CS152/252A
Fall 2026

Architectures, Microcoding

Discussion 1

1 Architecture vs Microarchitecture

True or false: The following is architecturally visible (exposed by the architecture)?

(a) Register file entries in a classical RISC pipeline

True - This is true because the number and semantics of architectural registers are explicitly defined by the ISA. Instructions name registers directly, and program correctness depends on their existence and behavior. Therefore, register file entries are architecturally visible state.

(b) The stack in a stack architecture

True - In a stack-based ISA, the stack is the primary operand storage mechanism and is explicitly defined by the architecture. Instructions implicitly read from and write to the stack, making it part of the architecturally visible state.

(c) Pipeline registers

False - Pipeline registers are a microarchitectural implementation detail used to increase instruction throughput. They are not defined by the ISA, cannot be accessed by software, and may differ across implementations without affecting architectural correctness.

(d) Branch-delay / load-delay slots

True - Branch-delay slots are defined by certain ISAs, which guarantee that the instruction following a branch is always executed. This affects program semantics and must be handled explicitly by compilers and programmers. Therefore, branch-delay slots are architecturally visible.

(e) NOPs

True - NOP (no-operation) instructions are explicitly defined by the ISA and can be inserted by software. They affect instruction sequencing and timing (e.g., filling delay slots), and thus are part of the architecturally visible behavior.

(f) Pipeline bubbles

False - Pipeline bubbles represent idle cycles introduced internally by the processor to resolve hazards. They are not instructions, are not specified by the ISA, and are invisible to software. Hence, they are not architecturally visible.

(g) Condition codes, status flags

True - Condition codes and status flags are defined by the ISA and are updated by instructions and read by subsequent instructions (e.g., conditional branches). Program behavior depends on their values, making them architecturally visible state.

(h) Memory address width
True - The memory address width (e.g., 32-bit vs. 64-bit) is defined by the architecture and determines the addressable memory space and pointer size. Software and operating systems rely on this property, making it architecturally visible.

(i) Instruction/data caches
Usually false - Caches are generally microarchitectural structures used to improve performance. The ISA does not require their presence nor define their size or organization, and cache hits or misses do not change architectural program semantics. There are some ISAs where cache management instructions exists but this is not generally true.

2 Microcoded vs Pipelined

How does a microcoded machine differ from a classic RISC pipeline?
Consider how instruction execution phases are implemented:

- Instruction Fetch: A microcoded machine and RISC pipeline fetch instructions from an instruction cache/memory in the same way.
- Decode: In a RISC pipeline, we decode fields like the opcode, funct3, and funct7 to determine control signals necessary for a given instruction as well as registers being used. For a microcoded machine, there is no need to determine control signals; we only need to determine where to send our μPC and follow the μinstructions and control signals defined there rather than determining control signals from the instruction itself.
- Register Read: A microcoded machine is generally only able to read one register value per cycle if there is only one bus available for it to be connected to. However, there can be multiple μinstructions to do multiple register reads for a single instruction. In a classic RISC pipeline, you are strictly limited to 2 register reads per instruction / cycle.
- ALU Operations: ALU operations are executed in the same way for both types of machines.
- Memory Operations: Memory operations are executed in the same way for both types of machines.
- Register Writebacks: Similar to register reads, microcoded machines can only write one register value per cycle, but there may be multiple μinstructions allowing for a single instruction to complete multiple register writes. A RISC pipeline is still limited to one register write per instruction / cycle.
- Next PC Calculation: RISC pipelines consistently increments the PC by 4 each cycle, except for when a jump occurs or branch is taken, in which case it takes the output of the ALU (PC+imm / Register+imm). In a microcoded machine, the PC is not incremented every cycle. It is incremented before an instruction is dispatched, where an arbitrary number of cycles may pass before the instruction completes and we are able to fetch the next instruction and increment the PC again. The microcoded machine handles jumps/branches similarly to the RISC pipeline where the ALU output is saved to the PC.

Why is a simpler microarchitecture generally possible with microcoding?

- μArchitecture: No complex control logic, no hazard detection, no forwarding, no speculation, no pipeline registers. Instead: microcode ROM says exactly what to do, one thing per cycle.
- RISC: Each stage has dedicated hardware. Instructions in different stages overlap in time and can result in some data/control hazards.

3 Microprogramming

Implement a conditional memory-to-memory move instruction in microcode for the single-bus RISC-V machine described in Handout 1 The instruction has the following format:

```
CMOVM (rd), (rs1), rs2
```

CMOVM performs the following operation: If the value in rs2 is true (non-zero), then the memory word loaded from the address in rs1 is stored to the address in rd.

```
if R[rs2] != 0
  M[R[rd]] := M[R[rs1]]
```

Fill in the following table with the microinstructions and control signals. Optimize your microprogram to minimize the number of cycles and to set entries to don't-cares (*) wherever possible.

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En	A Ld	B Ld	ALU Op	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	µBr	Next State
FETCH0	MA := PC A := PC	*	PC	0	1	1	*	*	0	1	0	0	*	0	N	*
	IR := Mem	1	*	0	0	0	*	*	0	0	0	1	*	0	S	*
	PC := A + 4	0	PC	1	0	0	*	INC_A_4	1	*	0	0	*	0	D	*
…																
NOP0	µBr to FETCH0	*	*	0	0	*	*	*	0	*	0	0	*	0	J	FETCH0
CMOVM0:	A := R[rs2]	0	rs2	0	1	1	*	*	0	*	0	0	*	0	N	*
	MA := R[rs1] if (A == 0): goto FETCH0	0	rs1	0	1	*	*	COPY_A	0	1	0	0	*	0	EZ	FETCH0
	A := Mem	0	*	0	0	1	*	*	0	0	0	1	*	0	S	*
	MA := R[rd]	*	rd	0	1	0	*	*	0	1	0	0	*	0	N	*
	Mem := A	*	*	0	0	*	*	COPY_A	1	0	1	0	*	0	S	*
	goto FETCH0	*	*	0	0	*	*	*	0	*	0	0	*	0	J	FETCH0
…																

Some extra explantations regarding certain entries in the above table:

***'s and 0's in IrLd:** We need the first couple rows of IrLd to have the value of 0 because we need to maintain the rs2, rs1, and rd value for them until MA := R[rd]. And after that, we don't need to read rs1/rs2/rd values which are part of the instruction, so we can leave them as *'s.

0 for ALd in Pseudocode MA := R[rd]: The next line uses the A value from the previous instruction, which is A := Mem, during the execution of MA := R[rd], we want to maintain the value of A as what we got from A := Mem, so we set ALd to 0 and it does not get overwritten.

0 for MALd in Pseudocode A:= Mem: Similar to the expatiation for the previous one, we want to maintain the value of MA as the one from the previous line, which was MA := R[rs1], so that when we are reading the data from Mem, it does not get misread by an overwritten memory address.