

Computer Architecture Comprehensive Exam Review

Jon Su, Philip Guo, Ewen Cheslack-Postava

Performance (Hennessy & Patterson Ch. 2)

Performance = $1 / \text{Execution Time}$

Increase Performance = Decrease Execution Time

Speedup:

Speedup of A over B is $(\text{Time}(B) / \text{Time}(A))$.

Performance Components:

IC – instruction count

-limited by ISA

CPI – clock cycles per instruction ($\text{IPC} = \text{instructions per cycle} = 1/\text{CPI}$)

-affected by memory system, processor structure, mix of instructions

-varies by application

CC – clock cycle time (seconds / cycle)

-fixed by implementation

Performance Equation

Time = IC * CPI * CC

Amdahl's Law

Overall speedup of improvement that gives speedup of S on fraction P of computation is $1 / ((1-P) + P/S)$

This is just the law of diminishing returns: if you speedup a part of the system by 100x, but that part is only used 1% of the time, your overall speedup is only ~ 1.01 , or about 1% faster. Continuing to improve that part of the system won't have much effect. (Corresponds to design principle – make the common case fast).

Performance Pitfalls and Fallacies

P: Expecting improvement of one aspect of a machine to increase performance by an amount proportional to the size of the improvement. (Note: Amdahl's Law)

F: Hardware independent metrics predict performance. (Ex. Code size as measure of speed)

P: Using MIPS as a performance metrics. Problems with MIPS:

1. Specifies instruction execution rate but not the capabilities
2. MIPS varies between programs on the same computer
3. MIPS can vary inversely with performance

F: Synthetic benchmarks predict performance. (Often have unrealistic patterns that are easily optimized or exploitable by benchmark specific optimization)

P: Using the arithmetic mean of normalized execution times to predict performance. (Result will depend on choice of reference machine, instead use geometric mean for normalized times).

F: The geometric mean of execution time ratios is proportional to total execution time. (Violates fundamental principle of performance measurement, they do not predict execution time.)

ISA (Hennessy & Patterson Ch. 3)

- 32 bit words
- 32 word-size registers

Load/Store – indexed, alignment restriction – can only load aligned words

Name	#	Usage	Callee Preserved
\$zero	0	Always zero	N/A
\$at	1	Assembler temp	No
\$v0..\$v1	2..3	Return values	No
\$a0..\$a3	4..7	Arguments	No
\$t0..\$t7	8..15	Temporaries	No
\$s0..\$s7	16..23	Saved	Yes
\$t8..\$t9	24..25	Temporaries	No
\$k0..\$k1	26..27	OS use	No
\$gp	28	Global pointer	Yes
\$sp	29	Stack pointer	Yes
\$fp	30	Frame pointer	Yes
\$ra	31	Return address	N/A

Instruction Formats

R-type (Register)

OpCode	SrcReg1	SrcReg2	DestReg	Shift amt	Function Code
6	5	5	5	5	6

I-type (Immediate)

OpCode	SrcReg	DestReg	Imm/Offset
6	5	5	16

J-type (Jumps)

OpCode	Address-offset
6	26

Too many to list all here, see

http://en.wikipedia.org/wiki/MIPS_architecture#Summary_of_R3000_instruction_set for a full table.

Note, any address offsets are **word** offsets.

Functions

- Function Outline
 1. Prologue
 - a) Save \$s* registers to be used on the stack
 2. Body
 - a) Save return values in \$v0..\$v1
 3. Epilogue
 - a) Restore values saved on stack to registers
 - b) Restore \$sp to old value
 - c) Return to caller with “jr \$ra”

- Leaf doesn't call any other functions, doesn't need to save any caller saved registers
- Nested calls other procedures, must save any \$t*, \$a*, and \$v* registers it will need after call
- \$fp – frame pointer – points to the current stack frame
- \$sp – stack pointer, current stack position, adjusted for saved argument register, saved return address, saved \$s registers, local arrays and structures
- \$gp – global pointer, used for accessing globals

Addressing Modes

Register addressing	R-type add
Base or displacement addressing	I-type ld/st
Immediate addressing	I-type addi
PC-relative addressing	I-type beq/bne
Pseudo-direct addressing	J-type j/jal

