Computer Architecture

EECS 470

University of Michigan

Harrison Centner

Prof. Jon Beaumont

February 13, 2023

Contents

INTRODUCTION & MOTIVATION

"Computer architecture is a contract between hardware and software."

Textbooks:

(i) *Computer Architecture* by Patterson & Hennessey

Content:

(a) Instruction Set Architecture design

instructions,

memory management and protection,

interrupts, and

floating point.

(b) Computer Organization also called microarchitecture.

number & location of functional units,

pipeline & cache configuration, and

datapath connections.

(c) Implementation:

low-level circuits.

In 470 we want to understand the processors in modern desktops and servers not microwaves (which would be 370). At a high level, (1) dynamic out-of-order processing, (2) memory architecture, and (3) multicore & multiprocessor issues. At a low level, (1) microarchitecture and out-of-order machines and (2) cache designs.

Project 3 will take at least 20 hours and ends up being about 10 lines of code to modify.

Power is the instantaneous usage of energy. **Power efficiency** is important because you don't want your phone to light your pants on fire. **Energy efficiency** is important because energy is expensive.

FUNDAMENTAL CONCEPTS

Speed

There are three ways to speed up tasks (1) decrease workload, (2) increase throughput, or (3) increase parallelism.

We can use memoization, exploit locality, and parallelization.

Most of 470 will focus on parallelization. Including pipelining, speculative execution, dynamic scheduling, register renaming, branch prediction, superscaling, multiprocessing, VLIW, and more.

Proposition. (Amdahl's Law)
Suppose an enhancement speeds up a fraction f of a task by a factor S. Then the **speedup** is given by

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{S}}$$

The time it will take is

$$t = t_0 \left((1-f) + \frac{f}{S} \right)$$

Definition. (Parallelism)
Parallelism is the amount of independent sub-tasks available.
Work is the time to complete a computation on a sequential system and we write T_1. We write T_n to be the amount of time to complete a computation given n parallel workers.
The **Critical Path** is given by T_∞.
The average parallelism is given by $\frac{T_1}{T_\infty}$.

Moore's Law has continued to hold: transistors per die area has doubled every two years. power, frequency, and single thread performance are approaching asymptotes caused by the limits of air cooling of chips.

Definition. (Performance)
Latency is the time to finish a fixed task. **Throughput** is the number of tasks in a fixed time. Parallelism speeds throughput. Cars have lower latency; buses has higher throughput.

Latency is more relevant to measure a single benchmark.
$\text{Latency}(A+B) = \text{Latency}(A) + \text{Latency}(B)$.
$\text{Throughput}(A+B) = \frac{1}{\frac{1}{\text{Throughput}(A)} + \frac{1}{\text{Throughput}(B)}}$.

Definition. (Averages)
Arithmetic mean is good for latency.

$$\frac{1}{n}\sum_{i\in[n]} a_i$$

Harmonic mean is good for throughput.

$$n\left(\sum_{i\in[n]}\frac{1}{a_i}\right)^{-1}$$

Geometric mean is good for speedup.

$$\sqrt[n]{\prod_{i\in[n]} a_i}$$

Law. (Iron Law of Performance)
We will be working to improve cycles per instruction.

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

Compiler optimizations can make MIPS count go down because there are more dependencies etc.

Instruction Set Architecture

The ISA is a contract between software and hardware containing functional definitions and precise descriptions of how to invoke and access them. The ISA components include:

(i) Programmer-visible state (program counter, general purpose register, memory, control reg)

(ii) Programmer-visible behaviors (state transitions)

(iii) A binary encoding of instructions.

ISAs last 25+ years generally because of software costs.

RISC and **CISC** are two classifications of ISAs. CISC ie easy for assembly level programmers and good for code density → complex hardware. RISC increases instructions per program (hopefully not much if the compiler is smart) and perhaps improves the clock cycle (aggressive implementations are allowed by simpler instructions).

Nowadays, CISC has become much less popular. Internally CISC processors decompose instructions into RISC-like instructions.

A good ISA requires:

(i) Programmability

(ii) Implementability

(iii) Compatibility.

Instructions for an ISA are load/store, conditional branches, add. NAND with load/store on its own is functionally complete.

Power & Energy

We care about power consumptions because power usage impacts device temperature, energy usage, cost, and environmental concerns.

Power density in a processor is about the same as in a nuclear reactor.

Dynamic power is the power used to switch the transistor states. **Static power** relates to leakage. As transistors become smaller quantum electron tunneling is causing more static power leakage.

