Computer Architecture Midterm Review

liuhanzuo

November 4, 2024

1 Basic

1.1 Cost

Moore's Law The number of transistors per square inch on integrated circuits is doubling every two years. – Gordon E. Moore, 1965, revised in 1975.

$$\text{ingot} \to \text{wafer} \to \text{die(unpackaged chip)} \to \text{chip}$$

$$\textbf{yield} = \frac{\text{good chips}}{\text{total chips}}$$

within certain technology, cost $\sim$ area$^{\alpha}$($\alpha > 1$ constant). Bigger chips – more expensive, fewer chips per wafer, lower yield. Usually use area as proxy for cost.
Testing Cost depends on chip complexity, test time, yield...
Packaging Cost depends on pin count, die size, power dilivery...
Design Cost (NRE, non-recurring engineering cost) depends on design complexity, can be amortized over high volume.

System Cost = Cost of power+cost of cooling+total cost of ownership(TCO)+availability

1.2 Parralelism

Inter-task parallelism: Do not seperate tasks, run many synchronously. Improve throughput, not latency.
Intra-task parallelism: Break a task into subtasks, run them in parallel. Improve latency and throughput.
Buffering/queuing/batching improves throughput but may hurt latency.

$$\text{Performance} = \frac{1}{\text{Execution Time}}, \text{Speed up of } X \text{ over } Y = \frac{\text{Performance of } X}{\text{Performance of } Y}$$

$$\text{Execution Time} = \text{CCT} \times \text{Num cycles}, \text{Num Cycle} = \text{IC} \times \text{CPI}$$

$$\text{Execution Time} = \text{IC} \times \text{CPI} \times \text{CCT}, \text{CPI} = \frac{\text{Num Cycles}}{\text{IC}} = \sum_i \frac{\text{IC}_i}{\text{IC}} \times \text{CPI}_i$$

software determines IC, ISA determines CPI, microarchitecture/circuit determines CCT

1.3 Memory

Processor: Execution Time $= \frac{\text{ops}}{\text{throughput}}$, Memory: Execution Time $= \frac{\text{bytes}}{\text{bandwidth}}$
Operational Intensity(Arithmetix intensity): how many ops to perform for each byte.
Roofline Model: Perf $= \min\{\text{Processor Throughput}, \text{Memory Bandwidth} \times \text{OI}\}$

1.4 Power

Dynamic/active power $\alpha C V_{dd}^2 f$, $\alpha = \frac{\text{switch0}\to 1}{\text{op num}}$
Static/leakage power $V_{dd} I_{leak}$, power $= \alpha C V_{dd}^2 f + V_{dd} I_{leak}$

$$\text{Power Density} = \frac{\text{power}}{\text{chip area}}, \text{Energy} = \text{average power} \times \text{execution time}$$

Limits: power–infrastucture, power densety–thermal dissipation, energy–battery capacity.
First line: Past Denard Scaling, Second line: Current Scaling

Feature Size	Area	Capacitance	Voltage	Current	Freq	Energy	Power	Pwr Density
$1/S$	$1/S^2$	$1/S$	$1/S$	$1/S$	S	$1/S^3$	$1/S^2$	1
$1/S$	$1/S^2$	$1/S$	~ 1	~ 1	$< S$	$1/S$	< 1	$< S^2$

Table 1: Scaling of various parameters with feature size

$$\text{Energy Efficiency} = \frac{\text{Performance}}{\text{Power}} = \frac{\text{Operaion}}{\text{Energy}}$$

Pareto-optimal curve: find the optimal curve – within $x\%$ of the best performace and $y\%$ of the best energy efficiency.
Scalability: N Processor's speedup verses N. Strong scalability: with fixed total workload size. Weak scalability: with fixed workload per processor.
Balance work – static load balancing(partition input), dynamic load balancing(work dispatch more work, work stealing work on other's work) Amdhal's law: $S = \frac{1}{(1-f)+\frac{f}{S}}$, No prematurally optimize, no overlook opportunities.
Benchmark: programs for evaluaion. Benchmark suite: a set of benchmarks.
amean for absolutes; hmean for rates; gmean for ratios. $\to$ weighted average.

