Programming Microprocessor – Instruction Set Architecture I

CS2053 Computer Architecture

Computer Science & Engineering

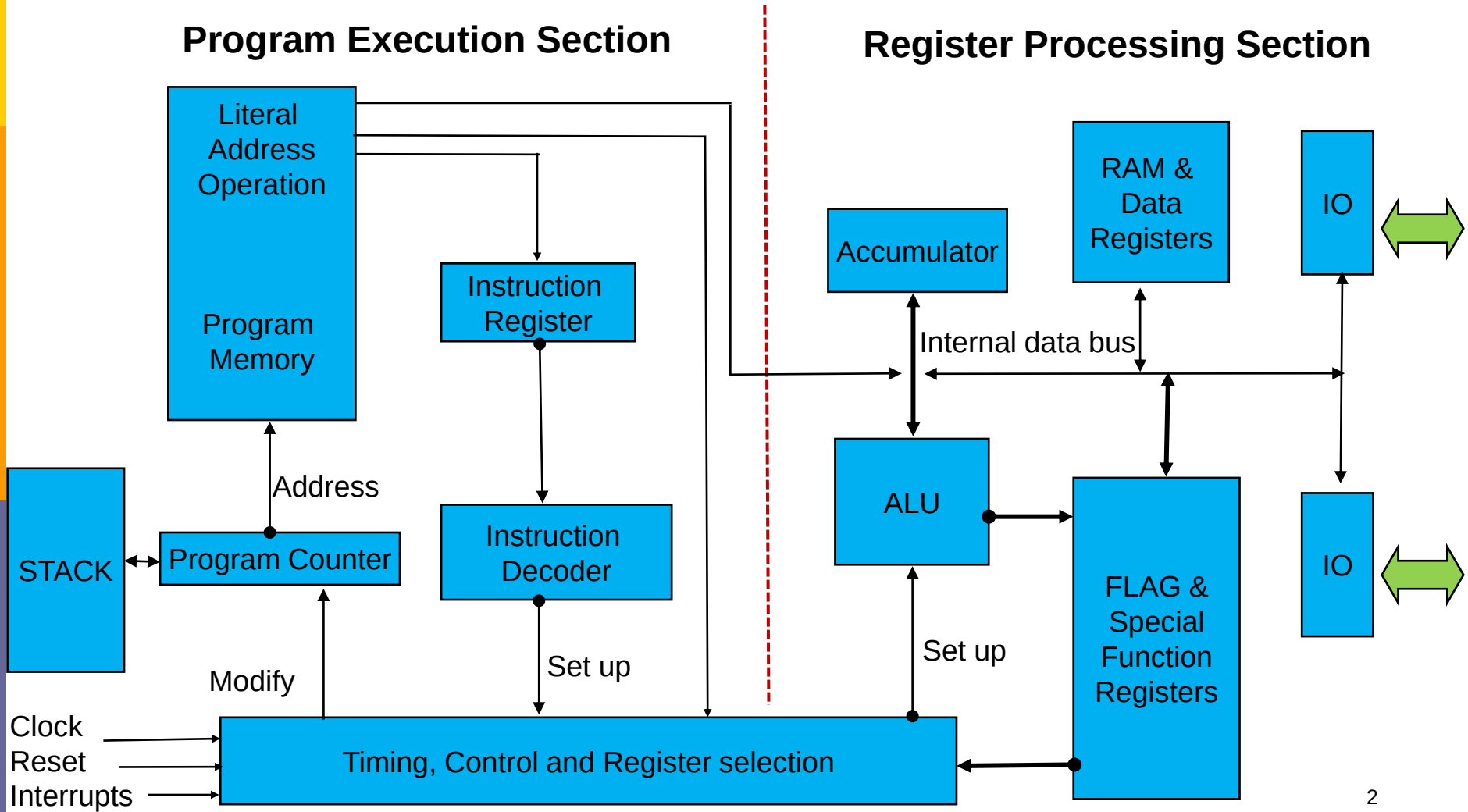
University of Moratuwa

Sulochana Sooriyaarachchi

Chathuranga Hettiarachchi

Slides adopted from Dr.Dilum Bandara

Blocks of a Microprocessor



Source: Makis Malliris & Sabir Ghauri, UWE

So many types of devices!