Law. (Approximation of Power)
Power is approximately given by

$$\frac{1}{2}C \cdot V^2 \cdot A \cdot f$$

C is capacitance, V is the supply voltage, A is the activity factor, and f is the frequency.
As architects we have the most control over V and f.

There is a linear relationship between voltage and frequency. So reducing power requires a cubic reduction in voltage frequency. This is the reasoning for the trend to multicore processors.

The units of power are **Watts** and determines the battery life in hours as well as packaging limits.

Energy is the integral of power and its units are **Joules**.

Pipelining

This is the traditional in-order five stage pipeline. Recall that before pipelining there was the **single-cycle** (in a single clock cycle complete a single instruction). This results in poor performance because all instructions are performed at the instruction with the longest latency. Instead we develop **multi-cycle** which as a short clock period but high cycles per instruction (CPI). Multi-cycle has potentially better overall latency.

Ultimately no processors use single-cycle designs.

We need to parallelize ... **superscalar** and **pipelining** are the two methods. Superscalar hardware runs two instructions at once, but has twice the hardware overhead. Pipelining overlaps instruction execution when possible.

Pipelining improves throughput at the expense of latency. We need to clock pipeline registers (sometimes called latches) independently will add delays. In general, bandwidth will increase inversely with the number of pipeline stages.

The five stages of our pipeline are:

(1) **F**etch. Use the Program Counter (PC) to fetch the next instruction from memory. ++PC. Place the instruction bits and PC+1 value into the IF/ID register.

(2) **D**ecode. Place register names from the instruction bits into the register file. Store the contents of regA, regB, and PC+1 into the ID/EX register. Use a control ROM to determine behavior.

(3) **E**xecute. Store data from the DestReg and Data into the register file (wires from **W**rite-Back stage). Perform ALU operations; inputs may be register contents or control signals (immediates). Store ALU result, regB contents, and PC+1+offset in EX/Mem register.

(4) **M**emory. Put ALU result into the address port and regB contents into the data port of Data Memory. Pass ALU result, memory read data, and control signals to to MEM/WB register. Determine if a branch should be taken and pass the PC+1+offset back to **F**etch stage.

(5) **W**rite-back. Mux the ALU result and Memory read data to send back to data input of register file. Pass control signals back to the destination register specifier.

Hazards & Forwarding

Arbitrarily deep pipelines are not possible because of data dependencies.

Definition. (Hazard)
A **Hazard** refers to a data dependency in which a failure to act will produce incorrect computation. **Pipeline hazards** are caused by close together data dependencies. **Structural hazards** result from poor hardware choices. **Control hazards** are hazards that result from branching.

Hazard resolution can be performed in software (static) or in hardware at run time (dynamic). One method is **pipeline interlock** or **forwarding** which provides dynamic hazard resolution through detection and enforcement of dependencies.

There are three naive main methods to deal with data hazards:

(1) Avoidance the programmer or compiler inserts noops between dependent instructions. Results in slower performance, larger program size, and breaks backwards compatibility (not agnostic to the microarchitectural implementations).

(2) Detect & Stall a three bit comparator will pass noops to execute stage when a dependent instruction is detected. Results in slower performance and increases CPIs upon hazards.

(3) Forwarding adds data paths to pass ALU result to future computations.

There are three main ways to deal with control hazards:

(1) Remove branches with branch-less programming

(2) Detect & Stall means that we stall until the branch is resolved.

(3) Speculate & Squash assume that a branch is (not) taken and continue to execute subsequent instructions. If our guess was incorrect, then replace incorrect instructions with noops. Branch prediction can be improved by exploiting temporal locality. Modern processors use complicated branch predictors.

DYNAMIC SCHEDULING

Scalar pipelines are fundamentally limited by the "Flynn Bottleneck" which is the fact that the instructions per clock (IPC) is at most 1. Unification into a single pipeline causes long latency for each instruction. Performance is lost in rigid in-order pipelines by unnecessary stalls.

A scalar pipeline with D pipeline stages can have peak instruction parallelism of D. Operation latency and peak IPC are 1.

A **superscalar** pipeline has multiple pipelines, so there are duplicates of each pipeline stage. Instruction parallelism peaks at $D \times N$ where D is the pipeline depth and N is the pipeline width. When $D \times N$ approaches average distance between dependent instructions the superscalar performance degrades and forwarding is no longer effective.$D \times N$ approaches average distance between dependent instructions the superscalar performance degrades and forwarding is no longer effective.

Definition. (Instruction Level Parallelism)
ILP is a number that represents the average number of instructions that can be processed in parallel. An ILP of 1 means that the code must be executed serially. ILP can be arbitrarily large.

