RISC-V

System-on-Chip Design

Harris, Stine, Thompson & Harris

Chapters 2:

RISC-V Introduction

Chapter 2 :: Topics

RISC-V Introduction

2.1 RISC-V Assembly Language

2.2 RISC-V Machine Language

2.3 Have a Hart

2.4 Memory Map

2.5 RISC-V Extensions & Profiles

2.6 Comparison with other Architectures

2.7 RISC-V Microarchitecture

2.8 Wally SoC

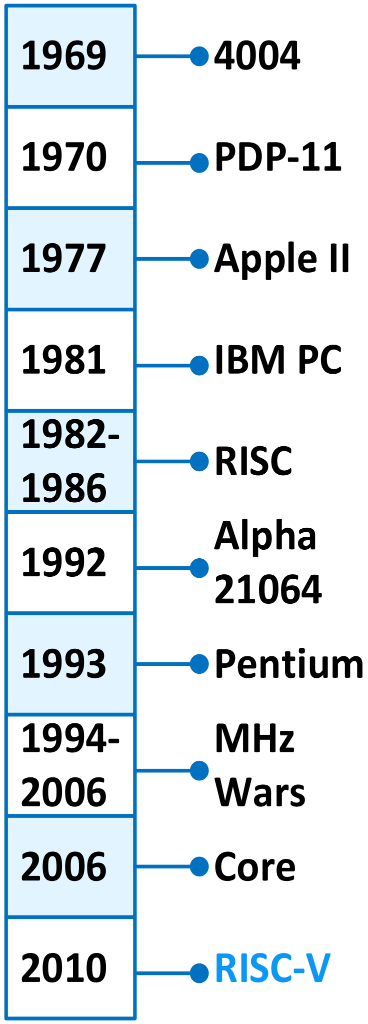
2.9 RISC-V Market

Chapter 1: RISC-V Introduction

RISC-V Intro

RISC-V

- Developed by Krste Asanovic, Andrew Waterman, Yunsup Lee, David Patterson and their colleagues at UC Berkeley in 2010.
- First widely accepted open-source computer architecture
- Called RISC-V (pronounced “risk five”) because it was the 5th generation RISC processor developed at Berkeley



1969	4004
1970	PDP-11
1977	Apple II
1981	IBM PC
1982-1986	RISC
1992	Alpha 21064
1993	Pentium
1994-2006	MHz Wars
2006	Core
2010	RISC-V

Kriste Asanovic

- Professor of Computer Science at the University of California, Berkeley
- Began developing RISC-V during summer 2010
- Chairman of the Board of RISC-V International, RISC-V's governing board
- Co-Founder of SiFive, a company that commercializes and develops supporting tools for RISC-V

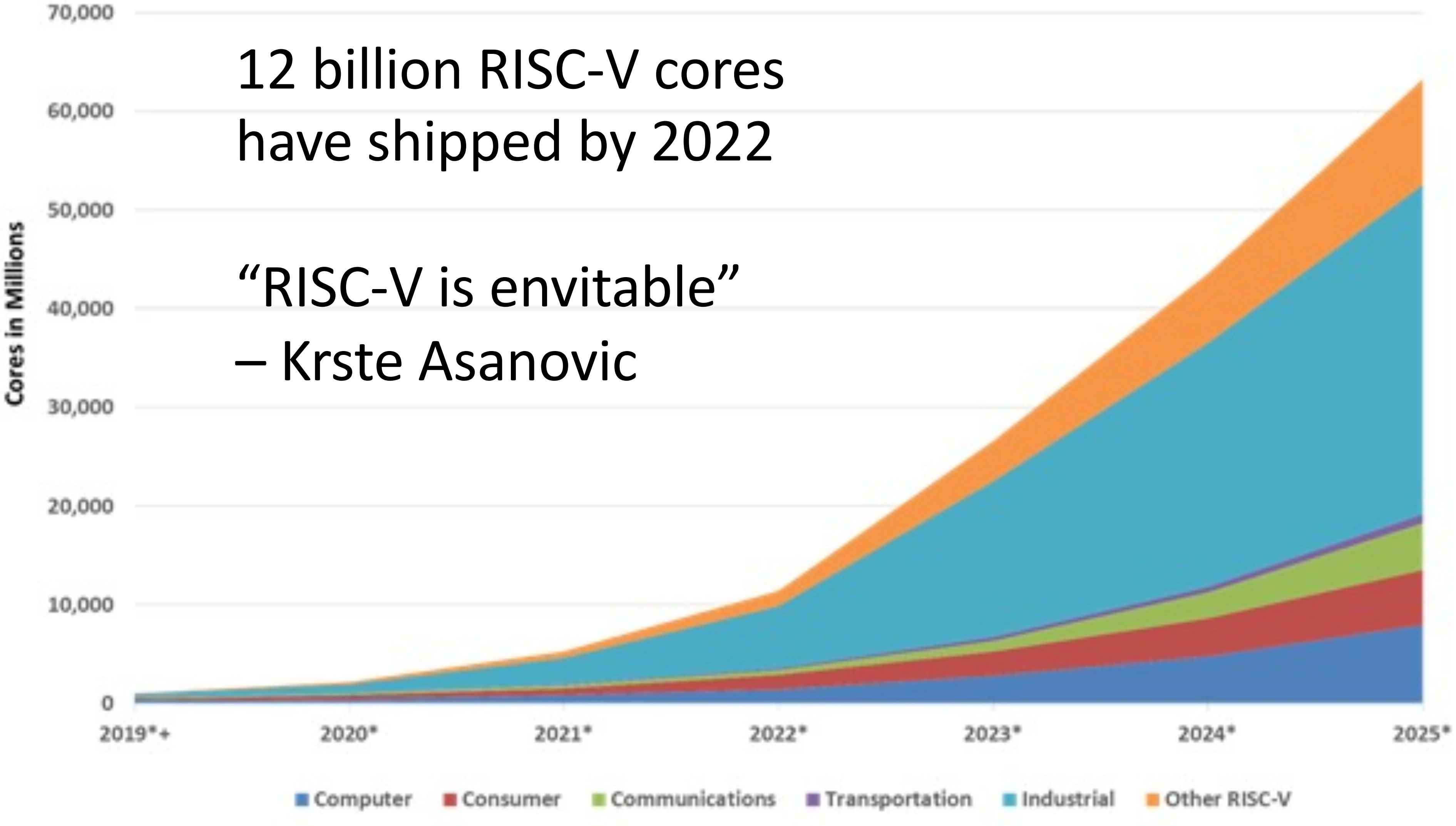
Photo used with permission.

Andrew Waterman

- Weary of existing instruction set architectures (ISAs), he co-designed the RISC-V architecture and the first RISC-V cores
- Co-founded SiFive with Krste Asanovic
- Earned his PhD in computer science from UC Berkeley in 2016

Photo used with permission.

Forecast RISC-V Market Growth



12 billion RISC-V cores
have shipped by 2022

“RISC-V is envitable”
– Krste Asanovic

Source: Semico Research Corp.

RISC-V ISA

- **Officially defined at:** https://riscv.org/specifications/ratified/
- **Base ISAs:**
  - RV32I (**XLEN** = 32)
  - RV64I (**XLEN** = 64)

RV128I and RV32E (embedded) are also in development

- **Extensions:**
  - **A:** Atomic memory accesses
  - **C:** Compressed instructions
  - **F, D:** Float/double precision
  - **M:** Integer multiply/divide
  - **G:** IACFDM
- **Applications processors:**
  - Run multitasking OS with graphical user interface
  - Typically RV64GC
- **Embedded processors:**
  - Execute smaller operations, including driving motors, flashing LEDs, etc.
  - Typically RV32IC

RISC-V International: the governing organization and community for developing RISC-V technical specifications (riscv.org)

Chapter 1: RISC-V Introduction

Assembly Language

RISC-V Assembly Language

- **RV32I:**
  - Integer operations
  - 40 instructions
- **RV64I** adds instructions:
  - For 64-bit (double-word) loads/stores
  - To operate on bottom 32 bits of 64-bit register

See inside cover of textbook for list of RISC-V instructions.

RV32I: Integer Instructions

- **R-Type** (Register-Type): Register operands only
- **I-Type** (Immediate-Type): Immediate & register operands
- **S-Type** (Store-Type): Stores
- **B-Type** (Branch-Type): Branches
- **J-Type** (Jump-Type): Jump
- **U-Type** (Upper-Imm.-Type): Upper-immediate operand

All RV32I instructions are also part of RV64I. These instructions operate on XLEN-bit operands.

R-Type Instructions

Register-Type Instructions:

- Arithmetic: `add`, `sub`
- Logical: `and`, `or`, `xor`
- Shift: `sll`, `srl`, `sra` (shift left/right logical/arithmetic)
- Comparison: `slt`, `sltu` (set if less than/unsigned)
  - **`sltu`** (set if less than unsigned) compares the operands as if they're unsigned. `slt` treats them as signed (2's complement)

Examples:

```
add s0, s1, s2  # s0 = s1 + s2
or  t0, t1, t2  # t0 = t1 | t2
```

I-Type Instructions

Immediate-Type Instructions:

– Arithmetic:	`addi`
– Logical:	`andi`, `ori`, `xori`
– Shift:	`slli`, `srli`, `srai`
– Comparison:	`slti`, `sltiu`
– Loads:	`lw`, `lh`, `lb`, `lhu`, `lbu`
– Jumps:	`jalr`

- The processor **sign-extends** the immediate before using it in **all operations** except shifts.
- So, the immediate is sign-extended for **logical operations** and before treating it as an unsigned number in **`sltiu`**.

Examples:

```
addi s0, s1, -15   # s0 = s1 - 15
xori t2, t3, -1    # t2 = t3 ^ 0xFFFFFFFF
slli t0, t1, 7     # t0 = t1 << 7
lw   t0, 0x24(s3)  # t0 = memory[s3 + 0x24]
jalr s5            # PC = s5, ra = PC + 4
```

S-Type Instructions

Store-Type Instructions:

- sw, sh, sb
- Store word, half, byte

Examples:

```
sw s0, 0x2c(t1)  # memory[t1 + 0x2c] = s0
sh t1, 0x30(s2)  # memory[s2 + 0x30]_{15:0} = t1_{15:0}
sb t3, 0x8c(s7)  # memory[s7 + 0x8c]_{7:0}  = t3_{7:0}
```

B-Type Instructions

Branch-Type Instructions:

– Branch if equal:	`beq`
– Branch if not equal:	`bne`
– Branch if less than:	`blt`
– Branch if greater than or equal:	`bge`
– Branch if less than (unsigned):	`bltu`
– Branch if greater than or equal (unsigned):	`bgeu`

Examples:

```
beq s0, s1, L1  # if (s0 == s1), goto L1
blt t0, t1, L3  # if (t0 <  t1), goto L3
```

J-Type Instruction

Jump-Type Instruction:

- Jump and Link: `jal`

Example:

```
jal     L1  # ra = PC + 4, goto L1
jal s3, L7  # s3 = PC + 4, goto L7
```

U-Type Instructions

Upper-Immediate-Type Instructions:

- Load upper-immediate: `lui`
- Add upper-immediate to PC: `auipc`

Examples:

```
# Load 20-bit immediate into upper bits, lower bits are 0s
lui  t1, 0xABCDE      # t1 = 0xABCDE000
# Load a 32-bit immediate (0xBBCCD123) into a register
lui  s3, 0xBBCCD      # s3 = 0xBBCCD000
addi s3, s3, 0x123    # s3 = 0xBBCCD000+0x123 = 0xBBCCD123
# Load a 32-bit immediate (0x12345ABC) into a register
lui  s1, 0x12346      # s1 = 0x12346000
addi s1, s1, 0x123    # s1 = 0x12346000+0xFFFFFABC = 0x12345ABC
# Move PC (+ upper-immediate) into a register
auipc s5, 0           # s5 = PC
auipc s11, 0xAABBC    # s11 = PC + {0xAABBC, 12’b0}
```

RV64I: Additional Instructions

- **Load and store double-words (64-bit word):** `ld`, `sd`
- **Load word unsigned:** `lwu`
- **32-bit (word) operations in 64-bit registers:** `addw`, `subw`, `sllw`, `srlw`, `sraw`
  - **Immediate versions:** `addiw`, `slliw`, `srliw`, `sraiw`

RV64I **word operations** operate on the **lower 32 bits** of a register, **discard the upper 32 bits** and **sign-extend** the result into the upper bits of the register. This is important to support 32-bit *int* data types efficiently on a 64-bit processor for languages such as C.

RV64I Word Operations

Example: `addw s0, s1, s2`

`s1` and `s2` should be treated as 32-bit operands (in lower half of register).

