ARM vs RISC-V

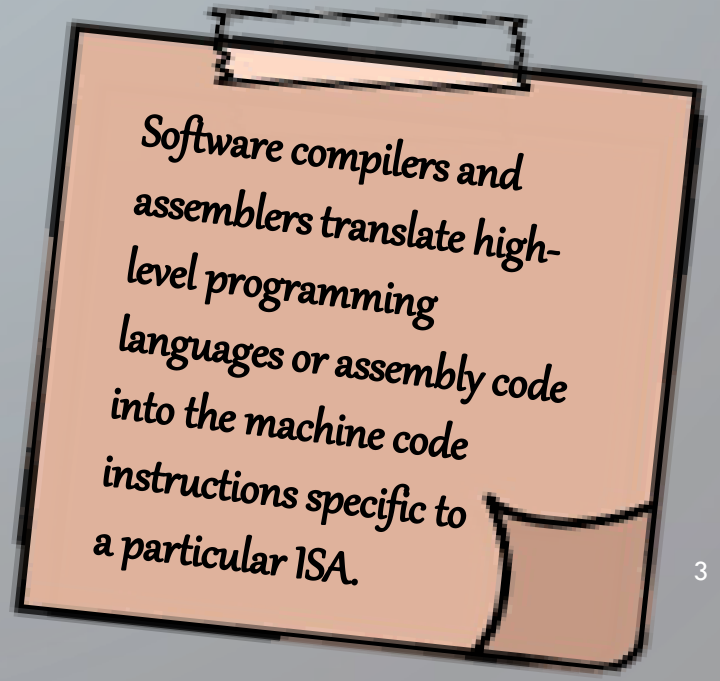
MELIKA RAJABI

OUTLINE

- What is ISA
- Why is ISA important
- The Story Begins
- RISC Design Principles
- Why RISC
- Exploring ARM ISA
  - ARM vs Thumb
  - Registers
  - Operations
- ARM ABI
- ARM Evolution Over Time
- What is Cortex
- ARMv8
- ARM Licensing Model
- Which Companies Use ARM
- ARM Product Code Description
- ARM Advantages
- Exploring RISC-V ISA
  - Registers
  - Operations
  - Memory Model
- RISC-V Evolution Over Time
- What is Open-Source
- Open-Source Advantages
- What is Modular Design
- Some RISC-V extensions
  - RV32I, RV64I, RV128I
  - M, A, F,D
- Which Companies Adopt RISC-V
- ARM or RISC-V
- References

What is an ISA?

- An Instruction Set Architecture (ISA) refers to the set of instructions that a computer processor understands and can execute.
- The interface between low-level hardware implementation and higher-level software programming of a computer system.
- It defines …
  - the operations that a processor can perform
  - the data types it can handle
  - the memory model it uses
  - The registers available for storing data



Why is ISA important?

Because it …

- Provides a standardized way for software developers to write programs that can run on different computer systems. (as long as they adhere to the same ISA.) ⟶ Software compatibility and portability
- Enables software developers to write a software program or application that will efficiently work with a particular computer system
- Reduces the complexity of programming by providing a standard interface between hardware and software

The story begins ...

ISAs can vary widely, depending on the architecture and design principles of the processor. Some popular ISAs include x86, ARM, MIPS, PowerPC, and RISC-V.

ISAs can be classified into different categories based on their design philosophy:

- Complex Instruction Set Computer (CISC): Large & diverse set of instructions that can perform complex operations
- Reduced Instruction Set Computer (RISC): Simpler & more streamlined set of instructions that typically execute in a single clock cycle

RISC Design Principles

Simple Instructions

- So that each can execute in one cycle

Register-to-Register Operations

- Just load/store operations can access memory
- To have simpler control unit

Simple Addressing Modes

- All operations except load/store use register based addressing
- load/store use memory addressing modes

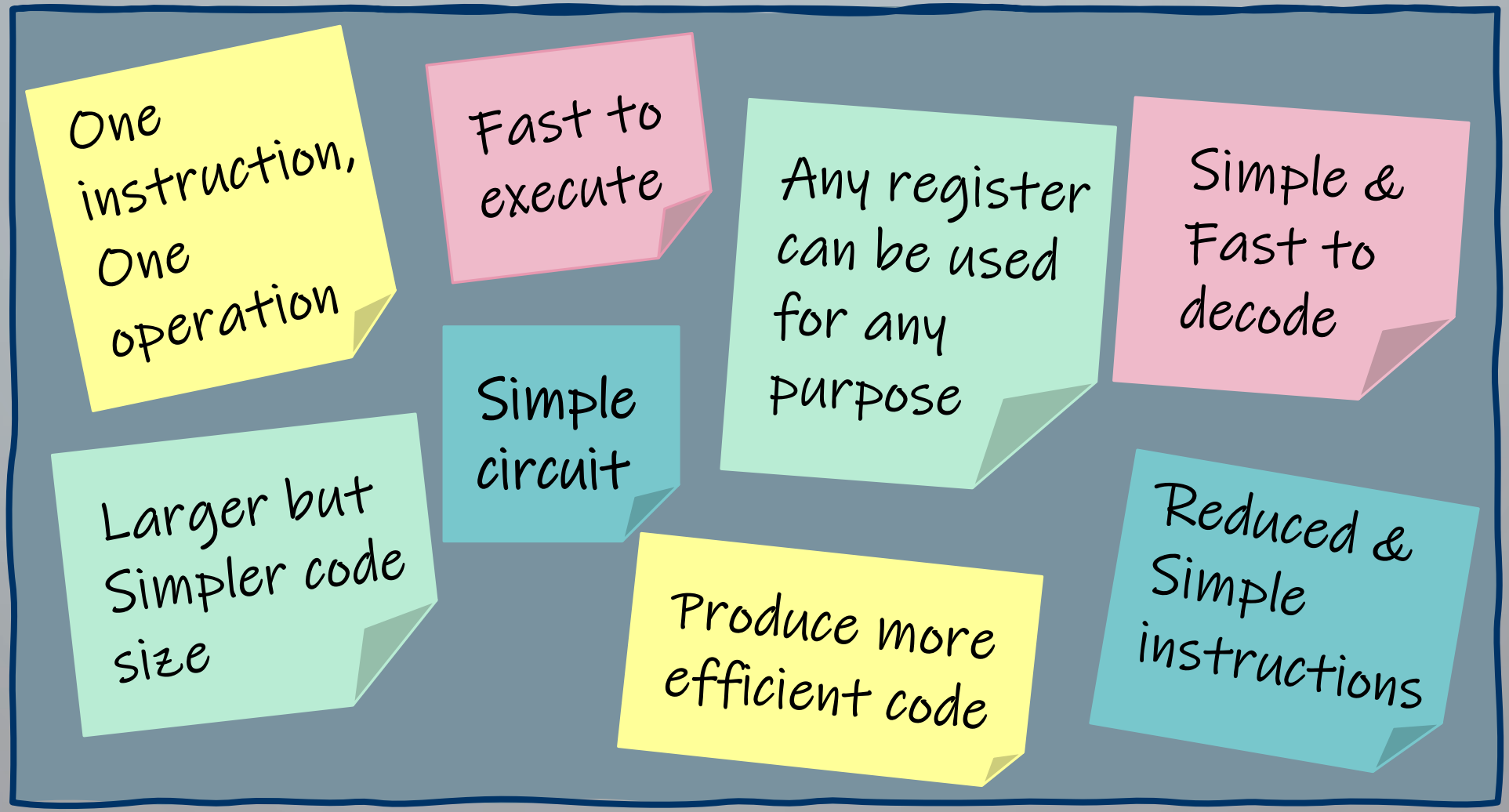
Large Register Set

- Large number of registers required for reg-to-reg operations
- Provide ample opportunities for the compiler to optimize their usage

Fixed Length

- Instruction lengths are fixed
- Implementation & execution efficiency
- Efficient decoding & scheduling

Why RISC?


One instruction, One operation
Fast to execute
Any register can be used for any purpose
Simple & Fast to decode
Simple circuit
Larger but Simpler code size
Produce more efficient code
Reduced & Simple instructions


ARM
ADVANCED RISC MACHINES

Exploring ARM ISA

- RISC based ISA
- 2 separate instruction set states:
  - 32-bit ARM
  - 16-bit Thumb
- 7 basic operating modes:
  - User : unprivileged mode under which most tasks run
  - FIQ : entered when a high priority (fast) interrupt is raised
  - IRQ : entered when a low priority (normal) interrupt is raised
  - Supervisor : entered on reset and when a Software Interrupt instruction is executed
  - Abort : used to handle memory access violations
  - Undef : used to handle undefined instructions
  - System : privileged mode using the same registers as user mode

ARM state features

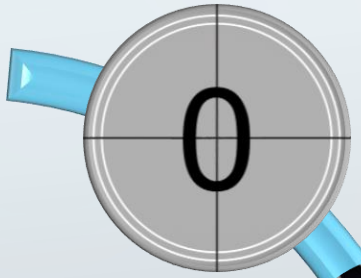
- 3-address data processing instructions
- Conditional execution of each instruction
- Shift and ALU operations in single instruction
- Load-Store and Load-Store multiple instructions
- Single cycle execution of all instructions
- Instruction set extension through coprocessor instructions

Thumb state features

- Works on 32-bit values, produces 32-bit addr for mem access
- Access to low registers (r0 - r7) similar to ARM state
- Restricted access to high registers (r8 - r15), MOV, ADD, CMP can access
- Is enabled by 'T' bit (if set) of CPSR
- If enabled, fetches 16-bit code from HW aligned addresses, PC is incremented by two bytes
- New Instructions: LSL, LSR, ASR, ROR (barrel shifter can't be combined with any instruction)
- Not conditionally executable (except 'B' branch instruction)
- Instructions removed: MLA, RSB, RSC, MSR, MRS, SWP, SWPB and coprocessor instructions

REGISTERS

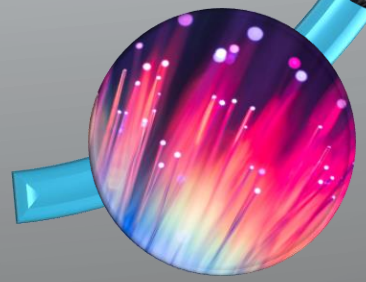
ARM has 37 registers all of which are 32-bits long



1 dedicated program counter (pc)

1 dedicated current program status register (cpsr)

5 dedicated saved program status registers (spsr)



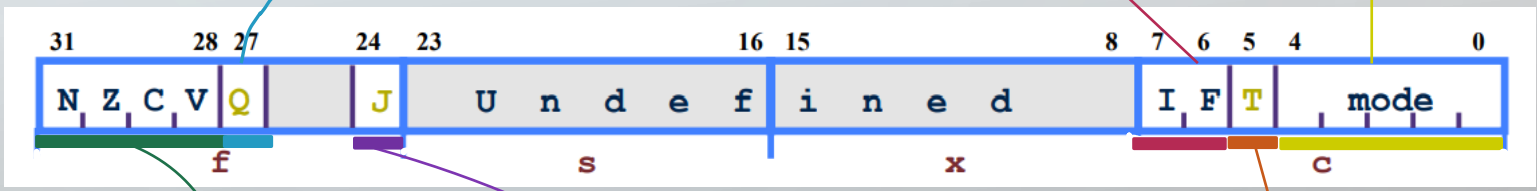
30 general purpose registers

Program status registers (cpsr & spsr)

Sticky Overflow flag - Q flag:
Indicates if saturation has occurred

Interrupt Disable bits:
I = 1: Disables the IRQ
F = 1: Disables the FIQ

Mode bits:
Specify the processor mode



Condition code flags:
N = Negative result from ALU
Z = Zero result from ALU
C = ALU operation Carried out
V = ALU operation overflowed

J bit:
J = 1: Processor in Jazelle state

T Bit:
T = 0: Processor in ARM state
T = 1: Processor in Thumb state

8-bit Jazelle instruction mode is a mode that allows java bytecode to be directly executed in ARM architecture in order to improve performance.

Program counter (pc)

When the processor is executing in ...

ARM state:

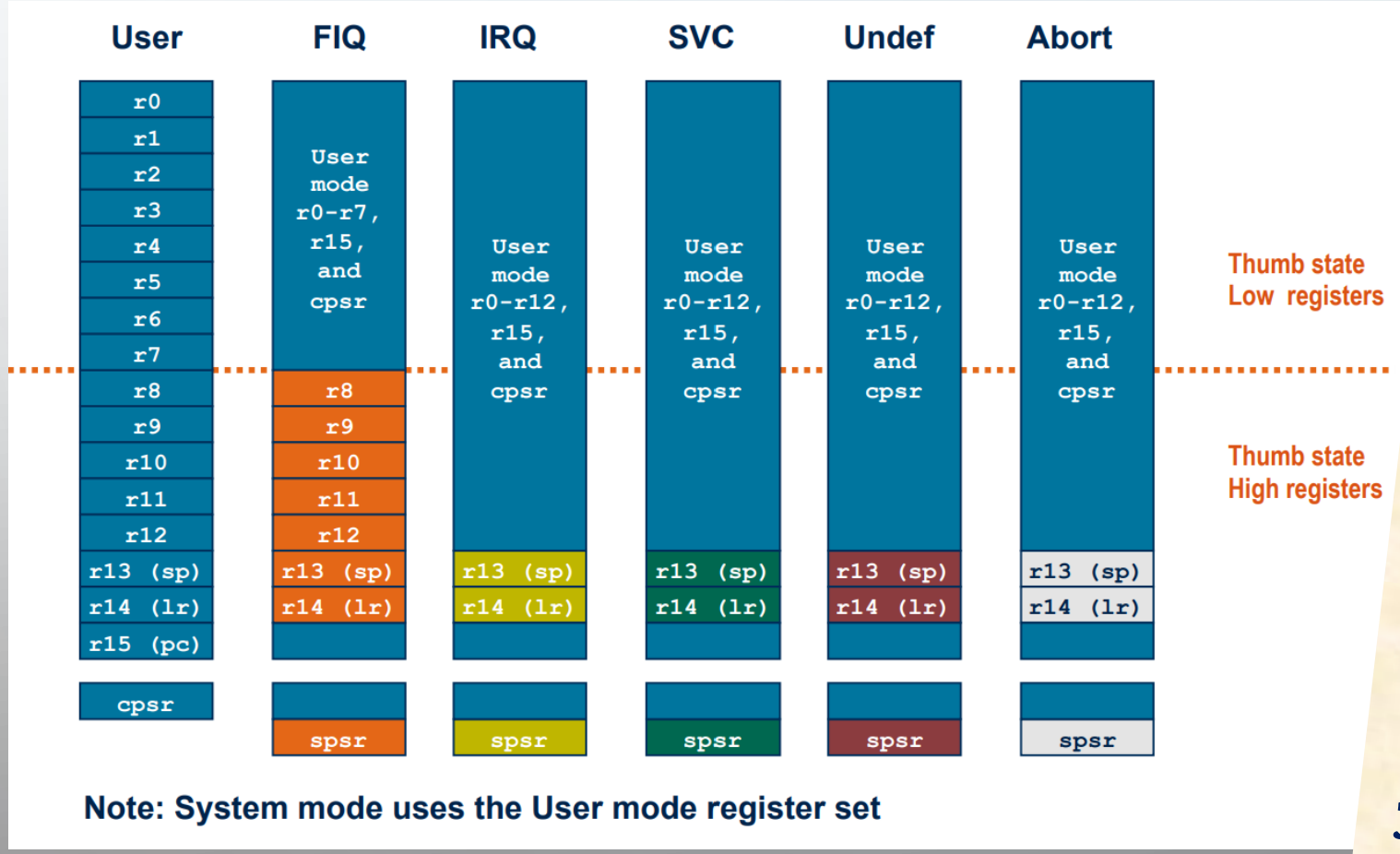
- All instructions are 32 bits wide
- All instructions must be word aligned (4 bytes)
- The pc value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned)

Thumb state:

- All instructions are 16 bits wide
- All instructions must be halfword aligned (2 bytes)
- The pc value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned)

Jazelle state:

- All instructions are 8 bits wide
- Processor performs a word access to read 4 instructions at once



As it is shown, each mode has access to a particular register set from all 37 registers.

