University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 1998 D.A. Patterson

# Quiz 1 Solutions
## CS252 Graduate Computer Architecture

Notes for future semesters: This quiz was long. If we were going to give this quiz again, we would probably drop the third part of question 2, and parts (b), (c), and (i) of question 3.

## Question 1: Calculate your Cache

A certain system with a 350 MHz clock uses a separate data and instruction cache, and a unified second-level cache. The first-level data cache is a direct-mapped, write-through, write-allocate cache with 8kBytes of data total and 8-Byte blocks, and has a perfect write buffer (never causes any stalls). The first-level instruction cache is a direct-mapped cache with 4kBytes of data total and 8-Byte blocks. The second-level cache is a two-way set associative, write-back, write-allocate cache with 2MBytes of data total and 32-Byte blocks.

The first-level instruction cache has a miss rate of 2%. The first-level data cache has a miss rate of 15%. The unified second-level cache has a local miss rate of 10%. Assume that 40% of all instructions are data memory accesses; 60% of those are loads, and 40% are stores. Assume that 50% of the blocks in the second-level cache are dirty at any time. Assume that there is no optimization for fast reads on an L1 or L2 cache miss.

All first-level cache hits cause no stalls. The second-level hit time is 10 cycles. (That means that the L1 miss penalty, assuming a hit in the L2 cache, is 10 cycles.) Main memory access time is 100 cycles to the first bus width of data; after that, the memory system can deliver consecutive bus widths of data on each following cycle. Outstanding non-consecutive memory requests can not overlap; an access to one memory location must complete before an access to another memory location can begin. There is a 128-bit bus from memory to the L2 cache, and a 64-bit bus from both L1 caches to the L2 cache. Assume a perfect TLB for this problem (never causes any stalls).

a) (2 points) What percent of all data memory references cause a main memory access (main memory is accessed before the memory request is satisfied)? **First show the equation, then the numeric result.**

If you did not treat all stores as L1 misses:
$=$ (L1 miss rate) $\times$ (L2 miss rate)
$= (.15) \times (.10)$
$= 1.5\%$

If you treated all stores as L1 misses:
$=$ (% of data ref that are reads)$\times$(L2 miss rate) + (% of data ref that are writes)$\times$(L1 miss rate)$\times$(L2 miss rate)
$= (.4)\times(.1) + (.6)\times(.15)\times(.1)$
$= 4.9\%$

b) (3 points) How many bits are used to <u>index</u> each of the caches? Assume the caches are presented physical addresses.

Data = 8K/8 = 1024 blocks = 10 bits
Inst = 4K/8 = 512 blocks = 9 bits
L2 = 2M/32 = 64k blocks = 32k sets = 15 bits

## Question 1 (continued)

c) (3 points) How many cycles can the longest possible data memory access take? Describe (briefly) the events that occur during this access.

L1 miss, L2 miss, writeback.
$1 + 10 + 2\times101 = 213$ cycles

Note that the time to read an L2 cache line from memory is 101 cycles (the first 16 B returns in 100 cycles; the next 16 return the next cycle).

d) (4 points) What is the average memory access time in cycles (including instruction and data memory references)? **First show the equation, then the numeric result.**

If you did not treat all stores as L1 misses:
$\mathrm{AMAT}_{\mathrm{total}} = \frac{1}{1.4}\mathrm{AMAT}_{\mathrm{inst}} + \frac{.4}{1.4}\mathrm{AMAT}_{\mathrm{data}}$
AMAT = (L1 hit time) + (L1 miss rate)× [(L2 hit time) + (L2 miss rate) × (mem transfer time)]
$\mathrm{AMAT}_{\mathrm{inst}} = 1 + 0.02(10 + .10\times1.5\times(101)) = 1.503$
$\mathrm{AMAT}_{\mathrm{data}} = 1 + .15(10 + .10\times1.5\times(101)) = 4.7725$
AMAT = 2.44

Note that the mem transfer time is multipled by 1.5 to account for writebacks in the L2 cache.

If you treated all stores as L1 misses:
$\mathrm{AMAT}_{\mathrm{total}} = \frac{1}{1.4}\mathrm{AMAT}_{\mathrm{inst}} + \frac{.24}{1.4}\mathrm{AMAT}_{\mathrm{loads}} + \frac{.16}{1.4}\mathrm{AMAT}_{\mathrm{stores}}$
AMAT = (L1 hit time) + (L1 miss rate)× [(L2 hit time) + (L2 miss rate) × (mem transfer time)]
$\mathrm{AMAT}_{\mathrm{inst}} = 1 + 0.02(10 + .10\times1.5\times(101)) = 1.503$
$\mathrm{AMAT}_{\mathrm{loads}} = 1 + .15(10 + .10\times1.5\times(101)) = 4.7725$
$\mathrm{AMAT}_{\mathrm{stores}} = 1 + 1(10 + .10\times1.5\times(101)) = 26.15$
AMAT = 4.88

Note that the mem transfer time is multipled by 1.5 to account for writebacks in the L2 cache.

## Question 2: Tomasulo's Revenge

Using the DLX code shown below, show the state of the Reservation stations, Reorder buffers, and FP register status for a speculative processor implementing Tomasulo's algorithm. Assume the following:

- Only one instruction can issue per cycle.
- The reorder buffer has 8 slots.
- The reorder buffer implements the functionality of the load buffers and store buffers.
- All FUs are fully pipelined.
- There are 2 FP multiply reservation stations.
- There are 3 FP add reservation stations.
- There are 3 integer reservation stations, which also execute load and store instructions.
- No exceptions occur during the execution of this code.
- All integer operations require 1 execution cycle. Memory requests occur and complete in this cycle. (For this problem, assume that, barring structural hazards, loads issue in one cycle, execute in the next, write in the third, and a dependent instruction can start execution on the fourth.)
- All FP multiply operations require 4 execution cycles.
- All FP addition operations require 2 execution cycles.
- On a CDB write conflict, the instruction issued earlier gets priority.
- Execution for a dependent instruction can begin on the cycle after its operand is broadcast on the CDB.
- If any item changes from "Busy" to "Not Busy", you should update the "Busy" column to reflect this, but you should not erase any other information in the row (unless another instruction then overwrites that information).
- Assume the all reservation stations, reorder buffers, and functional units were empty and not busy when the code show below began execution.
- The "Value" column gets updated when the value is broadcast on the CDB.

Integer registers are not shown, and you do not have to show their state.

## Question 2 (continued)

a) (4 points) The tables below show the state after the cycle in which the second `SUBI` from the code below issued. Show the state after the next cycle.

```
Lp: LD     F0, 0(R1)
    LD     F2, 0(R2)
    MULTD  F4, F0, F2
    ADDD   F6, F0, F0
    SUBI   R1, R1, 8
    SUBI   R2, R2, 8
    ADDI   R3, R3, 1
```

| Reservation stations | | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
| Add1 | *N* | `ADDD` | F0 | F0 | | | #4 |
| Add2 | | | | | | | |
| Add3 | | | | | | | |
| Mult1 | Y | `MULTD` | F0 | F2 | | | #3 |
| Mult2 | | | | | | | |
| Int1 | Y | `SUBI` | R2 | 8 | | | #6 |
| Int2 | *Y* | *`ADDI`* | *R3* | *1* | | | *#7* |
| Int3 | Y | `SUBI` | R1 | 8 | | | #5 |

| Reorder buffer | | | | | |
|---|---|---|---|---|---|
| Entry | Busy | Instruction | State | Destination | Value |
| 1 | N | `LD F0, 0(R1)` | Commit | F0 | Mem[0(R1)] |
| 2 | N | `LD F2, 0(R2)` | Commit | F2 | Mem[0(R2)] |
| 3 | Y | `MULTD F4, F0, F2` | Execute | F4 | |
| 4 | Y | `ADDD F6, F0, F0` | *Write* | F6 | *F0 + F0* |
| 5 | Y | `SUBI R1, R1, 8` | Execute | R1 | |
| 6 | Y | `SUBI R2, R2, 8` | *Execute* | R2 | |
| 7 | *Y* | *`ADDI R3, R3, 1`* | *Issue* | R3 | |
| 8 | | | | | |

| FP register status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Field | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ... | F30 |
| Reorder # | 1 | 2 | 3 | 4 | | | | ... | |
| Busy | N | N | Y | Y | | | | ... | |

## Question 2 (continued)

b) (8 points) The tables below show the state during the cycle in which the second `MULTD` from the code below issued. Show the state after ***two*** cycles.

```
MULTD  F0, F2, F4
ADDD   F6, F6, F0
ADDD   F2, F2, F8
LD     F4, 0(R2)
ADDI   R1, R1, 8
MULTD  F8, F10, F12
ADDD   F4, F4, F10
ADDI   R2, R2, 1
```

| **Reservation stations** | | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
| Add1 | Y | ADDD | F6 | *F0* | | *blank* | #2 |
| Add2 | *N* | ADDD | F2 | F8 | | | #3 |
| Add3 | *Y* | *ADDD* | *F4* | *F10* | | | *#7* |
| Mult1 | N | MULTD | F2 | F4 | | | #1 |
| Mult2 | Y | MULTD | F10 | F12 | | | #6 |
| Int1 | *N* | LD | R2 | 0 | | | #4 |
| Int2 | Y | ADDI | R1 | 8 | | | #5 |
| Int3 | *Y* | *ADDI* | *R2* | *1* | | | *#8* |

| **Reorder buffer** | | | | | |
|---|---|---|---|---|---|
| Entry | Busy | Instruction | State | Destination | Value |
| 1 | N | MULTD F0, F2, F4 | *Commit* | F0 | F2×F4 |
| 2 | Y | ADDD F6, F6, F0 | *Execute* | F6 | |
| 3 | Y | ADDD F2, F2, F8 | *Write* | F2 | *F2+F8* |
| 4 | Y | LD F4, 0(R2) | *Write* | F4 | *Mem[0(R2)]* |
| 5 | Y | ADDI R1, R1, 8 | Execute | R1 | |
| 6 | Y | MULTD F8, F10, F12 | *Execute* | F8 | |
| 7 | *Y* | *ADDD F4, F4, F10* | *Issue* | *F4* | |
| 8 | *Y* | *ADDI R2, R2, 1* | *Issue* | *R2* | |

| **FP register status** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Field | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ... | F30 |
| Reorder # | 1 | 3 | *7* | 2 | 6 | | | ... | |
| Busy | N | Y | Y | Y | Y | | | ... | |

## Question 2 (continued)

c) (8 points) The tables below show the state after the cycle in which the `SUB` from the code below issued. Show the state after the next ***four*** cycles.

```
ADDD   F4, F0, F0
SUBD   F4, F4, F2
ADDI   R2, R2, 1
ADDI   R3, R3, 1
ADDD   F2, F6, F8
MULTD  F0, F6, F8
```

| