Chapter 4: MIPS instructions and datapath, pipelines, hazards, dependencies, branching, datapath with floating-point operations, instruction-level parallelism and real processor design principles (Lectures 9 – 16)

Lectures 9 & 10 – MIPS Instructions and Datapath

Elements in the datapath

* Program counter
* Instruction memory
* ALU
* Data memory and Register file

R-Type Instructions

* Read two registers, write one
* Arithmetic (add $rd, $rs1, $rs2)
* Logical (and $rd, $rs1, $rs2)

J-type Instructions

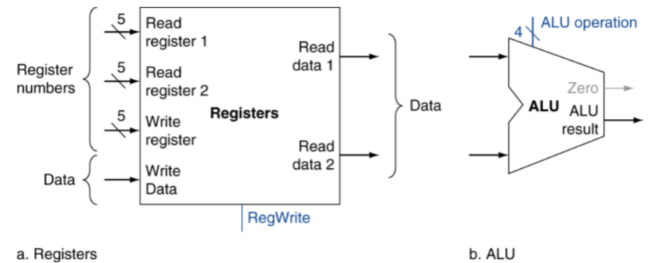
* No register operand
* Jump to an address

I-Type Instructions

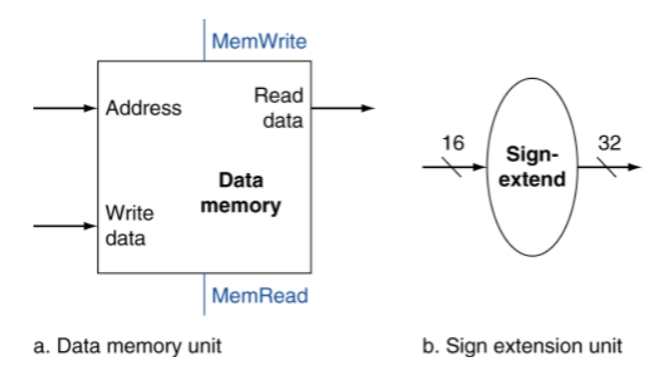
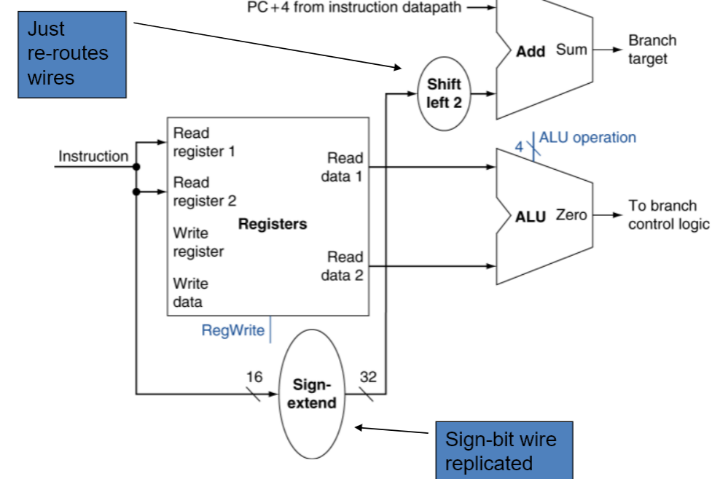
* Two register operands
* Arithmetic computation with constants (immediate)
  + Read one register, write one register
  + Add $rt, $rs, 10
* Branch (if-else, for, while, switch etc)
  + Read two registers
  + Beq $t0, $t1, Target (Branch to target if $t0 == $t1)
* Load data from memory or store data to memory
  + Load: read one register or write one register
    - Lw $8, 12($16) ($8 <- mem($16 + 12))
  + Store: Read two registers
    - Sw $8, 12($16) ($8 -> mem($16 + 12))

format of each type of instruction

Process of instruction execution

* PC -> instruction memory, fetch instruction
* Register numbers -> register file, read registers
* Depending on instruction class –
  + Use ALU to calculate
  + Memory address for load/store
  + Branch target address
* Access data memory for load/store
* PC <- target address or PC + 4

Elements in the datapath utilized for specific instructions

* R-type
  + Read two register operands
  + Perform arithmetic / logical operation
  + Write register result
* load / store
  + Read register operands
  + Calculate address using 16-bit offset (constant)
    - Use ALU, but sign-extent offset
  + Load: Read memory and update register
  + Store: Write register value to memory
* Branch
  + Read register operands
  + Compare operands
    - Use ALU, subtract and check zero output
  + Calculate target address
    - Sign-extend displacement
    - Shift left 2 places (word displacement)
    - Add to PC + 4
      * Already calculated by instruction fetch

Target address calculation for branch instructions

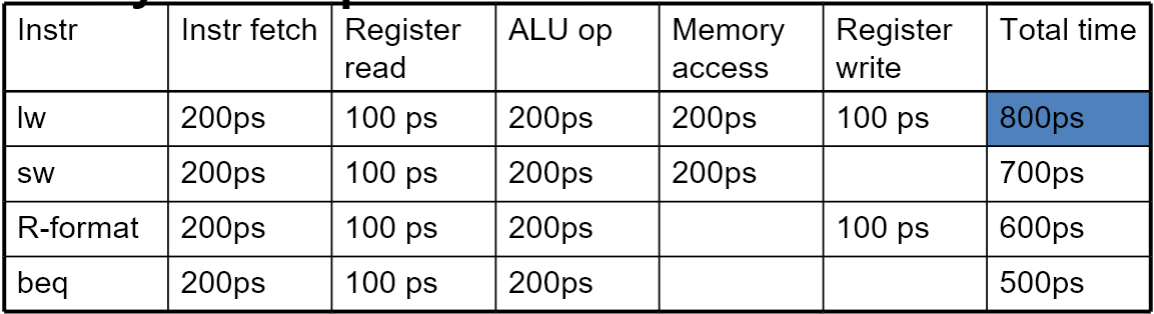
Datapath with control lines

Performance issues with single-cycle datapath

* Longest delay determines clock period
  + Critical path: load instruction
  + Instruction memory -> register file -> ALU -> data memory -> register file
* Not feasible to very period for different instructions
* Violates design principle
  + Making the common case fast

Lecture 11 – Pipeline Datapath

Five stages: One step per stage:

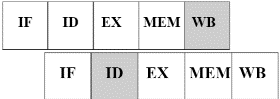
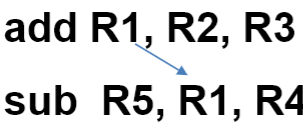
* IF: **Instruction** **fetch** from memory
* ID: **Instruction** **decode** & register read
* EX: **Execute** operation or calculate address
* MEM: Access **memory** operand
* WB: **Write** result **back** to register

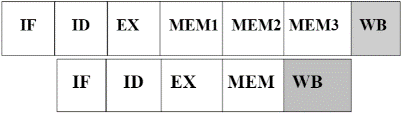
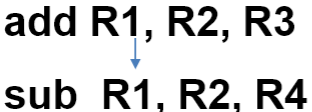
Performance of single-cycle instructions vs pipelined instructions

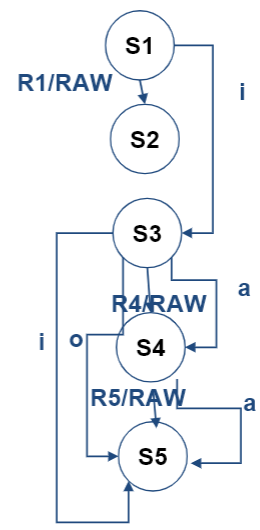
* Assume 100ps for read / write, 200ps for other
* Speedup:
  + If stages are balanced: Nonpipelined time / pipelined time = number of stages
  + If not balanced, speedup less. Speed up due to increased throughput (latency does not decrease)

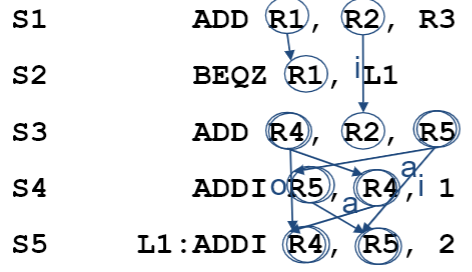
Performance of pipeline, time for stage in pipeline to complete vs. entire instruction, latency vs. throughput

Pipeline diagrams

Data hazards, data hazard types (RAW / WAW / WAR)

* RAW – Read After Write (true dependence)
  + J tries to read operand before I writes it
* WAR – Write After Read (antidependence, a)
  + J tries to write operand before I reads it
* WAW – Write after Write (output dependence, o)
  + J tries to write operand before I writes it
* RAR – Read After Read (Input dependence, i)
  + I and j read the value of some variable

Strategies for avoiding hazards (stalling vs using additional hardware to avoid the hazard)

* Stall (pause by inserting no-op)
  + Avoids overlap
  + Slows things down
* Add hardware to avoid hazard