Fallacies and Pitfalls

F: More powerful instructions mean higher performance

F: Write in assembly language to obtain the highest performance

P: Forgetting that sequential word address in machines with byte addressing do not differ by one

P: Using a pointer to an automatic variable outside its defining function

Design Principles

1. Simplicity favors regularity
2. Smaller is faster
3. Good design demands good compromises
4. Make the common case fast

Pipelining

Overview

- Work can be divided up into discrete stages
- Analogies – doing laundry, automobile assembly line
- Pipelining aims to maximize utilization of resource by keeping as many stages busy as possible
- Increases throughput – the amount of things done per unit time
 - Throughput with pipelining is limited by the SLOWEST stage
 - In contrast, throughput without pipelining considers the SUM of all stages
- Does NOT improve latency – the amount of time before one thing gets done (because it still must go through all the stages of the pipeline).

MIPS Pipeline

- 5 stages, each stage takes 1 clock cycle to execute
- Abbreviation used on exams: F, D, X, M, W
 1. Instruction Fetch (F)
 - a) Fetch instruction from instruction memory (cache)
 2. Instruction Decode (D)
 - a) Decode instruction and set the appropriate pins of the various control circuits in the CPU
 - b) Read the values of relevant registers from the register file
 3. Execute (X)

- a) Perform arithmetic calculation in the ALU
- b) Note that for certain instructions, like memory references, this stage calculates a memory address
- 4. Memory (M)
 - a) Load instruction: Read data from memory
 - b) Store instruction: Write data to memory
 - c) Assume that this completes in 1 cycle because there is a cache hit during memory access; otherwise will stall for many cycles
- 5. Write Back (W)
 - a) Write data (e.g. The result of an ALU calculation) to register in the register file
- Every instruction takes up all 5 stages, but for some kinds of instructions, some stages will be idle (e.g. for 'add', there is nothing to do for the memory stage)

Pipeline Hazards

- Why can't the pipeline perform at maximum speed, having all instructions fill up all stages at all times?
- Because of several types of hazards
 - Structural hazard – hardware doesn't have support for doing several of the same thing at once
 - Data hazard – later instructions depend on data that isn't yet ready from earlier instructions
 - Control hazard – from branch and jump instructions
- Hazards slow down the pipeline by forcing stalls on certain clock cycles (or, equivalently, the insertion of a NOP)
- However, hazards can often be eliminated by adding additional hardware and cleverly re-ordering the assembly code

Structural hazard

- Memory


```
Load:   F D X *M* W
Inst 1:  F D  X  M W
Inst 2:   F  D  X M W
Inst 3:   *F* D X M W
```
- Problem: The F stage from Inst 3 reads from instruction memory at the same time that the M stage from Load reads from data memory. In real computers, there is only one memory, so you can only read/write to one address in a given clock cycle.
- Solution: This structural hazard is overcome by having a separate instruction cache and data cache, so the two instructions are actually reading from seemingly separate memories.

Register File

- ```
Load: F D X M *W*
Inst 1: F D X M W
Inst 2: F D X M W
Inst 3: F *D* X M W
```
- Problem: In the same clock cycle, the W stage from the Load attempts to write the value just read from memory into a register in the register file, but the D stage from Inst 3 attempts to read a register from the register file.
- Solution: Because register reads/writes are very fast, often faster than 1/2 of a clock cycle, we solve this simultaneous read/write problem by introducing a convention:
  - Write to registers during the 1<sup>st</sup> half of a clock cycle
  - Read from registers during the 2<sup>nd</sup> half of a clock cycle
- This way it's possible to perform Read and Write in the same cycle. Sometimes this is known as

'forwarding through the register file'

## Control Hazard

- Branching

```
Branch: F D X * M W
Inst 1: F D X M W
Inst 2: F D X M W
Inst 3: * F D X M W
```

