div x5, x6, x7`	`x5 = x6 / x7`	Divide signed 64-bit numbers
	Divide unsigned	`divu x5, x6, x7`	`x5 = x6 / x7`	Divide unsigned 64-bit numbers
	Remainder	`rem x5, x6, x7`	`x5 = x6 % x7`	Remainder of signed 64-bit division
	Remainder unsigned	`remu x5, x6, x7`	`x5 = x6 % x7`	Remainder of unsigned 64-bit division

3.3 실수 연산 명령

RISC-V floating-point operands

Name	Example	Comments
32 floating-point registers	f0–f31	An f-register can hold either a single-precision floating-point number or a double-precision floating-point number.
2^{61} memory double words	Memory[0], Memory[8], ..., Memory[18,446,744,073,709,551,608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

RISC-V floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	fadd.s f0, f1, f2	f0 = f1 + f2	FP add (single precision)
	FP subtract single	fsub.s f0, f1, f2	f0 = f1 - f2	FP subtract (single precision)
	FP multiply single	fmul.s f0, f1, f2	f0 = f1 * f2	FP multiply (single precision)
	FP divide single	fdiv.s f0, f1, f2	f0 = f1 / f2	FP divide (single precision)
	FP square root single	fsqrt.s f0, f1	f0 = √f1	FP square root (single precision)
	FP add double	fadd.d f0, f1, f2	f0 = f1 + f2	FP add (double precision)
	FP subtract double	fsub.d f0, f1, f2	f0 = f1 - f2	FP subtract (double precision)
	FP multiply double	fmul.d f0, f1, f2	f0 = f1 * f2	FP multiply (double precision)
	FP divide double	fdiv.d f0, f1, f2	f0 = f1 / f2	FP divide (double precision)
	FP square root double	fsqrt.d f0, f1	f0 = √f1	FP square root (double precision)
Comparison	FP equality single	feq.s x5, f0, f1	x5 = 1 if f0 == f1, else 0	FP comparison (single precision)
	FP less than single	flt.s x5, f0, f1	x5 = 1 if f0 < f1, else 0	FP comparison (single precision)
	FP less than or equals single	fle.s x5, f0, f1	x5 = 1 if f0 <= f1, else 0	FP comparison (single precision)
	FP equality double	feq.d x5, f0, f1	x5 = 1 if f0 == f1, else 0	FP comparison (double precision)
	FP less than double	flt.d x5, f0, f1	x5 = 1 if f0 < f1, else 0	FP comparison (double precision)
	FP less than or equals double	fle.d x5, f0, f1	x5 = 1 if f0 <= f1, else 0	FP comparison (double precision)
Data transfer	FP load word	flw f0, 4(x5)	f0 = Memory[x5 + 4]	Load single-precision from memory
	FP load doubleword	fld f0, 8(x5)	f0 = Memory[x5 + 8]	Load double-precision from memory
	FP store word	fsw f0, 4(x5)	Memory[x5 + 4] = f0	Store single-precision from memory
	FP store doubleword	fsd f0, 8(x5)	Memory[x5 + 8] = f0	Store double-precision from memory

3.4 명령 사용 빈도

RISC-V Instruction	Name	Frequency	Cumulative
Add immediate	addi	14.36%	14.36%
Load doubleword	ld	8.27%	22.63%
Load fl. pt. double	fld	6.83%	29.46%
Add registers	add	6.23%	35.69%
Load word	lw	4.38%	40.07%
Store doubleword	sd	4.29%	44.36%
Branch if not equal	bne	4.14%	48.50%
Shift left immediate	slli	3.65%	52.15%
Fused mul-add double	fmadd.d	3.49%	55.64%
Branch if equal	beq	3.27%	58.91%
Add immediate word	addiw	2.86%	61.77%
Store fl. pt. double	fsd	2.24%	64.00%
Multiply fl. pt. double	fmul.d	2.02%	66.02%
Load upper immediate	lui	1.56%	67.59%
Store word	sw	1.52%	69.10%
Jump and link	jal	1.38%	70.49%
Branch if less than	blt	1.37%	71.86%
Add word	addw	1.34%	73.19%
Subtract fl. pt. double	fsub.d	1.28%	74.47%
Branch if greater/equal	bge	1.27%	75.75%

FIGURE 3.24 The frequency of the RISC-V instructions for the SPEC CPU2006 benchmarks.