Note that in all cases, just 32 registers are visible and others are banked out.

OPERATIONS

1	Data processing instructions
Branch instructions	2
3	Load-Store instructions
Software interrupt instructions	4
5	Program status register instructions
Coprocessor instructions	6

ARM state

Data processing Instructions 1/3

- ✓ 16 basic instructions (excluding two basic multiply instrs.)
- ✓ Move, Arithmetic, Logical, Compare and Multiply operations
- ✓ Operate on two 32-bit operands, produce 32-bit result
- ✓ All operations except multiply instructions are carried out in ALU
- ✓ Multiply instructions are carried out in multiplier block
- ✓ Do not access memory
- ✓ Can pre-process one operand using barrel shifter

Data processing Instructions 2/3

Default: AL

✓ Syntax:

<opcode>{<cond>} {S} Rd, Rn, n

"cond": Indicates flags to test

"S": Set condition flags in CPSR

"Rd": Destination

"Rn": 1st operand

"n": May be "Rm", "#const" or "Rs, <shift | rotate> n"

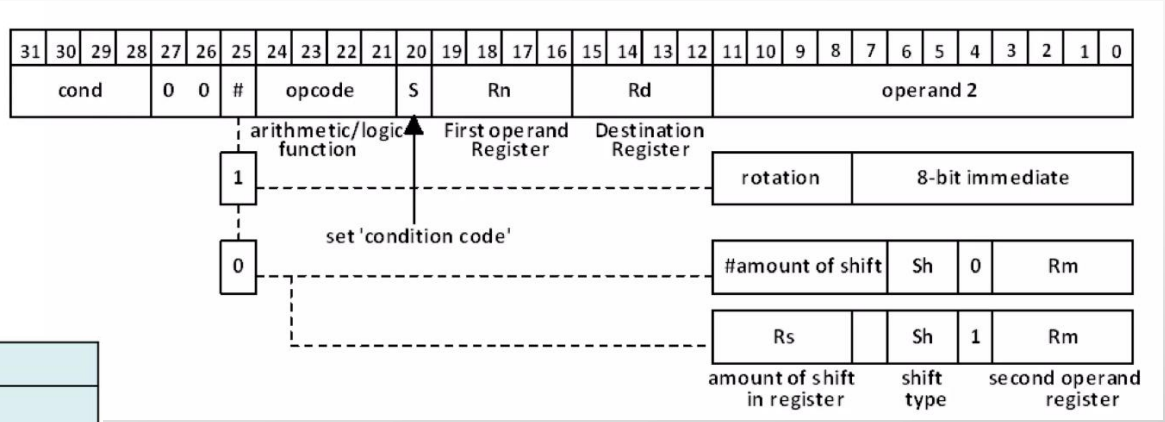
Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

Rn/Rm/Rs remains unchanged

Condition codes & Flag states tested

Data processing Instructions 3/3

Binary encoding



List of instructions

MOV	Move a 32-bit value	MOV Rd,n	Rd = n
MVN	Move negated (logical NOT) 32-bit value	MVN Rd,n	Rd = ~n
ADD	Add two 32-bit values	ADD Rd,Rn,n	Rd = Rn+n
ADC	Add two 32-bit values and carry	ADC Rd,Rn,n	Rd = Rn+n+C
SUB	Subtract two 32-bit values	SUB Rd,Rn,n	Rd = Rn-n
SBC	Subtract with carry of two 32-bit values	SBC Rd,Rn,n	Rd = Rn-n+C-1
RSB	Reverse subtract of two 32-bit values	RSB Rd,Rn,n	Rd = n-Rn
RSC	Reverse subtract with carry of two 32-bit values	RSC Rd,Rn,n	Rd = n-Rn+C-1
AND	Bitwise AND of two 32-bit values	AND Rd,Rn,n	Rd = Rn AND n
ORR	Bitwise OR of two 32-bit values	ORR Rd,Rn,n	Rd = Rn OR n
EOR	Exclusive OR of two 32-bit values	EOR Rd,Rn,n	Rd = Rn XOR n
BIC	Bit clear. Every '1' in second operand clears corresponding bit of first operand	BIC Rd,Rn,n	Rd = Rn AND (NOT n)
CMP	Compare	CMP Rd,n	Rd-n & change flags only
CMN	Compare Negative	CMN Rd,n	Rd+n & change flags only
TST	Test for a bit in a 32-bit value	TST Rd,n	Rd AND n, change flags
TEQ	Test for equality	TEQ Rd,n	Rd XOR n, change flags

MUL	Multiply two 32-bit values	MUL Rd,Rm,Rs	Rd = Rm*Rs
MLA	Multiple and accumulate	MLA Rd,Rm,Rs,Rn	Rd = (Rm*Rs)+Rn

N. Mathivanan

Barrel Shifter

LSL : Logical Left Shift

Multiplication by a power of 2

ASR: Arithmetic Right Shift

Division by a power of 2, preserving the sign bit

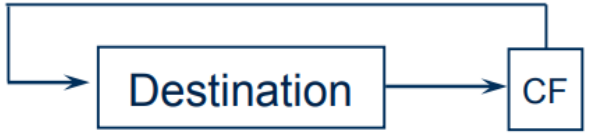
LSR : Logical Shift Right

Division by a power of 2

ROR: Rotate Right

Bit rotate with wrap around from LSB to MSB

RRX: Rotate Right Extended



Single bit rotate with wrap around from CF to MSB

Immediate Constant

- ★ No ARM instruction can contain a 32 bit immediate constant
- ★ To allow larger constants to be loaded, the assembler offers a pseudo-instruction:

  LDR rd, =const
- ★ This will either:

  1. Produce a MOV or MVN instruction to generate the value (if possible)

  Or

  2. Generate a LDR instruction with a PC-relative address to read the constant from a literal pool (Constant data area embedded in the code)

Other Multiply Instructions

[U|S] MULL {<cond>} {S} RdLo, RdHi, Rm, Rs

RdHi , RdLo = Rm * Rs

[U|S] MLAL {<cond>} {S} RdLo, RdHi, Rm, Rs

RdHi , RdLo = (Rm * Rs) + RdHi, RdLo

Branch Instructions 1/2

- ✓ Divert sequential execution / CALL a subroutine
- ✓ Range: +/- 32 MB from current position (i.e. PC-8), PC relative offset
- ✓ Has 24 bits allocated for specifying word offset
- ✓ With 24-bit offset append '00' at LSB, extend sign bit, place into PC
- ✓ PC is set to point to new address
- ✓ Syntax:

Branch : B{<cond>} label

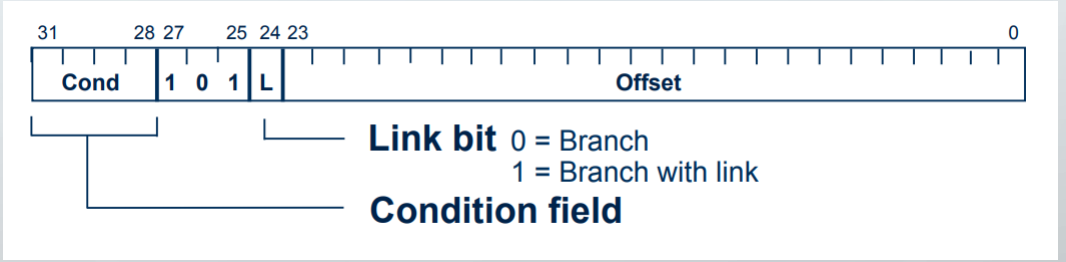
Branch with Link : BL{<cond>} subroutine_label

B	Branch	**B label**	PC = label, (unconditional branch)
BL	Branch and Link	**BL label**	LR = PC-4, PC = label, (CALL functionality)
BX	Branch and Exchange	**BX Rm**	PC = Rm, 'T' bit of CPSR = 1 (to ARM state)
BLX	Branch with Link Exchange	**BLX Rm**	LR = PC-4, PC = Rm, 'T' bit of CPSR = 1
		BLX label	LR = PC-4, 'T' bit of CPSR = 1, PC = label

N. Mathivanan

Branch Instructions 2/2

Binary encoding



- ✓ BL stores return address in LR
- ✓ For non-leaf functions, LR will have to be stacked

Load/Store Instructions 1/4

Single Transfer Instruction

- ✓ Transfers boundary aligned Word/HW/Byte between mem & reg
- ✓ Address of mem loc. is given by Base Addr. +/- Offset
- ✓ There are some methods of providing offset (Addressing modes):
  - Register addressing - A register contains offset
  - Immediate addressing - Immediate constant is offset
  - Scaled addressing - Offset in a reg is scaled using shift operation

	Addressing mode	Instruction	Operation
STR	Register addressing	`STR Rd,[Rn,Rm]`	`mem32[Rn+Rm]=Rd`
	Immediate addressing (with offset zero)	`STR Rd,[Rn]`	`mem32[Rn]=Rd`
	Immediate addressing	`STR Rd,[Rn,#offset]`	`mem32[Rn+offset]=Rd`
	Scaled addressing	`STR Rd,[Rn,Rm LSL #n]`	`mem32[Rn+(Rm<<n)]=Rd`
LDR	Register addressing	`LDR Rd,[Rn,Rm]`	`Rd=mem32[Rn+Rm]`
	Immediate addressing (with offset zero)	`LDR Rd,[Rn]`	`Rd=mem32[Rn]`
	Immediate addressing	`LDR Rd,[Rn,#offset]`	`Rd=mem32[Rn+offset]`
	Scaled addressing	`LDR Rd,[Rn,Rm LSL #n]`	`Rd=mem32[Rn+(Rm<<n)]`

Load/Store Instructions 2/4

✓ Choice of pre-indexed or post-indexed addressing

Indexing	Instruction	Operation
Preindex	LDR Rd,[Rn,n]	Rd=[Rn+n],
	STR Rd,[Rn,n]	[Rn+n]=Rd
Preindex with write back	LDR Rd,[Rn,n]!	Rd=[Rn+n],Rn=Rn+n
	STR Rd,[Rn,n]!	[Rn+n]=Rd, Rn=Rn+n
Postindex	LDR Rd,[Rn],n	Rd=[Rn],Rn=Rn+n
	STR Rd,[Rn],n	[Rn]=Rd,Rn=Rn+n

✓ Syntax:

<opcode>{<condition>} {<type>} Rd, [Rn{,<offset>}]

"type": H, HS, B, BS

"Rd": source/destination register

"Rn": Base address

"<offset>": "Rm" or "#(0-4095)" or "Rm, <shift>#n"

LDR	STR	Word
LDRB	STRB	Byte
LDRH	STRH	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

Load/Store Instructions 3/4

Multiple Transfer Instruction

- ✓ Transfers data between multiple regs & mem in single instruction
- ✓ Useful in stack, block move, temporary store & restore
- ✓ Advantages: small code size, single instruction fetch from memory
- ✓ Disadvantages: can't be interrupted, increases interrupt latency
- ✓ Syntax: LDM, STM

<opcode>{<cond>} <mode>Rn{!}, <registers>

"Rn": Base register

"!": Update base reg. after data transfer (option)

<mode>	Description	Start address	End address	Rn!
IA	Increment After	Rn	Rn+N*4-4	Rn+N*4
IB	Increment Before	Rn+4	Rn+N*4	Rn+N*4
DA	Decrement After	Rn-N*4+4	Rn	Rn-N*4
DB	Decrement Before	Rn-N*4	Rn-4	Rn-N*4

Load/Store Instructions 4/4

Stack operations

- ✓ Alternative suffixes to IA, IB, DA and DB
- ✓ Full/Empty, Ascending/Descending (FD, FA, ED, EA)

Stack type	Store multiple instruction (Push operation)	Load multiple instruction (Pop operation)
Full Descending	STMFD (STMDB)	LDMFD (LDMIA)
Full Ascending	STMFA (STMIB)	LDMFA (LDMDA)
Empty Descending	STMED (STMDA)	LDMED (LDMIB)
Empty Ascending	STMEA (STMIA)	LDMEA (LDMDB)

Swap Instructions (also known as semaphore)

SWP	Swap a word between register and memory	SWP Rd,Rm,[Rn]	temp = mem32[Rn] mem32[Rn] = Rm Rd = temp
SWPB	Swap a byte between register and memory	SWPB Rd,Rm,[Rn]	temp = mem8[Rn] mem8[Rn] = Rm Rd = temp

N. Mathivanan

Software Interrupt Instructions

- ✓ Used in User mode applications to execute OS routines
- ✓ When executed, mode changes to supervisor mode
- ✓ Syntax: SWI{cond} SWI_number
- ✓ Return instruction from SWI routine: MOVS PC, r14
- ✓ Distinguish subroutine call and SWI instruction execution

Program Status Register Instructions

- ✓ Instructions to read/write from/to CPSR or SPSR
- ✓ Syntaxes:

MRS {<cond>} Rd, <CPSR | SPSR>

MSR{<cond>} <CPSR | SPSR>, Rm

MSR{<cond>} <CPSR | SPSR>_<fields>, Rm

MSR{<cond>} <CPSR | SPSR>_<fields>, #immediate

Coprocessor Instructions 1/2

- ✓ ARM supports 16 coprocessors
- ✓ Coprocessor implemented in hardware, software or both
- ✓ Coprocessor contains instruction pipeline, instruction decoding logic, handshake logic, register bank, special processing logic with its own data path
- ✓ Also connected to data bus
- ✓ Instructions in program memory are available for coprocessors too
- ✓ Works in conjunction with ARM core and process same instruction stream as ARM, but executes only instructions meant for coprocessor and ignores ARM and other coprocessor's instructions
- ✓ 3 types of instructions: Data processing, Register transfer, Memory transfer

