Processor FSM Design

Mainak Chaudhuri
Indian Institute of Technology Kanpur

Sketch

- Abstract model of computer
- Single-cycle instruction execution
- Multi-cycle instruction execution
- Pipelined instruction execution

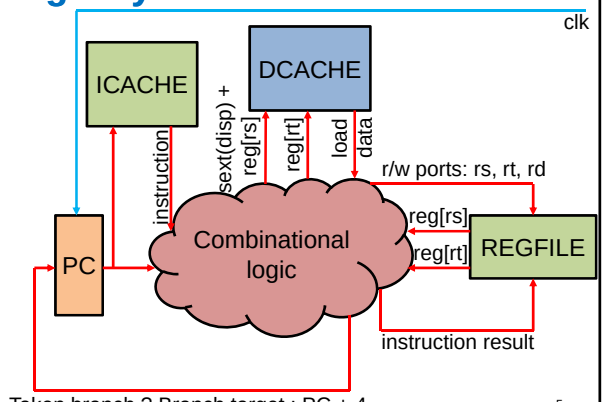
Abstract model of computer

- Each instruction undergoes five stages
  - Stage 0 (IF): fetch the instruction pointed to by program counter from memory
  - Stage 1 (ID/RF): decode the instruction to extract various fields and read source register operands
    - Source registers can be read in parallel with instruction decoding in MIPS due to fixed positions of rs and rt in all formats of the ISA
  - Stage 2 (EX): execute the instruction in ALU; compute address of load/store instructions; update program counter to PC+4 or branch target (if this instruction is control transfer ins)
  - Stage 3 (MEM): access memory if load/store instruction; use address computed in stage 2
  - Stage 4 (WB): write result back to destination register if the instruction produces a result

Single-cycle instruction execution

- Possible to implement all five stages as a big combinational logic
  - All the memory elements are also combinational meaning that the input address can be presented any time within a cycle and the output data is available within the same cycle
    - Register file (in stages 1 and 4), icache (in stage 0), and dcache (in stage 3) are the combinational memory elements
    - PC, icache[PC], reg[rs], reg[rt], dcache[disp(rs)] are inputs
      - dcache[disp(rs)] is a relevant input for loads
    - PC, reg[rd] or reg[rt], dcache[disp(rs)] are outputs
      - dcache[disp(rs)] is a relevant output for stores
  - The instruction opcode and function (for R format) fields decide the control of combinational logic e.g., which register operand(s) is/are valid sources, which ALU operation to invoke, etc.

Single-cycle instruction execution



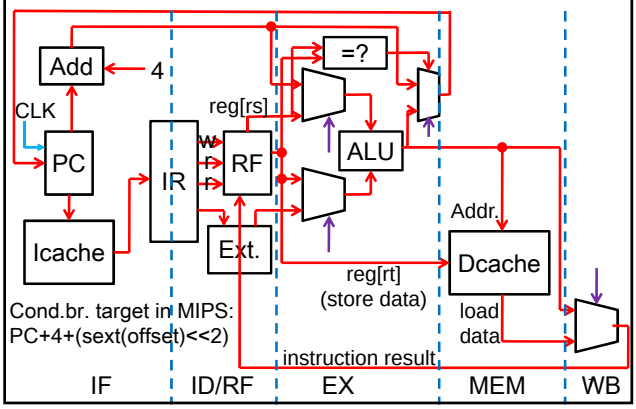
5

Single-cycle instruction execution

- Taken branch
  - A branch instruction whose condition dictates that the next instruction to be executed is the one at the branch target
    - beq $1, $2, label (the branch is taken if $1 == $2)
    - Unconditional direct and indirect jumps and procedure calls are all taken branches
- Not taken branch
  - A branch instruction whose condition dictates that the next instruction to be executed is the instruction right next to the branch
    - A la Frost's "Road Not Taken"
    - beq $1, $2, label (the branch is not taken if $1 != $2)
    - A conditional branch can be taken or not taken depending on how the condition evaluates at run time