4. RISC-V Addressing Mode

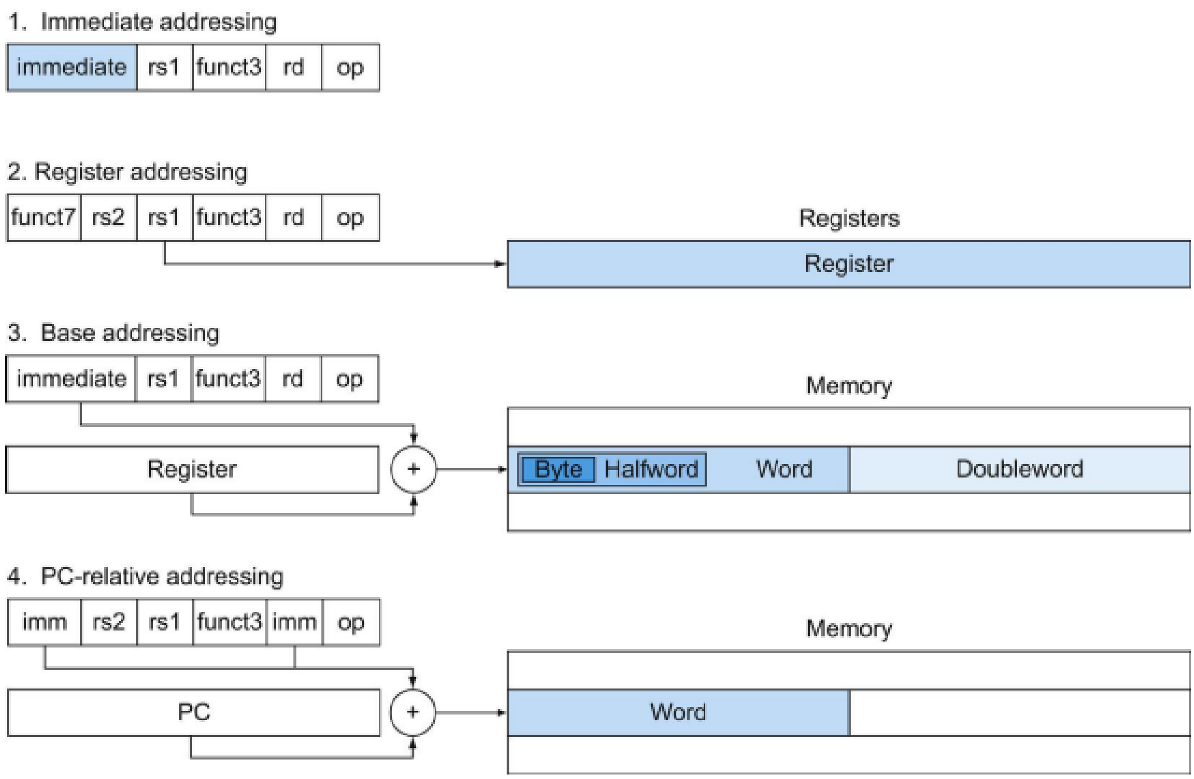


FIGURE 2.17 **Illustration of four RISC-V addressing modes.**

출처: Computer Organization and Design RISC-V Edition (David A. Patterson, John L. Hennessy)