Definition. (Dynamic Scheduling)
Dynamic scheduling or **out-of-order processing** occurs when a processor fetches/decodes instructions into a *instruction buffer* (or *windor* or *scheduler*) which allows the processor to reorder instructions. Instructions can leave the buffer when ready in arbitrary order.

Immediately many problems present including new types of hazards:

(1) Read-after-write (RAW) occurs when we must wait to read for a previous write instruction.

(2) Write-after-read (WAR) also called "anti-dependencies" occur when we must wait to write to a previous read instruction.

(3) Write-after-write (WAW) also called "output dependencies" when an earlier instruction overwrites a subsequent write instruction.

These latter two hazards occur because we are reusing registers (i.e. they are structural dependencies). It might be a good idea to increase the number of registers.

Definition. (Register Renaming)
Given infinite registers WAR/WAW hazards can be eliminated. The processor dynamically renames instructions to use new registers which are not visible to the programmer.

A **MapTable** is used to track which opcode registers correspond to physical registers.

Current machines generally can fit 100+ instructions into the instruction window.

Scoreboard Scheduling

Scoreboarding is the first OoO (out of order execution) method, it uses no register renaming. We call the instruction buffer the **insn buffer**.

We partition the **D**ecode stage in two:[1]

(1) **D**ispatch where we accumulate instructions into the insn buffer in order. A new structural hazard occurs if the insn buffer is full. A stall back-propagates to younger instructions.

(2) **I**ssue where we send instructions from the insn buffer to the execution units. Occurs in an OoO manner. Wait does not back-propagate to younger instructions.

Scoreboard Scheduling uses a centralized control scheme in which instruction statuses are explicitly tracked. The insn buffer is called the **Functional Unit Status Table** or (**FUST**). No register renaming occurs.

In our "Simple Scoreboard" there are five functional units each of which has 1 ALU, 1 load, 1 store, 2 floating points (3-cycle, pipelined).

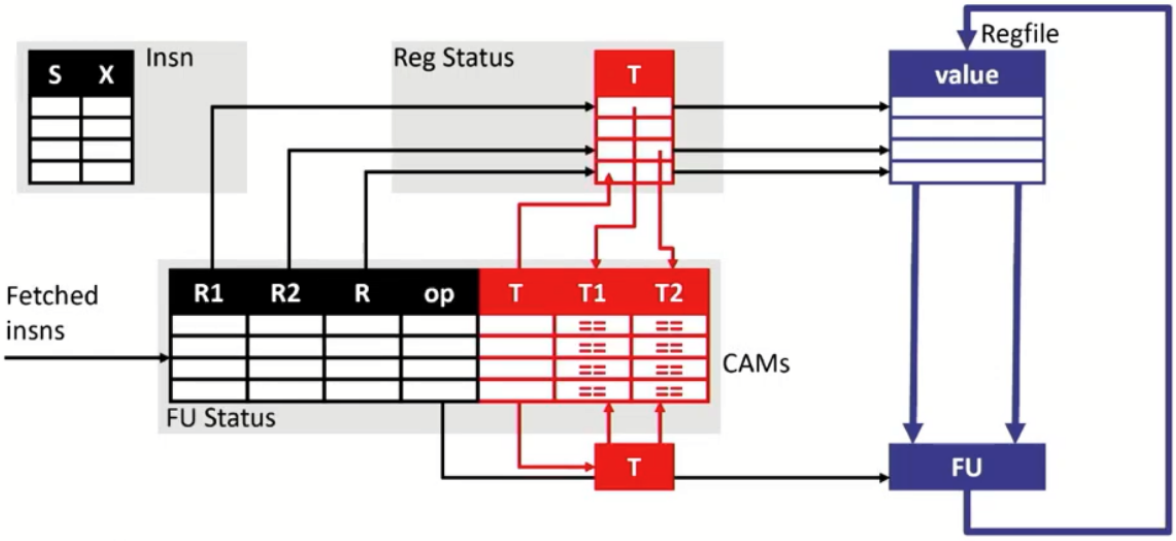

Figure 1: Simple Scoreboard Data Structures.

Without register renaming we will need to stall for WAW hazards. The processor needs to choose an issue policy (or a priority selector) to choose which instruction to execute if multiple are ready.

Upon WAR hazards we will need to clear the Reg Status entry.

Scoreboard Redux has cheap hardware and pretty good performance. Unfortunately our scheme does not include bypassing, dealing with branch mispredicts, and a limited scheduling scope.

[1] Confusingly, some computer architects swap the names of these stages.