Suppose:

`s1` = `s2` = `0xFFFF_FFFF_`**`8000_0000`**

Performed on 32-bit ALU:

`0x8000_0000 + 0x8000_0000 =` **`0x0000_0000`**

Performed on 64-bit ALU (numbers are sign-extended):

`0xFFFF_FFFF_8000_0000 + 0xFFFF_FFFF_8000_0000 =`
`0xFFFF_FFFF_0000_0000`

But the result should be 0 for a 32-bit operation. So,

- Discard upper 32 bits: 0x~~**FFFF_FFFF**~~_0000_0000
- Sign-extend to 64 bits: 0x**0000_0000**_0000_0000

Chapter 1: Introduction

Registers & Conventions

RISC-V Register

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

Register Naming & Conventions

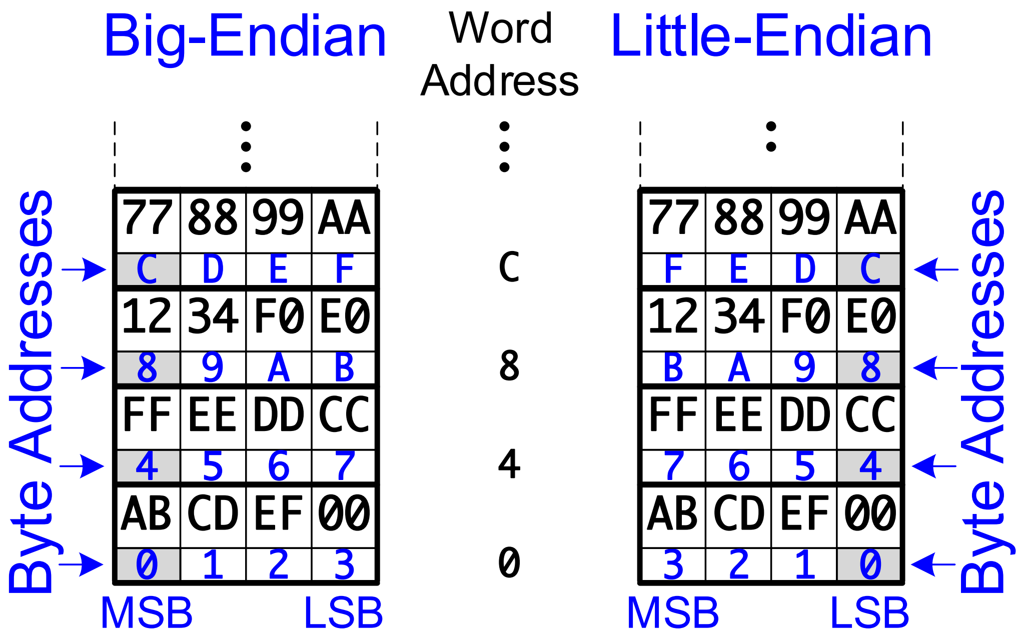
- **Registers:**
  - Can use either name (i.e., `ra`, `zero`) or `x0`, `x1`, etc.
  - Using name is preferred
- Registers used for **specific purposes**:
  - **`zero`**: always holds the **constant value 0**.
  - **`s0-s11`**: the ***saved registers***, used to hold variables
  - **`t0-t6`**: the ***temporary registers***, used to hold intermediate values during a larger computation
  - **`ra`**: return address register
  - **`sp`**: stack pointer (address of top of stack)
  - **`a0-a7`**: argument registers
  - **`a0`**: also the return register

RV32E

- Register file is the biggest part of a simple embedded RV32I core
  - RV32E reduces register file to only 16 regs
  - x0-x15
    - Still includes t0-t2, s0-s1, a0-a5
    - And all the special purpose registers

Endianness

- How bytes are numbered (addressed) within a word in memory:
  - **Big-endian:** Address numbers start at the big end (MSB) and increase toward the LSB.
  - **Little-endian:** Address numbers start at the LSB and increase toward the MSB.



Big-Endian				Word Address	Little-Endian			
⋮				⋮	⋮			
77	88	99	AA		77	88	99	AA
C	D	E	F	C	F	E	D	C
12	34	F0	E0		12	34	F0	E0
8	9	A	B	8	B	A	9	8
FF	EE	DD	CC		FF	EE	DD	CC
4	5	6	7	4	7	6	5	4
AB	CD	EF	00		AB	CD	EF	00
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB

Big-Endian & Little-Endian Example

- Suppose 0x23456789 is stored at address 0x3C.
- What value is in `s0` after this instruction executes in a **big-endian** system? `lb s0, 0x3D(zero)`
- In a **little-endian** system?

Big-Endian & Little-Endian Example

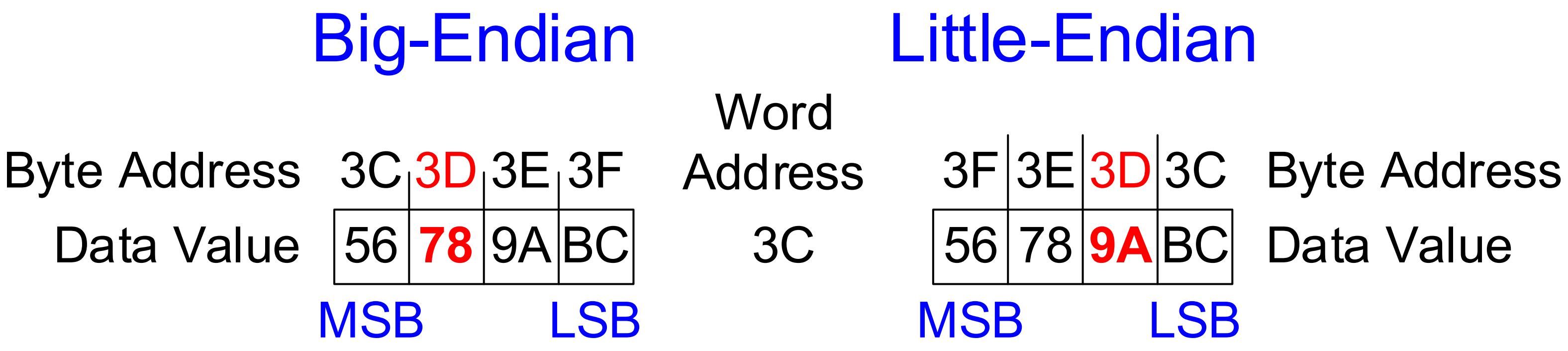
- Suppose 0x56789ABC is stored at address 0x3C.
- What value is in `s0` after this instruction executes in a **big-endian** system? `lb s0, 0x3D(zero)`
- In a **little-endian** system?

Big-endian:

`s0` = 0x000000**78**

Little-endian:

`s0` = 0xFFFFFF**9A**



Big-Endian

Byte Address	3C	3D	3E	3F
Data Value	56	78	9A	BC
	MSB			LSB

Word Address: 3C

Little-Endian

3F	3E	3D	3C	Byte Address
56	78	9A	BC	Data Value
MSB			LSB	

Remember that `lb` sign-extends the byte.

Chapter 1: Introduction

Pseudoinstructions

Pseudoinstructions

- **Pseudoinstructions:** instructions that are convenient for the programmer that can be implemented with existing RISC-V instructions.
- Assembler converts them to real RISC-V instructions.

Some RISC-V Pseudoinstructions

Pseudoinstruction	RISC-V Instructions
j label	jal zero, label
jal label	jal ra, label
jr ra	jalr zero, ra, 0
mv t5, s3	addi t5, s3, 0
not s7, t2	xori s7, t2, -1
nop	addi zero, zero, 0
li s8, 0x56789DEF	lui s8, 0x5678A addi s8, s8, 0xDEF
bgt s1, t3, L3	blt t3, s1, L3
bgez t2, L7	bge t2, zero, L7
call L1	auipc ra, imm$_{31:12}$ jalr ra, ra, imm$_{11:0}$
ret	jalr zero, ra, 0

See inside covers for more pseudoinstructions.

C to Assembly Examples

C to Assembly Examples: Variables

- For the following programs, assume these variables have been declared and are kept in these registers:

```
int a, b, c;     // in s0, s1, s2, respectively
int i, j;        // in s3, s4
int grades[100]; // base address in s5
char name[80];   // base address in s6
```

Simple Operations

C Code	RISC-V Assembly Code
a = b + c;	add s0, s1, s2 # a = b + c
a = b & 0xFF;	andi s0, s1, 0xFF # 0xFF = 8 1s

Conditional Statements: if, if/else

C Code	**RISC-V Assembly Code**

`if (i == j)`
`  a = 1;`
	` bne s3, s4, done # skip if i != j`
	` li s0, 1          # a = 1`
	`done:`

`if (i == j)`
`  a = 1;`
 	
`else`
`  a = 2;`
	` bne s3, s4, else # skip if i != j`
	` li s0, 1          # a = 1`
	` j done`
	`else:`
	` li s0, 2          # a = 2`
	`done:`

The C code checks (`i == j`) and the assembly code checks the opposite condition (`i != j`, using bne).

Loop: while

C Code

```c
a = 0;
i = 0;

while (i <= 10){

  a = a + i;
  i = i + 2;
}
```

RISC-V Assembly Code

```
 li s0, 0         # a = 0
 li s3, 0         # i = 0
 li t0, 10        # need 10 for compare
while:
 bgt s3, t0, done # done if i > 10
 add s0, s0, s3   # a = a + i
 addi s3, s3, 2   # i = i + 2
 j while          # repeat while loop
done:
```

Loop: for

C Code

```
a = 1;

for (i=1; i<=b; i++)

  a = a * 2;
```

RISC-V Assembly Code

```
 li s0, 1         # a = 1
 li s3, 1         # i = 1
for:
 bgt s3, s1, done # done if i > b
 slli s0, s0, 1   # a = a * 2
 addi s3, s3, 1   # i++
 j for            # repeat for loop
done:
```

Arrays Example 1

C Code

```
grades[1] = grades[0] + 17;
```

RISC-V Assembly Code

```
# t0 = grades[0]
lw t0, 0(s5)

# t0 = t0 + 17
addi t0, t0, 17

# grades[1] = t0
sw t0, 4(s5)
```

Arrays Example 2

C Code

```
for (i=0; i<100; i++)

 grades[i]=grades[i]+14;
```

RISC-V Assembly Code

```
 li s3, 0           # i = 0
 li t0, 100         # t0 = 100 for <
for:
 bge s3, t0, done # t0 >= 100? done
 slli t1, s3, 2     # t1 = i*4
 add t1, s5, t1     # t1 = &grades[i]
 lw t2, 0(t1)       # t2 = grades[i]
 addi t2, t2, 14    # grades[i] += 14
 sw t2, 0(t1)       # grades[i] = t2
 addi s3, s3, 1     # i++
 j for              # repeat
done:
```

Arrays Example 3: String Handling

C Code

```
i = 0;
while (name[i])
  i++;
```

RISC-V Assembly Code

```
       li s3, 0
while: add t0, s6, s3
       lb t1, 0(t0)
       beq  t1, zero, done
       addi s3, s3, 1
       j while
done:
```

Function Call: Factorial

C Code

```c
// s0 = result
// s1 = i
int fact(int n) {
 int result = 1;
 int i;
 for (i=1; i<=n; i++)
  result = result*i;


 return result;
}
```

RISC-V Assembly Code

```
fact: addi sp, sp, -8   # room on stack
      sw s0, 0(sp)      # save s0
      sw s1, 4(sp)      # save s1
      li s0, 1          # result = 1
      li s1, 1          # i = 1
for:  bgt s1, a0, done  # i > n?
      mul s0, s0, s1    # result *= i
      addi s1, s1, 1    # i++
      j for             # repeat
done: mov a0, s0        # return result
      lw s1, 4(sp)      # restore s1
      lw s0, 0(sp)      # restore s0
      addi sp, sp, 8    # restore stack
      ret               # return
```

Recursive Function Call: Factorial

C Code

```c
int factr(int n) {
 if (n<=1)
   return 1;

 else



  return n*factr(n-1);
}
```

RISC-V Assembly Code

```asm
factr: li t0, 1          # for compare
       bgt a0, t0, else # n > 1
       li a0, 1          # return 1
       ret               # return
else:  addi sp, sp, -8   # room on stack
       sw ra, 0(sp)      # save ra
       sw a0, 4(sp)      # save n
       addi a0, a0, -1   # n-1
       jal factr         # factr(n-1)
       lw t0, 4(sp)      # restore n
       lw ra, 0(sp)      # restore ra
       addi sp, sp, 8    # restore stack
       mul a0, t0, a0    # n*factr(n-1)
       ret
```

RISC-V Machine Language

RISC-V Instruction Formats

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
funct7	rs2	rs1	funct3	rd	op	**R-Type**
$\text{imm}_{11:0}$		rs1	funct3	rd	op	**I-Type**
$\text{imm}_{11:5}$	rs2	rs1	funct3	$\text{imm}_{4:0}$	op	**S-Type**
$\text{imm}_{12,10:5}$	rs2	rs1	funct3	$\text{imm}_{4:1,11}$	op	**B-Type**
$\text{imm}_{31:12}$				rd	op	**U-Type**
$\text{imm}_{20,10:1,11,19:12}$				rd	op	**J-Type**
20 bits				5 bits	7 bits	

RISC-V Instruction Fields

- **Operands:**
  - `rs1`, `rs2`: Source registers
  - `rd`: Destination register
  - `imm`: 12-, 13-, 21-, or 32-bit immediate
- **Control fields** – indicate what operation to perform:
  - `op`: 7-bit operation code or opcode
  - `funct7`, `funct3`: 7- and 3-bit function

RISC-V Instruction Fields

When present in an instruction format, all fields (except `imm`) are located in same instruction bits:

- `op`: Bits 6:0
- `rs2`, `rs1`, `rd`: Bits 24:20, 19:15, 11:7
- `funct7`, `funct3`: Bits 31:25, 14:12

R-Type

31:25	24:20	19:15	14:12	11:7	6:0
funct7	rs2	rs1	funct3	rd	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

I-Type

31:20	19:15	14:12	11:7	6:0
$\text{imm}_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

RISC-V Immediate Fields

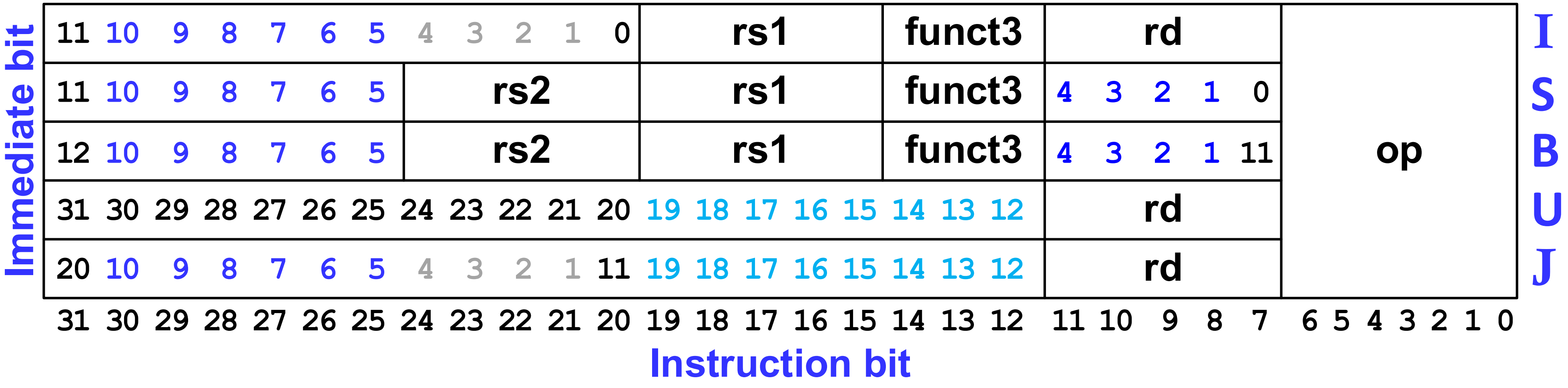
Immediate bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	11	10	9	8	7	6	5	4	3	2	1	0	rs1					funct3			rd					op							I
	11	10	9	8	7	6	5	rs2					rs1					funct3			4	3	2	1	0								S
	12	10	9	8	7	6	5	rs2					rs1					funct3			4	3	2	1	11								B
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	rd												U
	20	10	9	8	7	6	5	4	3	2	1	11	19	18	17	16	15	14	13	12	rd												J

Instruction bit

Immediate bit aims to stay in the same instruction bit, but varying immediate sizes and operands require exceptions.