Coprocessor Instructions 2/2

CDP	Coprocessor data processing	`CDP p5,2,c12,c10,c3,4`	Coprocessor number 5, opcode1 – 2, opcode2 – 4, Coprocessor destination register – 12, Coprocessor source registers – 10 & 3
MCR	Move to coprocessor from ARM register	`MCR p14,1,r7,c7,c12,6`	Coprocessor number 14, opcode1 – 2, opcode2 – 4, ARM source register – r7, Coprocessor destination registers – 10 & 3
MRC	Move to ARM register from coprocessor	`MRC p15,5,r4,c0,c2,3`	Coprocessor number 15, opcode1 – 5, opcode2 – 3, ARM destination register – r4, Coprocessor source registers – 0 & 2
MCRR	Move to coprocessor from two ARM registers	`MCRR p10,3,r4,r5,c2`	Coprocessor number 10, opcode1 – 3, ARM source registers – r4, r5 Coprocessor destination register – 2
MRRC	Move to two ARM registers from coprocessor	`MRCC p8,4,r2,r3,c3`	Coprocessor number 8, opcode1 – 4, ARM destination registers – r2, r3 Coprocessor source register – 3

LDC	Load coprocessor register	`LDC p6,c1,[r4]`	Coprocessor number 6, Coprocessor register c1 is loaded with data from memory address in r4.
		`LDC p6,c4,[r2,#4]`	Coprocessor number 6, Coprocessor register c4 is loaded with data from memory address in r2 +4.
STC	Store coprocessor register	`STC p8,c8,[r2,#4]!`	Coprocessor number 8, Memory address is [r2] + 4 Store c8 in memory and then r2 =r2+4
		`STC p8,c9,[r2],#-16`	Coprocessor number 8, Memory address is [r2] Store c9 in memory and then r2 =r2-16

Thumb state

Thumb uses less memory space than ARM though more instructions

Data processing instructions

- 2-addr format, No conditional execution, No 'S' suffix (always ON)
- MOV, MVN, ADD, ADC, SUB, SBC, MUL, AND, ORR, EOR, BIC, NEG, LSL, LSR, ASR, ROR, CMP, CMN, TST

Load-Store instructions

- Offset: register, immediate (0-124 words), relative to PC/SP
- LDR, STR, LDMIA, STMIA, PUSH, POP

Branch instructions

- Branch instruction ('B') is conditionally executable
- B{cond}, B, BL, BX, BLX

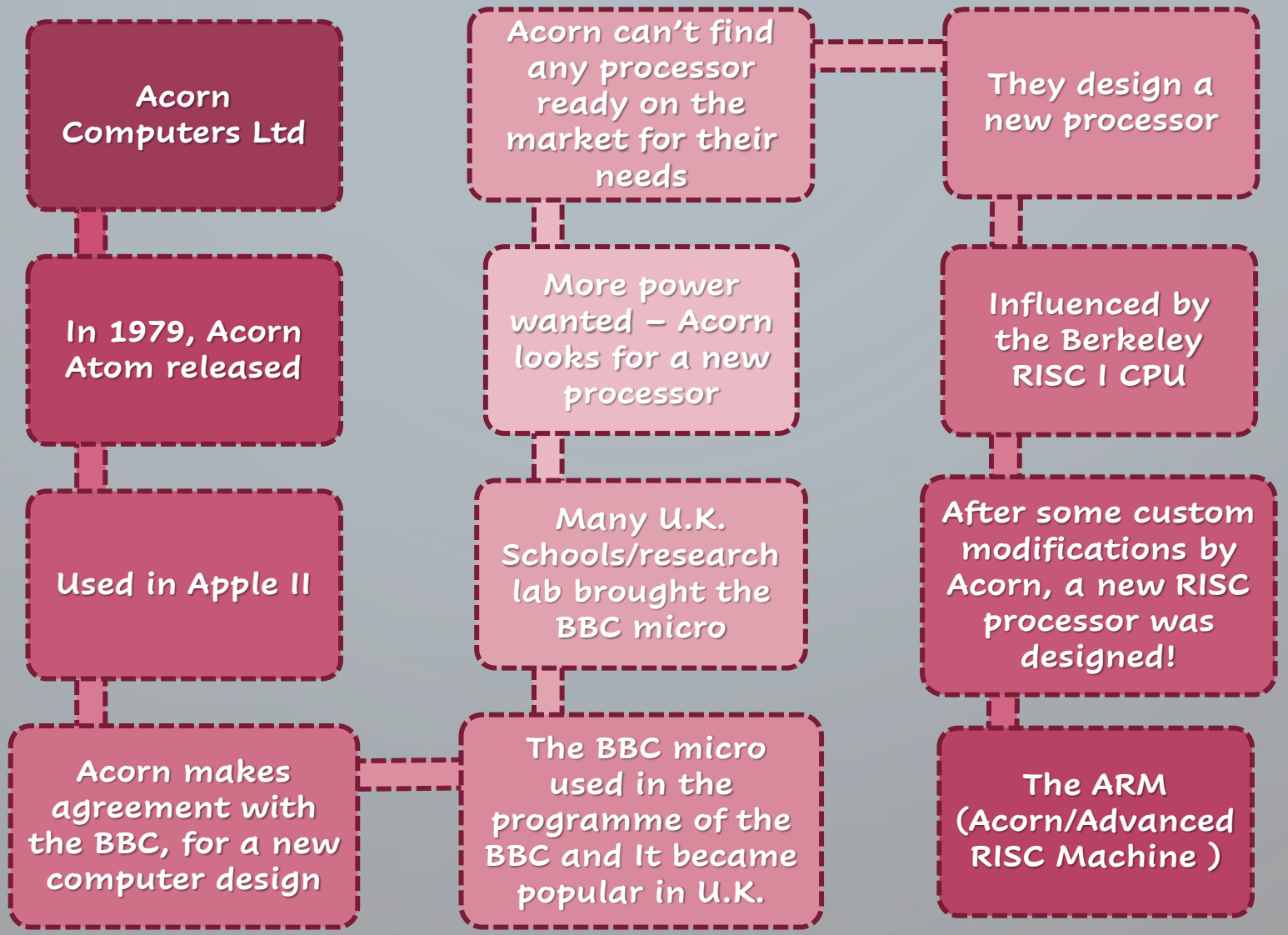
Software interrupt instructions

- Generates exception, Use just 8-bit interpreted value
- Handler enters in ARM mode

ARM ABI

- The ARM Application Binary Interface (ABI) is a set of conventions and rules that dictate how software components, such as applications and libraries, should interact with each other when running on ARM-based systems. The ABI defines various aspects of software execution, including Function calling conventions, Data representation, Register usage, System libraries, Object file formats, Name mangling.
- Adhering to the ARM ABI ensures that software components built by different compilers or written in different programming languages can work together seamlessly on ARM-based systems. This is essential for creating a consistent and predictable environment for software development and execution on ARM devices.

From Acorn to ARM



ARM Limited

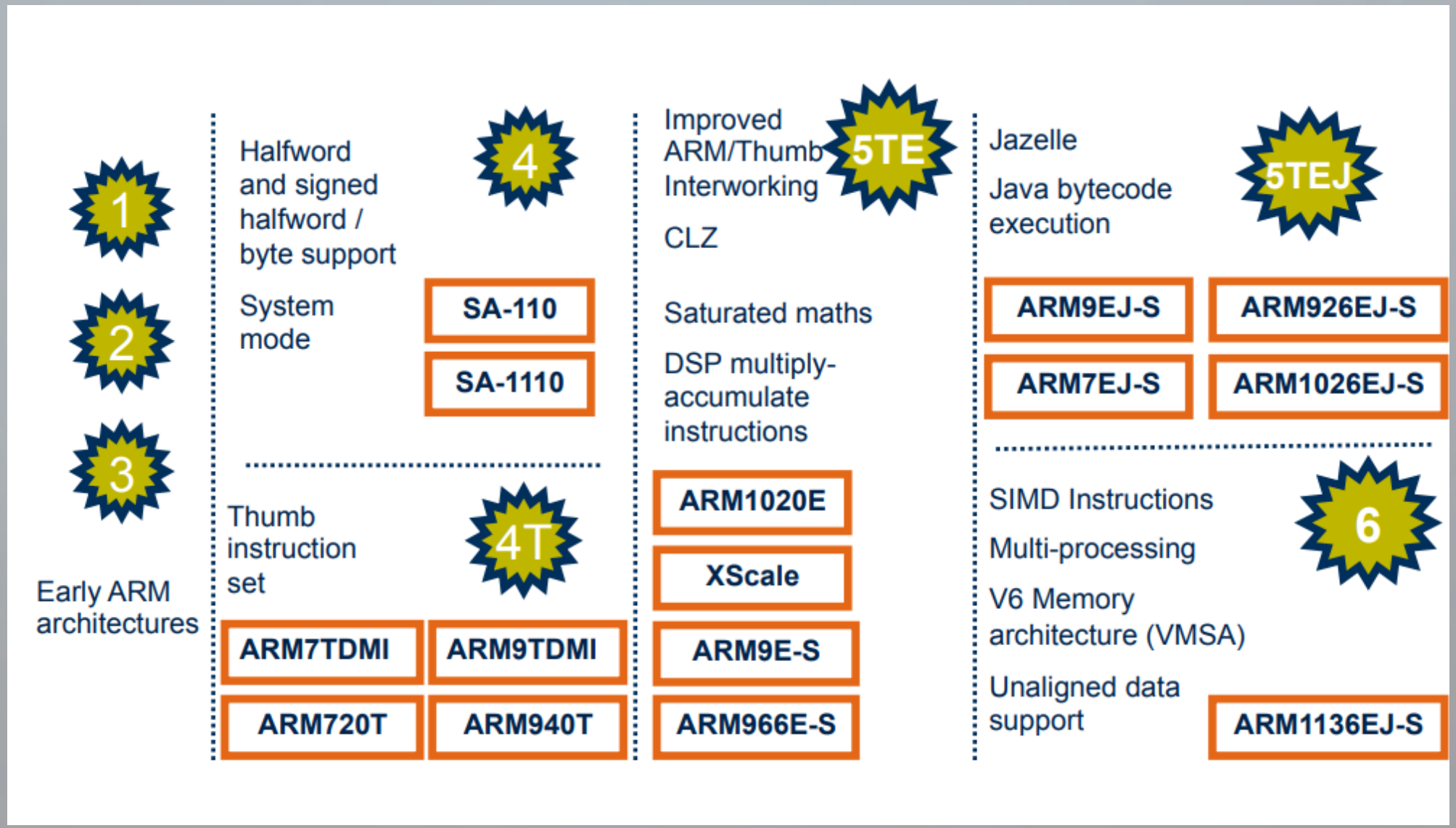
ARM = RISC with a few CISC features

Founded in November 1990
(Spun out of Acorn Computers)

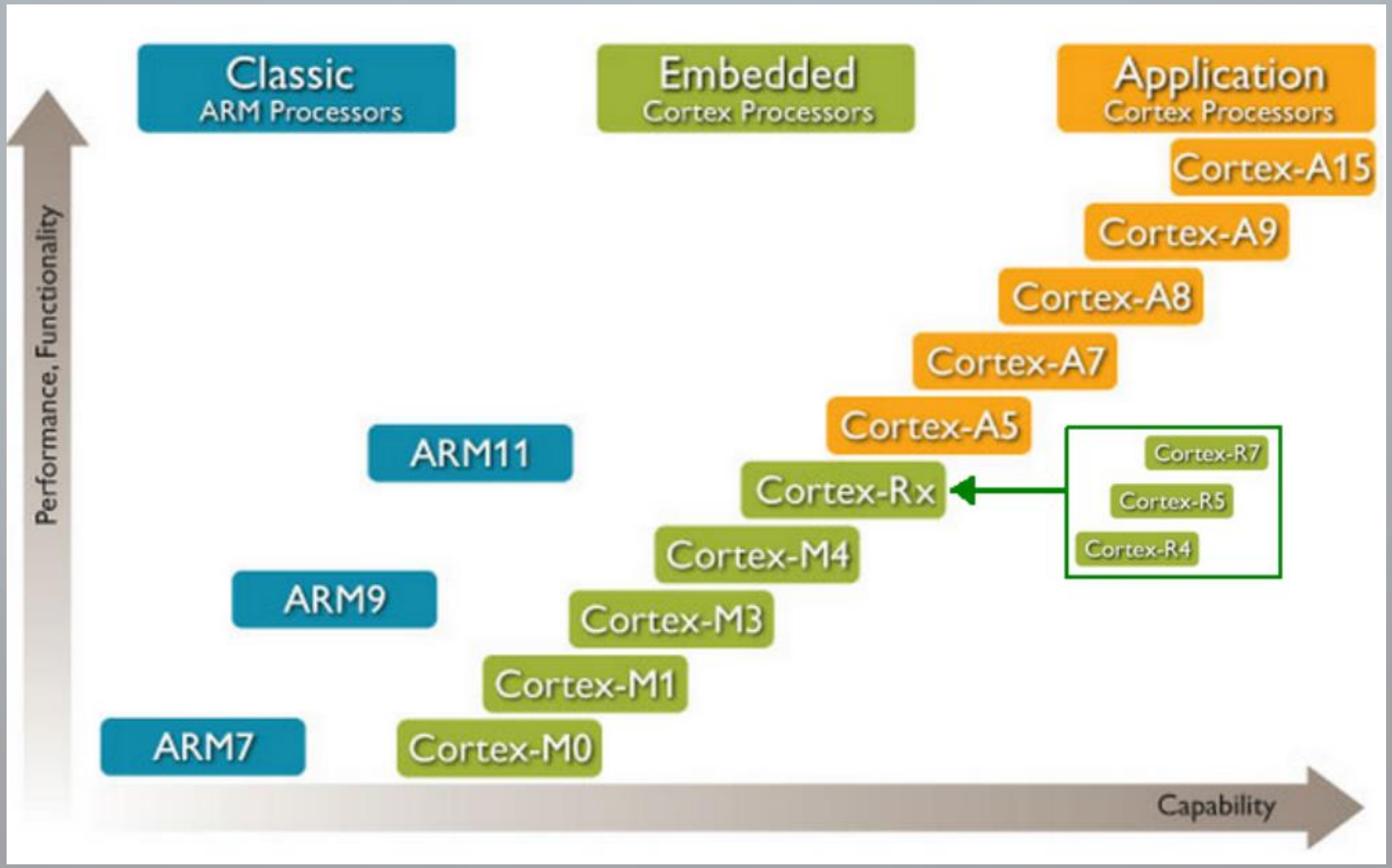
ARM does not fabricate silicon itself

Also develop technologies to assist with the design-in of the ARM architecture
(Software tools, boards, debug hardware, application software, bus architectures, peripherals etc.)

ARM Evolution



ARM Evolution



What is Cortex?

- Cortex is a family of ARM cores that have replaced Arm's classic cores with more market-specific variations.

- Cortex can be broadly categorized into three types of workloads:
  - Applications
  - Real-time (R)
  - Microcontroller (M)

Cortex-A

Designed for fully capable computers, running typical operating systems (Android, Windows, Linux, iOS). Those processors are found in anything from smartphones to tablets and laptops.

Cortex-X

Slightly enhanced version of the Cortex-A designs in order to further optimize the Cortex-A design (in terms of PPA) for certain markets or workloads

Cortex-R

Designed for real-time operating systems with very deterministic behavior. Those cores are typically less powerful than the A series and are used for things such as controllers, factory equipment, medical devices, and other machines.