6

Single-cycle instruction execution



Single-cycle instruction execution

- Clock cycle time must be large enough to accommodate the lengthiest instruction
  - This is typically the load instruction whose critical path includes all five stages: IF: fetch instruction, ID/RF: decode while reading reg[rs] and sign extending displacement, EX: add reg[rs] with sext(disp) to generate address, MEM: read dcache to get load data, WB: write load data back to reg[rt]
  - Not a good idea since many other instructions take smaller amount of time
    - Control transfer instructions require only three stages
    - Stores require only four stages
    - All ALU instructions can bypass MEM stage

8

Multi-cycle instruction execution

- Each stage takes one cycle to execute
  - Processor is now a five state FSM and the memory elements are sequential
  - Need flip-flops at stage boundaries and each stage is a combinational logic
  - Clock cycle time = latency of the longest stage
  - Most instructions take five cycles
    - ALU instructions do nothing in MEM stage, but will have to go through it if FSM state transition is implemented using an incrementer
    - Some instructions may take longer e.g., multiply/divide, load/store (in case of dcache miss)
    - Some instructions may take shorter time e.g., stores take four cycles, control transfer instructions take three cycles
  - Overall CPI depends on instruction mix

9

Multi-cycle instruction execution

- Example
  - IF: 2 ns, ID/RF: 1 ns, EX: 1 ns, DMEM: 3 ns, WB: 1 ns
  - Branch frequency: 20%
  - Store frequency: 10%
  - Multiply/divide frequency: 5%, latency: 30 ns
  - Total instruction count: 100
- Multi-cycle
  - Cycle time: 3 ns, frequency: 333 MHz
  - CPI: 0.2*3+0.1*4+0.05*10+0.65*5 = 4.75
  - Execution time: 100*4.75*3 ns = 1425 ns
- Single-cycle
  - Cycle time: 8 ns, frequency: 125 MHz
  - CPI: 1*0.95+ceil(30/8)*0.05 = 1.15
  - Execution time: 100*1.15*8 ns = 920 ns

10

Multi-cycle instruction execution

- Same example with a balanced pipeline
  - IF: 2 ns, ID/RF: 2 ns, EX: 2 ns, DMEM: 2 ns, WB: 2 ns
  - Branch frequency: 20%
  - Store frequency: 10%
  - Multiply/divide frequency: 5%, latency: 30 ns
  - Total instruction count: 100
- Multi-cycle
  - Cycle time: 2 ns, frequency: 500 MHz
  - CPI: 0.2*3+0.1*4+0.05*15+0.65*5 = 5
  - Execution time: 100*5*2 ns = 1000 ns
- Single-cycle
  - Cycle time: 10 ns, frequency: 100 MHz
  - CPI: 1*0.95+ceil(30/10)*0.05 = 1.1
  - Execution time: 100*1.1*10 ns = 1100 ns (worse than multi-cycle)

11

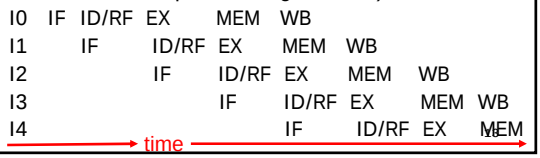
Pipelined instruction execution

- Observations from multi-cycle design
  - In the second cycle, I know if it is a branch; if not, start fetching the next instruction?
  - When the ALU is doing an addition (say), the decoder is sitting idle; can we use it for some other instruction?
  - In summary, exactly one stage is active at any point in time: wastes hardware resources
- Form a pipeline
  - Process five instructions in parallel
  - Each instruction is in a different stage of processing (called pipe stage)

12

Pipelined instruction execution

- Individual instruction latency is five cycles, but ideally can finish one instruction every cycle after the pipeline is filled up
  - Ideal CPI of 1.0 at the clock frequency of multi-cycle design
  - Execution time is ideally one-fifth of the multi-cycle design
    - Instruction throughput improves five times (number of instructions completed in a given time)