- **I-type:** 12-bit signed immediate
- **S-type:** 12-bit signed immediate
- **B-type:** 13-bit signed immediate
- **U-type:** 32-bit upper immediate
- **J-type:** 21-bit signed immediate

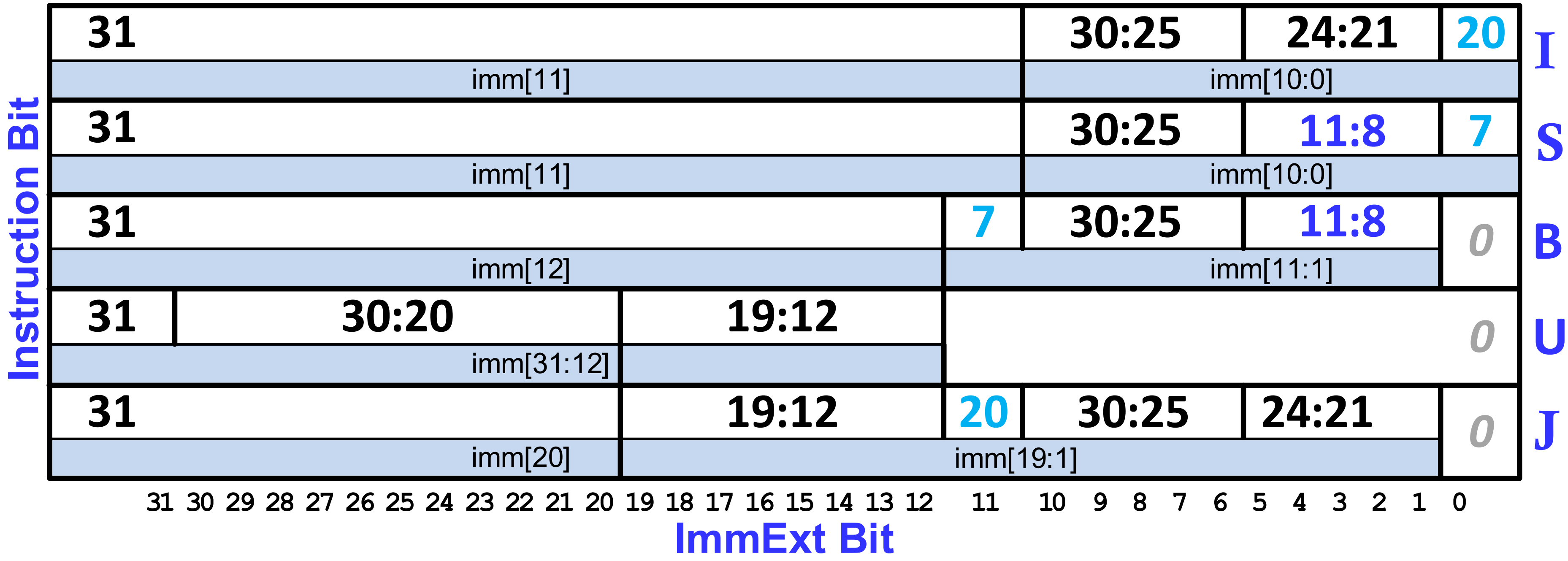
Composition of 32-bit Immediates



Immediate bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
11	10	9	8	7	6	5	4	3	2	1	0	rs1					funct3			rd					op							I
11	10	9	8	7	6	5	rs2					rs1					funct3			4	3	2	1	0								S
12	10	9	8	7	6	5	rs2					rs1					funct3			4	3	2	1	11								B
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	rd												U
20	10	9	8	7	6	5	4	3	2	1	11	19	18	17	16	15	14	13	12	rd												J

Instruction bit



Instruction Bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
31																					30:25						24:21				20	I
imm[11]																					imm[10:0]											
31																					30:25						11:8				7	S
imm[11]																					imm[10:0]											
31																				7	30:25						11:8				0	B
imm[12]																				imm[11:1]												
31	30:20										19:12								0													U
imm[31:12]																																
31												19:12								20	30:25						24:21				0	J
imm[20]												imm[19:1]																				

ImmExt Bit

Assembly to Machine Code

Convert to machine language:

```
    sub x3, x4, x5
```

Solution: According to Table 1.2, `sub` is an R-type instruction with:

- `op` = 0110011
- `funct3` = 000, `funct7` = 0100000
- `rs1` = 4 = 00100, `rs2` = 5 = 00101, `rd` = 3 = 00011

funct7	**rs2**	**rs1**	**funct3**	**rd**	**op**	
0100000	00101	00100	000	00011	0110011	= **0x405201B3**

Machine Code to Assembly

Convert to assembly language:

`0x03E90523`

Solution:

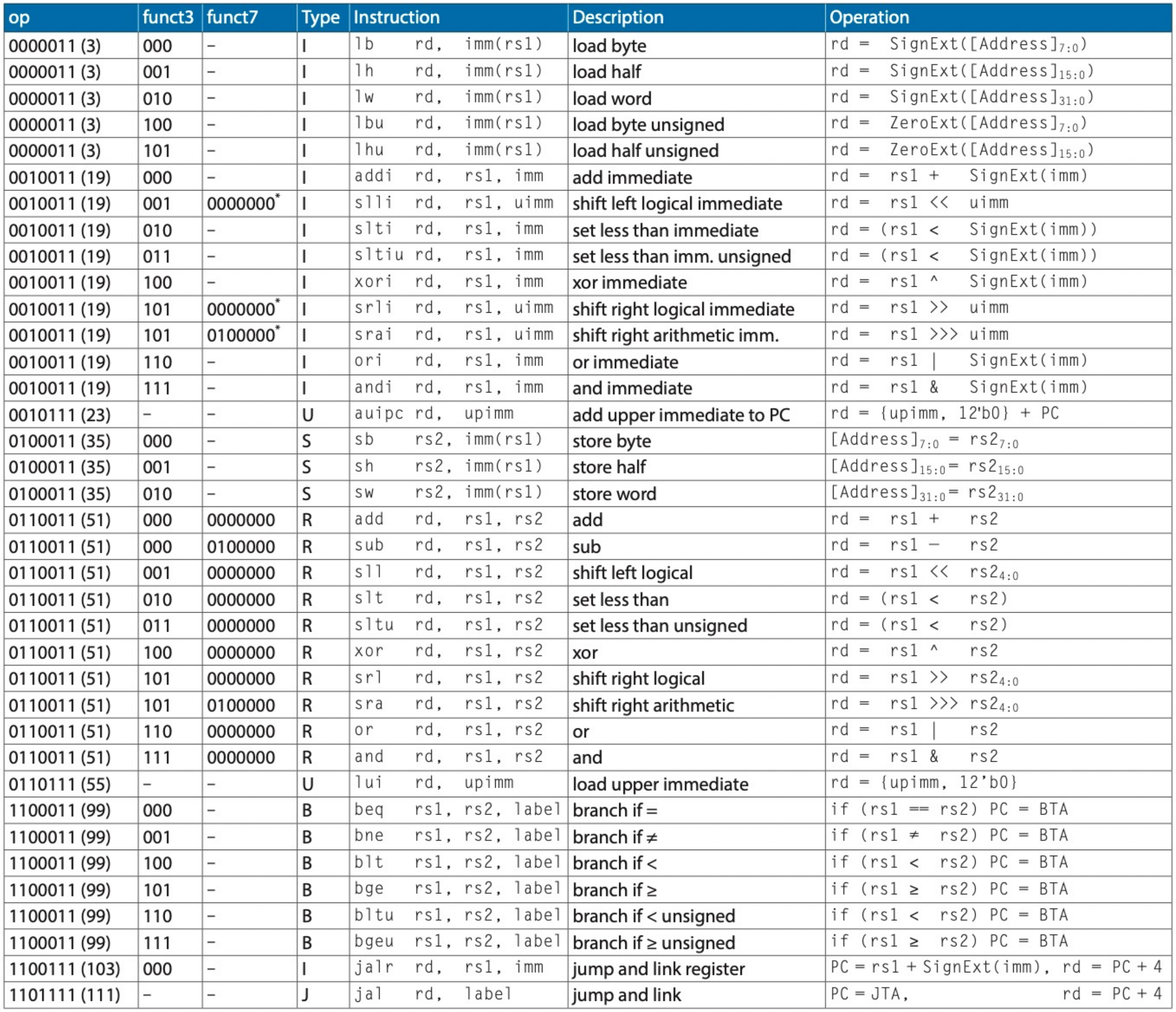
- Write in binary: 0000 0011 1110 1001 0000 0101 0**010 0011**
- Start with **op**: tells how to parse rest: **010 0011** = S-type
- Extract fields:

$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
0000001	11110	10010	000	01010	0100011

- op = 0100011, funct3 = 00: sb
- rs2 = 11110 (x30), rs1 = 10010 (x18)
- imm = 000000101010 = 42

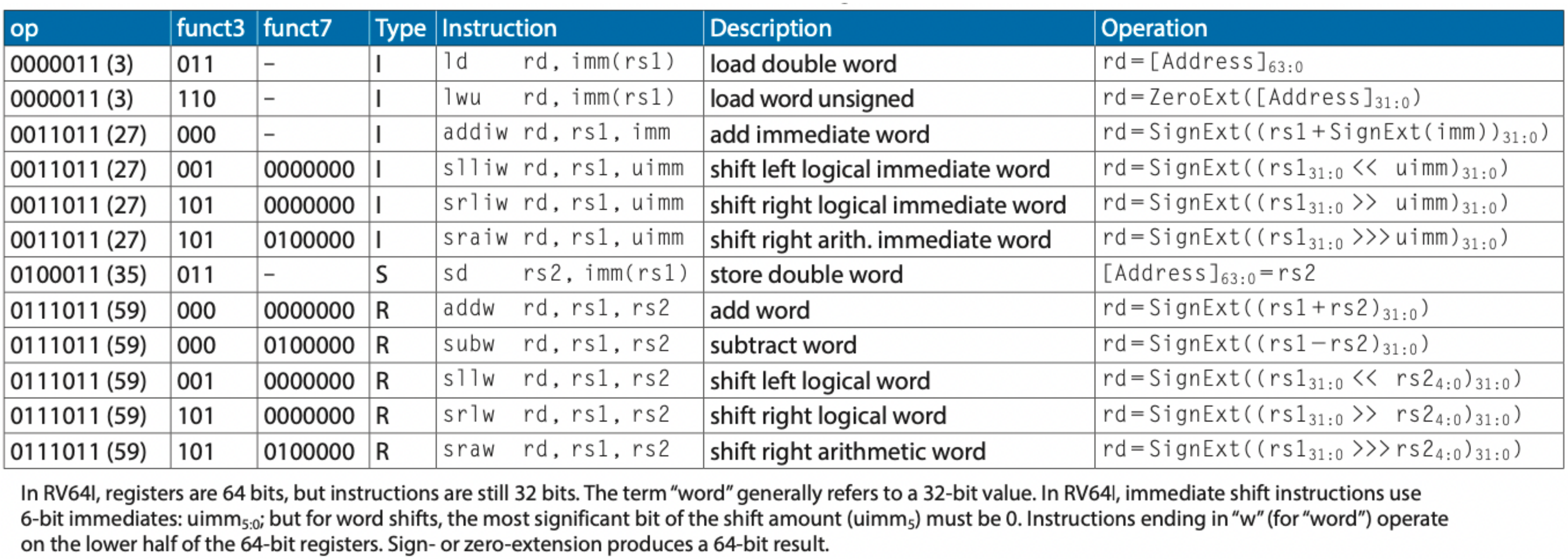
`sb x30, 42(x18)`

RISC-V Integer Instruction Summary



op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = SignExt([Address]$_{31:0}$)
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	[Address]$_{15:0}$= rs2$_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address]$_{31:0}$= rs2$_{31:0}$
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 – rs2
0110011 (51)	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2$_{4:0}$
0110011 (51)	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2$_{4:0}$
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2$_{4:0}$
0110011 (51)	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
1100111 (103)	000	–	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
1101111 (111)	–	–	J	jal rd, label	jump and link	PC = JTA, rd = PC + 4