Cortex-M

Designed for ultra-low power, typically small factor and low-performance systems. Those are found in cost-sensitive devices and electronics, automated systems, and many other embedded devices.

Let's Focus on the ARMv8

- The ARMv8 architecture is a 64-bit ISA.
- It represents a significant advancement over its predecessor, ARMv7, by introducing several architectural improvements and enhancements, making it suitable for a wide range of computing devices, including smartphones, tablets, servers, and embedded systems.
- One of the key features of ARMv8 is its support for two execution states: AArch32 and AArch64.

AArch32

- The execution state that maintains compatibility with the previous ARM architectures.
- A 32-bit execution state that supports the execution of 32-bit ARM and Thumb instructions.

Backward-compatible, meaning that it can run legacy 32-bit ARM software without any modifications.

AArch64

- Also known as ARMv8-A.
- The 64-bit execution state
- Supports 64-bit memory addressing and 64-bit arithmetic operations
- Increased performance, larger addressable memory space
- Not backward-compatible with AArch32

Software designed for AArch32 cannot be executed directly in AArch64 mode and requires recompilation or emulation.

Other features of ARMv8

- **Energy efficiency** ARMv8 incorporates power-saving features, including improved branch prediction, instruction prefetching, and power gating, to optimize energy efficiency and extend battery life in mobile and embedded devices.
- **Scalability** Supports a wide range of implementations, from low-power embedded devices to high-performance servers, enabling scalability across different market segments.
- **Cryptography support** ARMv8 includes cryptographic instructions to accelerate encryption and decryption operations, enhancing the performance of cryptographic algorithms.
- **Improved security** ARMv8 includes enhanced security features, such as address space layout randomization (ASLR) and pointer authentication, to mitigate security vulnerabilities and protect against exploits.
- **Virtualization support** ARMv8 architecture includes virtualization extensions, allowing efficient and secure virtualization of resources, enabling consolidation of multiple virtual machines on a single physical device.
- **Increased performance** The 64-bit execution state of AArch64 provides access to a larger number of registers, larger memory address space, and improved SIMD (Single Instruction, Multiple Data) capabilities, resulting in improved performance for compute-intensive workloads.

ARM licensing model

ARM's licensing model is based on the concept of intellectual property (IP) licensing. ARM, designs and licenses processor architectures and related technologies rather than manufacturing and selling physical chips. This approach allows ARM to focus on innovation and design, while partnering with semiconductor companies to produce chips based on their architectures.

ARM offers two main types of licenses:

Architecture License: This license grants the licensee the right to use ARM's processor architecture to design their own chips. It provides access to the instruction set architecture (ISA) and the necessary design specifications. Companies with an architecture license can create custom processors tailored to their specific requirements.

Processor IP License: This license provides the licensee with a pre-designed processor core, known as intellectual property (IP), which they can integrate into their own chips. ARM offers a range of processor IP cores, from low-power microcontrollers to high-performance application processors. Licensees can customize and optimize these cores to suit their target markets and applications.

Which companies use ARM?

Apple's iPhone and iPad devices use ARM-based processors. Apple's custom-designed chips, such as the Apple A-series and M-series processors, are based on ARM architecture. These chips provide high-performance and energy-efficient computing capabilities for Apple's mobile devices.

Qualcomm, a leading semiconductor company, utilizes ARM architecture in its Snapdragon processors. Snapdragon chips power many smartphones and tablets running Android operating systems. They are known for their integration of processing power, graphics performance, and connectivity features.

Samsung incorporates ARM-based processors in its mobile devices, including smartphones and tablets. Their Exynos processors, based on ARM architecture, provide computing power for Samsung's flagship devices.

NVIDIA, known for its graphics processing units (GPUs), aims to leverage ARM's architecture and expertise to enhance its GPU capabilities and expand its reach into various markets, including edge computing and artificial intelligence.

MediaTek designs and manufactures system-on-chips (SoCs) for smartphones, tablets, and other consumer electronics. Their SoCs are often based on ARM architecture, delivering a balance of performance and affordability.

Amazon's AWS Graviton processors, used in their Amazon Web Services (AWS) cloud infrastructure, rely on ARM-based architecture. These processors offer efficient and cost-effective options for cloud-based computing.

Microsoft employs ARM-based processors in its Surface Pro X devices, providing a balance between performance and battery life. They also utilize ARM architecture in their cloud services, such as Azure, for specific workloads.

ARM Partnership

ARM Product Code Description

- ✓ **M**: Multiplier

ARM processor have hardware multiplier unit doing multiplication

- ✓ **I**: Embedded ICE Macrocel

Hardware circuit used to generate trace information. Used in advance debugging.

- ✓ **E**: Enhanced Instruction Set
- ✓ **J**: Java Acceleration by Jazelle mode
- ✓ **F**: Vector Floating point

Hardware implementation of floating operations.

- ✓ **S**: Synthesizable Version

The ARM architecture can be modified as it comes in terms of soft processor core.

So …

Cheap

Easy to Develop

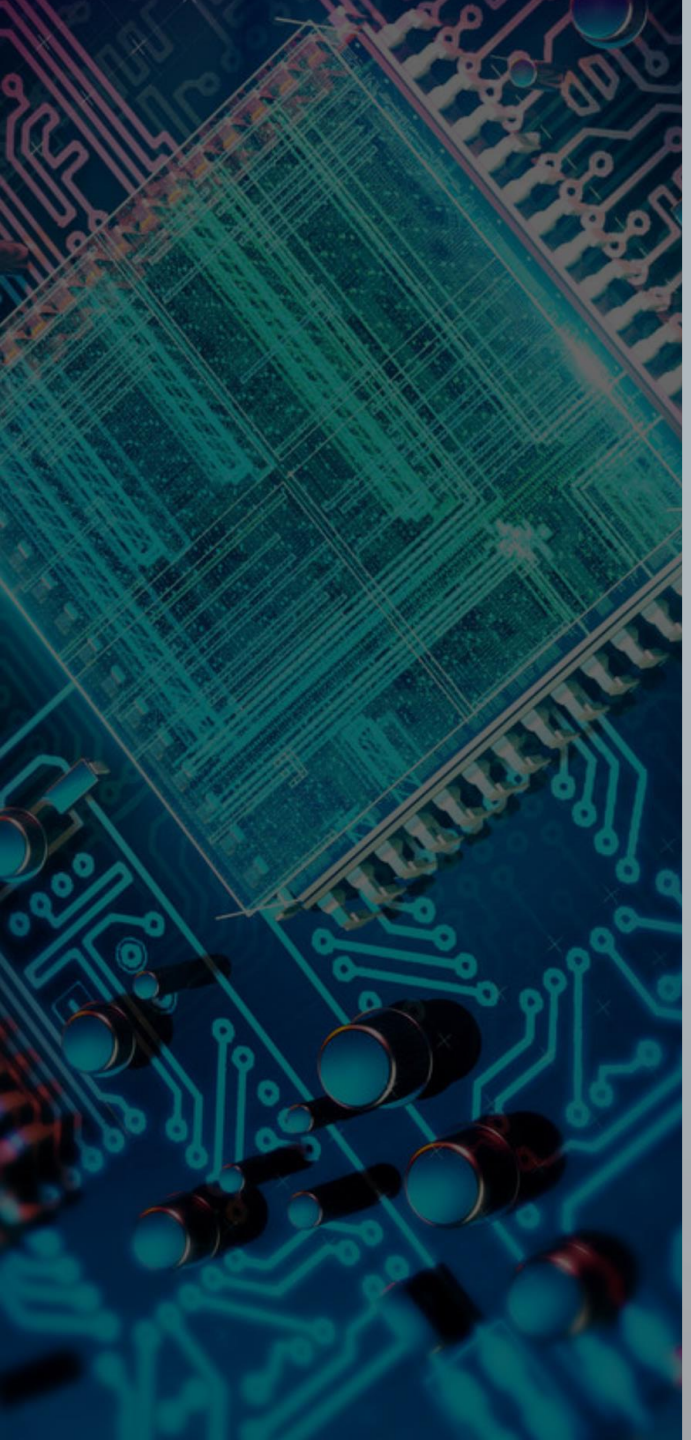
Great Performance

Cost Sensitive Applications

Low Power Consumption

Great Computing Power

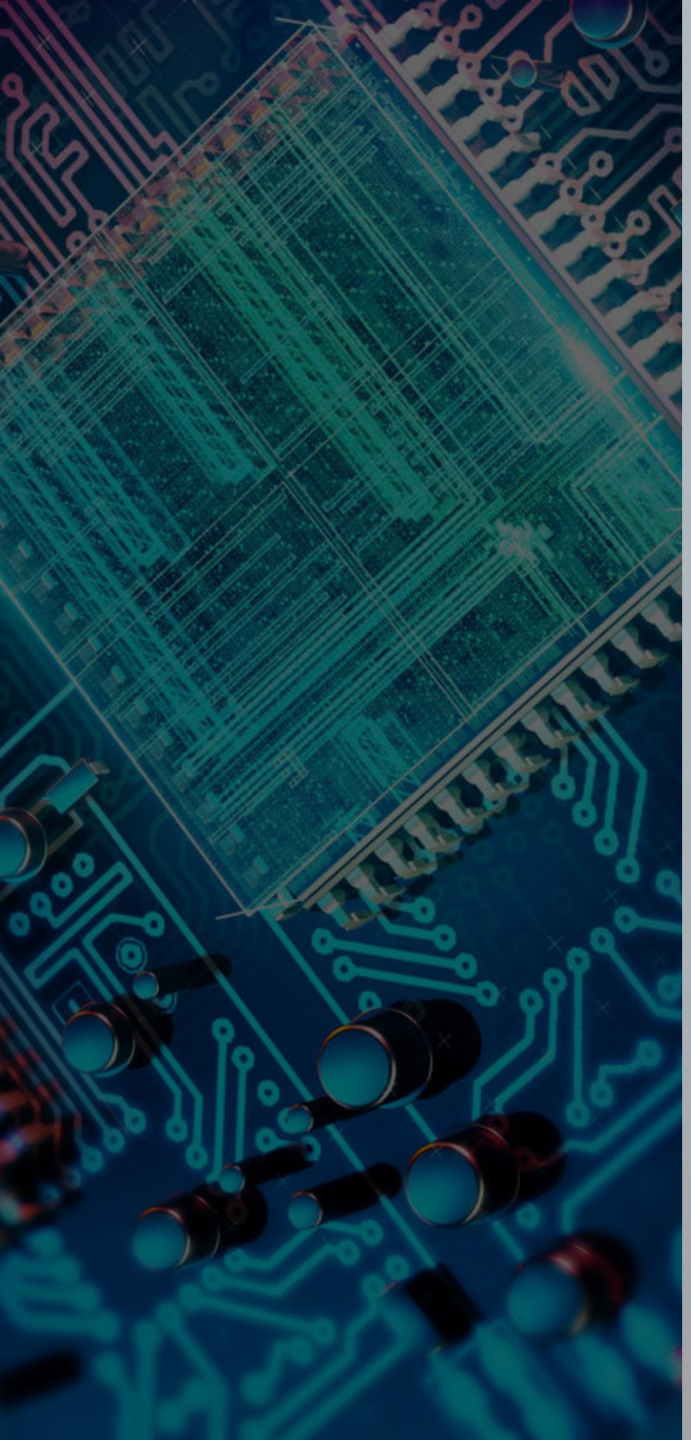

RISC-V



Exploring RISC-V ISA

REGISTERS

- There are 31 general-purpose registers, which hold fixed-point values + 1 register hardwired to the constant 0.
- For RV64, the registers are 64 bits wide, and for RV32, they are 32 bits wide.
- + There are 32, 64-bit registers, which hold single- or double-precision floating-point values.
- + There are 2 special user-visible registers. The program counter (pc) holds the address of the current instruction. The floating-point status register (fsr) contains the operating mode and exception status of the floating-point unit.

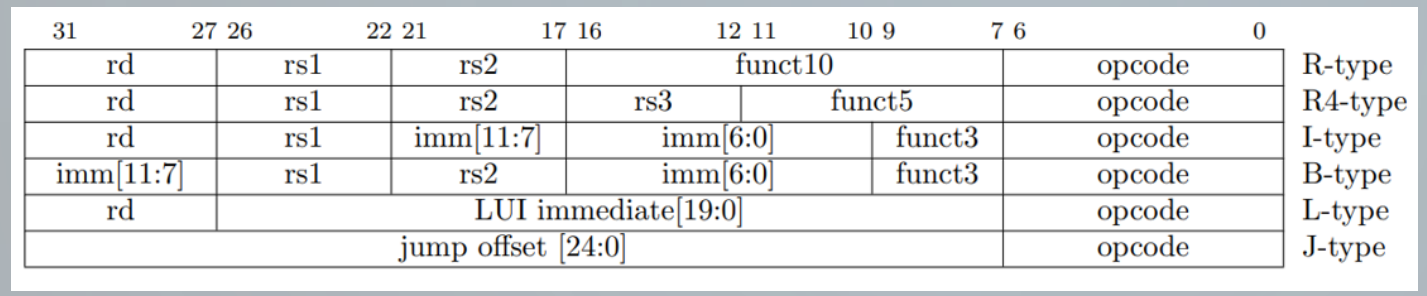


- The base RISC-V ISA has fixed-length 32-bit instructions that must be naturally aligned on 32-bit boundaries.

  However, the RISC-V encoding scheme is designed to support ISA extensions with variable-length instructions, where each instruction can be any number of 16-bit instruction parcels in length. Parcels are naturally aligned on 16-bit boundaries.

OPERATIONS

In the base ISA, there are six basic instruction formats.



31 27	26 22	21 17	16 12	11 10	9 7	6 0	
rd	rs1	rs2	funct10			opcode	R-type
rd	rs1	rs2	rs3	funct5		opcode	R4-type
rd	rs1	imm[11:7]	imm[6:0]		funct3	opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]		funct3	opcode	B-type
rd	LUI immediate[19:0]					opcode	L-type
jump offset [24:0]						opcode	J-type

R-type instructions specify two source registers (rs1 and rs2) and a destination register (rd). The funct10 field is an additional opcode field.

