Module 3: "Recap: Single-threaded Execution"

Lecture 5: "Pipelining and Hazards"

The Lecture Contains:

RECAP: SINGLE-THREADED EXECUTION

Previous Next

Module 3: "Recap: Single-threaded Execution"

Lecture 5: "Pipelining and Hazards"

Long history

- Starting from long cycle/multi-cycle execution
- Big leap: pipelining
  - Started with single issue
  - Matured into multiple issue
- Next leap: speculative execution
  - Out-of-order issue, in-order completion
- Today's microprocessors feature
  - Speculation at various levels during execution
  - Deep pipelining
  - Sophisticated branch prediction
  - And many more performance boosting hardware

Single-threaded execution

- Goal of a microprocessor
  - Given a sequential set of instructions it should execute them correctly as fast as possible
  - Correctness is guaranteed as long as the external world sees the execution in-order (i.e. sequential)
  - Within the processor it is okay to re-order the instructions as long as the changes to states are applied in-order
- Performance equation
  - Execution time = average CPI × number of instructions × cycle time

CPI equation: analysis

- To reduce the execution time we can try to lower one or more the three terms
- Reducing average CPI (cycles per instruction):
  - The starting point could be CPI=1
  - But complex arithmetic operations e.g. multiplication/division take more than a cycle
  - Memory operations take even longer
  - So normally average CPI is larger than 1
  - How to reduce CPI is the core of this lecture
- Reducing number of instructions
  - Better compiler, smart instruction set architecture (ISA)
- Reducing cycle time: faster clock

Life of an instruction

- Fetch from memory
- Decode/read (figure out the opcode, source and dest registers, read source registers)
- Execute (ALUs, address calculation for memory op)
- Memory access (for load/store)
- Writeback or commit (write result to destination reg)
- During execution the instruction may talk to
  - Register file (for reading source operands and writing results)
  - Cache hierarchy (for instruction fetch and for memory op)

Previous Next

Multi-cycle execution

- Simplest implementation
  - Assume each of five stages takes a cycle
  - Five cycles to execute an instruction
  - After instruction i finishes you start fetching instruction i+1
  - Without “long latency” instructions CPI is 5
- Alternative implementation
  - You could have a five times slower clock to accommodate all the logic within one cycle
  - Then you can say CPI is 1 excluding mult/div, mem op
  - But overall execution time really doesn’t change
- What can you do to lower the CPI?

Pipelining

- Simple observation
  - In the multi-cycle implementation when the ALU is executing, say, an add instruction the decoder is idle
  - Exactly one stage is active at any point in time
  - Wastage of hardware
- Solution: pipelining
  - Process five instructions in parallel
  - Each instruction is in a different stage of processing
  - Each stage is called a pipeline stage
  - Need registers between pipeline stages to hold partially processed instructions (called pipeline latches): why?

More on pipelining

- What do you gain?
  - Parallelism: called instruction-level parallelism (ILP)
  - Ideal CPI of 1 at the same clock speed as multi-cycle implementation: ideally 5 times reduction in execution time
- What are the problems?
  - Slightly more complex
  - Control and data hazards
  - These hazards put a limit on available ILP

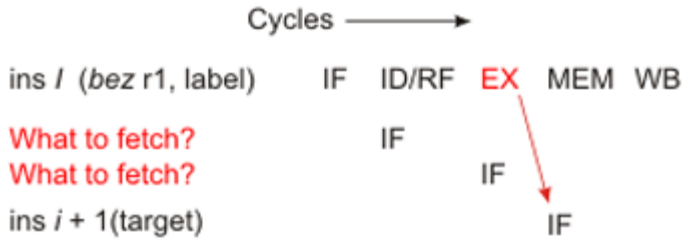
Previous Next

Module 3: "Recap: Single-threaded Execution"

Lecture 5: "Pipelining and Hazards"

Control hazard

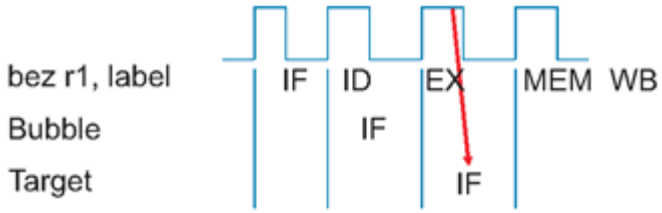
- Branches pose a problem



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

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
- Can we utilize the branch delay slot?
  - Ask the compiler guy
  - The delay slot is always executed (irrespective of the fate of the branch)
  - Boost instructions common to fall through and target paths to the delay slot
  - Not always possible to find
  - You have to be careful also
  - Must boost something that does not alter the outcome of fall-through or target basic blocks
  - 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

What else can we do?