RV64I Additional Instructions



op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	011	–	I	ld rd, imm(rs1)	load double word	rd = [Address]$_{63:0}$
0000011 (3)	110	–	I	lwu rd, imm(rs1)	load word unsigned	rd = ZeroExt([Address]$_{31:0}$)
0011011 (27)	000	–	I	addiw rd, rs1, imm	add immediate word	rd = SignExt((rs1 + SignExt(imm))$_{31:0}$)
0011011 (27)	001	0000000	I	slliw rd, rs1, uimm	shift left logical immediate word	rd = SignExt((rs1$_{31:0}$ << uimm)$_{31:0}$)
0011011 (27)	101	0000000	I	srliw rd, rs1, uimm	shift right logical immediate word	rd = SignExt((rs1$_{31:0}$ >> uimm)$_{31:0}$)
0011011 (27)	101	0100000	I	sraiw rd, rs1, uimm	shift right arith. immediate word	rd = SignExt((rs1$_{31:0}$ >>> uimm)$_{31:0}$)
0100011 (35)	011	–	S	sd rs2, imm(rs1)	store double word	[Address]$_{63:0}$ = rs2
0111011 (59)	000	0000000	R	addw rd, rs1, rs2	add word	rd = SignExt((rs1 + rs2)$_{31:0}$)
0111011 (59)	000	0100000	R	subw rd, rs1, rs2	subtract word	rd = SignExt((rs1 – rs2)$_{31:0}$)
0111011 (59)	001	0000000	R	sllw rd, rs1, rs2	shift left logical word	rd = SignExt((rs1$_{31:0}$ << rs2$_{4:0}$)$_{31:0}$)
0111011 (59)	101	0000000	R	srlw rd, rs1, rs2	shift right logical word	rd = SignExt((rs1$_{31:0}$ >> rs2$_{4:0}$)$_{31:0}$)
0111011 (59)	101	0100000	R	sraw rd, rs1, rs2	shift right arithmetic word	rd = SignExt((rs1$_{31:0}$ >>> rs2$_{4:0}$)$_{31:0}$)

In RV64I, registers are 64 bits, but instructions are still 32 bits. The term "word" generally refers to a 32-bit value. In RV64I, immediate shift instructions use 6-bit immediates: uimm$_{5:0}$; but for word shifts, the most significant bit of the shift amount (uimm$_5$) must be 0. Instructions ending in "w" (for "word") operate on the lower half of the 64-bit registers. Sign- or zero-extension produces a 64-bit result.

Chapter 1: RISC-V Introduction

RISC-V Hart

RISC-V Hart

- A **hart** is an abstraction of a ***har***dware ***t***hread, but harts may be physical (in hardware), virtual, or simulated.
- Each hart:
  - Has a PC (program counter) and registers
  - Executes independently of other harts
- **Example:** a hart may correspond to a physical core or an operating system (OS) may time-multiplex many user-level virtual harts onto a smaller number of physical harts.

Privilege Modes

- **3 privilege modes:**
  - Machine: M-Mode (highest)
  - Supervisor: S-Mode
  - User: U-Mode (lowest)
- **They offer protection between modes:**
  - Lower privilege modes have restricted access to some memory and some control registers
  - M-mode has access to all memory and registers

Control & Status Registers (CSRs)

- Each privilege mode has a set of **CSRs**
- CSRs include:
  - Address of trap handler
  - Flags from a floating-point operation
  - Performance counters
  - Etc.

Processes & Threads

- **Process:** a running program.
  - Operating systems often support multiple processes running at once.
- **Thread:** a process is divided up into one or more threads.
  - E.g., a word processor printing, spell-checking, and allowing text entry at the same time
  - Threads share common memory and code space
- **Hart:** contains architectural state (PC, registers, CSRs) to run independently/concurrently with other threads.

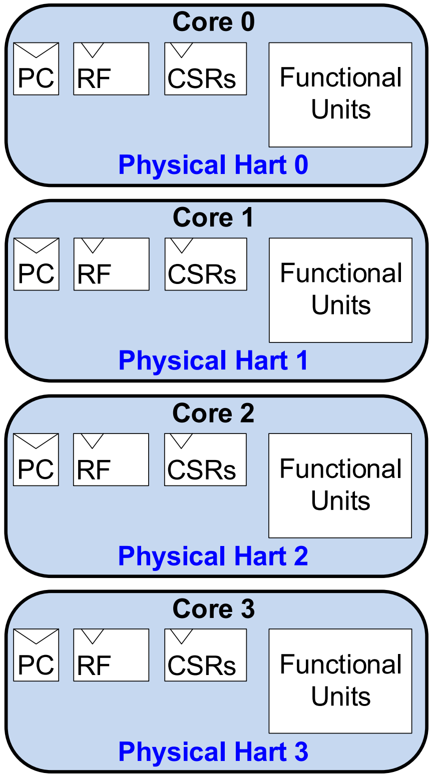
Cores & Harts

- **Core:** single processor on one chip. At a minimum, includes:
  - Architectural state: PC, Registers, CSRs
  - Execution units needed to run a thread
- **Multithreaded core:** single processor that:
  - Contains multiple copies of architectural state to support multiple threads
  - Shares execution units amongst threads
- **Multicore processor:** on the same chip, contains:
  - Multiple cores
  - Interconnection network
  - Upper-level cache (usually)
- **Microprocessor:** partitioned into:
  - Core: architectural state and execution units
  - Uncore: peripherals, shared memory or caches, external interfaces

Harts revisited

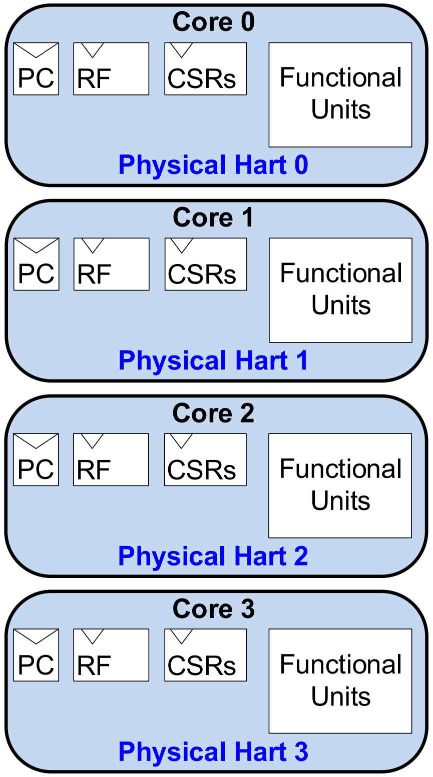
- **Multicore processor:**
  - Each core is a hart capable of running a separate thread
- **Single Multithreaded processor:**
  - Can switch between threads (architectural state), so acts as multiple harts
- **Example: Multicore processor**
  - 4 cores
  - Each core is 2-way multithreaded
  - Thus, has 8 physical harts
- **Other cases:**
  - An OS may time-multiplex multiple virtual harts onto a single physical hart
  - An emulator, such as QEMU, may run RISC-V programs on emulated harts

Harts: Examples

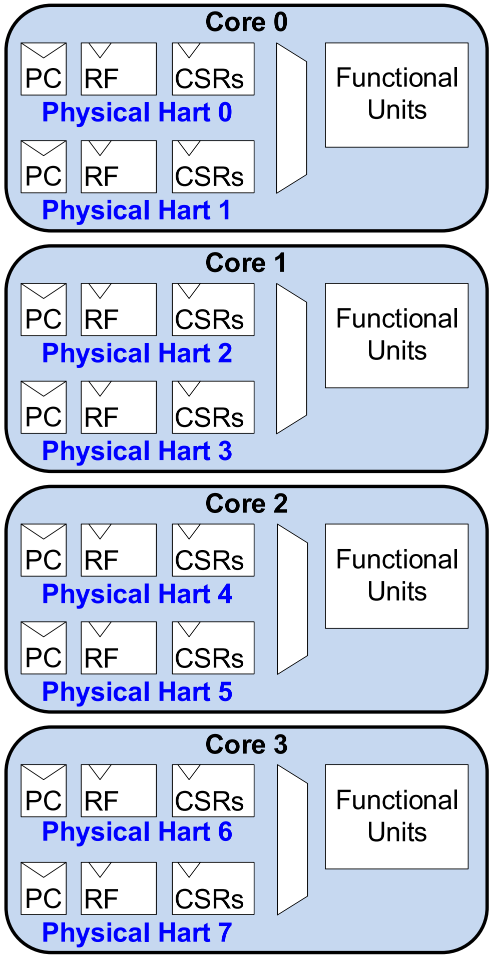


4 Cores
4 Physical Harts

Harts: Examples

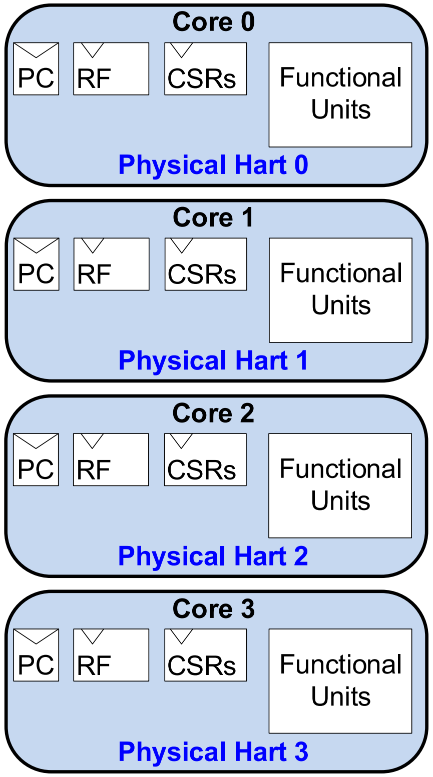


4 Cores
4 Physical Harts

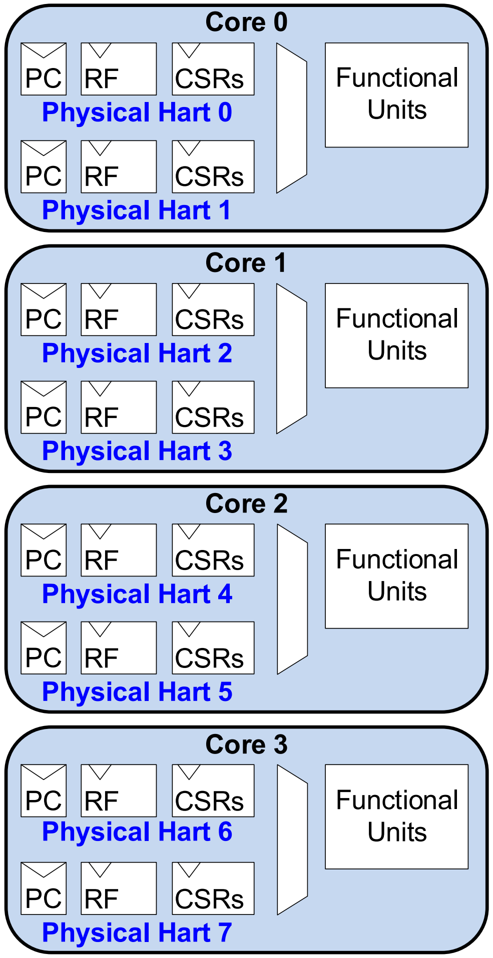


4 Multithreaded Cores
8 Physical Harts

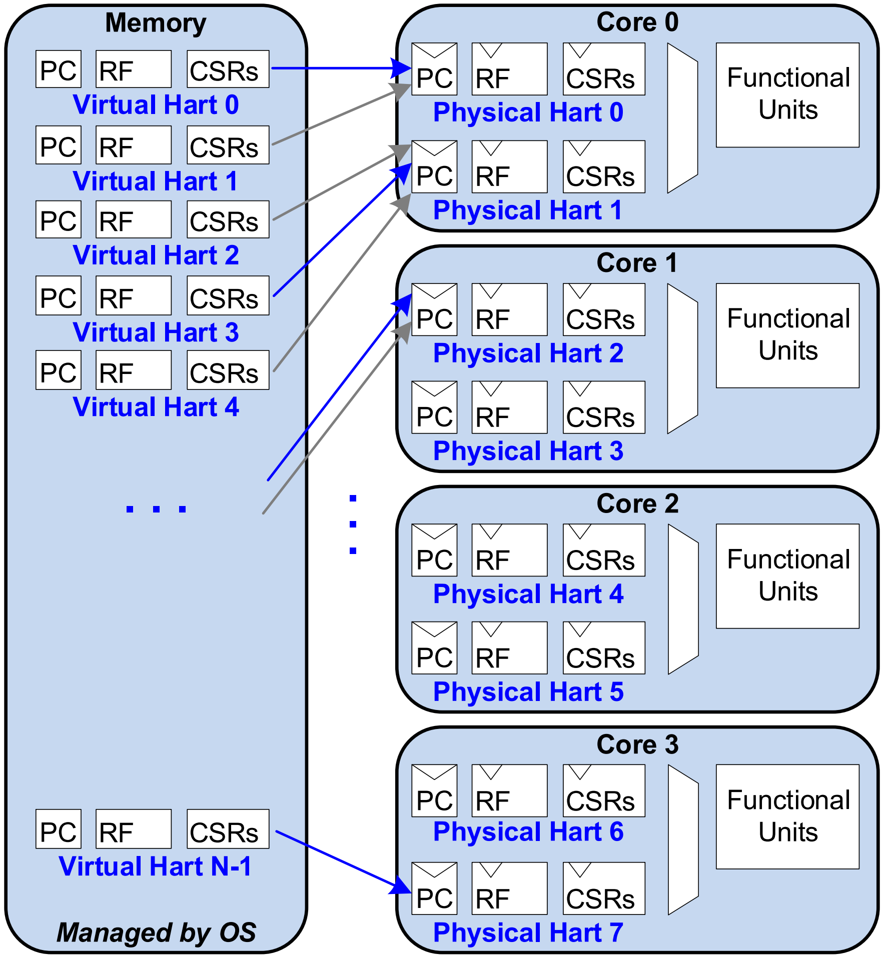
Harts: Examples



4 Cores
4 Physical Harts



4 Multithreaded Cores
8 Physical Harts



4 Multithreaded Cores
N Virtual Harts mapped to 8 Physical Harts

Traps

- **Trap:** an exception or interrupt
  - **Exception:** illegal instruction, ecall, access to non-existent memory, etc.
  - **Interrupt:** asynchronous external event, like a button being pushed or serial port data arriving
- Traps cause a **trap handler** to execute. Related CSRs:
  - **`mepc`:** Exception PC = PC of instruction when trap occurred
  - **`mcause`:** Cause of trap
  - **`mtval`:** Additional information, if needed
  - **`mtvec`:** Address of trap handler
  - S-mode versions also exist (`sepc`, `scause`, `stvec`, etc.)
- Traps **return to program** using:
  - ***`x`*`ret`** (**`mret`**, **`sret`**)**:** return to address in *`x`*`epc`

Execution Environments

- **Purpose:** enable lower-level privilege modes to interact with higher-level privilege modes.
  - **ABI (application binary interface):** an API that enables application (U-mode) harts to interact with the OS (S-mode) harts that reside in the application execution environment.
  - **SBI (supervisor binary interface)**: an API that enables OS (S-mode) harts to interact with M-mode harts that reside in the supervisor execution environment.