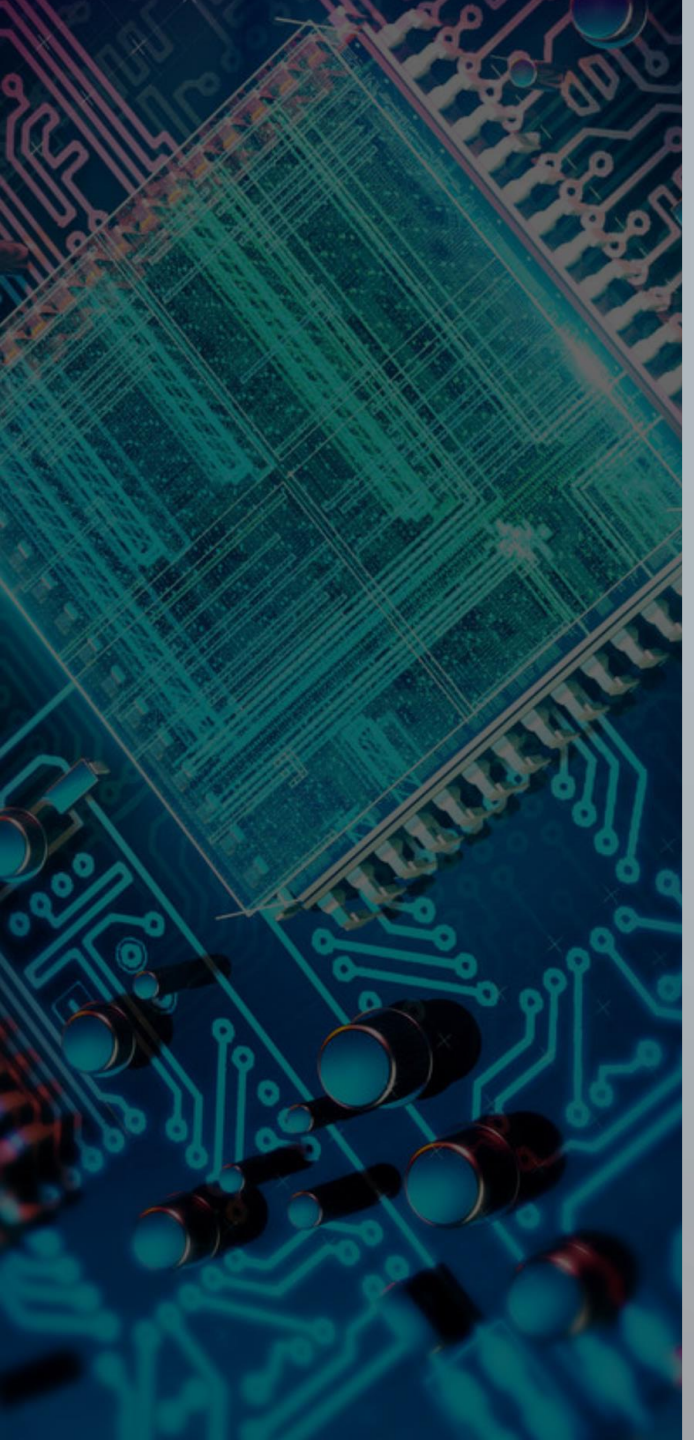
R4-type instructions specify three source registers (rs1, rs2, & rs3) and a destination register (rd). The funct5 field is a second opcode field. This format is only used by the floating-point fused multiply-add instructions.

B-type instructions specify two source registers (rs1 and rs2) & a third source operand encoded as a sign-extended 12-bit immediate. The immediate is encoded as the concatenation of the bits 31–27, and bits 16–10. The funct3 field is a second opcode field.

I-type instructions specify one source register (rs1) and a destination register (rd). The second source operand is a sign-extended 12-bit immediate. The funct3 field is a second opcode field.

L-type instructions specify a destination register (rd) and a 20-bit immediate value. lui is the only instruction of this format.

J-type instructions encode a 25-bit jump target address as a PC-relative offset. The 25-bit immediate value is shifted left one bit and added to the current PC to form the target address.



Load and Store Instructions

Integer Computational Instructions

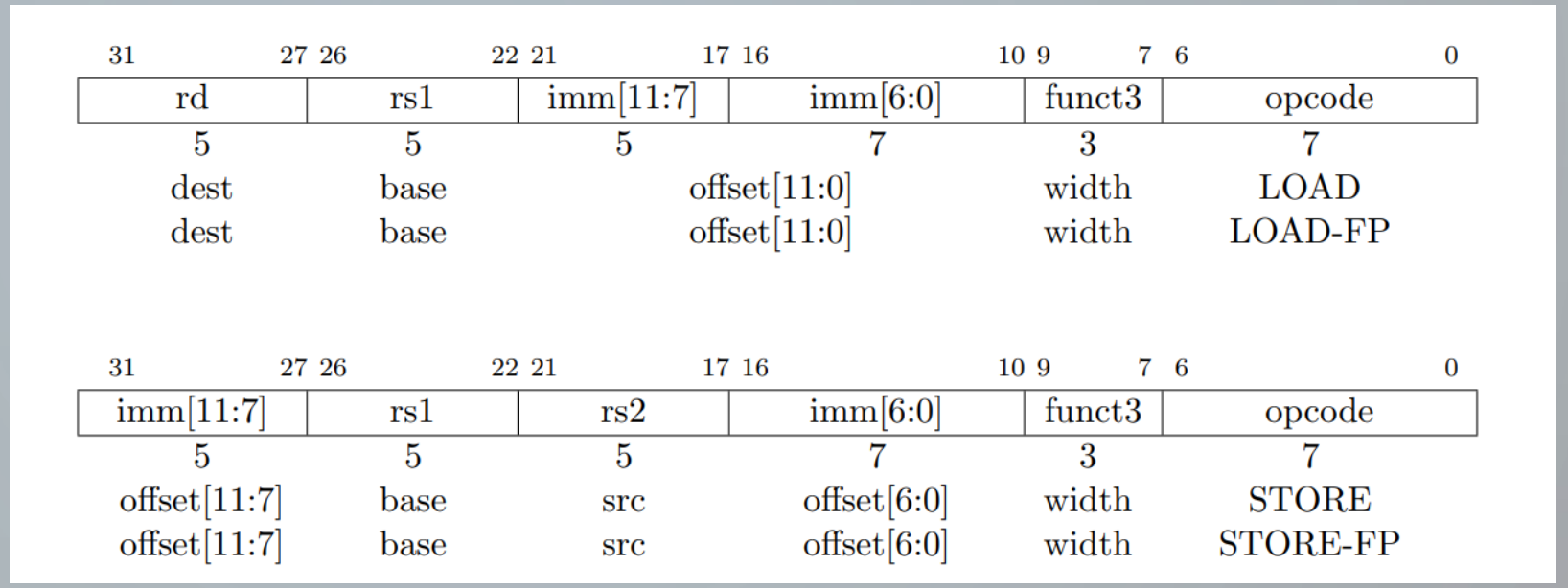
Control Transfer Instructions

Floating-Point Instructions

Memory Model

System Instructions

Load & Store Instructions 1/3



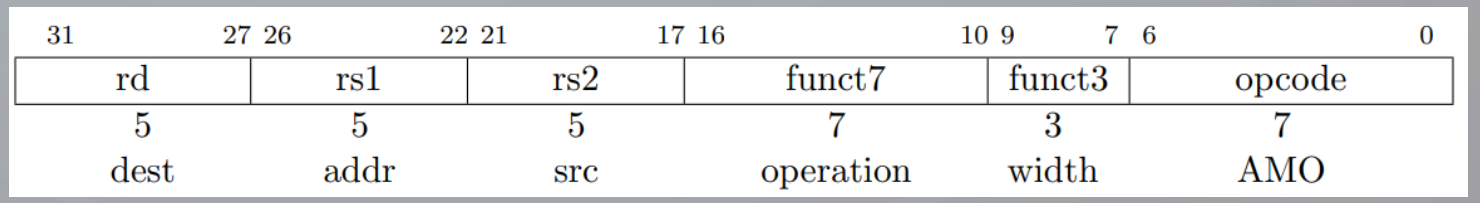
31–27	26–22	21–17	16–10	9–7	6–0
rd	rs1	imm[11:7]	imm[6:0]	funct3	opcode
5	5	5	7	3	7
dest	base	offset[11:0]		width	LOAD
dest	base	offset[11:0]		width	LOAD-FP

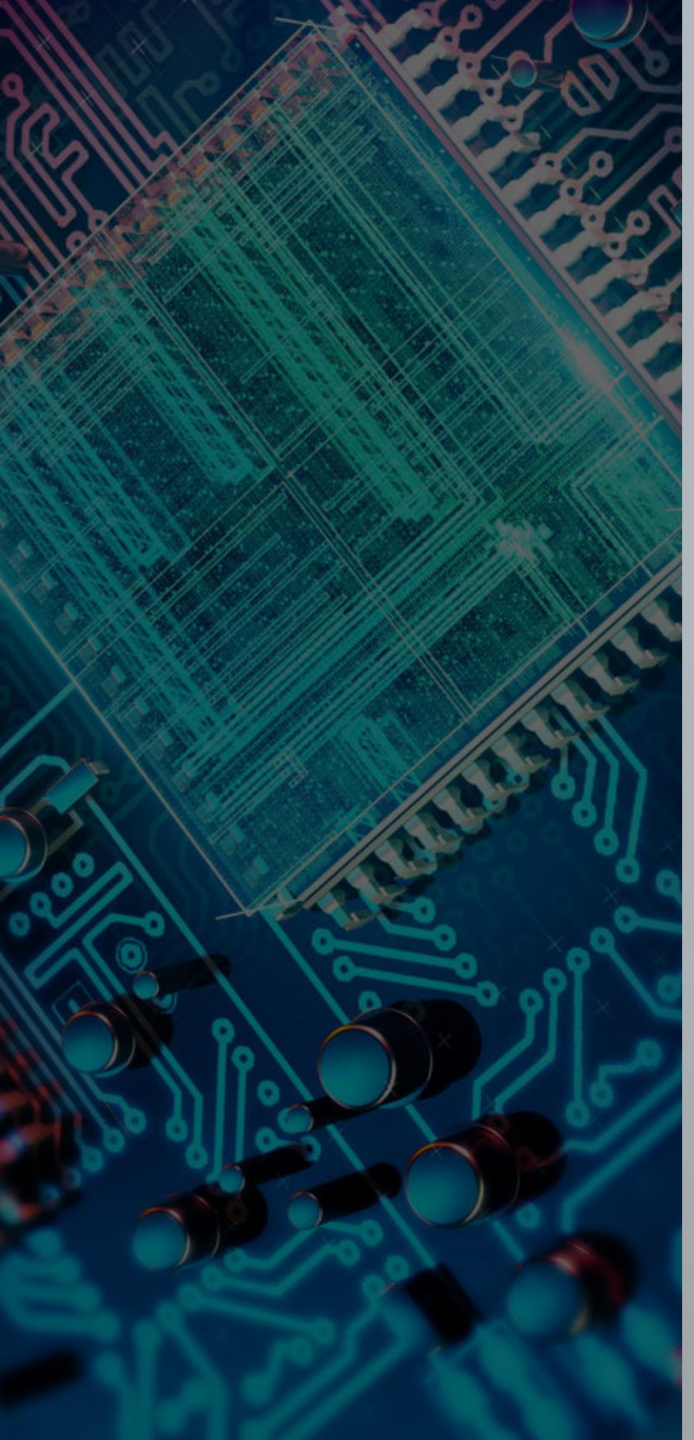
31–27	26–22	21–17	16–10	9–7	6–0
imm[11:7]	rs1	rs2	imm[6:0]	funct3	opcode
5	5	5	7	3	7
offset[11:7]	base	src	offset[6:0]	width	STORE
offset[11:7]	base	src	offset[6:0]	width	STORE-FP

- ✓ Transfer a value between the registers and memory.
- ✓ Loads are encoded in the I-type format. Stores are B-type.
- ✓ The effective byte address is obtained by adding register rs1 to the sign-extended immediate.
- ✓ Loads write to register rd a value in memory. Stores write to memory the value in register rs2.

Load & Store Instructions 2/3

- ✓ Instructions: LD, LW, LWU, LH, LHU, LB, LBU, SD, SW, SH, SB, FLD, FLW, FSD, FSW.
- ✓ L stands for load & S for store. F also indicates the floating point type of the register. The D, W, H, B letters correspond to the type of the register based on its length. U indicates zero or sign extending of the load instruction.
- ✓ Atomic Memory Operation Instructions (AMO): perform read-modify-write operations for multiprocessor synchronization and are encoded with an R-type instruction format.



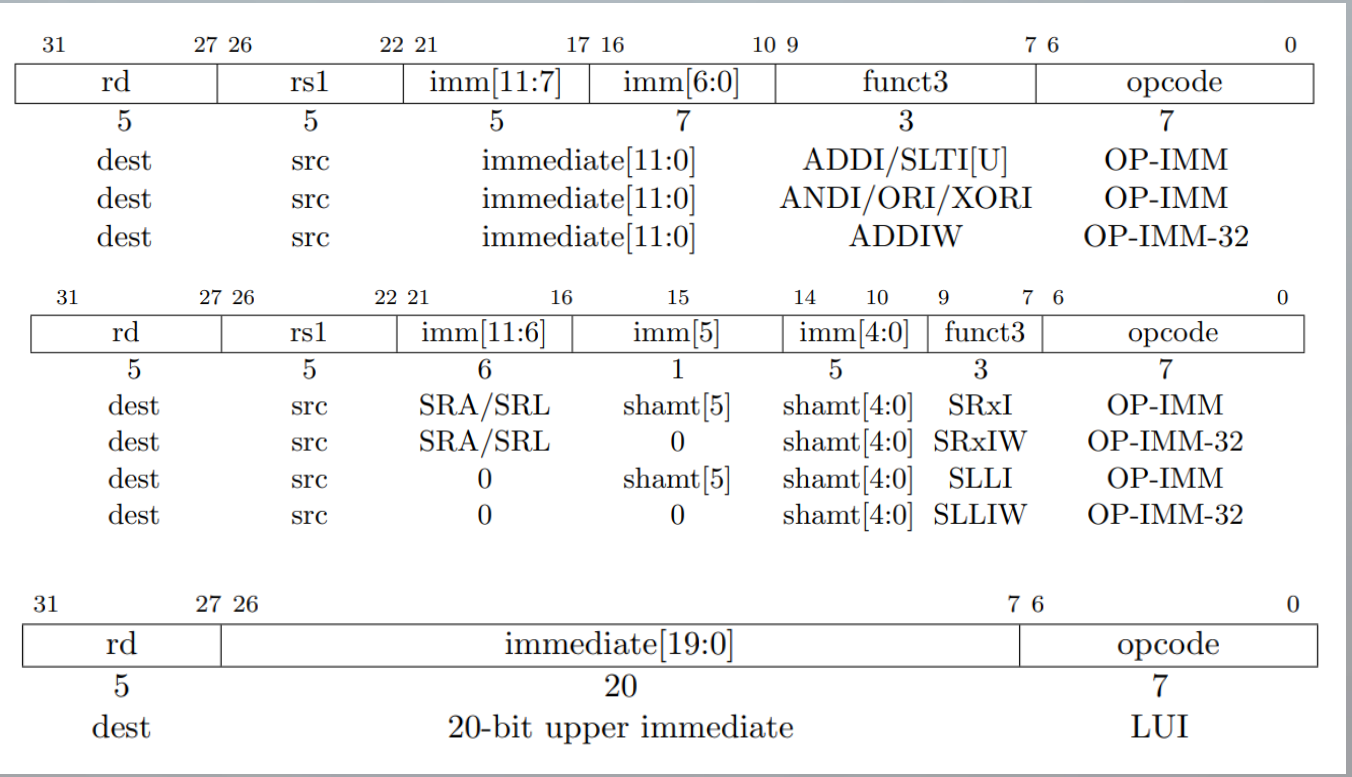


Load & Store Instructions 3/3

- ✓ AMO atomically load a data value from the address in rs1, place the value into register rd, apply a binary operator to the loaded value and the value in rs2, then store the result back to the address in rs1.
- ✓ The operations supported are integer add, logical AND, logical OR, swap, and signed and unsigned integer maximum and minimum.