13

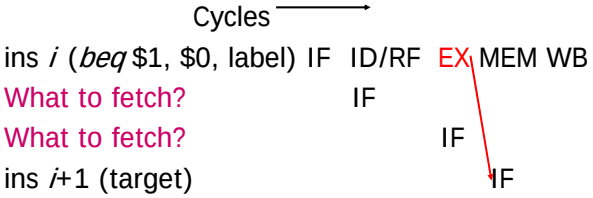
Pipelined instruction execution

- Gains
  - Extracting parallelism from a sequential instruction stream: known as instruction-level parallelism (ILP)
  - Can complete one instruction every cycle (ideally)
- Loss
  - Each pipe stage may get lengthened a little bit due to control overhead (skew time, setup time, propagation delay): limits the gain due to pipelining
  - Each instruction may take slightly longer for this reason
  - Bigger aggregate memory bandwidth: icache and dcache may miss in the same cycle
  - Pipeline hazards

14

Pipeline hazard: control

- Branches pose a problem

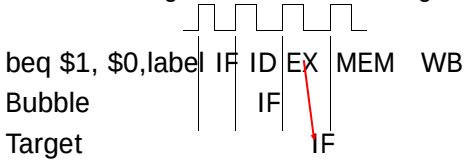


- Two pipeline bubbles: increases average CPI
- Can we reduce it to one bubble?

15

Branch delay slot

- MIPS R3000 has one bubble
  - Called branch delay slot
  - Exploit clock cycle phases
  - On the positive half compute branch condition
  - On the negative half fetch the target



- The PC update hardware (selection between target and next PC) works on the lower edge

16

Branch delay slot

- Can we utilize the branch delay slot?
  - The delay slot is always executed (irrespective of the fate of the branch)
    - Reason why branch target is PC+4+(sext(offset)<<2)
  - Boost instructions common to fall through and target paths to the delay slot or from earlier than the branch
  - Not always possible to find
  - Must boost something that does not alter the outcome of the computation
  - If the BD slot is filled with useful instruction then we don't lose anything in CPI; otherwise we pay a branch penalty of one cycle

17

Branch prediction

- Today all processors rely on branch predictors that observe the behavior of individual branches and learn to predict their future behavior
  - Makes it possible to infer the next instruction's PC even before the branch executes
  - Pipeline must be flushed on a wrong prediction

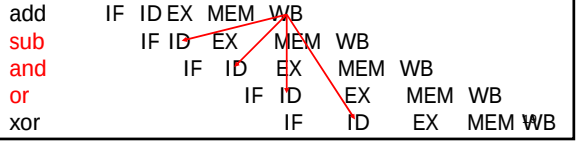
18

Pipeline hazard: data

- Pipelining disturbs the sequential thought-process
  - Data dependencies among instructions start to show up

```
add $1, $2, $3
sub $4, $1, $5
and $6, $1, $7
or  $8, $1, $9
xor $10, $1, $11
```

  - Result of add is needed by all instructions (RAW hazard: read after write hazard)



19

Pipeline hazard: data

- How to avoid increasing CPI?
  - Can we forward the correct value just in time?

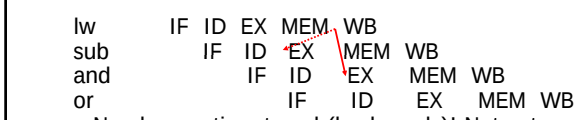
  - Read wrong value in ID/RF, but bypassed value overrides it (need a multiplexor for each ALU input to choose between the RF value and the bypassed value)

20

Pipeline hazard: data

- Can we always avoid stalling?

```
lw   $1, 0($2)
sub  $4, $1, $5
and  $6, $1, $7
or   $8, $1, $9
```



  - Need some time travel (backwards)! Not yet feasible!!
  - One option: hardware *pipeline interlock* to stall the *sub* by a cycle
  - Early generations of MIPS (Microprocessor without Interlocked Pipe Stages) had the compiler to fill the *load delay slot* with something independent or a NOP

21

Pipelined instruction execution

- Hazards can cause pipeline stalls and introduce bubbles in the pipeline depending on the pipeline organization
- Overall speedup of pipelining over multi-cycle non-pipelined implementation = number of pipe stages/(1 + average stall cycles per instruction)

22