Execution Environment

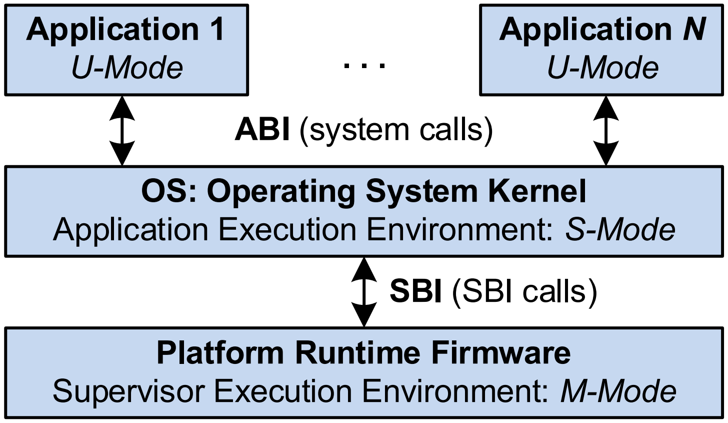
- **EEI** (execution environment interface, or simply execution environment) defines:
  - Initial state of program
  - Number & types of harts (including privilege mode, ISA, memory map & how traps are handled)
- **Example EEIs:**
  - **ABI** (application binary interface): defines calling conventions and data types
  - **SBI** (supervisor binary interface): defines how the OS calls the underlying platform runtime firmware (e.g., setting the system timer, starting/stopping/synchronizing cores, etc.)

Simple Execution Environment

- **Bare metal:** simplest execution environment:
  - Processor runs in machine mode with no OS
  - Physical harts correspond to physical cores (each core has a single copy of architectural state)
  - Programmer has access to all features and memory
  - No virtual harts exist
  - No protection between harts

Linux

- **Linux OS – More complex execution environment:**
  - OS runs in S-mode:
    - OS uses an SBI to access platform runtime firmware, which consists of the **supervisor execution environment**
  - Applications run in U-mode. OS provides each application with an **application execution environment**, which defines:
    - That application's virtual memory space
    - ABI provides a common set of system calls for applications to make requests to the kernel to manage processes, memory, and files.


Application 1
U-Mode
. . .
Application *N*
U-Mode
ABI (system calls)
OS: Operating System Kernel
Application Execution Environment: *S-Mode*
SBI (SBI calls)
Platform Runtime Firmware
Supervisor Execution Environment: *M-Mode*

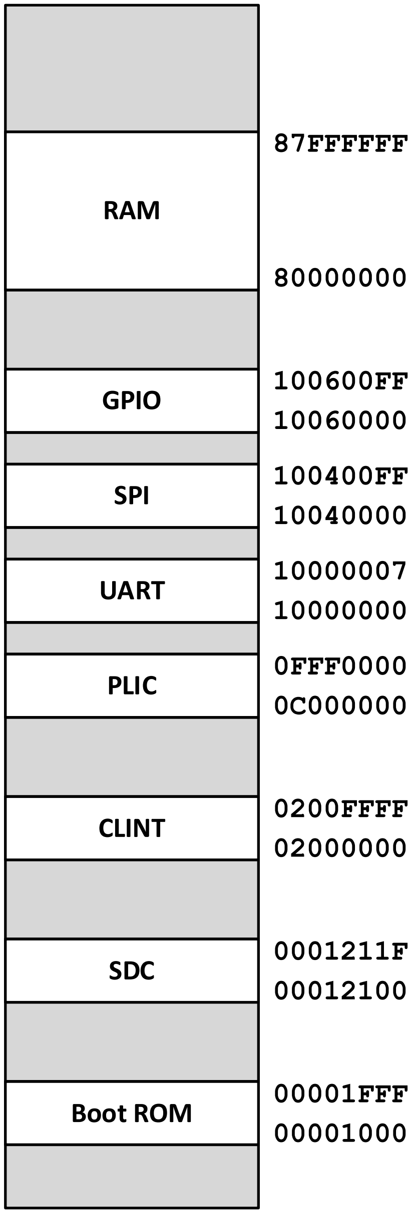
Chapter 1: RISC-V Introduction

Memory Map & Extensions

Wally Memory Map

- **Memory Map:** Locations (addresses) of memory & I/O (peripherals)
- **Wally memory map** contains:
  - A small boot ROM
  - A larger RAM
  - Peripherals including:
    - **GPIO:** General Purpose Input/Output for reading and writing binary values from pins
    - **UART:** Universal Asynchronous Receiver and Transmitter for serial communication, such as printing to a terminal
    - **CLINT:** Core Local Interrupter for handling software and timer interrupts
    - **PLIC:** Platform-Level Interrupt Controller for multiplexing external interrupts
    - **SDC:** Secure Digital (SD) Card controller

Wally Memory Map



87FFFFFF

RAM

80000000

100600FF

GPIO

10060000

100400FF

SPI

10040000

10000007

UART

10000000

0FFF0000

PLIC

0C000000

0200FFFF

CLINT

02000000

0001211F

SDC

00012100

00001FFF

Boot ROM

00001000

- Same as SiFive FU540 processor and QEMU *virt* machine (**virt:** QEMU virtual model with peripherals suitable for booting Linux)
  - **Except UART:**
    - 0x10010000 on SiFive FU540
    - 0x10000000 on Wally and virt
- On reset, processor starts at **reset vector**, which usually points to a boot ROM, which contains the boot loader (loads the next stage of the OS from SD card, flash, or hard drive).
  - The reset vector is platform-dependent:
    - **0x80000000:** Wally reset vector (it jumps to boot ROM)
    - **0x1000:** in QEMU & buildroot Linux configuration
    - **0x1004:** on SiFive FE540 chip

Common RISC-V Extensions

Extension	Instructions
M	Multiply, divide, and remainder
F, D, Q	Single-, double-, and quad-precision floating-point operations
A	Atomic read-modify-write for synchronization
C	Compressed (16-bit)
Zicsr	CSRs and CSR instructions
Zfencei	FENCE.I
Zfh	Half-precision floating-point
Zk*	Cryptography
V	Vector
B	Bit manipulations

RISC-V Microarchitecture

Simplified Example

- Simplified RV32I core that supports following subset of instructions:
  - **ALU Instructions:** `add`, `sub`, `and`, `or`, `slt`, `addi`, `andi`, `ori`, `slti`
  - **Memory Instructions:** `lw`, `sw`
  - **Branch Instructions:** `beq`, `jal`

Performance

- Performance measured as time to execute a program of interest, T_{CPU} (CPU time):

$$\mathbf{T_{CPU}} = \text{\# instructions} \times \text{CPI} \times T_c$$

CPI = clock cycles per instruction

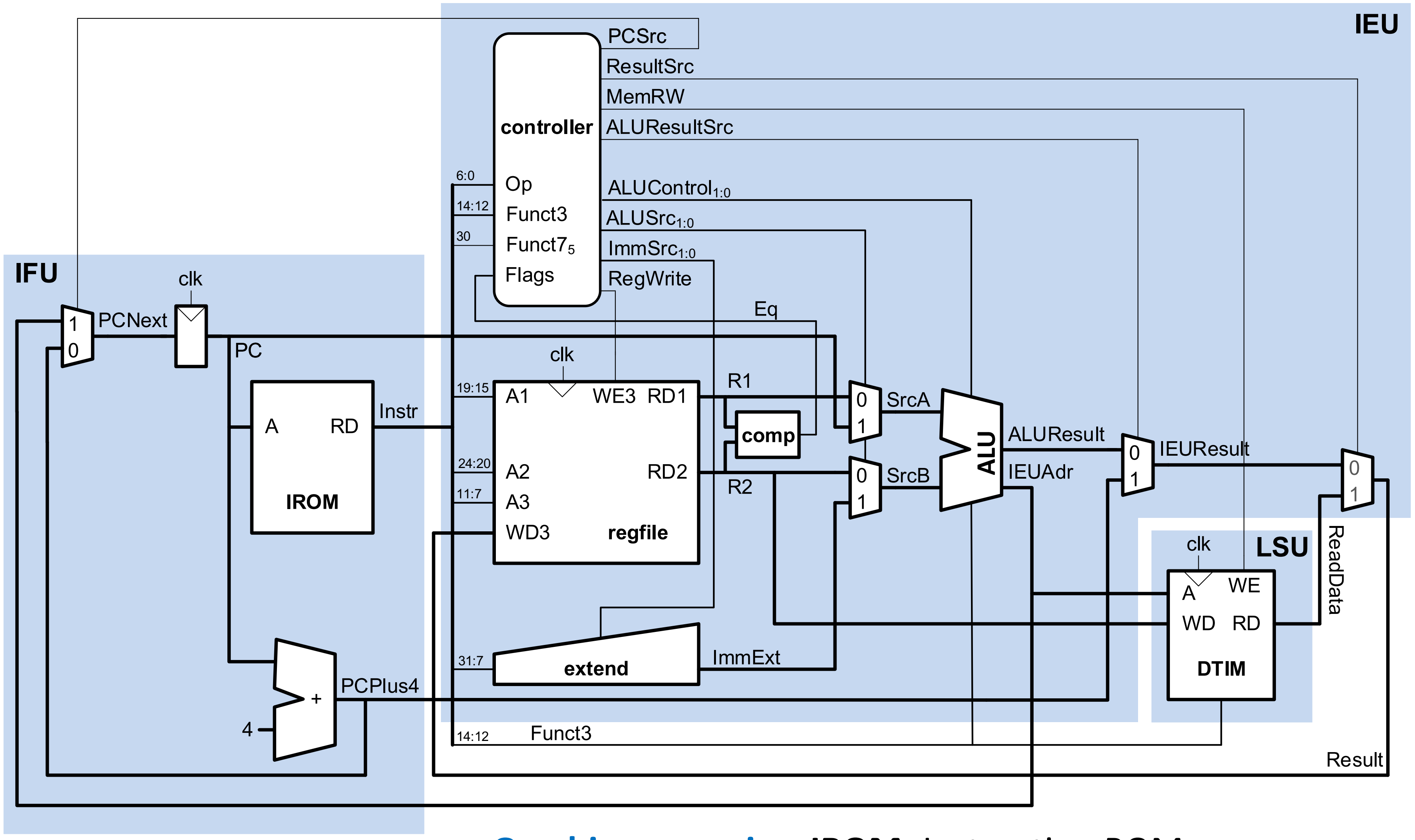
T_c = clock cycle time

Microarchitecture Overview

- Any microarchitecture must include:
  - **Architectural state:** PC, registers, memory
- Simplified Microarchitecture Examples
  - Single-cycle
  - Pipelined
- Both have architectural state and run instructions
- Differ in performance and cost
- Cores are partitioned into:
  - **IFU:** instruction fetch unit
  - **IEU:** instruction execution unit
  - **LSU:** load/store unit