5. Pipelining

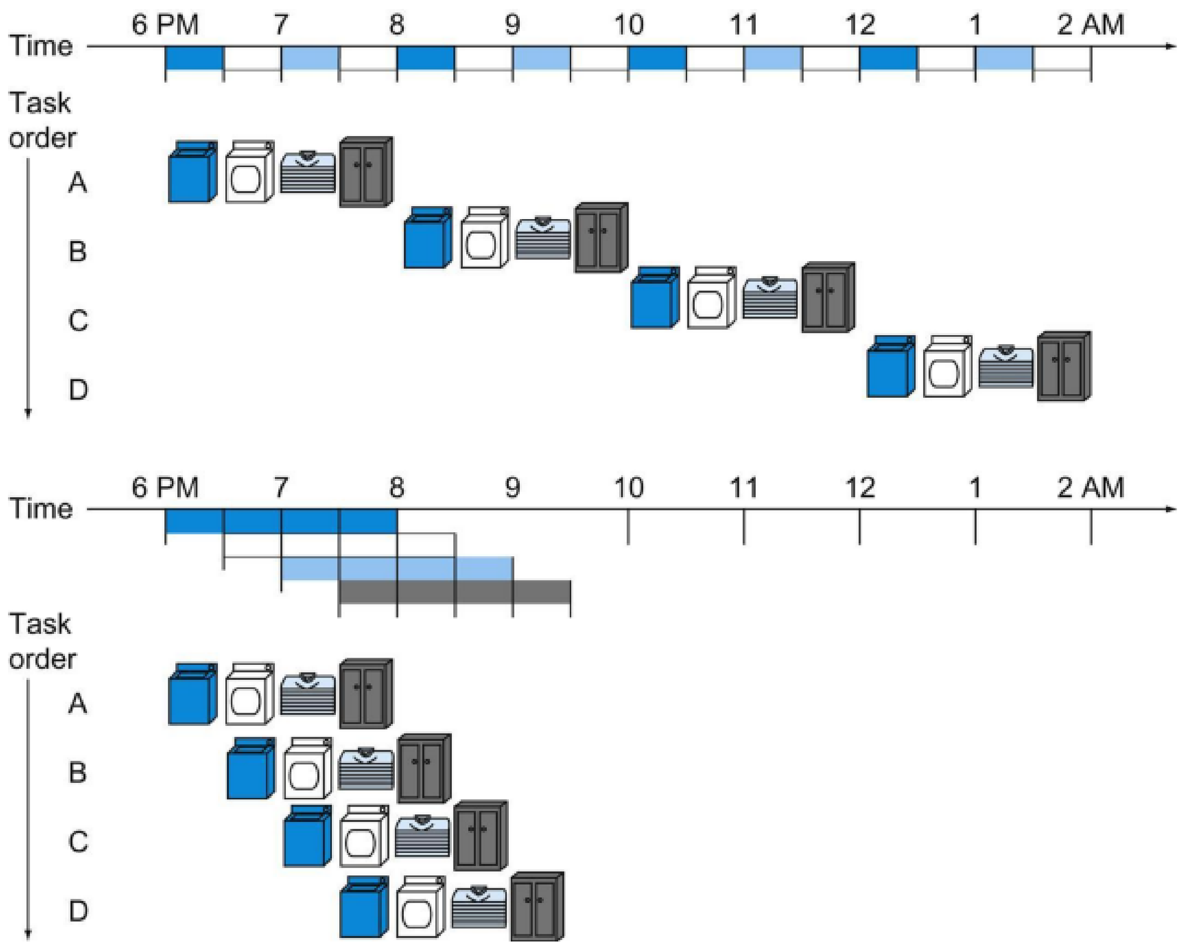


FIGURE 4.23 The laundry analogy for pipelining.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load doubleword (ld)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store doubleword (sd)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

FIGURE 4.24 Total time for each instruction calculated from the time for each component.

