Inst	Name	FMT	Opcode	funct3	funct7
add	ADD	R	0110011	0x0	0x00
sub	SUB	R	0110011	0x0	0x20
xor	XOR	R	0110011	0x4	0x00
or	OR	R	0110011	0x6	0x00
and	AND	R	0110011	0x7	0x00
sll	Shift Left Logical	R	0110011	0x1	0x00
srl	Shift Right Logical	R	0110011	0x5	0x00
sra	Shift Right Arith*	R	0110011	0x5	0x20
slt	Set Less Than	R	0110011	0x2	0x00
sltu	Set Less Than (U)	R	0110011	0x3	0x00
addi	ADD Immediate	I	0010011	0x0	
xori	XOR Immediate	I	0010011	0x4	
ori	OR Immediate	I	0010011	0x6	
andi	AND Immediate	I	0010011	0x7	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20
slti	Set Less Than Imm	I	0010011	0x2	
sltiu	Set Less Than Imm (U)	I	0010011	0x3	
lb	Load Byte	I	0000011	0x0	
lh	Load Half	I	0000011	0x1	
lw	Load Word	I	0000011	0x2	
lbu	Load Byte (U)	I	0000011	0x4	
lhu	Load Half (U)	I	0000011	0x5	
sb	Store Byte	S	0100011	0x0	
sh	Store Half	S	0100011	0x1	
sw	Store Word	S	0100011	0x2	
beq	Branch ==	B	1100011	0x0	
bne	Branch !=	B	1100011	0x1	
blt	Branch <	B	1100011	0x4	
bge	Branch ≥	B	1100011	0x5	
bltu	Branch < (U)	B	1100011	0x6	
bgeu	Branch ≥ (U)	B	1100011	0x7	
jal	Jump And Link	J	1101111		
jalr	Jump And Link Reg	I	1100111	0x0	
lui	Load Upper Imm	U	0110111		
auipc	Add Upper Imm to PC	U	0010111		
ecall	Environment Call	I	1110011	0x0	imm=0x0
ebreak	Environment Break	I	1110011	0x0	imm=0x1

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

x0: constant value 0

x1: return address

x2: stack pointer

x3: global pointer

x4: thread pointer

x5 – x7, x28 – x31: temporaries (np)

x8: frame pointer

x9, x18 – x27: saved registers

x10 – x11: function arguments/results (np)

x12 – x17: function arguments (np)

23, 52 8, 11 127, 1023

$$x = (-1)^S \times (1 + \text{Frac}) \times 2^{(\text{Expo}-\text{Bias})}$$

	0	denormalized	float	infinity	NaN
Expo	0	0	1–254	255	255
Frac	0	nonzero	anything	0	nonzero

CPU time = CPU clock cycles / Clock frequency

- Clock frequency (rate): cycles per second (Hz)
- Clock period: duration of a clock cycle (s)

CPU clock cycles = Instruction count × Cycles per instruction (CPI)

$$\text{CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Saving registers

```
sort:   addi sp, sp, -20        # make room on stack for 5 registers
        sw x1, 16(sp)           # save return address on stack
        sw x22, 12(sp)          # save x22 on stack
        sw x21, 8(sp)           # save x21 on stack
        sw x20, 4(sp)           # save x20 on stack
        sw x19, 0(sp)           # save x19 on stack
```

Procedure body

Move parameters

```
        addi x21, x10, 0        # copy parameter x10 into x21
        addi x22, x11, 0        # copy parameter x11 into x22
```

Outer loop

```
        addi x19,x0, 0          # i = 0
for1tst:bge x19, x22, exit1     # go to exit1 if i >= n
```

Inner loop

```
        addi x20, x19, -1       # j = i - 1
for2tst:blt x20, x0, exit2      # go to exit2 if j < 0
        slli x5, x20, 2         # x5 = j * 4
        add x5, x21, x5         # x5 = v + (j * 4
        lw x6, 0(x5)            # x6 = v[j]
        lw x7, 4(x5)            # x7 = v[j + 1]
        ble x6, x7, exit2       # go to exit2 if x6 < x7
```