RISC-V Single-Cycle Processor

Single-Cycle Processor

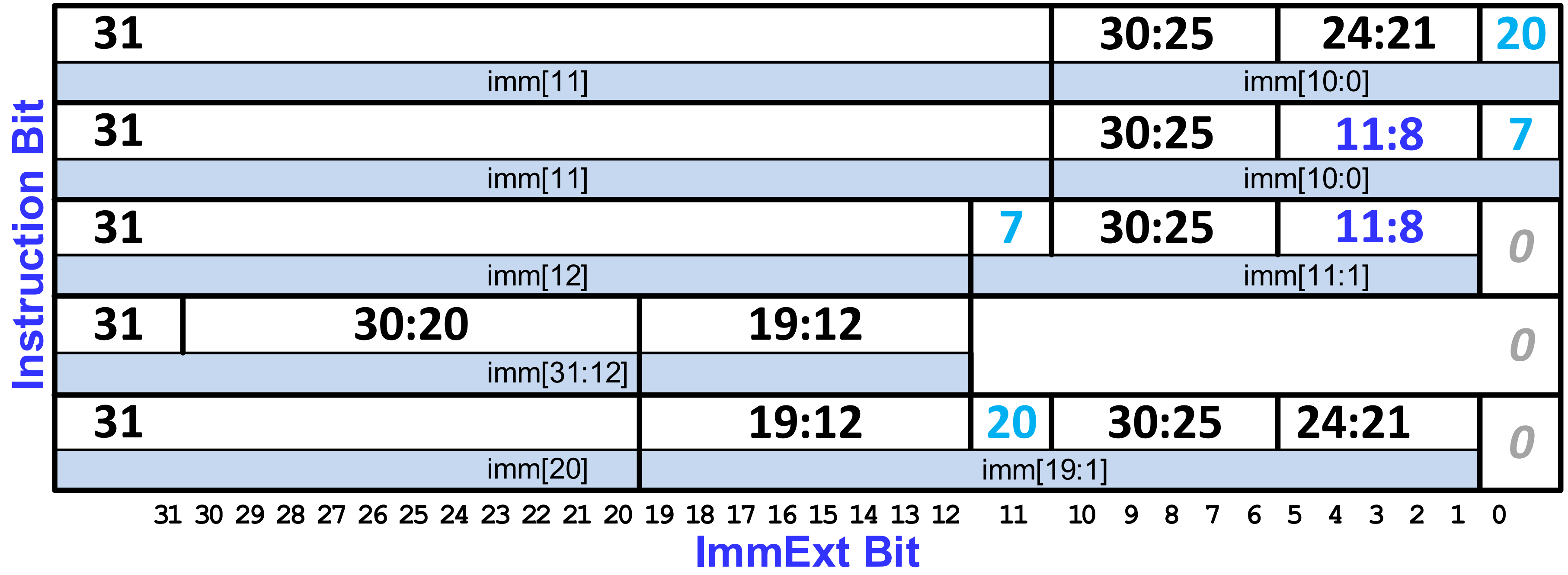


On-chip memories: IROM: Instruction ROM

DTIM: Data tightly integrated memory

Extend Operation

ImmSrc	ImmExt	Type
00	{{20{Instr[31]}}, Instr[31:20]}	I (12-bit)
01	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	S (12-bit)
10	{{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}	B (13-bit)
11	{{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0}	J (21-bit)



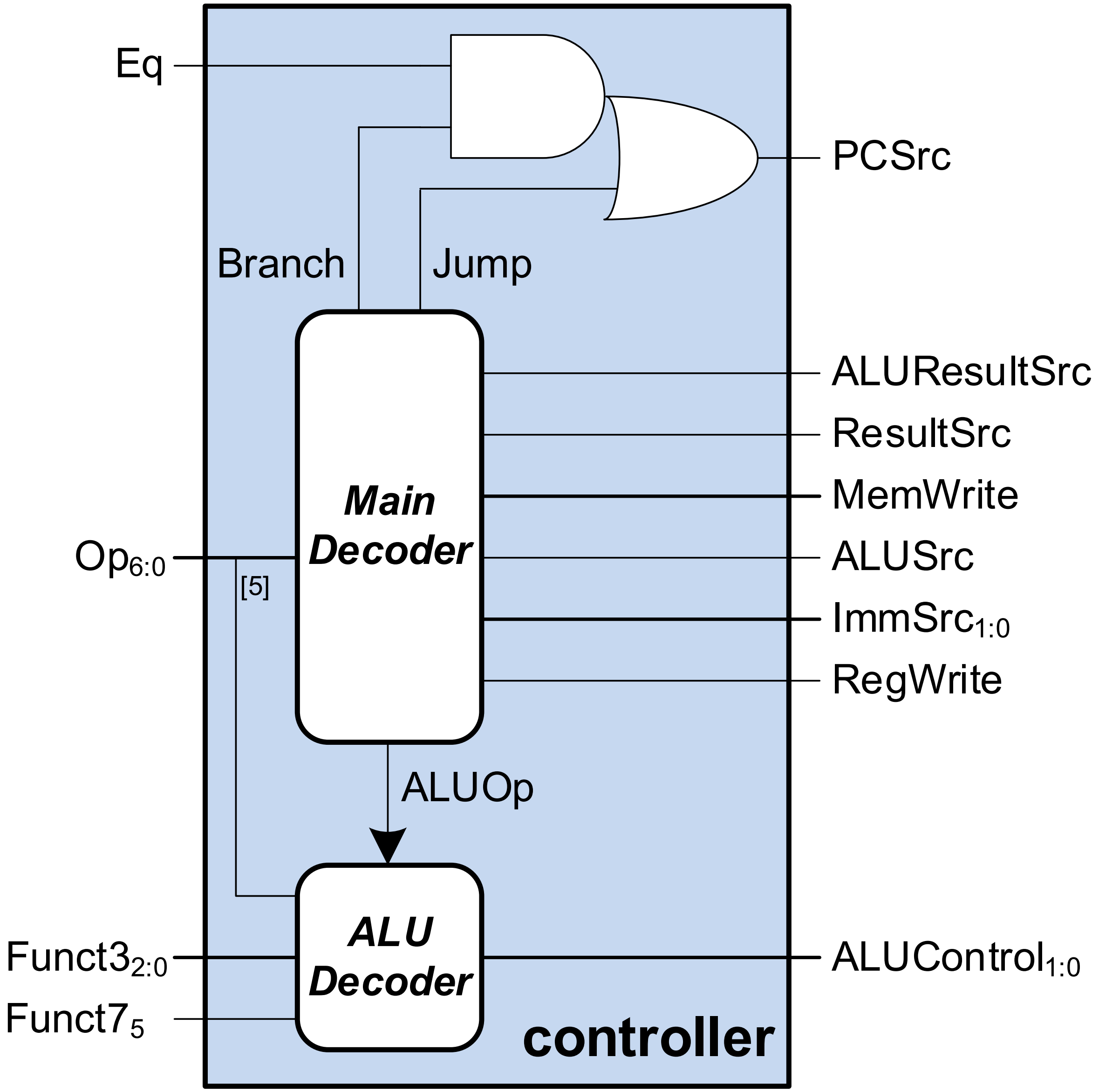
ALU Operations

ALUControl[1] (Sub)	ALUControl[0] (ALUOp)	Funct3	ALUResult
0	0	x	add
0	1	000	add
0	1	110	or
0	1	111	and
1	1	000	sub
1	1	010	slt

Main Decoder Truth Table

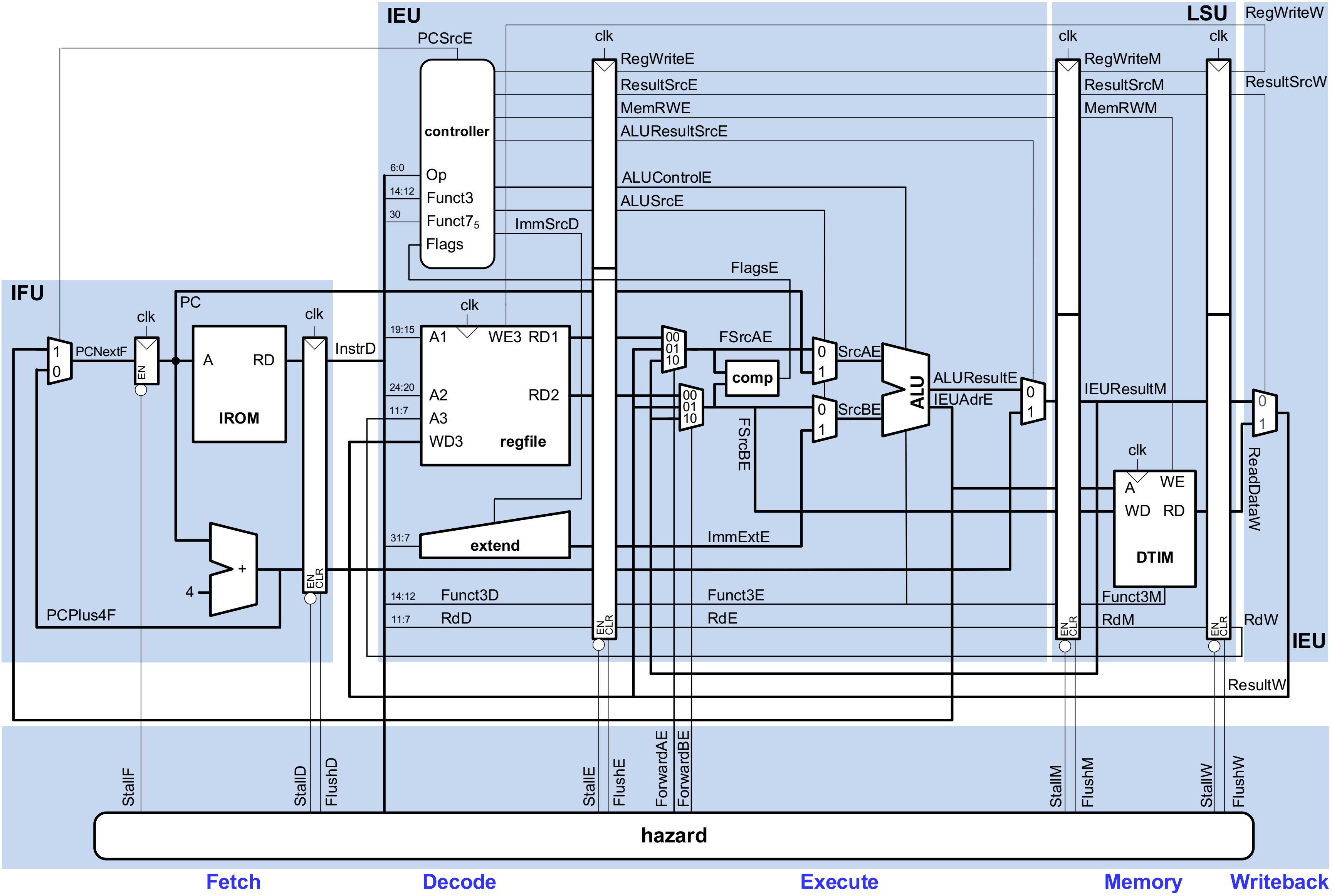
Instr	Op	RegWrite	ImmSrc	ALUSrc	ALUOp	ALU ResultSrc	MemWrite	ResultSrc	Branch	Jump
lw	0000011	1	00	01	0	0	0	1	0	0
sw	0100011	0	01	01	0	0	1	0	0	0
R-type ALU	0110011	1	xx	00	1	0	0	0	0	0
I-type ALU	0010011	1	00	01	1	0	0	0	0	0
beq	1100011	0	10	11	0	0	0	0	1	0
jal	1101111	1	11	11	0	1	0	0	0	1

Single-Cycle Control Unit



RISC-V Pipelined Processor

Pipelined Processor



Chapter 1: RISC-V Introduction

Wally RISC-V SoC

Wally Supported Features

- **XLEN** (32 or 64)
- **Primary Extensions**
  - **A:** Atomic
  - **C:** Compressed
  - **D:** Double-precision floating-point
  - **E:** Embedded 16 registers
  - **F:** Single-precision floating-point
  - **M:** Multiplication and division
  - **Q:** Quad-precision floating-point
  - **Zb*:** Bit manipulation
  - **Zicsr:** CSRs
  - **Zfencei:** FENCE.I instruction synchronization
  - **Zk*:** Cryptography
- **Additional Extensions**
  - **Zicsr:** CSRs
  - **Zicntr:** Performance counters
  - **Zifencei:** `fence.i` synchronization
  - **Zfh:** Half-precision floating-point
  - **Zfa:** Additional floating-point
  - **Zmmul:** Integer multiplication without division
  - **Zbc:** Carry-free multiplication
  - **Zca, Zcb, Zcf, Zcd:** Additional compressed instructions
  - **Zicbom, Zicboz, Zicbop:** Cache management operations
  - **Zicclsm:** Misaligned loads & stores
  - **Zihintpause:** Pause hint
  - **Zicond:** Conditional zero