Different demands

- Can you run all your desktop/laptop computer in the mobile phone as well?If not, why?
- Which battery lasts longer? Laptop battery or the mobile phone battery? Why?
- What computing devices are specialized and what are meant for general purpose tasks?

Major concerns of processor design

Available memory
Power consumption
Heat dissipation
Fabrication technology
Hardware complexity/ dedicated hardware
Connectivity
How to enable software development?

Levels of abstraction

What do you mean by x86 and x64?

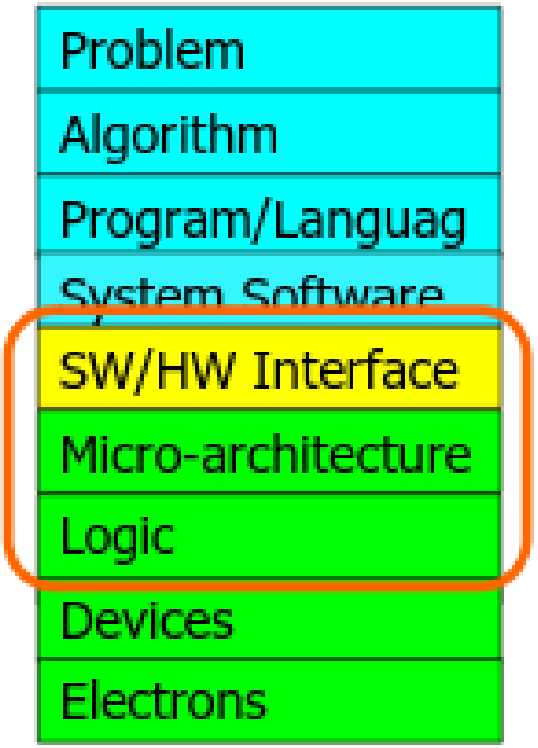
What are the other alternatives you know of?

We need programmable computers

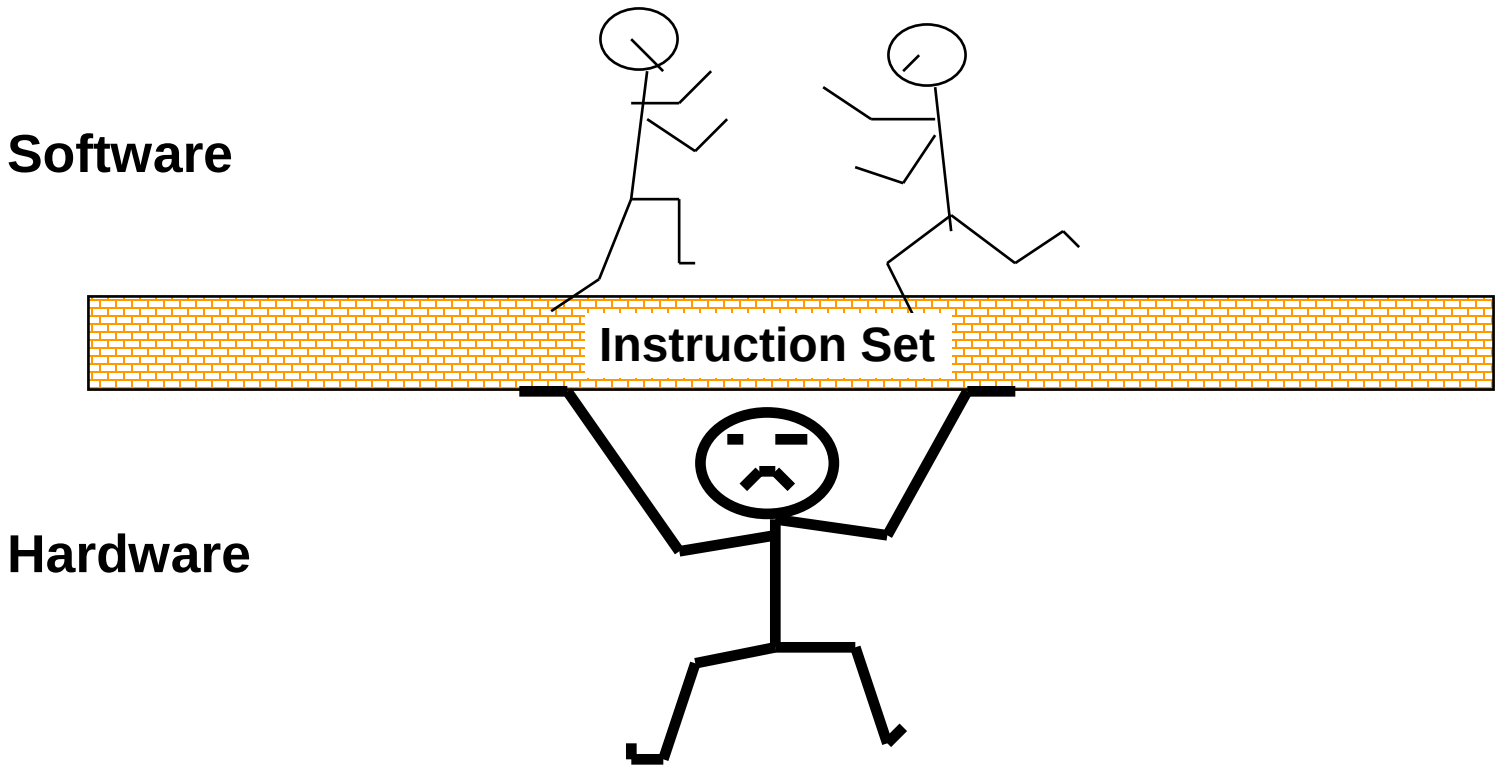
Computer is just a collection of hardware modules without a program.