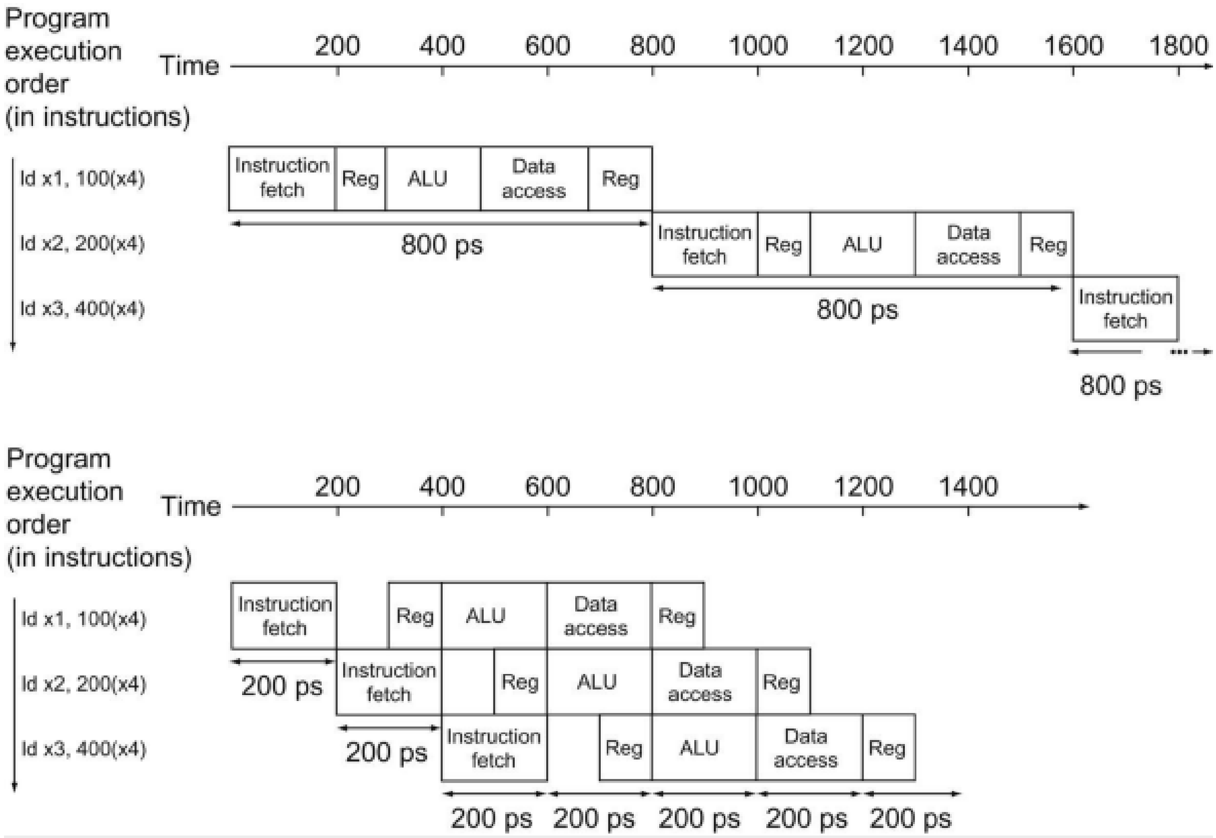
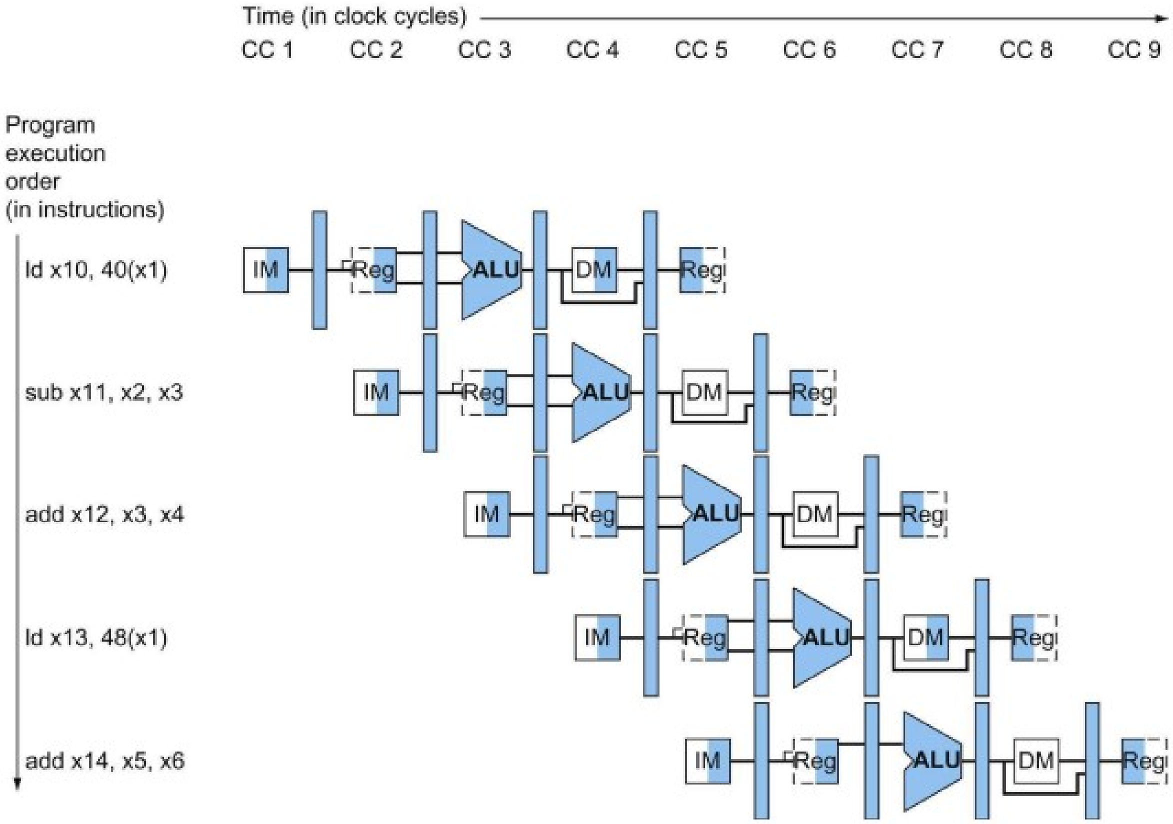


FIGURE 4.25 Single-cycle, nonpipelined execution (top) versus pipelined execution (bottom).



