Lec 9 - Control Hazards and Midterm Review

▼ Control Hazards / Recap of Data Forwarding

Data forwarding recap:

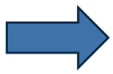
ALU to ALU
- forwarding from EX/MEM pipeline latch
— execute
- forwarding from MEM/WB pipeline latch
— memory / wb = writeback?

Memory-ALU
- NOP
- forwarding from MEM/WB pipeline latch

Control Hazards

branches i.e. loops

```
int main() {
    for (int i = 0; i < 10; i++)
{
        // loop body
    }
    return 0;
```



```
main:
    li t0, 0                    # i = 0
    li t1, 10                   # n=10
loop:
    addi t0, t0, 1              # i = i + 1
    bne t0, t1, loop            # if (i != n) => loop
exit:
    li a0, 0                    # return 0
    ret
```

Register mapping according to RISCV spec
t0 → x5
t1 → x6
//pseudo instructions
Li → LUI and ADDI; instructions determined by compiler

ret → JALR x6

Data and control dependence:

cannot jump before iterating and checking branch condition
- need to calculate next PC

Branches

Conditional:
- BEQ r1, r2, imm; if (r1 == r2) ⇒ PC = PC + imm

Unconditional:
- JAL rd, imm; know where to jump to
- JALR rd, rb, imm; dynamic address calculation, useful for when we don't know where to jump to (function call?)

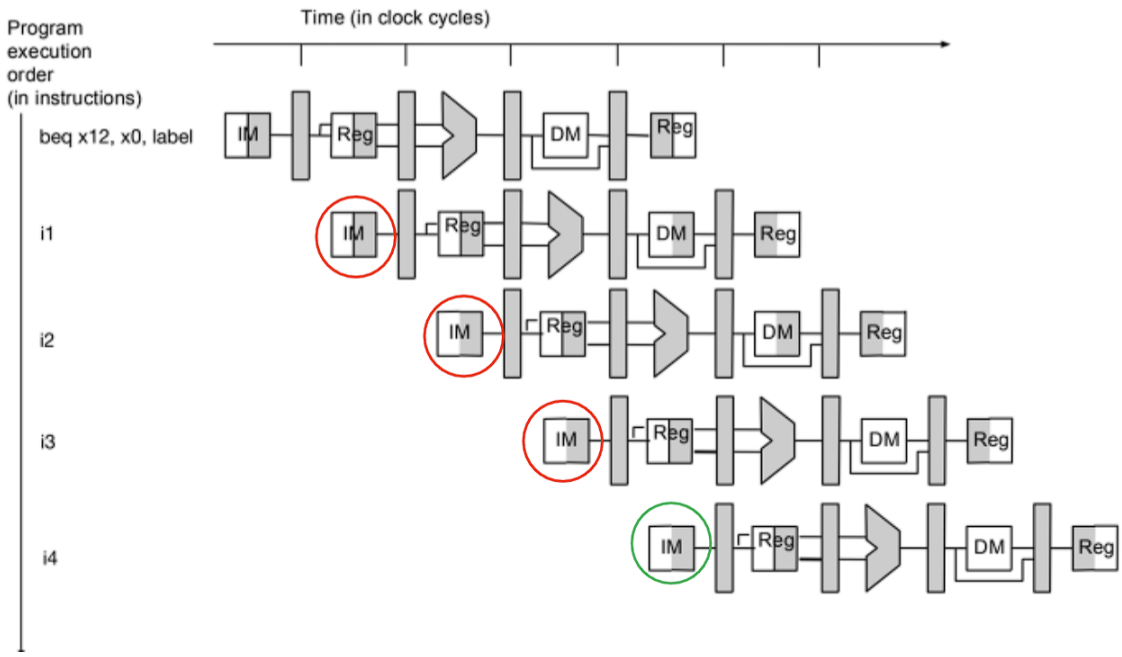
Whether to branch?
determined in memory stage (PCSrc)
control logic feeds to decode stage and selects PC+4 or calculated address for next instruction address

Where to branch
next instruction address computed in execute stage
→ need to store in Instruction Fetch stage since program counter is located there
PC is updated next cycle after PCSrc
- input from MEM stage to PC on same cycle as PCSrc, but since it's a flip flop, output is only propagated on next cycle

> Still specific to project microarchitecture. Basic CPU

Instruction data stored in latch
in each stage, we check if the instruction is valid (also stored in latches)

- can invalidate for next cycle; essentially NOP, stall progress, don't go to next stage
— use register to store valid bit, can modify and change this once we determine branch condition and instruction address → resume instruction



Determine branch condition in DM, requires 3 stall cycles (NOPs), branch address needs to propagate for 1 cycle so ready on the next cycle

Control Hazard Handling

Software solution: `insert NOPs`
Pros:
- no hardware cost
Cons:
- Bad performance: still stalling
- S/W depends on H/W stall count

Alternative software solution:
`Delayed branch`
S/W insert useful instructions in delay slots
Pros:

• No H/W cost
• Improved performance
Cons:
• Hard to find useful instructions
— instruction scheduling
• S/W depends on H/W stall count

Hardware solution:
`stall the pipeline`
-

> need to know how many NOPs to insert; software may break or become inefficient if hardware optimization improves

Hardware solution: Speculation

Prediction

Flushing the pipeline: convert to NOPs, performance cost is 3 cycles of instructions

Moving branch logic can be easily moved to Execute stage
- originally not in Execute because of propagation delay

More aggressive solution, BRANCH ONLY
- move NextPC comparison to ID
- move PC + imm addition to ID
Warning! r1/r2 RAW; need data forwarding

Better solution: always taken prediction instead
- need to know next PC for taken branch in order to execute instructions before next PC is computed
— problem!!
- small memory/cache/history to predict next PC

▼ Midterm Review

Benefit of latches:
write and read in the same clock cycle, reduces number of stores that may occur in the pipeline

Will provide format and addressing mode; important to know the operands in use and infer the types of instructions and cost associated (hardware cost)
Calculating clock speed from propagation delay of single cycle vs. multicycle/pipelined