Tomasulo's Algorithm

Scoreboarding does not do register renaming, Tomasulo's algorithm implements register renaming eliminating WAR and WAW hazards. **Architectural registers** are available to programmers and can be named. The first implementation was on the IBM 360/91 in 1967.

Most implementations use a MapTable which is an SRAM indexed by name (a look up table). We call the instruction buffer the **reservation stations** (**RS**). The **common data bus** (**CDB**) broadcasts results to the **RS**.

In our "Simple Tomasulo" there are five **RS** each of which has 1 ALU, 1 load, 1 store, 2 floating points (3-cycle, pipelined).

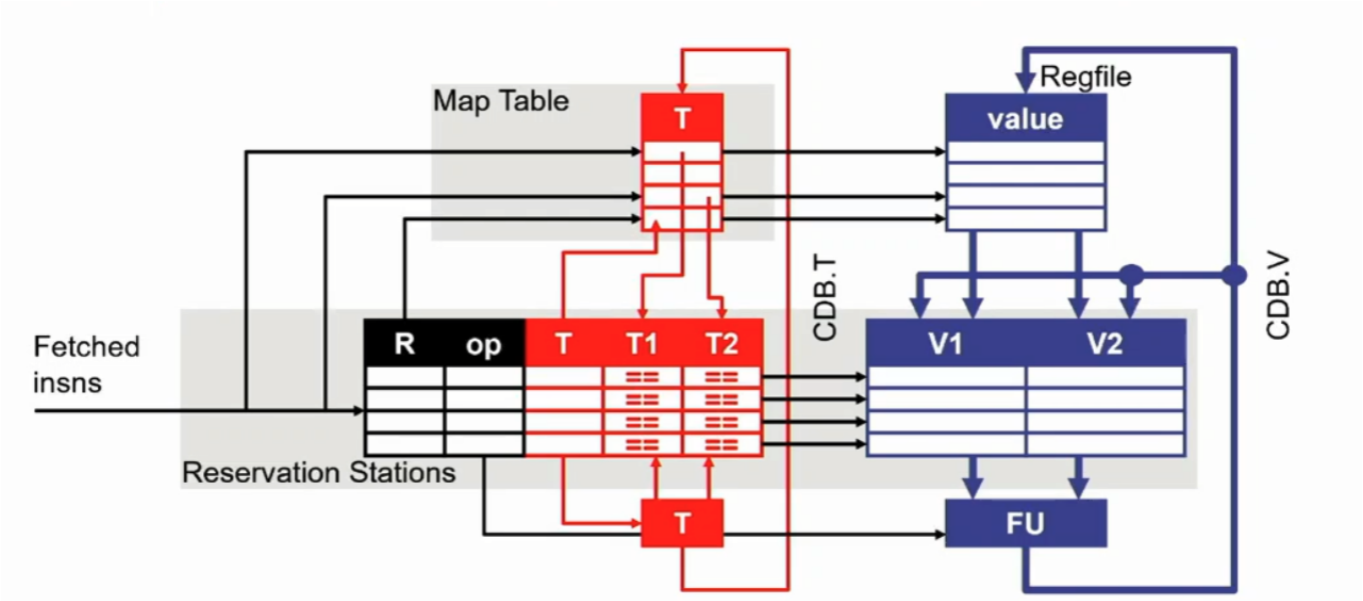

Figure 2: Simple Tomasulo Data Structures

Precise State with P6

Precise state is the ability to abort and restart a program after any instruction.