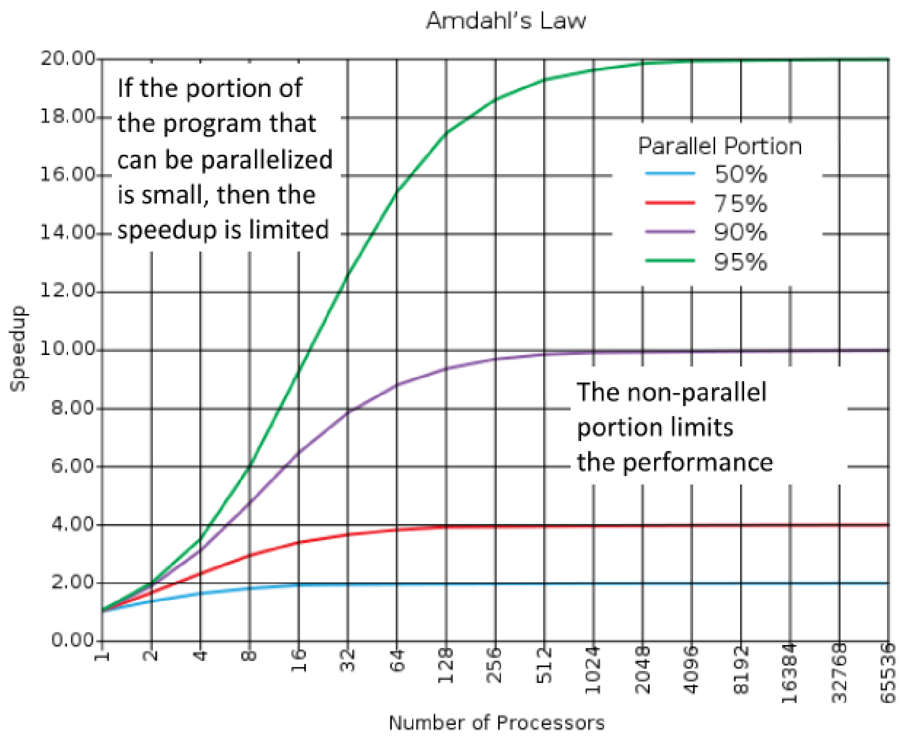
Program execution order (in instructions)	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
ld x10, 40(x1)	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
sub x11, x2, x3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
add x12, x3, x4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
ld x13, 48(x1)				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
add x14, x5, x6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

FIGURE 4.42 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 4.41.

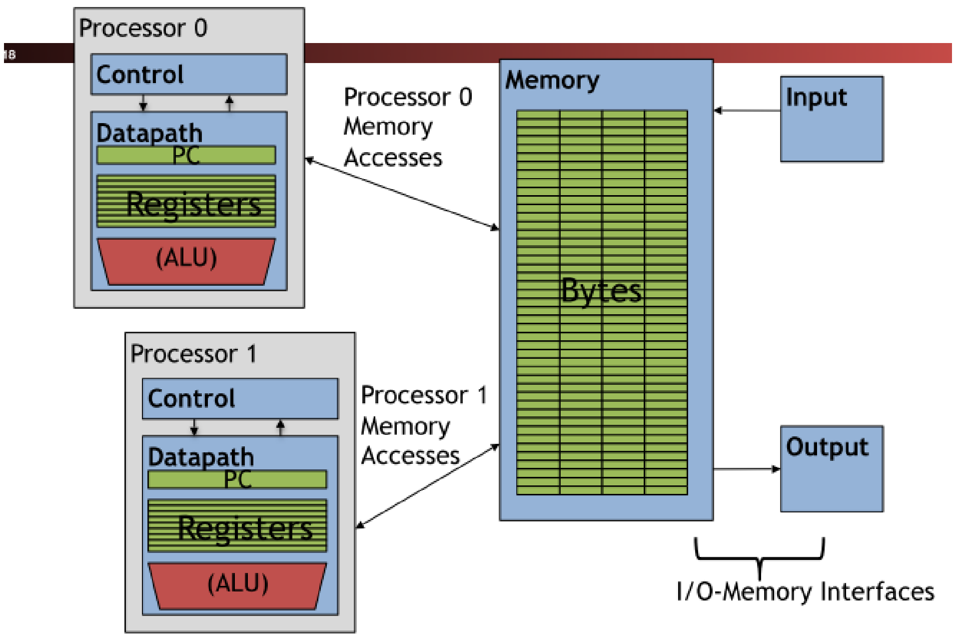
$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipestages}}$$

$$\frac{800{,}002{,}400 \text{ ps}}{200{,}001{,}400 \text{ ps}} \simeq \frac{800 \text{ ps}}{200 \text{ ps}} \simeq 4.00$$

6. 병렬처리



Multiprocessor



동기화, Locking, 메모리 배리어, Atomic 연산

Initial condition on both harts: s0 == address x; s1 == address y.
Initial conditions in memory: all locations initialized to 0

Hart 0	Hart 1

li t1, 1
sw t1, 0(s0)
fence w,w
sw t1, 0(s1)
	loop:
	lw a0, 0(s1)
	beqz a0, loop
	fence r,r
	lw a1, 0(s0)

Final output: what are the possible final values of a0 and a1 on hart 1?