Pass parameters and call

```
        addi x10, x21, 0        # first swap parameter is v
        addi x11, x20, 0        # second swap parameter is j
        jal x1, swap            # call swap
```

Inner loop

```
        addi x20, x20, -1       j for2tst
        jal, x0 for2tst         # go to for2tst
```

Outer loop

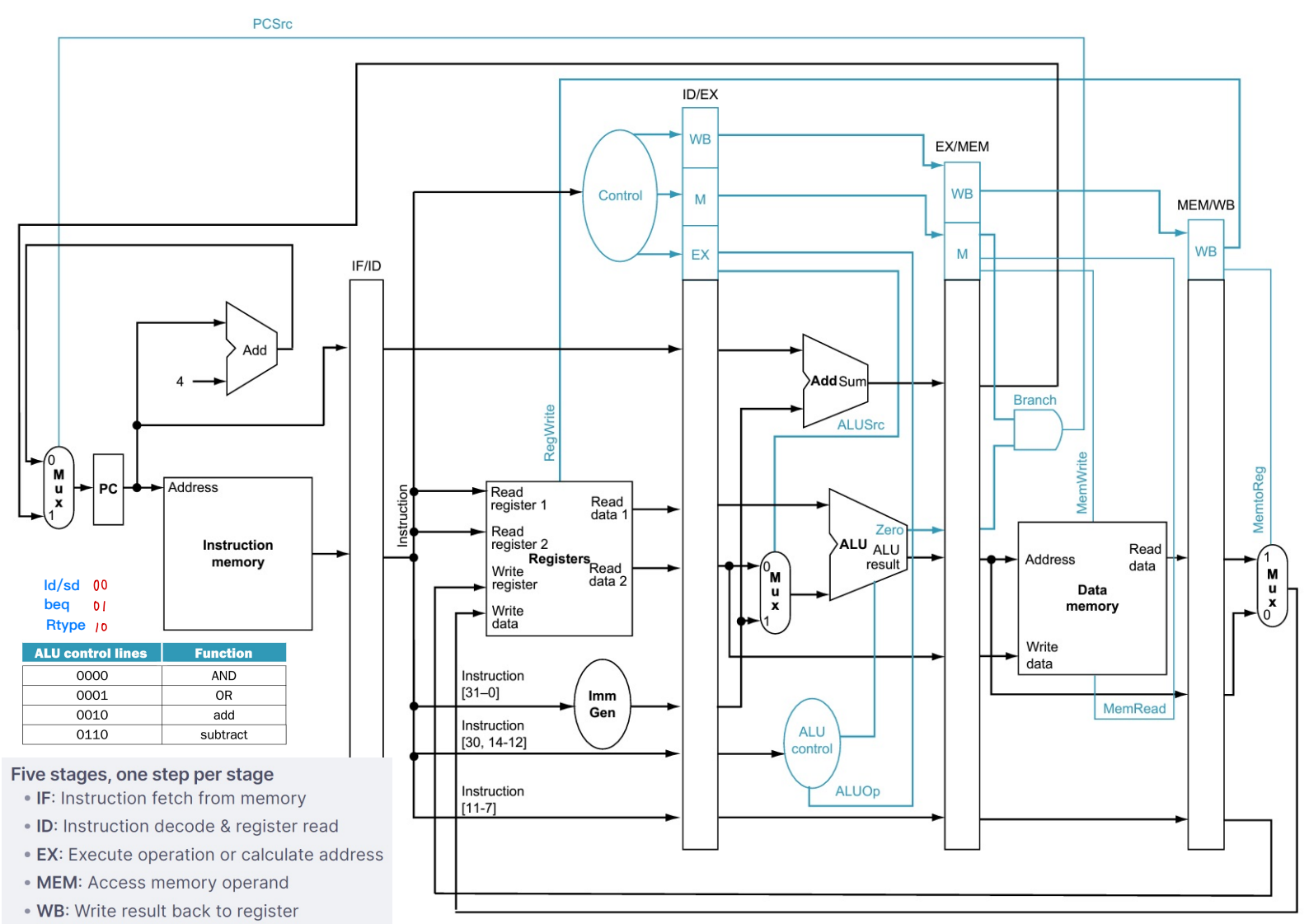
```
exit2:  addi x19, x19, 1        # i += 1
        jal, x0 for1tst         # go to for1tst
```