How to make all the hardware features available to programmers?

- More dedicated hardware modules => More instructions
- Less hardware modules => Less instructions



Instruction Set Architecture (ISA)



Source: Computer Architecture: A Quantitative Approach, J. L. Hennessy & D. A. Patterson, 3rd Edition.

ISA (Cont.)

- Part of computer architecture related to programming
- Include native data types, instructions, registers, addressing modes, memory architecture, interrupt & exception handling, & external I/O
  - e.g., Registers x0 – x31 , PC, Control/Status registers
  - e.g., Instructions: li, addi, andi, beq
- ISA specifies the set of opcodes (i.e., machine language instructions) & native commands implemented by a particular processor

CISC ISA

When there is a large number of dedicated hardware co-processors/ modules, it needs to use more instructions .

- CISC : Complex Instruction Set Computer
- Consumes more power, because large number of modules are active
- Large number of instructions for dedicated hardware operations
- Difficult to program. Too many ways to choose from!

RISC ISA

When we have a minimal set of hardware we can use a simple set of instructions.

- RISC : Reduced Instruction Set Computer
- Consumes less power, hardware is simple and minimal
- Only a handful of instructions
- Easy to program

Well Known ISAs

- x86
  - Based on Intel 8086 CPU in 1978
  - Intel family, also followed by AMD
  - X86-64
    - 64-bit extensions
    - Proposed by AMD, also followed by Intel
- ARM
  - 32-bit & 64-bit
  - Initially by Acorn RISC Machine (ARM)
  - ARM Holding
- MIPS
  - 32-bit & 64-bit
  - By Microprocessor without Interlocked Pipeline Stages (MIPS) Technologies

Well Known ISAs (Cont.)

- SPARC
  - 32-bit & 64-bit
  - By Sun Microsystems, today Oracle
- PIC
  - 8-bit to 32-bit
  - By Microchip
- Z80
  - 8-bit
  - By Zilog in 1976
- Many extensions
  - Intel – MMX, SSE, SSE2, AVX
  - AMD – 3D Now!

What is Instruction Set Architecture (ISA)?

A layer of abstraction between hardware implementations and the what the processor can do.

Processor developer may choose to keep the hardware implementation details a secret.

Argue:

List of available assembly instructions and register/ memory map in a processor provides ‘everything’ necessary to get work done using a processor.