2 ISA-ASM

2.1 RISCV

ISA defines the state of the system, instruction functionality and format.
Compute: arithmetic,logic,shift,compare. Memory: load,store. Control: jump,branch.
Overflow: When the resul of add/sub cannot be represented in n bits, simply

throw away the most significant bit. RISCV choose to ignore them in add and sub.
imm operands should be 12-bits long. always sign-extended.
shift left logically, right logically: zero-ext, right arithmetically: sign-ext.
In most RISC ISAs, all memory accesses happen through loads/stores, Other instructions only manipulate data within the processor.
Note lui and aupic uses 20-bits imm, but addi uses 12-bits imm. rd will read a 20 bits imm to the 20 most significant bits.
In RV64I, the RV32I instructions will be sign-ext to 64 bits (load/store a double word:ld/sd).
for loops, reduce instruction in the loop body and the number of branch/jump instructions.

2.2 Procedure Call and Calling Convention

A complete ISA also contains instructions to control and implement: user programs request service, such as syscalls. And OS kernel implements priviledged/protected operations.
Procedure arguments: Before calling jal, caller stores $a_0 \sim a_7$, others are passed through stacks.
Return values: Return value is stored in a_0, a_1, others are passed through stacks.
Caller Saved ra,$t_0 \rightarrow t_6$,$a_0 \rightarrow a_7$,caller save before call callee, callee can use freely. **Callee Saved** sp,$s_0 \rightarrow s_{11}$,callee save before call another function, caller can use freely.
sp(stack pointer):points to the top of the stack. Decrement when pushing frame, Increment when popped. Always the multiple of 16, 16-bits aligned.
fp(frame pointer):points to the bottom of the frame, moved with sp together.
frame have Return address to its parent (as needed), Arguments to the function, Local variables,Caller/callee-saved registers (as needed), All arguments of its children (first 8 have values in registers).

3 RISCV Encoding

Instructions are simply represented as binary data in the same memory (Von Neumann). the only difference is how bits are interpreted. Binary representation of instructions is part of the ISA.
Binary backward compatibility: old programs could run on new version

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

Table 2: RISC-V Instruction Formats

Note that all possible rd, rs1 and rs2 are in the same position for all kind of instructions.
R-Format: funct7+funct3 is the function code. example: add,sub,xor,or,and,slt,sltu.
I-Format: imm is 12 bit, used by register-immediate instructions. rd for destination, rs1 for source. Compare to R-Format, replace rs2 and funct7 with imm. example: lw,addi,slti,sltiu,xori,ori,andi,slli,srli,srai.
Shift logically or arithmetically is decided by imm[5:11]. load signed or not: funct3[2], size: funct3[1:0].
S-Format imm is splited into two parts imm[11:5] and imm[4:0], replacing funct7 and rd. example: sb,sh,sw,addi,slti,sltiu,xori,ori,andi,slli,srli,srai.
U-Fomat imm is 20 bits, used by lui and aupic.PC is 32 bit, but immediate value is 12-bit. But branch target is often near to the branch instruction. example: lui, aupic.
Encode address difference from current PC in signed immediate field.

$$\text{branch target} = \text{PC} + \text{imm}[12:1] \times 2$$

instructions address is always a multiple of 2 bytes(compressed extension allows 16-bit instructions). If target is too far away from the branch, turn into an unconditional jump
B-Format: reordered imm, imm[0] is 0, imm[12] is the most significant bit. example: beq,bne,blt,bge,bltu,bgeu.
J-Format: bit 31 in imm[20] for sign-ext. imm[10:1] matches I-format, imm[19:12] matches U-format. omit imm[0] to 0. jal can jump withi in 2^{20} bits. jalr can jump within 2^{11} bits. example: jal,jalr.

3.1 SSE/AVX Registers

Fixed total length can be split into various numbers of data types.
zmm is 512 bits, ymm is lower 256 bits, xmm is lower 128 bits.
SIMD intrinsics: use C functions. Benefits: better and simpler syntax, registers managed by compilers.
function name: _mm⟨width⟩_[function]_[type]
drawback: vector length is hard-coded in instructions. Each time we introduce a longer vector support, when introduce a longer vector support, we need add

more instructions. Old code that is written with old vectors cannot benefit from new hardware.No flexibility in microarchitecture design to select vector length for hardware cost