Restoring registers

```
exit1:  lw x19, 0(sp)           # restore x19 from stack
        lw x20, 4(sp)           # restore x20 from stack
        lw x21, 8(sp)           # restore x21 from stack
        lw x22, 12(sp)          # restore x22 from stack
        lw x1, 16(sp)           # restore return address from stack
        addi sp, sp, 20         # restore stack pointer
```

Procedure return

```
        jalr x0, 0(x1)          # return to calling routine
```



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Five stages, one step per stage

- **IF**: Instruction fetch from memory
- **ID**: Instruction decode & register read
- **EX**: Execute operation or calculate address
- **MEM**: Access memory operand
- **WB**: Write result back to register

Hazards

- **Structural Hazard**: conflict for use of a resource
  → pipelined datapaths need separate memories (caches)
- **Data Hazard**: instruction depends on previous instruction
  ex: `add x19, x0, x1`, `sub x2, x19, x3`
  → forawarding: retrieving data from internal buffers (after **EX**) instead of wait it to be stored in memory
- **Load-Use Data Hazard**: data loaded (at **MEM**) by load instruction have not yet become available when needed
  → reorder code to avoid using load result in next instruction
- **Control Hazards**: fetching instruction depends on branch result
  → branch prediction: only stall if prediction is wrong
  → static: based on typical branch behavior. ex: loop, if
  → dynamic: record recent history of branch, when wrong, stall while re-fetching, and update history

Handle Exceptions

- Save PC of interrupted instruction → SEPC (can be use to return)
- Save indication of the problem → SCAUSE
- Jump to handler → 0000 0000 1C09 0000
- **Vectored Interrupts**: determine handler address by adding exception vector address to vector table base register
- when multiple exceptions:
  - **precise**: deal with exception from earliest instruction, flush subsequent
  - **imprecise**: stop pipeline and save state → complex handler software

Instruction-Level Parallelism

- pipelining: executing multiple instructions in parallel
- increase ILP: deeper pipeline → less work per stage → shorter cc
- **static multiple issue**: compiler group instructions into "issue slots" by pipeline resources required, hazards detected by compiler
  - **RISC-V dual issue**: ALU/branch then load/store, pad unused with nop
  - **loop unrolling**: replicate loop body with different registers
- **dynamic multiple issue**: CPU choose instructions to issue each cycle, compiler can help reorder instructions, hazards resolved by CPU at runtime
  - avoid structural & data hazards, compiler scheduling
  - **dynamic scheduling**: execute instructions out of order to avoid stalls
  - **register renaming**: by copy operand to reservation stations
  - **speculation**: guess the outcome of instructions then start corresponding operation, if guess is wrong then roll back

EX hazard

```
if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10

if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```

Mux control	Source
ForwardA = 00	ID/EX
ForwardA = 10	EX/MEM
ForwardA = 01	MEM/WB
ForwardB = 00	ID/EX
ForwardB = 10	EX/MEM
ForwardB = 01	MEM/WB

MEM hazard

```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01

if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

Load-use hazard

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
    (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
    stall the pipeline (do nop)
```

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw x31, 0(x20)	1
	addi x20, x20, -4		2
	add x31, x31, x21		3
	blt x22, x20, Loop	sw x31, 4(x20)	4

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi x20, x20, -32	lw x28, 0(x20)	1
		lw x29, 12(x20)	2
	add x28, x28, x21	lw x30, 8(x20)	3
	add x29, x29, x21	lw x31, 4(x20)	4
	add x30, x30, x21	sw x28, 16(x20)	5
	add x31, x31, x21	sw x29, 12(x20)	6
		sw x30, 8(x20)	7
	blt x22, x20, Loop	sw x31, 4(x20)	8

```
Loop: lw   x31, 0(x20)    // x31=array element
      add  x31, x31, x21  // add scalar in x21
      sw   x31, 0(x20)    // store result
      addi x20, x20, -4   // decrement pointer
      blt  x22, x20, Loop // compare to loop limit,
                          // branch if x20 > x22
```