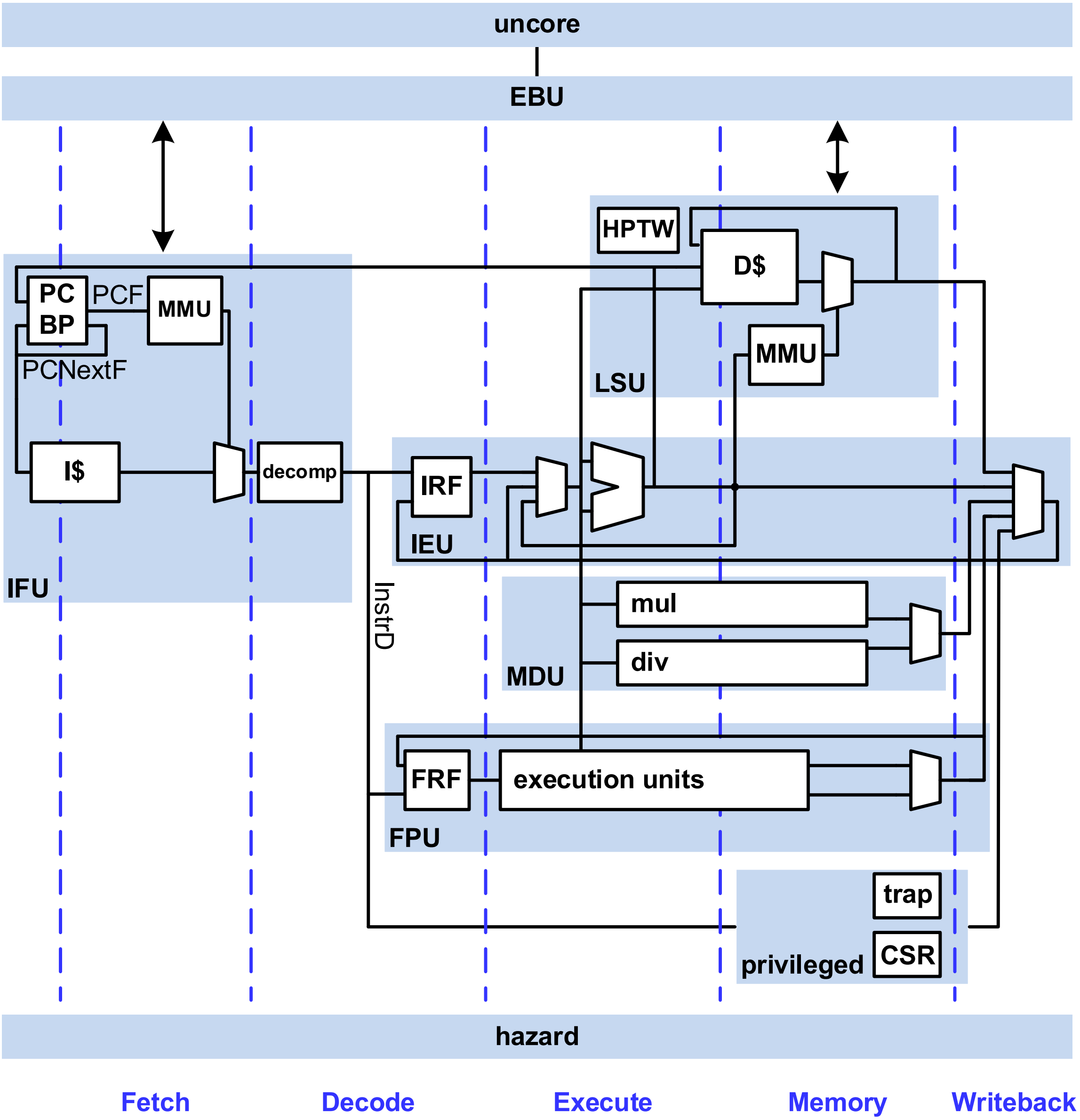
Wally Supported Features

- **Privileged Modes and Features**
  - M, S, U
  - Reset vector address
  - Virtual memory (ITLB, DTLB)
  - Physical memory protection
  - Vectored interrupts
- **Caches and Branch Prediction**
- **Peripherals**
  - Core Local Interrupt Controller with timers (CLINT)
  - Platform Level Interrupt Controller (PLIC)
  - UART
  - GPIO
  - TIM

Not Supported in Wally

- Vector extension
- Hypervisor extension
- Advanced microarchitecture: superscalar, out-of-order, multithreading, multicore
- Unfrozen extensions
  - L: Decimal floating-point
  - J: Dynamically translated languages
  - T: Transactional memory
  - P: Packed SIMD

Wally Block Diagram



Wally Configurations

Configuration	Config	XLEN	DTIM / IROM Size	Bus	Periph	I$/D$ Size	Priv. Modes	Virt. Mem
Embedded	rv32e	32 bits	n/a	YES	NO	n/a	none	NO
Simple CPU	rv32i	32 bits	2K / 2K	NO	NO	n/a	none	NO
µcontroller	rv32ic	32 bits	4K / 16K	YES	YES	n/a	MU	NO
Apps Proc	rv32gc	32 bits	n/a	YES	YES	16K	MSU	YES
Simple CPU	rv64i	64 bits	2K / 2K	NO	NO	n/a	none	NO
Apps Proc	rv64gc	64 bits	n/a	YES	YES	16K	MSU	YES

RISC-V Market

Rapidly Changing Market

- **2010:** RISC-V developed at Berkeley
- **2011:** First RISC-V chip fabricated in 28 nm
- **2015:** RISC-V Foundation launched
- **2015:** SiFive founded by RISC-V inventors
- **2021:** 12 billion RISC-V cores shipped in commercial products
- **2022:** Seimco Research forecasts 40% annual growth through 2030

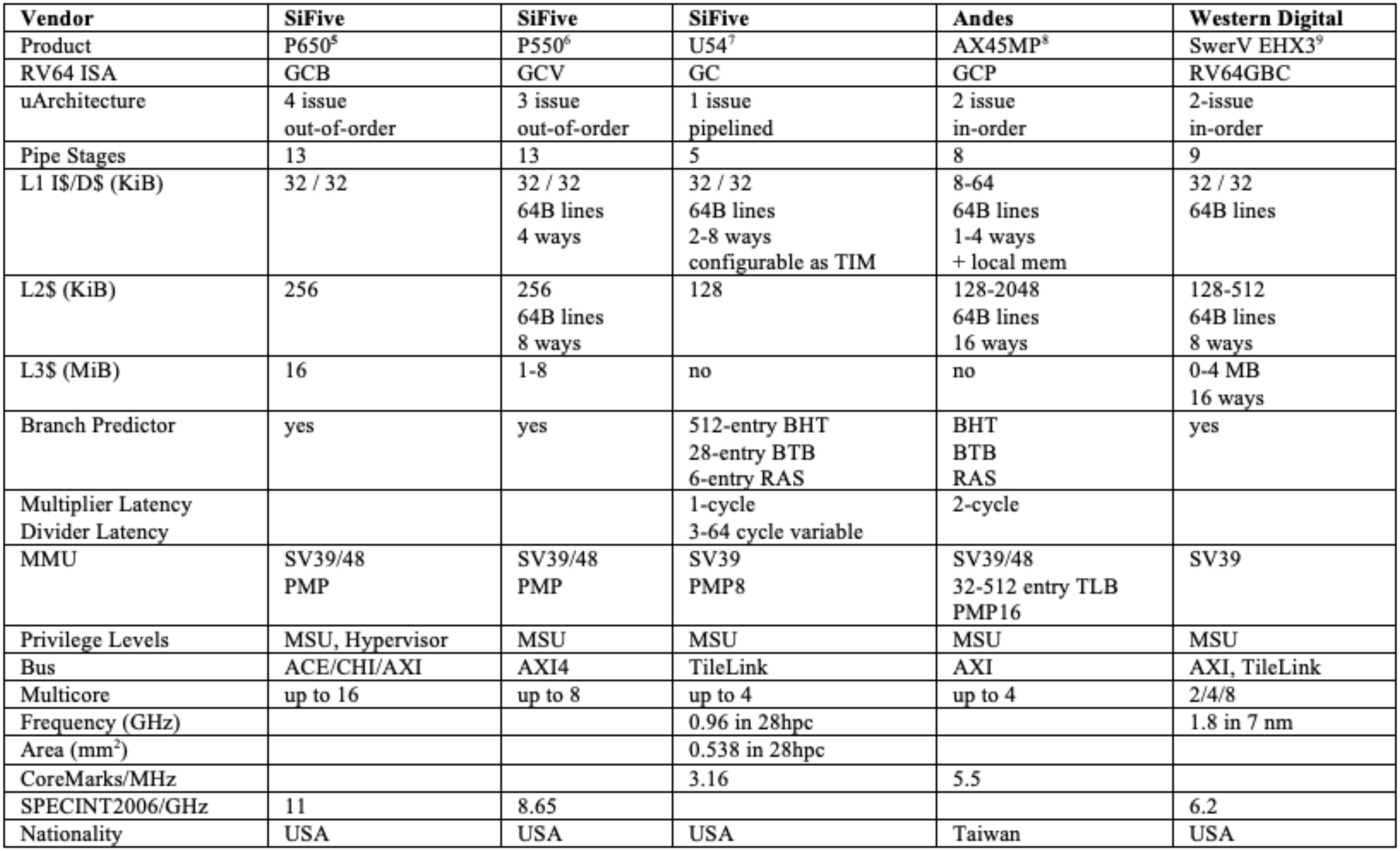
RISC-V International

- **Stewards the open RISC-V standard**
  - > 60 working groups
  - > 3000 individuals contributing
- **Processes for approving new extensions**
- **Infrastructure development**
  - Architectural Compatibility Tests
  - Compilers and simulators
  - Operating system support
  - Debug standards

RISC-V Processor Suppliers

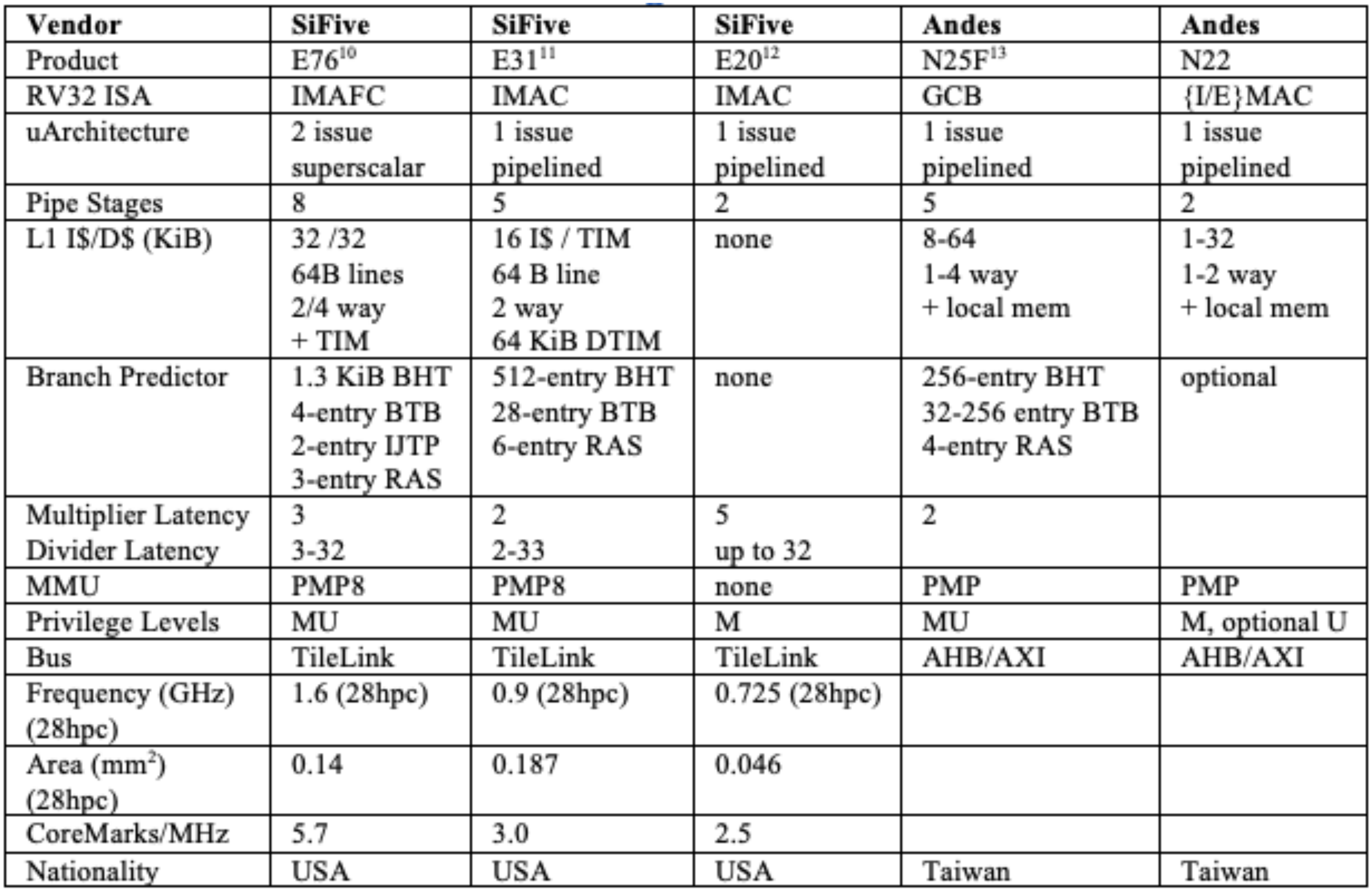
- **Startups and established companies selling IP**
  - Alibaba, Andes, Esperanto, Rivos, Semidynamics, SiFive,
  - Tenstorrent, Ventana, Western Digital
- **In-house use**
  - Qualcomm, Intel
- **Open Source IP**
  - OpenHW Group
- **Global competition and cooperation**
  - Especially United States, China, Taiwan, Europe, India
  - Open architecture reduces barriers to entry