Only possible outcome is a0 == a1 == 1

Synch	Synch thread	I	FENCE
	Synch Instr & Data	I	FENCE.I

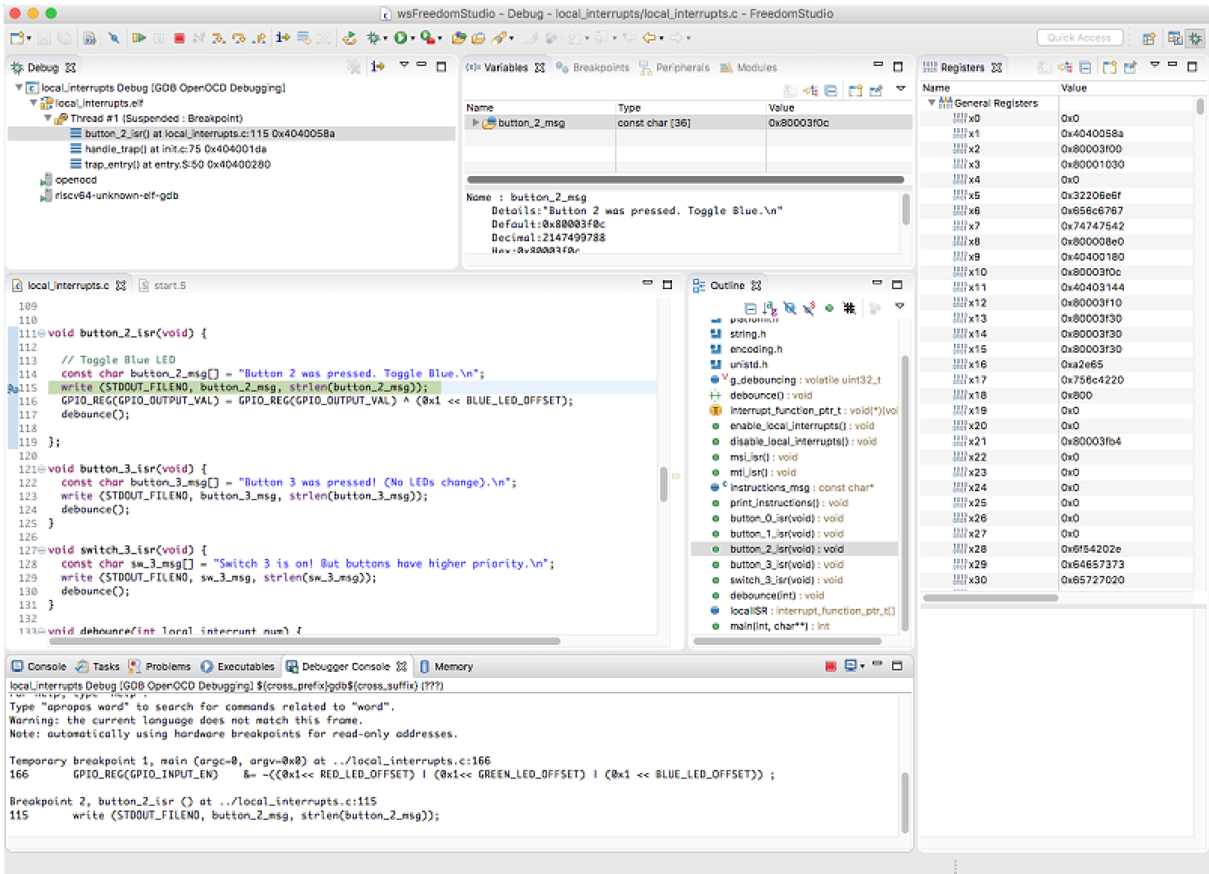
Load	Load Reserved	R	LR.W	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2

어셈블리, 컴파일러에 의존적

7. Freedom Studio

Freedom Studio 는 Eclipse 기반의 통합개발환경(IDE)이다. SiFive 에서 제작한 대부분의 제품들을 이 환경에서 프로그래밍할 수 있도록 툴체인 컴파일러를 포함하고 있다.

Freedom Studio 스크린샷



Freedom Studio 다운로드 경로:

https://www.sifive.com/products/tools/

위의 경로에서 다운로드 받은 파일을 압축해제하면 다음과 같은 파일들이 나타난다.

윈도우용 Freedom Studio 파일 내용

이름	수정한 날짜	유형	크기
build-tools	2018-02-22 오전 8:31	파일 폴더	
eclipse	2018-02-22 오전 8:31	파일 폴더	
jre	2018-02-22 오전 8:29	파일 폴더	
SiFive	2018-02-22 오전 8:31	파일 폴더	
Freedom Studio.exe	2018-01-22 오전 7:22	응용 프로그램	324KB
FreedomStudio_Manual.v1p3.pdf	2018-01-22 오전 7:22	PDF-XChange Vie...	1,314KB

위의 파일 목록에서 SiFive 경로를 보면, 다음과 같이 riscv 용 툴체인 컴파일러가 있는 경로를 확인할 수 있다.