Integer Computational Instructions 1/3

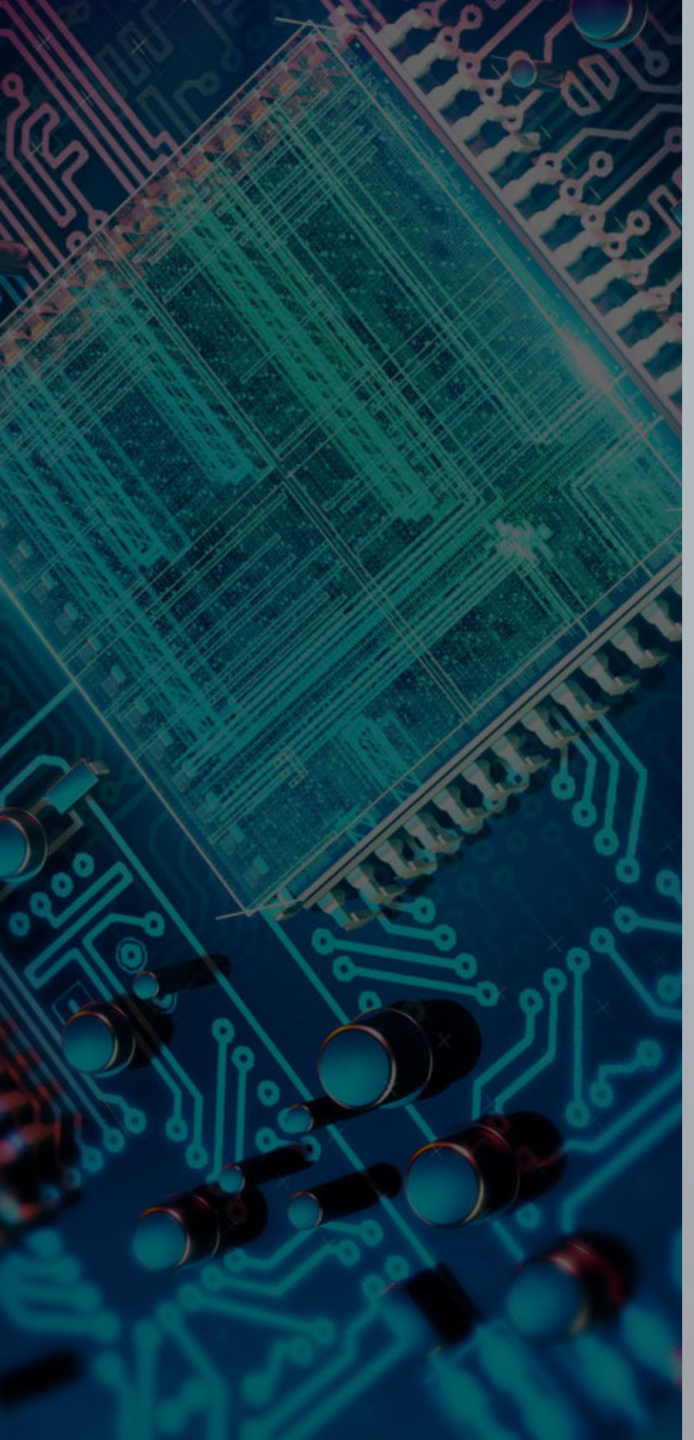
- ✓ Integer computational instructions are either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format.
- ✓ No integer computational instructions cause arithmetic traps.
- ✓ The destination is register rd for both register-immediate and register-register instructions.



31 27	26 22	21 17	16 10	9 7	6 0
rd	rs1	imm[11:7]	imm[6:0]	funct3	opcode
5	5	5	7	3	7
dest	src	immediate[11:0]		ADDI/SLTI[U]	OP-IMM
dest	src	immediate[11:0]		ANDI/ORI/XORI	OP-IMM
dest	src	immediate[11:0]		ADDIW	OP-IMM-32

31 27	26 22	21 16	15	14 10	9 7	6 0
rd	rs1	imm[11:6]	imm[5]	imm[4:0]	funct3	opcode
5	5	6	1	5	3	7
dest	src	SRA/SRL	shamt[5]	shamt[4:0]	SRxI	OP-IMM
dest	src	SRA/SRL	0	shamt[4:0]	SRxIW	OP-IMM-32
dest	src	0	shamt[5]	shamt[4:0]	SLLI	OP-IMM
dest	src	0	0	shamt[4:0]	SLLIW	OP-IMM-32

31 27	26 7	6 0
rd	immediate[19:0]	opcode
5	20	7
dest	20-bit upper immediate	LUI



Integer Computational Instructions 2/3

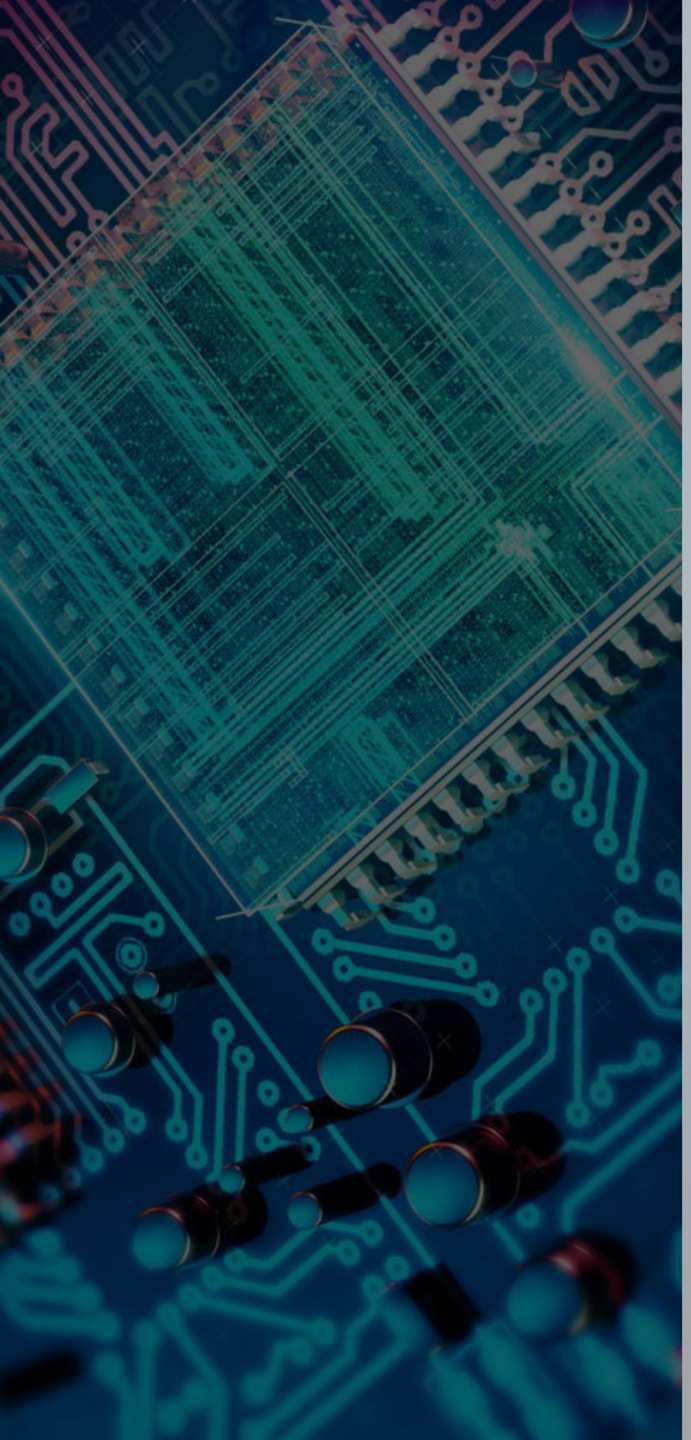
- ✓ Instructions: ADDI, ADDIW, SLTI, SLTIU, ANDI, ORI, XORI, AND, OR, XOR, SLLI, SRLI, SRAI, SLLIW, SRLIW, SRAIW, LUI.
- ✓ A stands for arithmetic (e.g. in SRAI the original sign bit is copied into the vacated upper bits).

Integer Computational Instructions 3/3

- ✓ In R-types also, all operations read the rs1 and rs2 registers as source operands and write the result into register rd. The funct field selects the type of operation.
- ✓ Different multiply and division so of course remainder operations are required for different register lengths & signed or unsigned conditions.

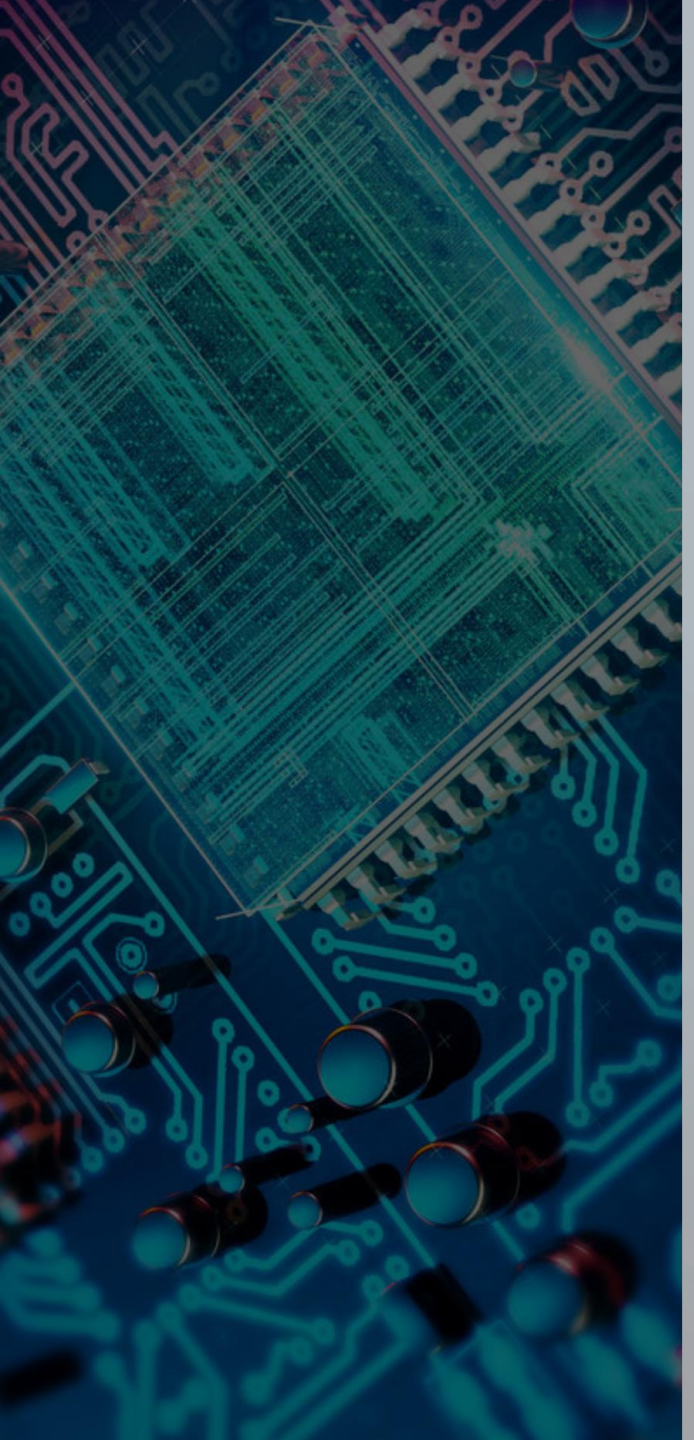
31 27	26 22	21 17	16 7	6 0
rd	rs1	rs2	funct10	opcode
5	5	5	10	7
dest	src1	src2	ADD/SUB/SLT/SLTU	OP
dest	src1	src2	AND/OR/XOR	OP
dest	src1	src2	SLL/SRL/SRA	OP
dest	src1	src2	ADDW/SUBW	OP-32
dest	src1	src2	SLLW/SRLW/SRAW	OP-32

31 27	26 22	21 17	16 7	6 0
rd	rs1	rs2	funct10	opcode
5	5	5	10	7
dest	src1	src2	MUL/MULH[[S]U]	OP
dest	dividend	divisor	DIV[U]/REM[U]	OP
dest	src1	src2	MUL[U]W	OP-32
dest	dividend	divisor	DIV[U]W/REM[U]W	OP-32



Control Transfer Instructions 1/4

- ✓ RISC-V provides two types of control transfer instructions: unconditional jumps and conditional branches
- ✓ Control transfer instructions in RISC-V do not have architecturally visible delay slots.
- ✓ Absolute jumps (J) and jump and link (JAL) instructions use the J-type format.
- ✓ The 25-bit jump target offset is sign-extended and shifted left one bit to form a byte offset, then added to the pc to form the jump target address.
- ✓ Jumps can therefore target a ±32 MB range.
- ✓ JAL stores the address of the instruction following the jump (pc+4) into register x1.

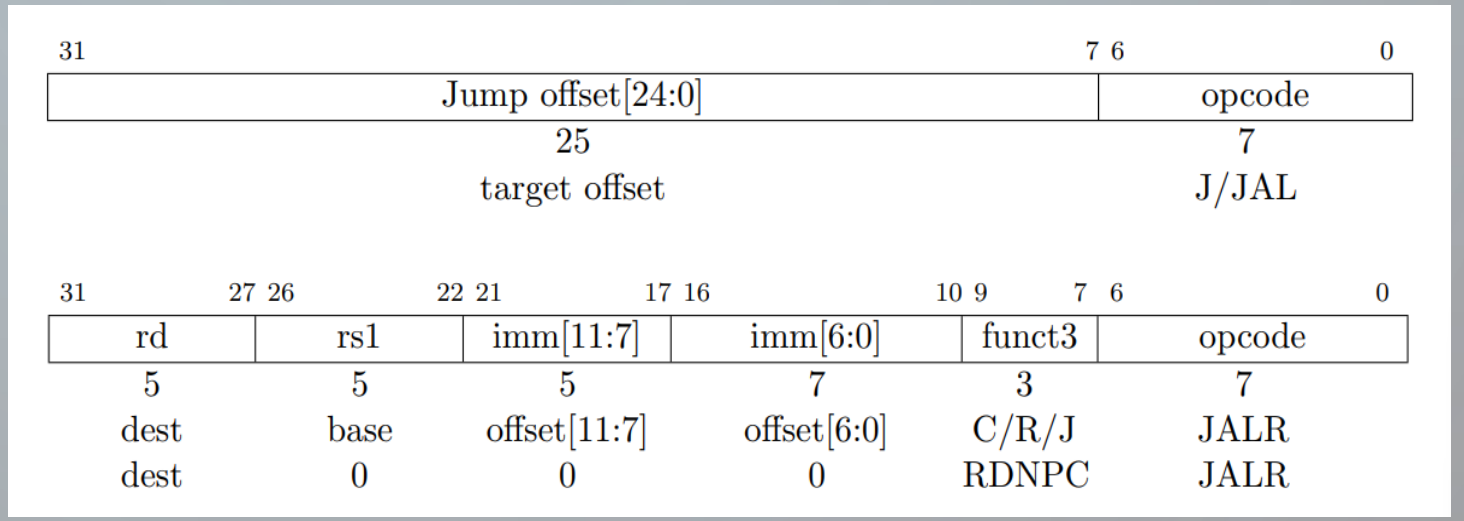


Control Transfer Instructions 2/4

- ✓ The indirect jump instruction JALR (jump and link register) uses the I-type encoding.
- ✓ It has three variants that are functionally identical but provide hints to the implementation:
- ✓ JALR.C is used to call subroutines;
- ✓ JALR.R is used to return from subroutines;
- ✓ JALR.J is used for indirect jumps.
- ✓ The target address is obtained by sign-extending the 12-bit immediate then adding it to the address contained in register rs1. The address of the instruction following the jump (pc+4) is written to register rd. Register x0 can be used as the destination if the result is not required.