- Branch prediction
  - We can put a branch target cache in the fetcher
  - Also called branch target buffer (BTB)
  - Use the lower bits of the instruction PC to index the BTB
  - Use the remaining bits to match the tag
  - In case of a hit the BTB tells you the target of the branch when it executed last time
  - You can hope that this is correct and start fetching from that predicted target provided by the BTB

- One cycle later you get the real target, compare with the predicted target, and throw away the fetched instruction in case of misprediction; keep going if predicted correctly

Branch prediction

- BTB will work great for
  - Loop branches
  - Subroutine calls
  - Unconditional branches
- Conditional branch prediction
  - Rather dynamic in nature
  - The last target is not very helpful in general (if-then-else)
  - Need a direction predictor (predicts taken or not taken)
  - Once that prediction is available we can compute the target
- Return address stack (RAS): push/pop interface

Previous Next

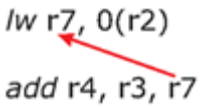
Data hazards

- Data dependency in instruction stream limits ILP
- True dependency (Read After Write: RAW)

- Need a bypass network to avoid losing cycles
- Without the bypass the fetching of subtraction would have to be delayed by three cycles
- This is an example of RAW hazard

More on RAW

- The most problematic dependencies involve memory ops
- The memory ops may take a large number of cycles to return the value (if missed in cache)



- This type of dependencies is the primary cause of increase in CPI and lower ILP

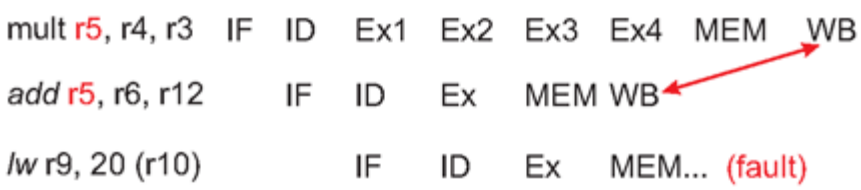
Multi-cycle EX stage

- Thus far we have assumed a single cycle EX
- Consider multiplication and division
- Assume a four-cycle multiplication unit: mult r5, r4, r3 IF ID EX1 EX2 EX3 EX4 MEM WB
- Normally the multiplier is separate
- So the next instruction can start executing when mult moves to EX2 stage and, in fact, can finish before mult
    - More data hazards

Previous Next

WAW hazard

- Write After Write (WAW)



- The problem: out-of-order completion
- The final value in r5 will nullify the effect of the add instruction
- The bigger issue: precise exception is violated
- Next load instruction raises an exception (may be due to page fault)
- You handle the exception and start from the load
- But value in r5 does not reflect precise state
- Solution: disallow out-of-order completion

Overall CPI

- CPI = 1.0 + pipeline overhead
- Pipeline overhead comes from
  - Branch penalty (useless delay slots, mispredictions)
  - True data dependencies
  - Multi-cycle instructions (load/store, mult/div)
  - Other data hazards
- So to boost CPI further
  - Need to have better branch prediction
  - Need to hide latency of memory ops, mult/div

Multiple issue

- Thus far we have assumed that at most one instruction gets advanced to EX stage every cycle
- If we have four ALUs we can issue four independent instructions every cycle
- This is called superscalar execution
- Ideally CPI should go down by a factor equal to issue width (more parallelism)
- Extra hardware needed:
  - Wider fetch to keep the ALUs fed
  - More decode bandwidth, more register file ports; decoded instructions are put in an issue queue
  - Selection of independent instructions for issue
  - In-order completion

Previous Next