3.2 RISCV Vector Extension

Decouple vector lengths in software application and harware implementation. No need to introduce new instructions when developing new hardware. No need to change software code when porting to new harware
32 vector data registers: v0 to v31. Each is **VLEN** bits long. Vector length register **vl**. **vl** is the length of software data(variable). **VLEN** is the length of hardware space(fixed). **vl** is number of elements, **VLEN** is number of bits, Vector type register **vtype**, only talk about LMUL and SEW here
LMUL= 2^{lmul} vector register grouping multiplier.
SEW:selected element width. SEW = 2^{3+vsew}.
Instruction types(FP) and SEW determine data types together(ideally SEW should be encoded in instruction).
Set vector configuration: vsetvli, vsetvl, vsetivli. rs1 or imm is the requested application vector length(AVI), rs2 is the value that encodes SEW, LMUL. $\text{vl} = \min(\text{AVL}, \text{VLMAX})$ is set $\text{VLMAX} = \text{LMUL} \times \text{VLEN/SEW}$, the resulting vl value is also returned in the destination register.
AVL: application vector length, the actual length of data vector, **VLEN:** hardware register size, maximum size that harware can process at a time **vl:** the actual vector length that hardware process in a time, in number of elmenets.

Instruction	Operands	Operation	Comment
vadd.vv	vd, vs2, vs1	vd = vs2 + vs1	Vector + vector
vadd.vx	vd, vs2, rs1	vd = vs2 + rs1	Vector + scalar
vadd.vi	vd, vs2, imm	vd = vs2 + imm	Vector + immediate scalar
vsub.vv	vd, vs2, vs1	vd = vs2 - vs1	Vector - vector
vsub.vx	vd, vs2, rs1	vd = vs2 - rs1	Vector - scalar
vrsub.vx	vd, vs2, rs1	vd = rs1 - vs2	Scalar - vector
vrsub.vi	vd, vs2, imm	vd = imm – vs2	Immediate scalar - vector
vmul.vv	vd, vs2, vs1	vd = vs2 * vs1	Vector * vector
vmul.vx	vd, vs2, rs1	vd = vs2 * rs1	Vector * scalar

Table 3: vectored instructions

Unit-stride (sequential): vle<w>.v vd, (rs1), vse<w>.v vd, (rs1)
<w> could be 8,16,32,64 denote the element width, rs1 is the base address, no offset.
Constant-strided: vlse<w>.v vd, (rs1), rs2, vsse<w>.v vd, (rs1), rs2
rs2 provide the stride, rs1 is the base address.
vd stores $\text{MEM}[\text{rs1}, \text{rs1} + \text{rs2}, ..., \text{rs1} + (\text{vl} - 1) \times \text{rs2}]$

Indexed (scatter/gather): vluxei<w>.v vd, (rs1), vs2, vsuxei<w>.v vd, (rs1), vs2
vs2 provide the list of the strides, as byte address offsets. vd stores MEM[rs1 + vs2[0], rs1 + vs2[1], ..., rs1 + vs2[vl − 1]]
mask: vector register v0 could hold a mask, written as v0.t, could be added as a bit for other possible instructions (such as vsub, do as conditional sub).
For insance, vsub.vv v3, v1, v2, v0.t means if v0.t[i] is 1, then v3[i]=v1[i]-v2[i], otherwise v3[i]=0.
Advanatages: Compact: a single instruction defines N operations. Reduce the cost of fetch, decode, issue, reduce number of branches.
Parallel: N operations are data parallel: no need for complex harware to dynamically detect parallelism.
Expressive: various expressive patterns – sequential, strided, indexed.
Compare to SSE/AVX: ISA is agnostic to hardware vector register length, automatically take advantage of new hardware. No need to create new instructions for longer vector length

4 Single-Cycle Processor

CPI=1, a processor consist of two parts datapath and control.
Datapath: collection of functional units, registers, and buses.Include combinational circuits (data processing) and state elements (data storage)
Control: the hardware that manages the datapath. Mostly combinational logic to generate control signals

4.1 Hardware

State elements, Combinational logic, Control logic.
Basic steps: fetch, decode, execute, memory, write back.
Steps for add/sub:

1. read two registers from register file
2. perform arithmetic/logic operation in ALU(a control signal ALUOP)
3. write register result back to register file(enabled by RegWrite)

imm instruction: get the second operand from instruction, sign-ext. need a MUX to control whether choose from the register file or imm: ALUSrc.
load: a control signal MemRead, read data from memory, note the address is rs1+imm, so need ALU. Write data from memory to register: need MemToReg.
store: part of the imm is from instr[11 : 7], need more complex combination logic ImmGen. Data of rs2 is passed to memory as write data, a control singal MemWrite indicate whether to write the memory.
branch and jump: PC-relative addressing: immediate specifies a PC offset.

AND with control signal **branch**. unconditional jump with signal **jump**, **two extra adders and a MUX** to select the next PC: PC+4 or PC+2×sign-ext(imm)

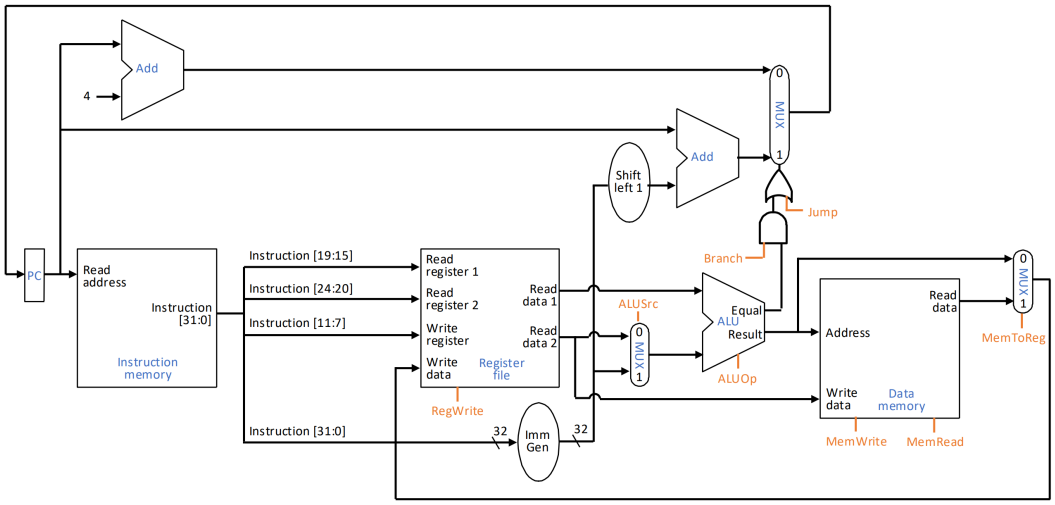

Figure 1: Description of the image

Control: decode instruction to generate control signal. **ALUOp** operation type for ALU, **RegWrite, MemRead, MemWrite** read/write control on registers and data memory, **ALUSrc, MemToReg** dataflow on the multiplexers, **Branch, Jump** control flow for next PC.

Ops	ALUOp	ALUSrc	MemToReg	RegWrite	MemWrite	MemRead	Branch	Jump
add	ADD	0	0	1	0	0	0	0
ori	OR	1	0	1	0	0	0	0
lw	ADD	1	1	1	0	1	0	0
sw	ADD	1	x	0	1	0	0	0
beq	SUB	0	x	0	0	0	1	0
j	x	x	x	0	0	0	x	1

Table 4: Control Signals

4.2 Multi-level Decoding

Two-level decoding to simplify control logic: pass opcode to the main control block, pass the partial decoded signal and funt3, funct7 to a local ALU control

block.
Two simpler blocks¡ One big block, fewer wires: fewer signals to route.

4.3 Pipeline

For a $k-$stage pipeline,

$$f = (\frac{t_{\text{comb}}}{k} + t_{\text{reg}})^{-1}$$

For N tasks, T time per task, unpipelined:
unpipelined time $= NT$, $k-$stage pipeline $= \frac{k-1}{k} \times T + N \times \frac{T}{k}$
Pipelining improves throughput of entire workload, but make latency of a single task worse. Carefully choose where to partition: Unbalanced stages reduce speedup, throughput is limited by the slowest stage.Register width is affected by signals across stages.
Need pipeline registers to store value of previous stage, pipeline the control signals along with data, propogate the control signal until consumed.

4.4 Pipeline Stalling

Note that 5-stage pipelining actually cause stalls– some data need for next instruction is not ready.

$$\text{effective CPI} = \text{base CPI} + \text{stall cycles per instruction}$$

generate some nop to fill the pipeline for stallings – prevent updates of PC and IF/ID, ID/EX registers

5 Hazards

$$\text{CPI} = \text{base CPI} + \text{stalls cycle per instruction}$$

Structural Hazards: A required resource is busy in this cycle.
Data Hazards: Must wait previous instruction's data.
Control Hazards: Next PC depends on previous instruction.