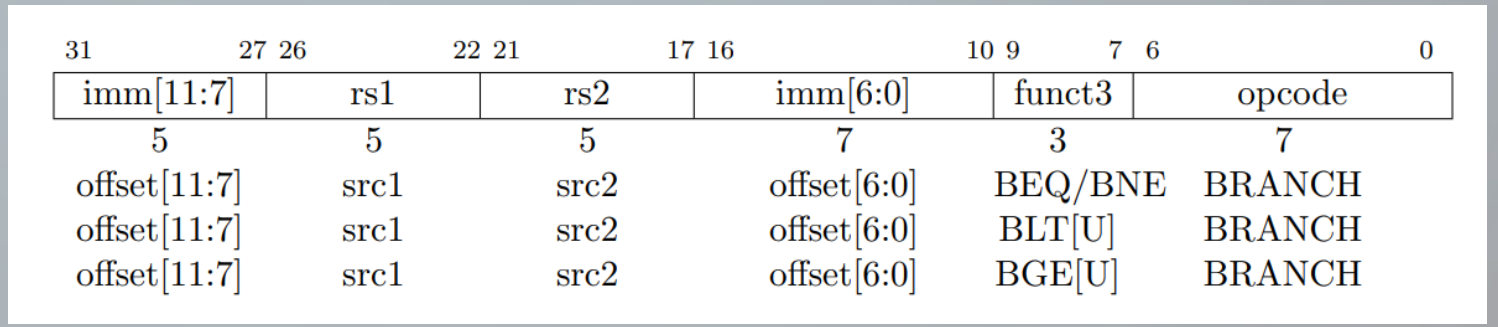
Control Transfer Instructions 3/4

- ✓ The JALR major opcode is also used to encode the RDNPC instruction, which writes the address of the following instruction (pc+4) to register rd without changing control flow.



Control Transfer Instructions 4/4

- ✓ All branch instructions use the B-type encoding.
- ✓ The 12-bit immediate is sign-extended, shiftedleft one bit, then added to the current pc to give the target address.

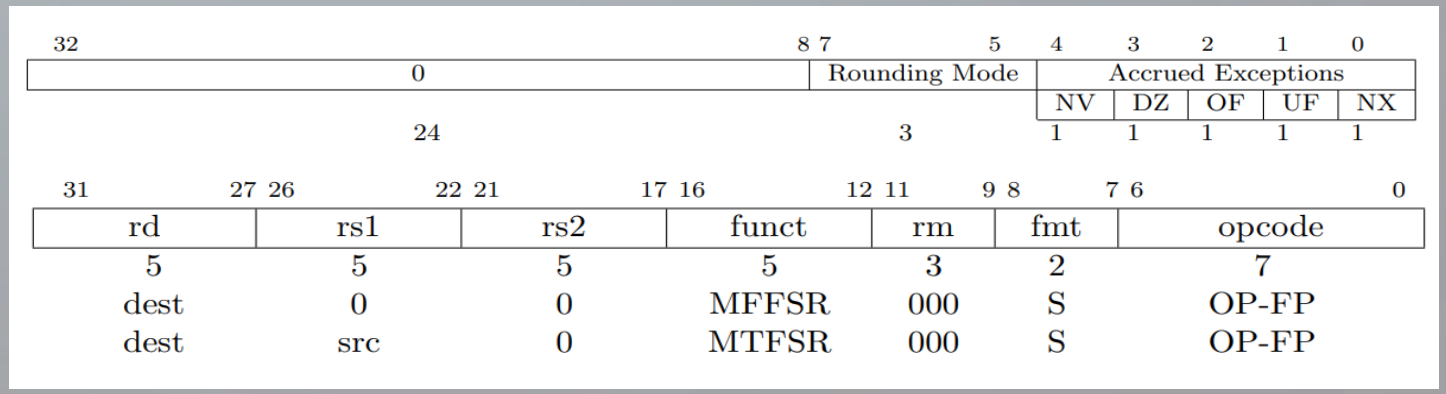


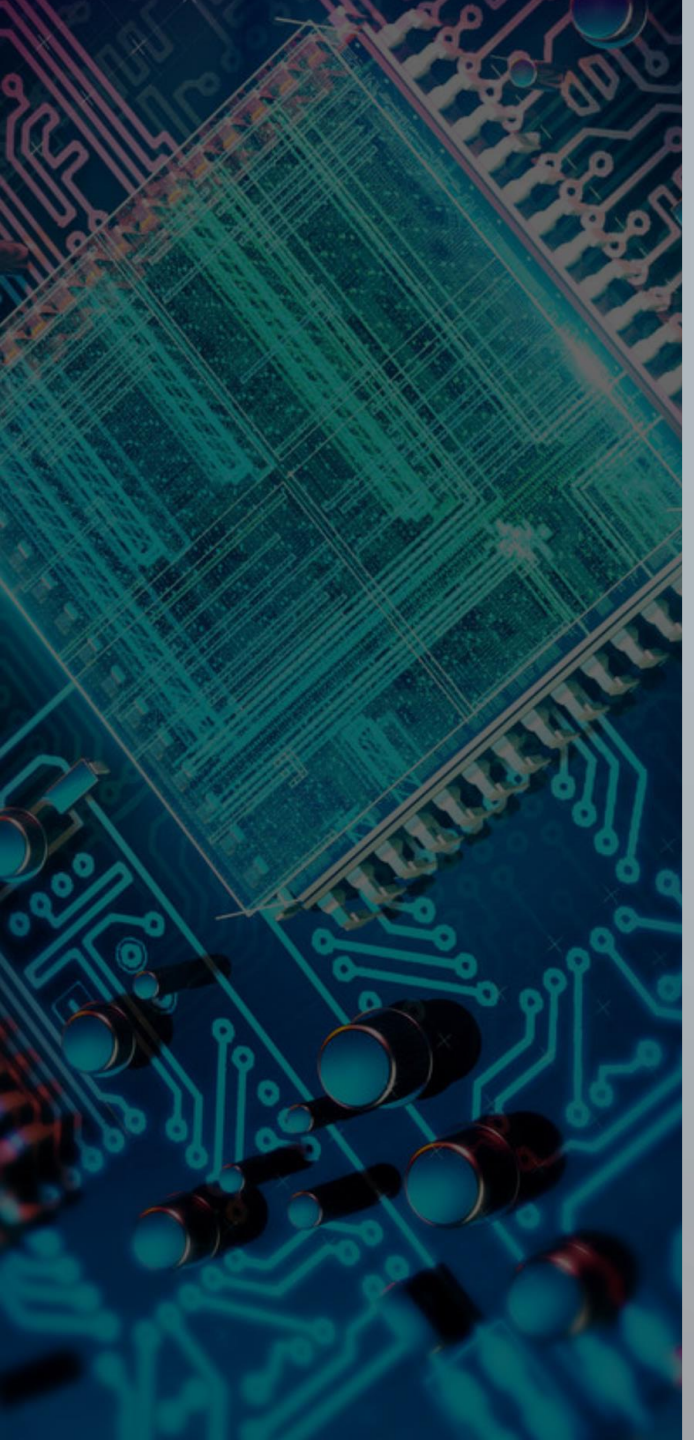
31 27	26 22	21 17	16 10	9 7	6 0
imm[11:7]	rs1	rs2	imm[6:0]	funct3	opcode
5	5	5	7	3	7
offset[11:7]	src1	src2	offset[6:0]	BEQ/BNE	BRANCH
offset[11:7]	src1	src2	offset[6:0]	BLT[U]	BRANCH
offset[11:7]	src1	src2	offset[6:0]	BGE[U]	BRANCH

- ✓ BEQ , BNE = Branch Equal, Branch Not Equal
- ✓ BLT and BLTU = Branch Less Than, Branch Less Than Unsigned
- ✓ BGE and BGEU = Branch Greater Than, Branch Greater Than Unsigned

Floating Point Instructions 1/5

- ✓ The base RISC-V ISA provides both single- and double-precision floating-point computational instructions compliant with the IEEE 754-2008 floating-point arithmetic standard.
- ✓ The fsr register is a 32-bit read/write register that selects the dynamic rounding mode for floating point arithmetic operations and holds the accrued exception flags.
- ✓ The fsr is read and written with the MFFSR and MTFSR floating-point instructions.





Floating Point Instructions

- ✓ Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in the Rounding Mode field and encoded as following.
- ✓ Some instructions are never affected by rounding mode, and should have their rm field set to RNE (000).

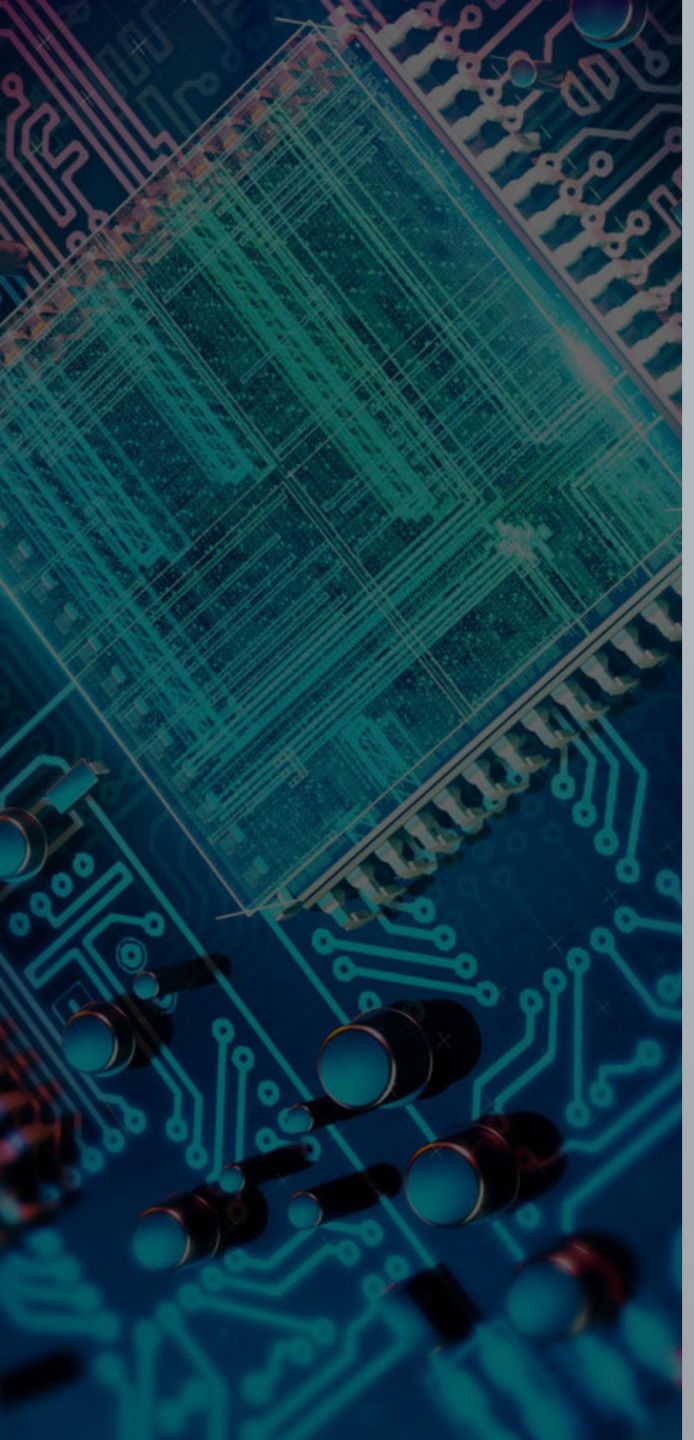
Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round toward Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101–111		*Invalid.*

- ✓ The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction.

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

Floating Point Instructions 3/5

- ✓ Floating-point arithmetic instructions with one or two source operands use the R-type format with the OP-FP major opcode.
- ✓ FADD.fmt, FSUB.fmt, FMUL.fmt, and FDIV.fmt perform floating-point addition, subtraction, multiplication, and division, respectively, between rs1 and rs2, writing the result to rd.
- ✓ FMIN.fmt and FMAX.fmt write, respectively, the smaller or larger of rs1 and rs2 to rd.
- ✓ FSQRT.fmt computes the square root of rs1 and writes the result to rd.
- ✓ The fmt field encodes the datatype of the operands and destination: S for single-precision or D for double-precision
- ✓ All floating-point operations that perform rounding can select the rounding mode statically using the rm field with the same encoding as shown previously. A value of 111 in the instruction's rm field selects the dynamic rounding mode held in the fsr.



Floating Point Instructions 4/5

- ✓ Floating-point fused multiply-add instructions are encoded as R4-type instructions.
- ✓ Multiply the values in rs1 and rs2, optionally negate the result, then add or subtract the value in rs3 to or from that result.
- ✓ FMADD.fmt computes rs1×rs2+rs3; FMSUB.fmt computes rs1×rs2-rs3; FNMSUB.fmt computes -(rs1×rs2-rs3); and FNMADD.fmt computes -(rs1×rs2+rs3).
- ✓ Other Instructions: Floating-point-to-integer and integer-to-floating-point conversion instructions, Floating-point to floating-point sign-injection instructions, Instructions to move bit patterns between the floating-point and fixed-point registers, floating-Point Compare Instructions.