Commercial RV64 Multicore Apps Processors



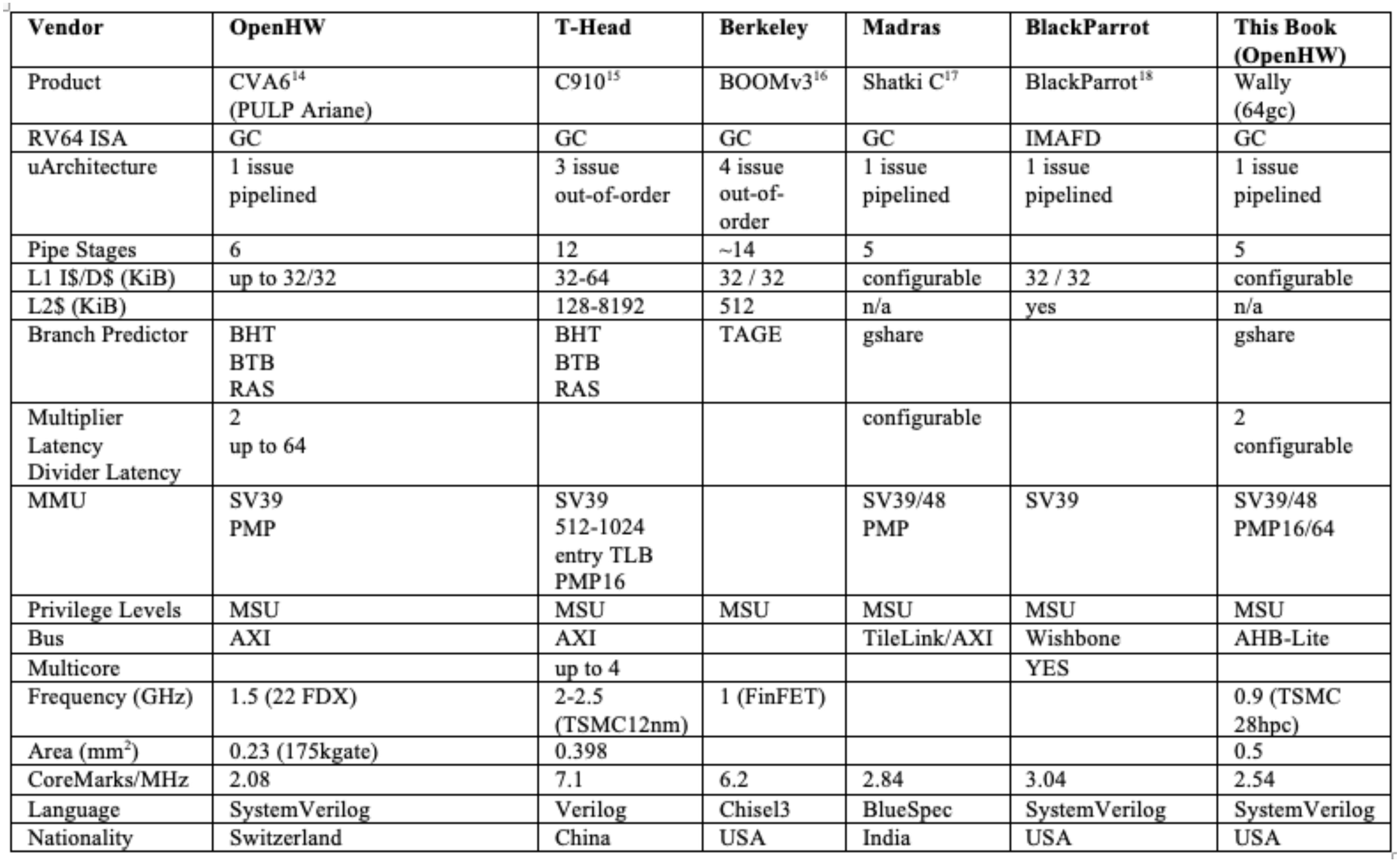
Vendor	SiFive	SiFive	SiFive	Andes	Western Digital
Product	P650[5]	P550[6]	U54[7]	AX45MP[8]	SwerV EHX3[9]
RV64 ISA	GCB	GCV	GC	GCP	RV64GBC
uArchitecture	4 issue out-of-order	3 issue out-of-order	1 issue pipelined	2 issue in-order	2-issue in-order
Pipe Stages	13	13	5	8	9
L1 I$/D$ (KiB)	32 / 32	32 / 32 64B lines 4 ways	32 / 32 64B lines 2-8 ways configurable as TIM	8-64 64B lines 1-4 ways + local mem	32 / 32 64B lines
L2$ (KiB)	256	256 64B lines 8 ways	128	128-2048 64B lines 16 ways	128-512 64B lines 8 ways
L3$ (MiB)	16	1-8	no	no	0-4 MB 16 ways
Branch Predictor	yes	yes	512-entry BHT 28-entry BTB 6-entry RAS	BHT BTB RAS	yes
Multiplier Latency Divider Latency			1-cycle 3-64 cycle variable	2-cycle	
MMU	SV39/48 PMP	SV39/48 PMP	SV39 PMP8	SV39/48 32-512 entry TLB PMP16	SV39
Privilege Levels	MSU, Hypervisor	MSU	MSU	MSU	MSU
Bus	ACE/CHI/AXI	AXI4	TileLink	AXI	AXI, TileLink
Multicore	up to 16	up to 8	up to 4	up to 4	2/4/8
Frequency (GHz)			0.96 in 28hpc		1.8 in 7 nm
Area (mm^2)			0.538 in 28hpc		
CoreMarks/MHz			3.16	5.5	
SPECINT2006/GHz	11	8.65			6.2
Nationality	USA	USA	USA	Taiwan	USA

Commercial RV32 Embedded Processors



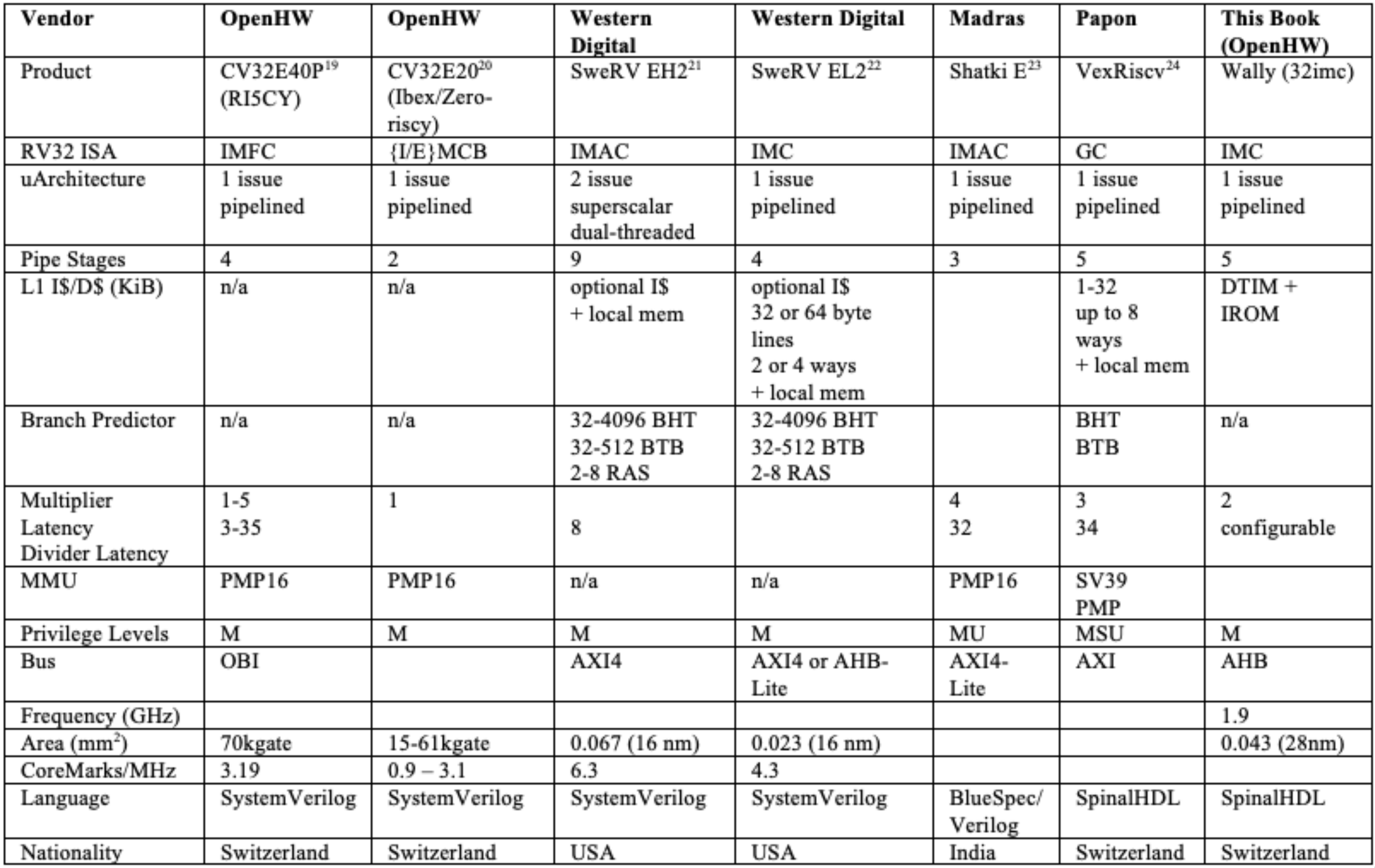
Vendor	SiFive	SiFive	SiFive	Andes	Andes
Product	E76[10]	E31[11]	E20[12]	N25F[13]	N22
RV32 ISA	IMAFC	IMAC	IMAC	GCB	{I/E}MAC
uArchitecture	2 issue superscalar	1 issue pipelined	1 issue pipelined	1 issue pipelined	1 issue pipelined
Pipe Stages	8	5	2	5	2
L1 I$/D$ (KiB)	32 /32 64B lines 2/4 way + TIM	16 I$ / TIM 64 B line 2 way 64 KiB DTIM	none	8-64 1-4 way + local mem	1-32 1-2 way + local mem
Branch Predictor	1.3 KiB BHT 4-entry BTB 2-entry IJTP 3-entry RAS	512-entry BHT 28-entry BTB 6-entry RAS	none	256-entry BHT 32-256 entry BTB 4-entry RAS	optional
Multiplier Latency Divider Latency	3 3-32	2 2-33	5 up to 32	2	
MMU	PMP8	PMP8	none	PMP	PMP
Privilege Levels	MU	MU	M	MU	M, optional U
Bus	TileLink	TileLink	TileLink	AHB/AXI	AHB/AXI
Frequency (GHz) (28hpc)	1.6 (28hpc)	0.9 (28hpc)	0.725 (28hpc)		
Area (mm^2) (28hpc)	0.14	0.187	0.046		
CoreMarks/MHz	5.7	3.0	2.5		
Nationality	USA	USA	USA	Taiwan	Taiwan

Open-Source RV64 Apps Processors



Vendor	OpenHW	T-Head	Berkeley	Madras	BlackParrot	This Book (OpenHW)
Product	CVA6[14] (PULP Ariane)	C910[15]	BOOMv3[16]	Shatki C[17]	BlackParrot[18]	Wally (64gc)
RV64 ISA	GC	GC	GC	GC	IMAFD	GC
uArchitecture	1 issue pipelined	3 issue out-of-order	4 issue out-of-order	1 issue pipelined	1 issue pipelined	1 issue pipelined
Pipe Stages	6	12	~14	5		5
L1 I$/D$ (KiB)	up to 32/32	32-64	32 / 32	configurable	32 / 32	configurable
L2$ (KiB)		128-8192	512	n/a	yes	n/a
Branch Predictor	BHT BTB RAS	BHT BTB RAS	TAGE	gshare		gshare
Multiplier Latency Divider Latency	2 up to 64			configurable		2 configurable
MMU	SV39 PMP	SV39 512-1024 entry TLB PMP16		SV39/48 PMP	SV39	SV39/48 PMP16/64
Privilege Levels	MSU	MSU	MSU	MSU	MSU	MSU
Bus	AXI	AXI		TileLink/AXI	Wishbone	AHB-Lite
Multicore		up to 4			YES	
Frequency (GHz)	1.5 (22 FDX)	2-2.5 (TSMC12nm)	1 (FinFET)			0.9 (TSMC 28hpc)
Area (mm^2)	0.23 (175kgate)	0.398				0.5
CoreMarks/MHz	2.08	7.1	6.2	2.84	3.04	2.54
Language	SystemVerilog	Verilog	Chisel3	BlueSpec	SystemVerilog	SystemVerilog
Nationality	Switzerland	China	USA	India	USA	USA

Open-Source RV32 Embedded Processors



Vendor	OpenHW	OpenHW	Western Digital	Western Digital	Madras	Papon	This Book (OpenHW)
Product	CV32E40P[19] (RI5CY)	CV32E20[20] (Ibex/Zero-riscy)	SweRV EH2[21]	SweRV EL2[22]	Shatki E[23]	VexRiscv[24]	Wally (32imc)
RV32 ISA	IMFC	{I/E}MCB	IMAC	IMC	IMAC	GC	IMC
uArchitecture	1 issue pipelined	1 issue pipelined	2 issue superscalar dual-threaded	1 issue pipelined	1 issue pipelined	1 issue pipelined	1 issue pipelined
Pipe Stages	4	2	9	4	3	5	5
L1 I$/D$ (KiB)	n/a	n/a	optional I$ + local mem	optional I$ 32 or 64 byte lines 2 or 4 ways + local mem		1-32 up to 8 ways + local mem	DTIM + IROM
Branch Predictor	n/a	n/a	32-4096 BHT 32-512 BTB 2-8 RAS	32-4096 BHT 32-512 BTB 2-8 RAS		BHT BTB	n/a
Multiplier Latency Divider Latency	1-5 3-35	1	 8		4 32	3 34	2 configurable
MMU	PMP16	PMP16	n/a	n/a	PMP16	SV39 PMP	
Privilege Levels	M	M	M	M	MU	MSU	M
Bus	OBI		AXI4	AXI4 or AHB-Lite	AXI4-Lite	AXI	AHB
Frequency (GHz)							1.9
Area (mm^2)	70kgate	15-61kgate	0.067 (16 nm)	0.023 (16 nm)			0.043 (28nm)
CoreMarks/MHz	3.19	0.9 – 3.1	6.3	4.3			
Language	SystemVerilog	SystemVerilog	SystemVerilog	SystemVerilog	BlueSpec/ Verilog	SpinalHDL	SpinalHDL
Nationality	Switzerland	Switzerland	USA	USA	India	Switzerland	Switzerland

About these Notes

RISC-V System-on-Chip Design Lecture Notes