5.1 Structural Hazards

WB and **ID** both access registers.
Solution: different read/write ports, half-cycle registers access.
Two read ports, one write port. Write data into register file in the first half cycle, read data in second half cycle. Ensure data consistency by internal design.

- let it stall: performance loss, tradeoff between performance and complexity.

- replicate resources: seperate instruction/data memory(multi-port memory/register file)
- design away the structural hazards, always use each resource once in the same stage. (delay R-formats WB to the 5th stage)

5.2 Data Hazards

RAW Read after Write, true dependency. Need stalls to handle it.
WAW Write after Write, output dependency.
WAR Write after Read, anti dependency.
forwarding:

$$\text{stalls} = \text{produce cycle} - \text{consume cycle}$$

Destination: consume cycle – EX, MEM, Sources: produce cycle – MEM, WB.

5.3 Control Hazards

predict the next PC value.
current PC: known in IF, current instruction: known in ID, register values: obtained in ID. need stall to 1 cycle.
Consider prediction: predict next instruction, if prediction is wrong, need to flush pipeline to nullify instructions and restart, penalty is 1 cycle + some extra penalty.

Control flow CPI overhead = % branch/jump×% misprediction×misprediction penalty

Additional forwarding paths of register values to ID stage, 1-cycle stall for previous ALU operation, 2-cycle stall for previous load.

5.4 Exceptions and Interrupts

OS – priviledged kernel handles exceptions(internal)/interrupts(external).
1. Save PC and current cause, transfer to kernel.
2. process exceptions/interrupts by exception handler.
3. return to the user program to continue or abort.
Handler process 1. Save PC and current cause, Supervisor Exception Program Counter(SEPC)
2. Save the reason for interrupt/exception: Supervisor Exception Cause Register (SCAUSE), encode the reason.
3. Jump to the handler at a fixed address in kernel
4. handler reads the cause register to decide what to do.
5. fixable: take corrective actions and use and use SEPC to return to user code. Otherwise, terminate program and report error using SEPC and SCAUSE **vectored Interrupts:** seperated handlers for different causes.
Each cause use a unique exception number to look up in the exception table –

interrupt vector.(direct deal/ jump to real handler)
When exception/interrupt occurs in the pipeline
1.Drain older instructions down the pipeline to complete
2.Nullify the current and younger instructions in earlier stages
3.Fetch from handler instruction address

6 Branch Prediction

6.1 BHT

Branch History Table: a table indexed by the lower bits of the PC, stores the history of the branch.
For a BHT table size 2^m, m bits of the PC are used to index the table.
Aliasing: two branches with different PC may be indexed to same BHT entry.
count +1 for an taken branch, -1 for an untaken branch.
1-bit states and 2-bit states. Simple 2-bit BHT already achieves 90% accuracy.
Correlated Behavior: two branches are correlated if the outcome of one branch is correlated with the outcome of the other.

6.2 BTT

Branch Target Buffer: Record previously executed branches and their most recent taken target addresses. Address: current PC, Data: next PC.

6.3 ILP

Instruction Level Parallelism: the number of instructions that can be executed in parallel.

$$T_n = \text{time to execute with } n \text{ processors}$$

$$\text{Average ILP} = \frac{T_1}{T_\infty}, \text{upper bound on the attainable IPC}$$

6.4 OoO Execution

Use diversified functional units to finish instructions ASAP to avoid blocking.
Use inter-stage buffers for decoupling stages and reordering instructions.
Basic steps: fetch, decode, dispatch/rename/allocation, issue/schedule, execute, commit/retire/wb.
Limitations: cannot faster than fetching and decoding steps.
Fetch: wide and speculative, Read many words and select those from current PC to the first-taken branch, Use complicated branch prediction to predict the next PC beyond branches.

Decode: CISC instructions are often translated into RISC-like micro-ops (uops)
Renaming, Execute: for deal WAR and WAW, use a new register for each new value definition.
physical register > architectual registers. Dynamically mapping PR to AR.
Renaming in order, exexution not in order.
Issue IW: instruction window, dispatch stage adds instruction to IW in order. Issue stage picks instructions from IW to functional units out of order.
scheduling instructions find the best execution order. **Scheduler:** decide order of instructions based on priority(FCFS, critically,latency...)
Commit an early instruction that executes late may trigger an exception after later instructions have executed, force commiting in order.
Each cycle, examing the oldest one and commit. If raising exceptions or mispredictions, flush the pipeline.
ROB(reorder buffer): a FIFO instruction tracking.