이름	수정한 날짜	유형
Documentation	2018-02-22 오전...	파일 폴더
Drivers	2018-02-22 오전...	파일 폴더
Examples	2018-02-22 오전...	파일 폴더
Licenses	2018-02-22 오전...	파일 폴더
Misc	2018-02-22 오전...	파일 폴더
riscv64-unknown-elf-gcc-20171231-x86_64-w64-mingw32	2018-02-22 오전...	파일 폴더
riscv-openocd-20171231-x86_64-w64-mingw32	2018-02-22 오전...	파일 폴더

SDK(Freedom E SDK) 및 예제 소스들은 아래 링크에서 다운로드할 수 있다.

Freedom E SDK 다운로드 경로:
https://github.com/sifive/freedom-e-sdk

8. RISC-V 커널 소스 분석

8.1 RISC-V 컴파일러(툴체인) 설치

참고 문서:

https://riscv.org/software-tools/

설치 스크립트

```
#!/bin/bash

export CC=gcc
export CXX=g++
export TOP=$(pwd)

git clone https://github.com/riscv/riscv-tools.git

cd $TOP/riscv-tools

git submodule update --init --recursive

sudo apt-get install autoconf automake autotools-dev curl device-tree-compiler libmpc-dev
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc
zlib1g-dev

export RISCV=$TOP/riscv

export PATH=$PATH:$RISCV/bin

./build.sh
```

automake update to 1.14

automake-1.14.tar

커널연구회(www.kernel.bz) 정재준 (rgbi3307@nate.com)

libjaylink/Makefile.am:23: error: Libtool library used but 'LIBTOOL' is undefined

$ libtool --version
libtool (GNU libtool) 2.4.2
Written by Gordon Matzigkeit <gord@gnu.ai.mit.edu>, 1996

libtool update to 2.4.6

```
wget http://ftp.gnu.org/gnu/libtool/libtool-2.4.6.tar.gz
tar -zxvf libtool-2.4.6.tar.gz
cd libtool-2.4.6
./configure --prefix=/usr/
make
sudo make install

echo /usr/share/aclocal > /usr/local/share/aclocal/dirlist
```

```
$ libtool --version
libtool (GNU libtool) 2.4.6
Written by Gordon Matzigkeit, 1996
```

cc1plus: error: unrecognized command line option '-std=c++11'
add -std=gnu++0x to your g++ command line. GCC 4.7 and later support -std=c++11 and -std=gnu++11 as well.

gcc update to 4.9

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install gcc-4.9 g++-4.9
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.9 60 --slave
/usr/bin/g++ g++ /usr/bin/g++-4.9
```

8.2 RISC-V 커널 소스 컴파일

참조: https://www.kernel.org/
참조: https://kernelnewbies.org/KernelBuild
참조: https://riscv.org/software-tools/

8.2.1 최신 리눅스 커널 소스(V4.15 이상) 받기

커널 소스 빌드 도구(패키지) 설치

```
sudo apt-get install libncurses5-dev gcc make git exuberant-ctags bc libssl-dev
```

리눅스 커널 git 보관소에서 최신 안정화 버전 소스를 다운로드 한다.

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
cd linux-stable
```

최신의 안정화된 커널의 tag 을 검색한다.

```
git tag -l | less
```

검색한 커널 tag 을 체크아웃 한다. 여기서 tag 는 보통 커널 버전(vX.Y.Z)으로 나타난다.

```
git checkout -b stable tag
```

참고로, 현재 개발 진행중인 최신의 리눅스 커널 –rc 소스는 다음과 같이 다운로드 한다.

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
cd linux
```

8.2.2 RISC-V 커널 소스 빌드

RISC-V 커널 소스 빌드(컴파일)는 다음과 같이 명령어를 실행한다.

환경 설정

```
$ make  ARCH=riscv  defconfig

또는

$ make  ARCH=riscv  menuconfig
```

.config 파일

```
CONFIG_RISCV=y
CONFIG_MMU=y
CONFIG_ARCH_PHYS_ADDR_T_64BIT=y
CONFIG_ARCH_DMA_ADDR_T_64BIT=y
CONFIG_PAGE_OFFSET=0xffffffe000000000
CONFIG_STACKTRACE_SUPPORT=y
CONFIG_RWSEM_GENERIC_SPINLOCK=y
CONFIG_GENERIC_BUG=y
CONFIG_GENERIC_BUG_RELATIVE_POINTERS=y
CONFIG_GENERIC_CALIBRATE_DELAY=y
CONFIG_GENERIC_CSUM=y
CONFIG_GENERIC_HWEIGHT=y
CONFIG_PGTABLE_LEVELS=3
CONFIG_DMA_NOOP_OPS=y