- Problem: For a branch instruction the branch condition must be calculated before the CPU decides whether the branch should be taken or not, so this might stall subsequent instructions on the pipeline.
- The result of the branch condition isn't available until right after the X stage of the Branch instruction (as denoted by the star in the diagram), so the earliest instruction that can fetch the correct instruction in the F stage is Inst 3, because that's the earliest time when the correct PC value is available (either [oldPC]+4 if branch is not taken, or some other address if branch is taken)
- This means that Inst 1 and Inst 2 are totally useless and must be killed, or that we must introduce 2 stall cycles in the pipeline where the instruction after the branch stalls in F:

```
Branch: F D X * M W
Inst: F F * F D X M W
```

- Branches take 3 cycles, which is bad :(
- Solutions:

1. Add an asynchronous comparator in the D stage so that as soon as an instruction is decoded, the CPU can figure out whether it's a branch and whether the branch condition is true so that the branch should be taken. This eliminates 1 stall cycle. Notice that the star (when the branch info is available) has been moved up to the end of the D stage, and only one stall cycle is needed. Branches will now take two cycles, which is an improvement.

```
Branch: F D * X M W
Inst: F * F D X M W
```

2. Delayed Branching

- Adding the comparator in the D stage still leaves one delay slot (which is better than 2), so how about if we redefine the semantics of branching to ALWAYS execute the instruction after the Branch instruction, regardless of whether the branch was taken? This is called delayed branching. It is up to the compiler (or human writing assembly code directly) to find an appropriate instruction to place in that branch delay slot
  - In the best case, a useful instruction is inserted, so the Branch only takes 1 instruction
  - In the worst case, a NOP is inserted, and the Branch takes 2 instructions
  - To calculate the average CPI (cycles per instruction) for Branch, assuming that the branch delay slot can be filled with a useful instruction x percent of the time:  
 $1 * (x/100) + 2 * (1-x/100)$
- Related: Delay slot for Jump instruction
  - Jumps are unconditional branches, but the correct PC can only be set after the instruction has been decoded in the D stage.

```
Jump: F D * X M W
Inst: F * F D X M W
```

- 1 delay slot is required. We can use the same trick as delayed branches to place 1 instruction in that slot to always execute.

### Data hazard

- Read after Write (RAW) dependency is the only problem for MIPS pipeline. This problem occurs when one instruction writes to a register but a subsequent instruction wants to read from that same register. The write may not be completed by the time that the read occurs, so pipeline stalls may be necessary.

```
Inst 1. add $t0 $t1 $t2
Inst 2. sub $t4 $t0 $t3
Inst 3. and $t5 $t0 $t6
Inst 4. or $t7 $t0 $t8
Inst 5. xor $t9 $t0 $t10
```

- Problem: Register \$t0 is the object of contention because instruction 1 writes to it but instructions 2-5 read from it. The semantics of assembly code ensures that the value of \$t0 will be the sum of \$t1 and \$t2 after instruction 1 completes. However, with pipelining, the subsequent instructions will execute BEFORE instruction 1 fully completes, so there is the danger of those instructions reading the incorrect (stale) value of \$t0. Refer to this pipeline diagram:

```
Inst 1: F D X M W *
Inst 2: F D X M W
Inst 3: F D X M W
Inst 4: F D X M W
Inst 5: F D X M W
```

- The new value of \$t0 isn't available until after the W stage (as denoted by the star). This means that, without introducing delays, instructions 2, 3, and 4 are going to read the incorrect old value of \$t0 in their respective D stages. Instruction 5 is fine because its D stage occurs one cycle after the W stage of instruction 1. 3 delays are required to ensure that instruction 2 doesn't read stale data (D stage comes after W stage of instruction 1), and the delays propagate to all later stages:

```
Inst 1: F D X M W *
Inst 2: F D D D D X M W
Inst 3: F F F F D X M W
Inst 4: F F F F D X M W
Inst 5: F F F F D X M W
```

- Notice that we can eliminate one delay cycle if we can assume that it is possible to read and write to the register file in one cycle using the convention of write during 1st half and read during 2<sup>nd</sup> half ('forwarding through the register file'). Now the W stage of instruction 1 can overlap with the D stage of instruction 2, making only 2 delay cycles necessary:

```
Inst 1: F D X M *W*
Inst 2: F D D *D* X M W
Inst 3: F F F D X M W
Inst 4: F F F D X M W
Inst 5: F F F D X M W
```

- Solution: However, 2 delay cycles is still not great, so we can eliminate all of these delays by introducing forwarding hardware to connect the output of the X and M stages back to the

beginning of the X stage. This way, subsequent instructions no longer need to wait for the W stage before the new register value is available, thus eliminating the need for delay slots for most instructions.