(OR) Programmer need to know some hardware aspects as well

Moodle Activity

A Good ISA

- Lasts through many implementations
  - Portability, compatibility
- Used in many different ways
  - Servers, desktop, laptop, mobile, tablet
- Provides convenient functions to higher layer
- Permit efficient implementation at lower layer

Introduction to RISC-V

- RISC-V is an open standard, in fact, the specification is public domain, and it has been managed since 2015 by the **RISC-V Foundation**, now called **RISC-V International**, a nonprofit organization promoting the development of hardware and software for RISC-V architectures

- Supported by 200 key players from research, academia, and industry, including Microchip, NXP, Samsung, Qualcomm, Micron, Google, Alibaba, Hitachi, Nvidia, Huawei, Western Digital, ETH Zurich, KU Leuven, UNLV, and UCM

- Modular rather than Incremental
  - x86 started with 80 instructions, and now has over 1300 (with 3600 machine opcodes)

Open Standard Specification

https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

RISC-V Base and Extensions

RISC-V Base and Extensions

Mnemonic	Description	Insn. Count
I	Base architecture	51
M	Integer multiply/divide	13
A	Atomic operations	22
F	Single-precision floating point	30
D	Double-precision floating point	32
C	Compressed instructions	36

FIGURE 2.38 The RISC-V instruction set architecture is divided into the base ISA, named I, and five standard extensions, M, A, F, D, and C.

Instruction Variants and Modules

- Base options/variants

  32 General Purpose Registers with ISA word size

  RV32I RV64I and RV128I

  RV32E (for embedded devices, only 16 registers available)

- Modular rather than Incremental
  - M Multiply/Division
  - A Atomic extensions
  - F Floating point
  - D Double precision Floating Point

  } RV32IMAFD or RV32IG (G - General, same as MAFD)

  - B Bit manipulation extension
  - C Compressed extension (More efficient encoding of instructions, reducing code size)
  - N User level interrupts
  - Proprietary extensions can be developed for your own applications.

- For example, a 64-bit RISC-V implementation, including all four general ISA extensions plus *Bit Manipulation* and *User Level Interrupts*, is referred to as an RV64GBN ISA
  - Can find out what modules are available by observing the “misa” status register

Example Implementations

- SweRV EH1 , SweRV EH2 , SweRV EL2
  - Some open-sourced hardware implementations by the company Western Digital
  - Apache 2.0 License : Use as-is or modified for free.
  - Can use in FPGAs since HDL code/IP is available
  - Modify and test
- Many companies have hardware boards with different RISC V processors as well
  - SiFive (Example: Red-V board)
  - Microchip

See for list of development boards: https://riscv.org/exchanges/boards/

Encoding Instructions

- Various instruction types
- Limited word size for registers, addresses, and instructions
  - Consider 32bit words in RV32I
  - All the instructions are 32bits
  - Example : If we need to load an immediate value to 32-bit register, how to fit all opcode and operands within 32-bit instruction?
    - Work with small numbers
    - Make compromises

Instruction Formats (6 Types)

- R-Format: instructions using 3 ***register*** inputs
  - `add`, `xor`, `mul` —arithmetic/logical ops
- I-Format: instructions with ***immediates***, loads
  - `addi`, `lw`, `jalr`, `slli`
- S-Format: ***store*** instructions: `sw`, `sb`
  - SB-Format: ***branch*** instructions: `beq`, `bge`
- U-Format: instructions with ***upper*** immediates
  - `lui`, `auipc` —upper immediate is 20-bits
  - UJ-Format: ***jump*** instructions: `jal`

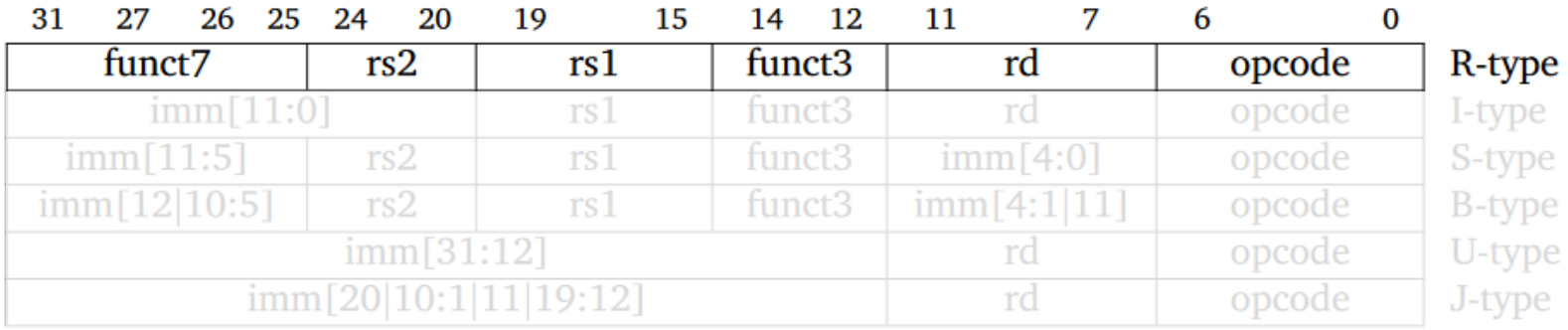
Instruction Formats(Types)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

- opcode 7bits
- funct 3 : 3 bit function
- funct 7 : 7 bit function
- rs1, rs2 : two source registers (5 bits each)
- rd : destination register (5 bits)

How many registers can be identified with 5 bits?