Floating Point Instructions *5/5*

31–27	26–22	21–17	16–12	11–9	8–7	6–0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	src1	src2	FADD/FSUB	RM	S/D	OP-FP
dest	src1	src2	FMUL/FDIV	RM	S/D	OP-FP
dest	src1	src2	FMIN/FMAX	000	S/D	OP-FP
dest	src	0	FSQRT	RM	S/D	OP-FP

31–27	26–22	21–17	16–12	11–9	8–7	6–0
rd	rs1	rs2	rs3	rm	fmt	opcode
5	5	5	5	3	2	7
dest	src1	src2	src3	RM	S/D	F[N]MADD/F[N]MSUB

31–27	26–22	21–17	16–12	11–9	8–7	6–0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	src	0	FCVT.W[U].*fmt*	RM	S/D	OP-FP
dest	src	0	FCVT.*fmt*.W[U]	RM	S/D	OP-FP
dest	src	0	FCVT.L[U].*fmt*	RM	S/D	OP-FP
dest	src	0	FCVT.*fmt*.L[U]	RM	S/D	OP-FP

31–27	26–22	21–17	16–12	11–9	8–7	6–0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	src	0	FCVT.S.D	RM	S	OP-FP
dest	src	0	FCVT.D.S	RM	D	OP-FP

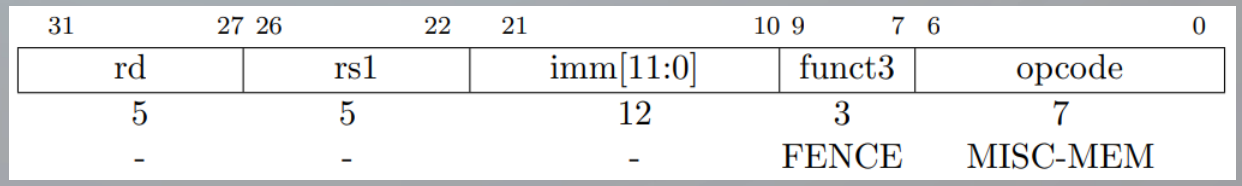
31–27	26–22	21–17	16–12	11–9	8–7	6–0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	src1	src2	FSGNJ[N]	000	S/D	OP-FP
dest	src1	src2	FSGNJX	000	S/D	OP-FP

31–27	26–22	21–17	16–12	11–9	8–7	6–0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	0	src	MFTX.*fmt*	000	S/D	OP-FP
dest	src	0	MXTF.*fmt*	000	S/D	OP-FP

31–27	26–22	21–17	16–12	11–9	8–7	6–0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	src1	src2	FEQ/FLT/FLE.*fmt*	000	S/D	OP-FP

Memory Model

- ✓ In the base RISC-V ISA, each hardware thread observes its own memory operations as if they executed sequentially in program order.
- ✓ RISC-V requires an explicit FENCE instruction to guarantee any specific ordering between memory operations from different threads.
- ✓ No other hardware thread can observe any memory operations following a FENCE before any memory operations preceding the FENCE.
- ✓ The FENCE instruction is used to order loads and stores as viewed by other hardware threads.

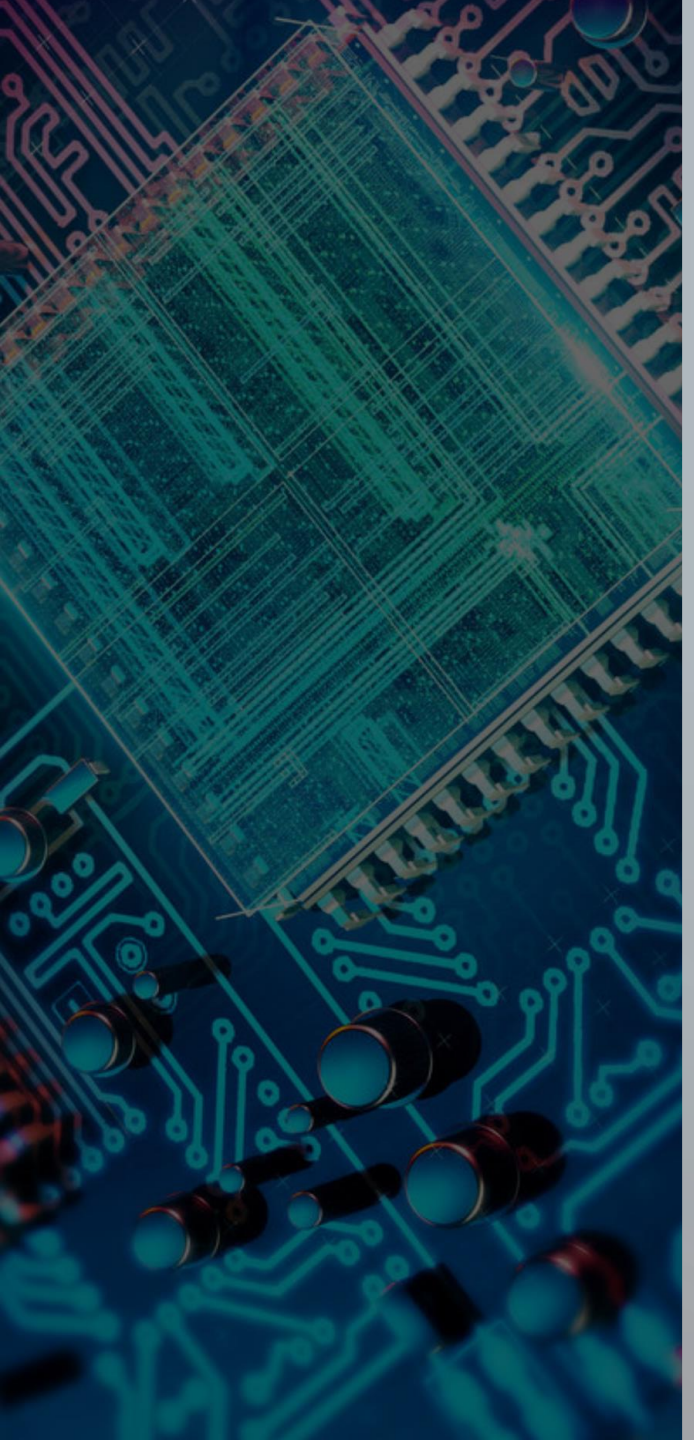


31 27	26 22	21 10	9 7	6 0
rd	rs1	imm[11:0]	funct3	opcode
5	5	12	3	7
-	-	-	FENCE	MISC-MEM

System Instructions

- ✓ SYSTEM instructions are used to access system functionality that might require privileged access.
- ✓ Encoded as R-type instructions
- ✓ SYSCALL: make a request to an operating system environment
- ✓ BREAK: used by debuggers to cause control to be transferred back to the debugging environment
- ✓ RDCYCLE: gives the number of clock cycles executed by the processor on which the hardware thread is running from an arbitrary start time in the past
- ✓ RDTIME: gives the real time that has passed from an arbitrary start time in the past
- ✓ RDINSTRET: gives the number of instructions retired by this hardware thread from some arbitrary start point in the past

00000	00000	00000	0000000	000	1110111	SYSCALL
00000	00000	00000	0000000	001	1110111	BREAK
rd	00000	00000	0000000	100	1110111	RDCYCLE rd
rd	00000	00000	0000001	100	1110111	RDTIME rd
rd	00000	00000	0000010	100	1110111	RDINSTRET rd



MEMORY MODEL

- RISC-V can be implemented with either big-endian or little-endian memory systems.
- Instructions are always stored in a little-endian sequence of parcels regardless of the memory system endianness.

Big Endian: the most significant byte (MSB) of a multi-byte value is stored at the lowest memory address, while the least significant byte (LSB) is stored at the highest memory address

Little Endian: the least significant byte (LSB) of a multi-byte value is stored at the lowest memory address, while the most significant byte (MSB) is stored at the highest memory address.

RISC-V Through Time

RISC-V is an open-source ISA that was developed at the University of California, Berkeley in 2010.

The project was initiated by researchers and faculty members at Berkeley's Parallel Computing Laboratory.

The development of RISC-V was driven by the desire to create a modern, open ISA that could be freely used for both academic research and commercial purposes.

The researchers believed that existing proprietary ISAs, such as ARM and x86, limited innovation and hindered collaboration.

The first version of the RISC-V ISA, known as the "RISC-V Classic", was designed to be simple and modular, making it easier to implement and customize for various applications and hardware platforms. It provided a small set of instructions, which allowed for efficient pipelining and execution of instructions.

As the RISC-V project gained traction and interest from both academia and industry, the RISC-V Foundation was established in 2015.

The foundation is a non-profit organization dedicated to managing and promoting the RISC-V architecture.

RISC-V Through Time

Since its inception, the RISC-V architecture has undergone several revisions and extensions to enhance its functionality and performance.
These extensions include support for floating-point operations, vector processing, and security features, among others.

The modular nature of RISC-V allows designers to choose the extensions that suit their specific requirements, making it highly customizable.

The open-source nature of RISC-V has fostered a vibrant ecosystem of hardware and software developers, leading to its rapid adoption in various domains.

The development of RISC-V continues to evolve, with ongoing research and industry collaborations driving its growth.

What is Open-Source?

The open-source nature of RISC-V refers to the fact that the architecture's specifications and design are publicly available and freely accessible. This openness brings several advantages

Open-Source Advantages

- Education and Research
- Collaboration & Innovation
- Industry Adoption and Standardization
- Reduced Dependency on Proprietary ISAs
- Security and Transparency
- Customization & Specialization

What is Modular Design?

In a modular design, each module is designed to be independent and self-contained, with well-defined interfaces that allow for communication and interaction with other modules. This allows for flexibility and scalability, as modules can be added, removed, or replaced without affecting the overall system.

The RISC-V ISA is designed with a modular approach, allowing for flexibility and customization based on specific application requirements. The modular design of RISC-V encompasses a base integer instruction set and optional extensions that provide additional functionality. The three commonly used base integer instruction sets are:

RV32I

- Designed for 32-bit implementations, where the register size and memory addressing are 32 bits.
- Includes 32 general-purpose registers (x0 to x31), where x0 is a constant zero register.
- Provides a set of integer arithmetic and logical operations, load and store instructions, control transfer instructions (branches and jumps), and system-level instructions.

RV64I

- An extension of RV32I and is designed for 64-bit implementations.
- Extends the register size and memory addressing to 64 bits, providing more addressable memory space.
- Includes the same instructions as RV32I but with wider data paths, allowing for more extensive computations and larger data manipulations.

RV128I

- Further extends the register size and memory addressing to 128 bits Catering to systems that require extremely large address spaces or deal with specialized data types
- Includes the same set of instructions as RV64I but with wider data paths and larger register sizes, enabling operations on 128-bit data types.

In addition to the base integer instruction sets, RISC-V allows for optional extensions that provide additional capabilities, such as floating-point operations (F), atomic operations (A), vector processing (V), and cryptography (C), among others. These extensions can be selectively added to the base integer instruction sets to meet specific application requirements, making RISC-V highly customizable and adaptable to various domains and use cases.

Multiply/Divide Extension (M)

- The M extension adds hardware support for integer multiplication and division operations.
- Includes instructions for signed and unsigned multiplication, division, and remainder operations.
- Particularly useful in applications that involve complex arithmetic computations, such as digital signal processing (DSP), graphics, and cryptography.

Atomic Operations Extension (A)

- The A extension introduces atomic memory operations, which are crucial for multi-threaded or concurrent programming.
- Atomic operations ensure that memory accesses are performed atomically, without interference from other threads or processes.
- Includes instructions for atomic read-modify-write operations, such as atomic compare-and-swap (CAS) and atomic load-linked/store-conditional (LL/SC).

Floating-Point Extensions (F and D)

- Add support for single-precision (F) and double-precision (D) floating-point operations, respectively.
- Enable efficient floating-point arithmetic and allow RISC-V processors to perform calculations on real numbers with decimal points.
- Vital for scientific computing, numerical simulations, graphics rendering, and other applications that require high-precision numerical computations.

Which companies adopt RISC-V?

SiFive offer customizable processor IP solutions based on the RISC-V architecture, enabling companies to design and implement their own RISC-V-based chips.

Western Digital, a major storage technology company, have developed RISC-V-based controllers for flash memory and SSDs, showcasing the applicability of RISC-V in storage systems.

Alibaba, one of the largest e-commerce and cloud computing companies globally, have developed their own RISC-V-based processor, Xuantie, to power their data centers and cloud computing infrastructure.

NVIDIA, have utilized RISC-V in their Deep Learning Accelerator (NVDLA) project, which is an open architecture for deep learning inference.

Google has made investments in the ecosystem. They have explored the use of RISC-V in various projects and initiatives, including hardware security and accelerators for machine learning.

Huawei, a leading telecommunications and consumer electronics company, has joined the RISC-V Foundation and actively participated in the development of the RISC-V ecosystem.

OpenAI, the organization behind ChatGPT, has also utilized RISC-V in its AI research. They have employed RISC-V-based processors in their infrastructure for high-performance computing and AI workloads.

ARM OR RISC-V

ARM

Instruction Set

Has a more extensive instruction set with a larger number of instructions and addressing Modes.

RISC-V

Instruction Set

Focuses on a minimalistic, modular, and extensible instruction set. This allows
RISC-V designers to implement only the instructions they need for their specific application, potentially resulting in more efficient and tailored processor designs.

Customization

RISC-V's open-source nature and modular ISA design allow for more customization and flexibility in processor design compared to ARM. Companies and researchers can develop custom extensions or create application-specific processor designs that better suit their needs, without being constrained by the licensing terms of a proprietary ISA.

ARM

RISC-V

Area & Cost

RISC-V processors tend to have smaller die sizes and lower production costs than ARM processors, making them more suitable for low-power and cost-sensitive applications.

Clock Speed

ARM processors typically have higher clock speeds than RISC-V processors, which can provide better performance for certain applications, such as high-performance computing or data-intensive tasks.

Power Consumption

RISC-V processors generally consume less power than ARM processors due to their simpler architecture and the ability to customise the ISA to match specific application requirements.

Memory Management Unit (MMU)

ARM processors have advanced memory management units which is not the case in RISC-V processors. This could be a limitation for some specific applications.

ARM

Licensing model

Companies that want to use ARM-based processor cores in their products need to obtain a license from ARM Holdings.

Ecosystem

Has a well-established ecosystem, with a wide range of processor cores available for various performance levels and applications, from low-power microcontrollers to high-performance application processors. ARM-based processors are used in numerous devices, such as smartphones, tablets, laptops, and embedded systems.

RISC-V

Licensing model

Anyone can use, modify, and develop their own processor designs based on the RISC-V ISA without the need for a license or royalties.

Ecosystem

A relatively new ISA, and its ecosystem is still growing. However, it has gained significant traction and support from various companies and organizations, leading to an increasing number of RISC-V-based processor designs and products.

ARM

Security

ARM processors have a proven track record in providing security features such as memory protection and cryptographic acceleration.

RISC-V

Security

RISC-V processors are still in the process of building similar features.

Usages

The use of RISC-V is increasing. It has found applications in embedded systems, Internet of Things (IoT) devices, high-performance computing, data centers, and even supercomputers. From microcontrollers to Academic research is using RISC-V.

REFERENCES

http://www.slideshare.net/

New Bing

chat.openai.com

https://en.wikipedia.org/wiki/Instruction_set_architecture

https://www.slideshare.net/gane309/risc-processors

https://www.slideshare.net/TusharSwami1/risc-reduced-instruction-set-computing?from_search=1

https://www.slideshare.net/MathivananNatarajan/arm-instruction-set-60665439

03-arm_overview.pdf in CW of the Embedded Systems Course

https://en.wikichip.org/wiki/arm_holdings/cortex

https://www.slideshare.net/poedja/introduction-to-arm?from_search=11

https://www.slideshare.net/PrashantSingh056/arm-processor-19617463

https://www.slideshare.net/black_pearl/arm-processor?from_search=0

https://people.eecs.berkeley.edu/~krste/papers/EECS-2011-62.pdf

THE END