# Platform type
# CONFIG_ARCH_RV32I is not set
CONFIG_ARCH_RV64I=y
# CONFIG_CMODEL_MEDLOW is not set
CONFIG_CMODEL_MEDANY=y
# CONFIG_MAXPHYSMEM_2GB is not set
```

```
CONFIG_MAXPHYSMEM_128GB=y
CONFIG_SMP=y
CONFIG_NR_CPUS=8
CONFIG_CPU_SUPPORTS_64BIT_KERNEL=y
CONFIG_TUNE_GENERIC=y
CONFIG_RISCV_ISA_C=y
CONFIG_RISCV_ISA_A=y

# Kernel type
CONFIG_64BIT=y
CONFIG_FLATMEM=y
CONFIG_FLAT_NODE_MEM_MAP=y
CONFIG_HAVE_MEMBLOCK=y
CONFIG_NO_BOOTMEM=y
CONFIG_SPLIT_PTLOCK_CPUS=4
CONFIG_COMPACTION=y
CONFIG_MIGRATION=y
CONFIG_PHYS_ADDR_T_64BIT=y
CONFIG_DEFAULT_MMAP_MIN_ADDR=4096
CONFIG_PREEMPT_NONE=y
CONFIG_HZ_250=y
CONFIG_HZ=250

//이하 생략..
```

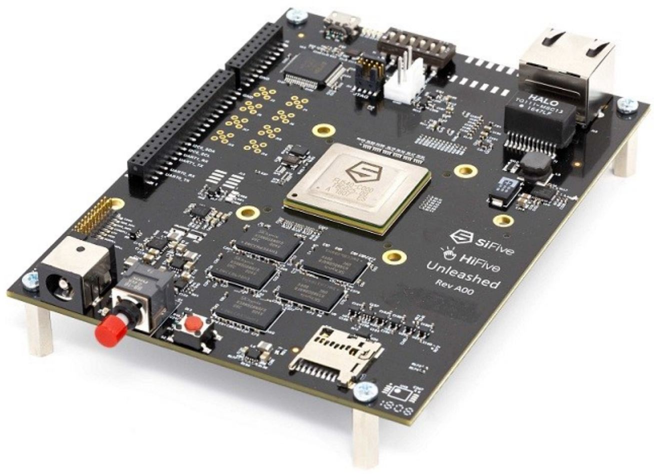
빌드(컴파일)

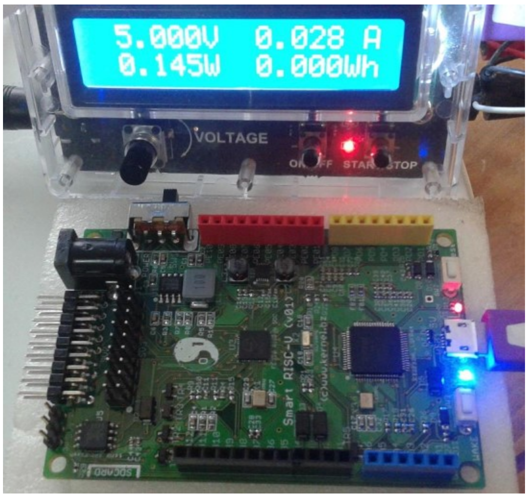
```
$ make ARCH=riscv CROSS_COMPILE=riscv64-unknown-elf-

또는

$ make -j# ARCH=riscv CROSS_COMPILE=riscv64-unknown-elf-
```

8.3 RISC-V 커널 소스 분석





RISC-V U540(64 비트) SiFive 제작　　**RISC-V E31(32 비트) 커널연구회 제작**

https://www.sifive.com/products/risc-v-core-ip/e5/e51/

https://elixir.bootlin.com/linux/latest/source

8.3.1 중요 소스 경로(요약)

```
arch/riscv/
init/
kernel/
kernel/irq/
kernel/time/
kernel/locking/
kernel/rcu/
kernel/sched/
mm/
fs/
fs/proc
fs/sysfs
drivers/base/
drivers/of/
```

기타 문의 사항은 커널연구회 정재준(rgbi3307@nate.com)에게 메일 주시기 바랍니다.

참고로, 리눅스 커널 RISC-V 소스 분석 스터디 모임을 정기적으로 하고 있습니다.

공부모임 내용공유:
https://github.com/kernel-bz/riscv-study

관련하여 커널연구회 웹사이트 참고해 주시기 바랍니다.

https://www.kernel.bz

감사합니다.