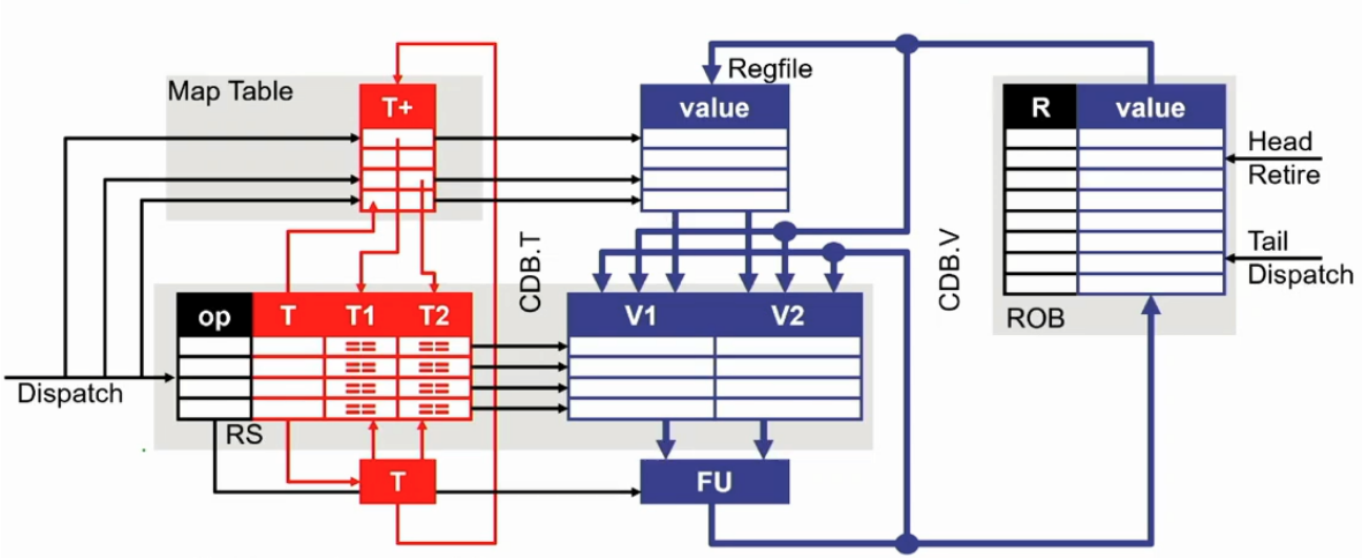

Figure 3: P6 Data Structures

Precise State with MIPS R10K

Precise state is a feature of programmability. Modern systems are expected to support features like exceptions.

Memory Disambiguation

Previously we have ignored memory instructions. Memory disambiguation provides a new hardware design called a load store queue which allow for out of order memory operation. Up until now we've discussed mostly register data flow and now we turn to memory operations. If you choose not to implement this in the final project, memory operations will likely be a bottleneck on your optimization.

Modern architectures support base + displacement memory operations, which means we need to execute an instruction before we know the address it writes to. We can guarantee that instructions are decoded in order, but we want to execute out of order. Therefore, memory operation dependencies are revealed until the execution stage. Data dependencies have ambiguity.

Because memory space is massive, we cannot perform memory renaming like we did for registers.

Naive Approach: Only issue a memory operation when all older memory operations have finished. This requires us to implement a load-store queue (basically a ROB but for memory ops).

Store-to-load-forwarding: Detect data dependencies and forward loads to memory before dependent (older) stores retire. We will effectively at a new functional unit that is a Store Queue (SQ) with 2 register inputs (addr, data in) one register output (data out) and 1 non-register input (load position). Addresses should be associatively searchable. Heuristically the size should be about $\frac{1}{5}$ the size of the ROB.

A conservative load scheduler could upon dispatch we will store the current store queue tail as a the "store position." We will need "age logic" to select the youngest store that is older than load. This means that loads must execute in-order.

Instead we can use speculative store-to-load forwarding. This means that we speculate there isn't an alias for a given load and issue to memory now. If we were wrong we will squash instructions and rewind. In practice about $\frac{1}{10}$ of loads are forwarded from stores. This means we need a store queue and a load queue.

A split load store queue has more complicated description, scalable, but easier hardware. A unified load store queue is simpler to describe, less scalable, but requires more expensive hardware (simply a $n \times n$ comparator matrix).

This concludes the class portion on OoO execution. □

INSTRUCTION FLOW

Branch Prediction

OoO processing speeds Register data flow. Branch prediction and target prediction speeds instruction flow.

General register comparison must be completed after an ALU computation. PC-relative branches can occur in Decode.

Superscalar designs leads to fragmentation of issue lines.

Branching strategies:

(a) Predict not taken.

(b) Software prediction. Allow a bit set by the compiler to determine speculative execution behavior (can use profiling).

(c) Static prediction (constant prediction).

(d) Predict based on branch offsets (i.e. predict not taken for positive offsets and negative offsets for predict taken).

(e) Dynamic resolution. Strategies include Branch Target Buffer (BTB) with RAS, Branch Direction Predictors.

Branch target buffers store a previous branch instructions. On a cache hit predict taken and a cache miss predict not taken. Aliasing on a tag match is OK because we don't need to be right.

BTBs work well for direct conditional branches, calls, & unconditional jumps. BTBs work well for indirect calls resulting from dynamically linked functions (e.g. system calls) but only ok for those from virtual functions (but we probably couldn't do much better.

A Return Address Stack (RAS) works in conjunction with a BTB and maintains a LIFO buffer that tracks whether a line in the BTB was a return or a conditional branch.

A branch direction predictor can be implemented with stateful branch predictors. A two-bit saturating counter adds "hysteresis" to the counter (we don't change our mind too much). A two-level predictor uses the PC to index into a branch history table each entry of which has a simpler counter.