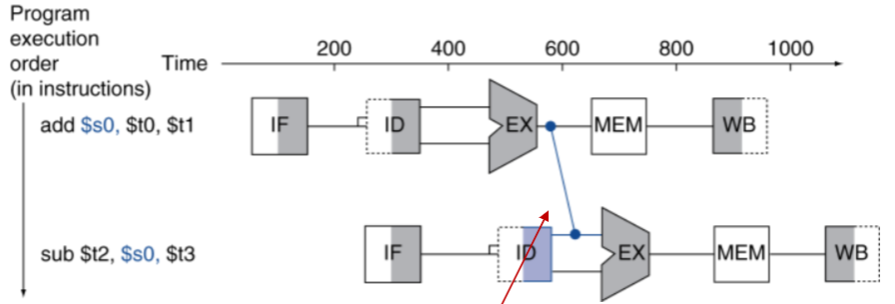
Data dependency diagrams

Lecture 12 – Pipeline Hazards

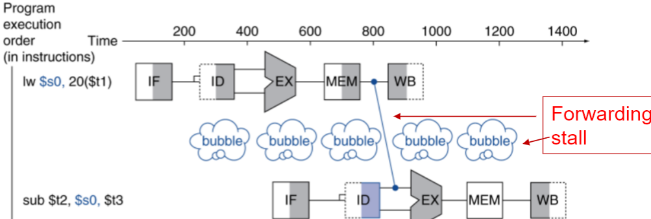
Structure hazard

* Conflict for use of a resource
* Single memory MIPS pipeline:
  + Load/store requires data access
  + Instruction fetch would have to stall for that cycle
    - Causing pipeline “bubble”
* Hence, pipelined datapaths require separate instruction / data memories
  + Or separate instruction / data caches

Data:

* Instruction depends on completion of previous instruction
* Stall pipeline by insertion NO-OP (0x0000000000)
* Instruction execution delayed, performance poor

Avoiding data hazards using forwarding

* Use result when computed
* Don’t wait for it to be stored in register
* Requires extra connections in the datapath

Load-use data hazards

* Can’t always avoid stalls, if value not computed when needed
* Can’t forward backward in time time

Code scheduling to avoid stalls

* Reorder code to avoid use of load result in next instruction

Control hazards and branching

* Branch determines flow of control
* Fetching next instruction depends on branch outcome
  + Following instruction may not be the one to be executed
  + Pipeline can’t always fetch correct instruction
  + Still working on ID stage of branch
* In MIPS Pipeline: Need to compare registers and compute target early in pipeline (add hardware to do it in ID stage)

Stall on branch (special treatment – execute target address by the end of the ID (2nd) stage)

Branch prediction (only stall when prediction is wrong)

* Longer pipelines can’t determine branch outcome early
  + Stall penalty become unacceptable
* Predict outcome of branch
  + Stall if prediction wrong
* In MIPS Pipeline
  + Can predict branches not taken
    - Pipeline executes as normal if prediction is correct
  + Fetch instruction after branch with no delay

Lecture 13 – Branch Hazards and Branch Prediction & Floating-Point Pipeline

Data hazards for branches (ALU instructions before branch instruction, load/store instruction before branch)

Reducing branch penalty

* Predict branch/jump instructions and branch direction
* Predict branch / jump target address
* Speculatively execute instructions along predicted path

Branch direction prediction (static prediction vs dynamic prediction)

* Static:
  + Compiler-determined
  + Fixed for life of program
* Dynamic:
  + Prediction changes as program behavior changes
  + Implemented in hardware
  + Algorithm based on branch history
    - Predict if branched last time
    - Don’t predict if not

Dynamic branch prediction: single bit vs. 2-bit branch prediction

* Single bit: Predict what did last time
* 2-bit: Changed after being wrong twice

Pipeline with floating-point instructions

Latency and initiation interval of instructions

Pipeline cycle diagrams containing both integer and floating-point instructions

WAW data hazard – out-of-order writebacks

Lecture 14 & 15 – Instruction Level Parallelism

Deeper pipeline

* Less work per stage -> shorter clock cycle

Multiple issue

* Replicate pipeline stages -> multiple pipelines
* Start multiple instructions per clock cycle
* CPI < 1, so used IPC. IPC > 1
* 4 GHz 4-way multiple issue
  + 16 billion IPS, peak CPI = .25, peak IPC = 4
* Dependencies reduce this in practice
* Static MI
  + Compiler groups instructions to be issued together
  + Packages them into “issue slots”
    - Instruction in same slot issued on same cycle
  + Compiler detects and avoids hazards
* Dynamic MI
  + CPU examines instruction stream and chooses instructions to issue each cycle
  + Compiler can help by reordering instructions
  + CPU resolves hazards using techniques at runtime

Compiler speculation

* Compiler can reorder instructions
* E.g. move load before branch
* Can include “fix-up” instructions to recover from incorrect guess

Hardware speculation

* Can look ahead for instructions to execute
* Buffer results until it determines they are needed
* Flush buffers on incorrect speculation

Scheduling static multiple issue with 1 ALU/branch + 1 load/store instruction issue packet

Changes to the pipeline with static multiple issue

Data hazards with a multiple issue pipeline

Performance improvement with multiple issue and impact of loop unrolling

Dynamically scheduled multiple issue

Lecture 16 – Real Processor Design Examples

Power efficiency (with respect to clock rate, pipeline depth, issue width and number of cores)

Comparison of ARM Cortex A8 vs. Intel Core i7

* Specifications
* Pipeline layout
* Performance

Performance impact of using code optimization techniques

Fallacies a pipelined datapath

* Easy
* Independent of technology

Pitfalls of pipelined datapath

* Poor ISA design makes harder