- Remember, the output of X can connect to the input of X, and the output of M can connect to the input of X.

```

add $t0 $t1 $t2 F D X M W
sub $t4 $t0 $t3 F D X M W (forward from X output of
'add' back to X input)
and $t5 $t0 $t6 F D X M W (forward from M output of
'add' back to X input)
or $t7 $t0 $t8 F D X M W (forward through register
file)
xor $t9 $t0 $t10 F D X M W (no forwarding needed)

```

- Caveat: Loads still require one delay slot, even with forwarding hardware. Why?

```

lw $t0 0($t1)
sub $t3 $t0 $t2

```

- The result of the load isn't available in \$t0 until the end of the M stage (with forwarding from M to X), and 'sub' needs the new value of \$t0 at the beginning of X.

```

lw $t0 0($t1) F D X M * W
sub $t3 $t0 $t2 F D X M W

```

- This doesn't work because M of lw and X of sub overlap. The new value of \$t0 isn't available until the star.

```

lw $t0 0($t1) F D X M * W
sub $t3 $t0 $t2 F D D X M W

```

- With one delay, it works. Again, like the branch delay slot, the compiler can try to place a useful instruction in the load delay slot (or a NOP if it fails) if it doesn't want to stall for one cycle:

```

lw $t0 0($t1) F D X M * W
Unrelated inst. F D X M W
sub $t3 $t0 $t2 F D X M W

```

## Caching

### Overview

- Reason for using caching is the Principle of Locality: only a small portion of a program's address space is needed at any instance in time
- Two types of locality:
  - Temporal Locality – If you reference a block, you will probably need it again soon. (exhibited by loops)
  - Spatial Locality – If you reference a block, you will probably reference blocks around it soon (exhibited by the sequential nature of program execution)

### Memory Hierarchies

- Why have a memory hierarchy – give the user the most amount of memory at the cheapest prices while still maintaining speed
- How to evaluate a memory hierarchy's performance:

- Hit Rate - % of memory accesses that are found in the upper level of the hierarchy
- Hit time – time to access a block in higher level memory
- Miss time – time to replace a block and retrieve it from lower memory

### **Basic Direct Mapped Caches**

- Direct Mapping is when each memory address can map to exactly one location in the cache
  - You index into the cache using the low order bits
  - The remaining bits form the tag
  - The valid bit is used to verify if the data is valid (it is invalid when it hasn't been written yet)
- Read misses: the CPU is stalled and all registers are frozen until memory is updated
- Write misses: the goal is to not have main memory and the cache become inconsistent (have different data stored for the same address)
- Write through: always write to both main memory and cache
  - pros: always consistent, cons: slow
  - can also use a write buffer to buffer main memory writes, which allows the CPU not to stall
- Write back: only update the cache, only writing back to main memory when a cache block is replaced
  - pros: fast, cons: it is complex to implement

### **Multiword Caches**

- Similar to direct mapped caches, but each block now contains multiple words
  - the index is now divided by # of words per block
  - on write misses, now we actually bring the entire block into memory, and overwrite the word we missed
  - improves spatial locality
  - but don't want too many words per block, because spatial locality within the blocks decreases, and fetch time increases

### **Flexible Word Placement**

- The idea is to reduce the number of cache misses by allowing more flexible placement of words in blocks
  - Fully associative: an address can be mapped to any block in the cache
  - Set associative: a block can be placed in a fixed n number of locations in the cache
    - when n approaches the # of blocks in the cache, it is equivalent to a fully associative cache
  - How to locate a block in the cache:
    - As before, the low bits are used to index into the cache, but now they index into a set instead of a single block. The tags of all blocks in the set are checked in parallel.
    - Replacement: typically choose which block to replace in a set using LRU