6.5 Modern Processor

Limitations: Limited ILP in programs, Pipelining overheads, Frontend bottleneck, Memory inefficiency, Implementation complexity.
Energy vs Performance is superlinear.
ALU is the really useful computation, but only 5%, others are instruction fetch and decode.
data level parralelism and custom design(AISC).
Memory Wall Memory performance and energy dominates with processors improved – memory accesses too expensive.

7 Appendix

7.1 Definitions

latency How long to complete a task.
Throughput How many tasks can be completed per unit time.
Clock Cycle Time(CCT) duration of a clock cycle.Clock Frequency $f = \frac{1}{\text{CCT}}$
Instruction count(IC) number of instructions executed.
Cycles per instruction(CPI) average number of cycles per instruction.

7.2 Zero and sign extension

Zero extension can pad all 0 to the front: $10110_2 \rightarrow \overline{000}10110_2$
Sign extension can pad the sign bit to the front: $10110_2 \rightarrow \overline{111}10110_2$, $01001_2 \rightarrow \overline{000}01001_2$

7.3 Registers

- zero(x_0): hardwired to 0
- ra,sp,gp,tp(x_1, x_2, x_3, x_4): return address, stack pointer, global pointer, thread pointer
- t_0-t_6($x_5 - x_{11}$): temporaries
- s_0-s_{11}($x_8 - x_{15}$): saved temporaries
- a_0-a_7($x_{10} - x_{17}$): arguments

7.4 Forwarding

IF	ID	EX	MEM	WB

Table 5: Basic concept of forwarding

- from MEM to EX

  $$\text{EX/MEM.RegWrite}, \text{EX/MEM.RegisterRd} \neq \text{x0}$$

  Ex/Mem.RegisterRd==ID/EX.RegisterRs1 or ID/EX.RegisterRs2, !EX/MEM.MemToReg

- from WB to EX

  $$\text{MEM/WB.RegWrite}, \text{MEM/WB.RegisterRd} \neq \text{x0}$$

  Mem/Wb.RegisterRd==ID/EX.RegisterRs1 or ID/EX.RegisterRs2, !ForwardFromMEM

  ForwardFromMEM is for double data hazard.

- load-use stall (use instruction in IF/ID and load is in EX/MEM)

  ID/EX.MemRead, ID/EX.RegisterRd = ID.RegisterRs1 or ID.RegisterRs2

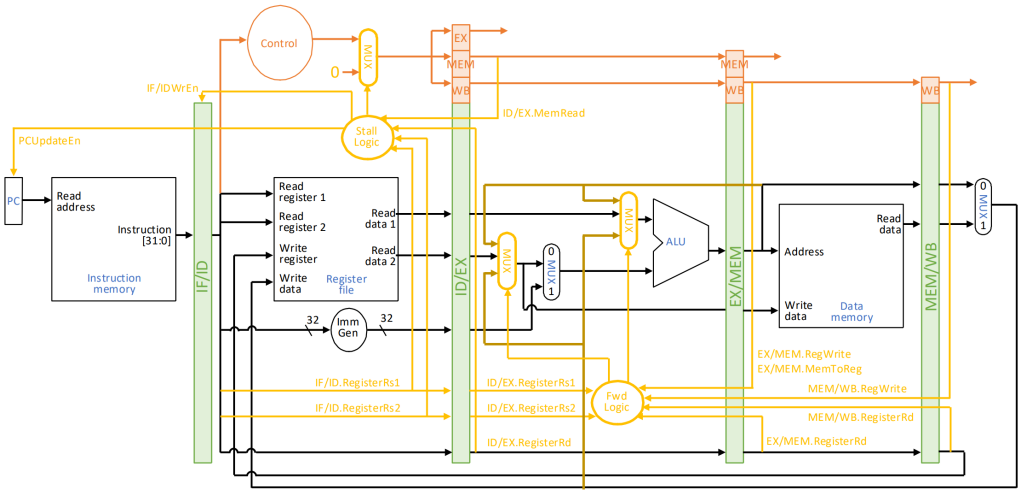

Figure 2: 5-Stage Pipeline with Stalling and Forwarding