Instruction Format-Register Type

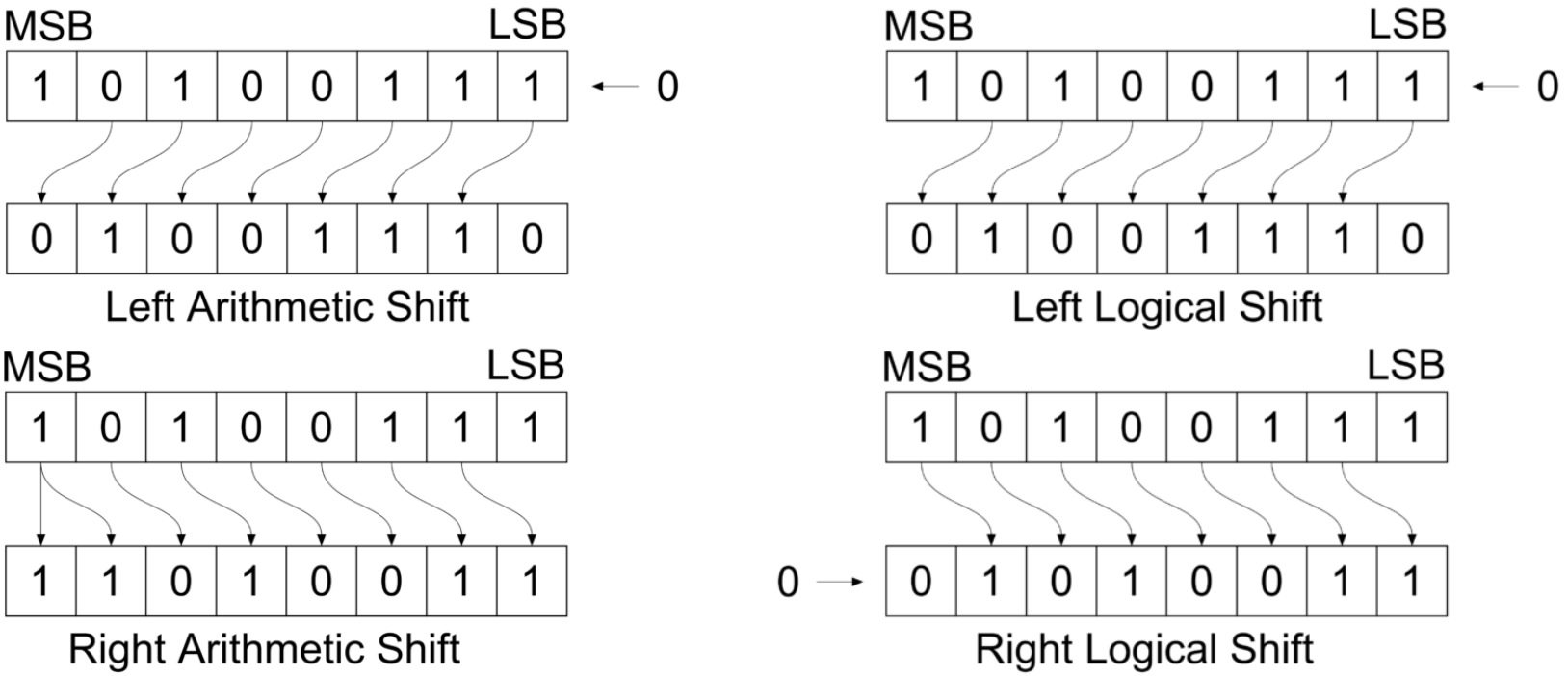


RV32I Base Integer Instructions (Register Type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

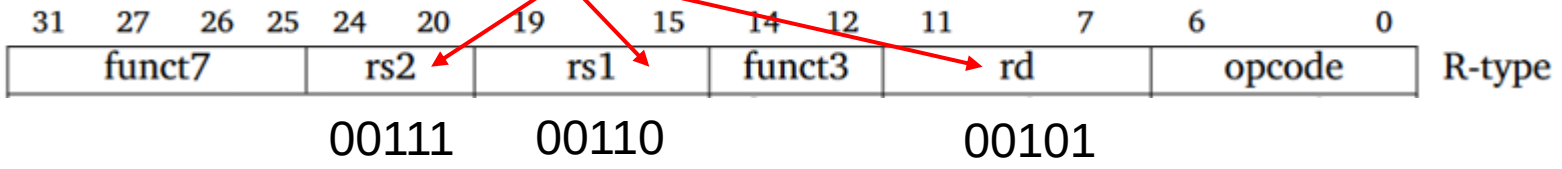
Logical Shift vs Arithmetic Shift



- **Right** Arithmetic shift preserve sign bit, whereas **Right** Logical shift can not preserve sign bit.
- Arithmetic shift perform multiplication and division operation, whereas Logical shift perform only multiplication operation.
- Arithmetic shift is used for signed interpretation, whereas Logical shift is used for unsigned interpretation.

Register Type Instruction Example

RISCV Instruction: `add x5,x6,x7`

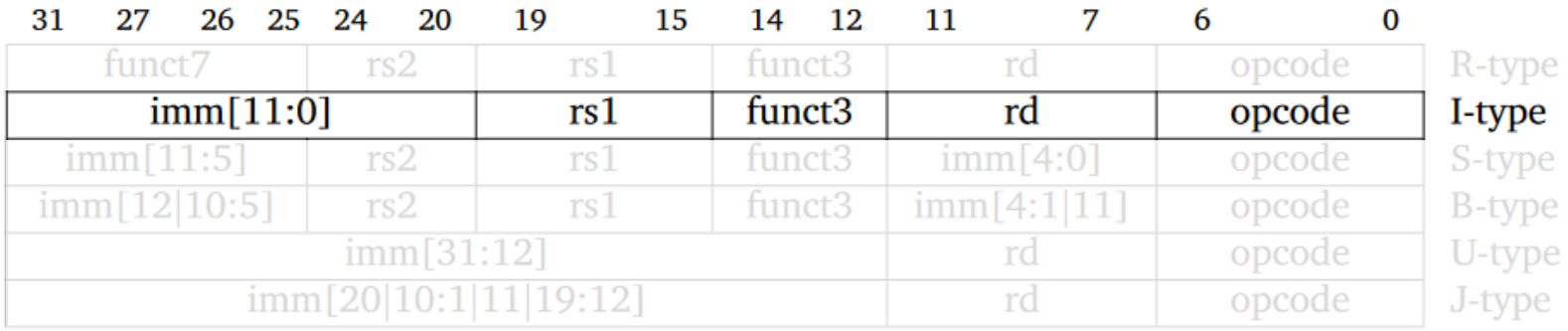


funct7	rs2	rs1	funct3	rd	opcode
0 0 0 0 0 0 0			0 0 0		0 1 1 0 0 1 1
	0 0 1 1 1	0 0 1 1 0		0 0 1 0 1	
0 0 0 0 0 0 0	0 0 1 1 1	0 0 1 1 0	0 0 0	0 0 1 0 1	0 1 1 0 0 1 1
0	0	7	3	0	2	B	3

Machine Instruction: **0x007302B3**

- Manipulating values in registers can be done
- Yet, how to get any value to registers in first place?
  - Immediate values are required.
    - Can we use rs1 or rs2 fields for am immediate value parameter?
      - rs1 and rs2 are just 5 bits each
      - Not enough to hold sufficiently large values
  - Dedicate both funct7 and rs2 for representing bigger immediate values
    - Immediate type instructions

Instruction Format-Immediate Type

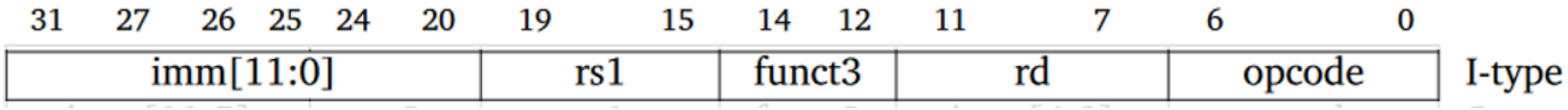


RV32I Base Integer Instructions (Immediate Type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends

Instruction	Format	immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
ld (load doubleword)	I	address	reg	011	reg	0000011

Immediate Type Example



RISCV Instruction: `addi x5,x6,0x04`

imm[11:0]	rs1	funct3	rd	opcode
		0 0 0		0 0 1 0 0 1 1
0 0 0 0 0 0 0 0 0 1 0 0	0 0 1 1 0		0 0 1 0 1	
0 0 0 0 0 0 0 0 0 1 0 0	0 0 1 1 0	0 0 0	0 0 1 0 1	0 0 1 0 0 1 1
0	0 4	3	0	2 9 3

Machine Instruction: **0x00430293**

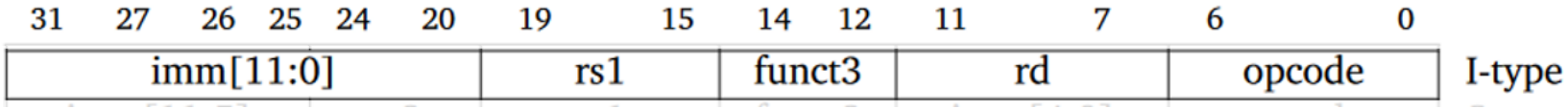
Immediate Values

- `immediate`(12): 12-bit number
  – All computations done in words, so 12-bit immediate must be extended to 32 bits
  – always sign-extended to 32-bits before use in an arithmetic operation

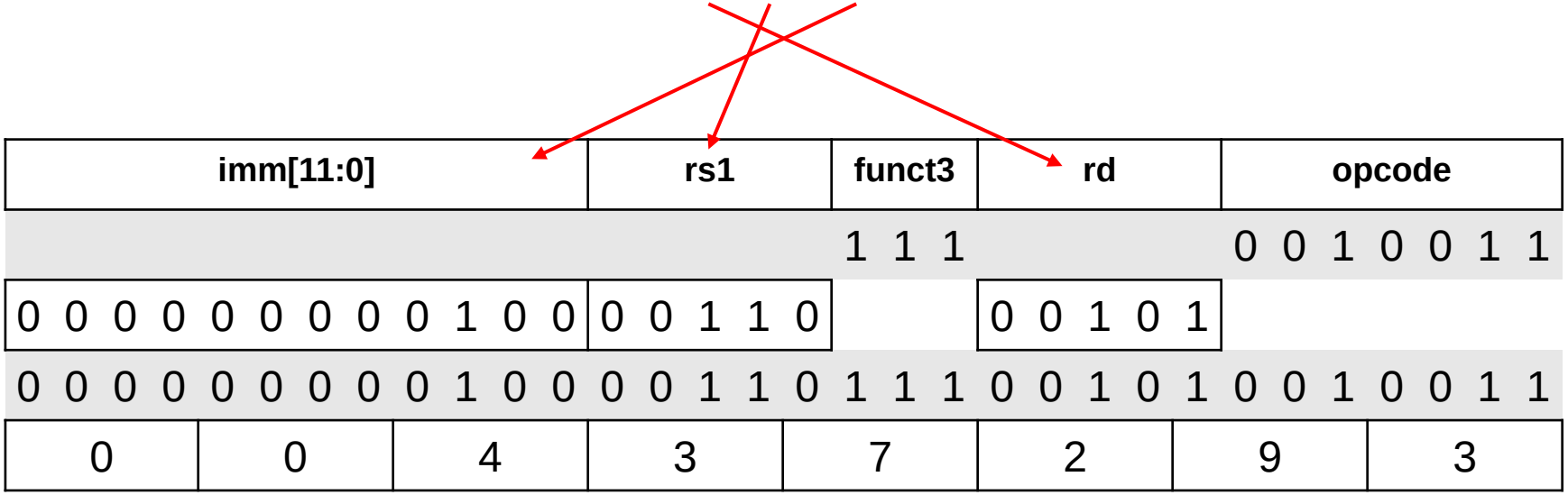
- Q: What is the largest Immediate value you can use with addi instruction?

2^{12} Range of Two's complement. $[-2^{11} , +2^{11})$
Immediate value should be between -2048 and **2047**

Immediate Type Example 2



RISCV Instruction: `andi x5, x6, 4`



Machine Instruction: **0x00437293**

Exercise in Moodle

- Assume x6 Register holds the value **0xFFFFFFFF**. The computer represent numbers in two's complement format.

- If you execute **andi x5,x6,-4** instruction, what will be the binary format of the following?
  - (a) Immediate value (12bits) encoded into the instruction?
  - (b) Value at x5 at the end of the instruction execution?

RISC –V Registers

32 32-bit registers

Name	Register Number	Use
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	Temporary variables
s0/fp	x8	Saved variable / Frame pointer
s1	x9	Saved variable
a0-1	x10-11	Function arguments / Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved variables
t3-6	x28-31	Temporary variables

RISC V Instruction Set

Common RISC-V Assembly Instructions & Pseudoinstructions

RISC-V Assembly	Description	Operation
`add s0, s1, s2`	Add	s0 = s1 + s2
`sub s0, s1, s2`	Subtract	s0 = s1 - s2
`addi t3, t1, -10`	Add immediate	t3 = t1 – 10
`mul t0, t2, t3`	32-bit multiply	t0 = t2 * t3
`div s9, t5, t6`	Division	t9 = t5 / t6
`rem s4, s1, s2`	Remainder	s4 = s1 % s2
`and t0, t1, t2`	Bit-wise AND	t0 = t1 & t2
`or t0, t1, t5`	Bit-wise OR	t0 = t1 | t5
`xor s3, s4, s5`	Bit-wise XOR	s3 = s4 ^ s5
`andi t1, t2, 0xFFB`	Bit-wise AND immediate	t1 = t2 & 0xFFFFFFFB
`ori t0, t1, 0x2C`	Bit-wise OR immediate	t0 = t1 | 0x2C
`xori s3, s4, 0xABC`	Bit-wise XOR immediate	s3 = s4 ^ 0xFFFFFABC
`sll t0, t1, t2`	Shift left logical	t0 = t1 << t2
`srl t0, t1, t5`	Shift right logical	t0 = t1 >> t5
`sra s3, s4, s5`	Shift right arithmetic	s3 = s4 >>> s5
`slli t1, t2, 30`	Shift left logical immediate	t1 = t2 << 30
`srli t0, t1, 5`	Shift right logical immediate	t0 = t1 >> 5
`srai s3, s4, 31`	Shift right arithmetic immediate	s3 = s4 >>> 31

RISC V Instruction Set

Common RISC-V Assembly Instructions & Pseudoinstructions (continued)

RISC-V Assembly	Description	Operation
`lw s7, 0x2C(t1)`	Load word	s7 = memory[t1+0x2C]
`lh s5, 0x5A(s3)`	Load half-word	s5 = SignExt(memory[s3+0x5A]$_{15:0}$)
`lb s1, -3(t4)`	Load byte	s1 = SignExt(memory[t4-3]$_{7:0}$)
`sw t2, 0x7C(t1)`	Store word	memory[t1+0x7C] = t2
`sh t3, 22(s3)`	Store half-word	memory[s3+22]$_{15:0}$ = t3$_{15:0}$
`sb t4, 5(s4)`	Store byte	memory[s4+5]$_{7:0}$ = t4$_{7:0}$
`beq s1, s2, L1`	Branch if equal	if (s1==s2), PC = L1
`bne t3, t4, Loop`	Branch if not equal	if (s1!=s2), PC = Loop
`blt t4, t5, L3`	Branch if less than	if (t4 < t5), PC = L3
`bge s8, s9, Done`	Branch if not equal	if (s8>=s9), PC = Done
`li s1, 0xABCDEF12`	Load immediate	s1 = 0xABCDEF12
`la s1, A`	Load address	s1 = Variable A's memory address (location)
`nop`	Nop	no operation
`mv s3, s7`	Move	s3 = s7
`not t1, t2`	Not (Invert)	t1 = ~t2
`neg s1, s3`	Negate	s1 = -s3
`j Label`	Jump	PC = Label
`jal L7`	Jump and link	PC = L7; ra = PC + 4
`jr s1`	Jump register	PC = s1

Next Lecture:

- More on ISA
  - Store Branch UpperImmediate and Jump instructions.
  - Pseudo Instructions
  - Compressed Instructions
  - (Addressing Modes)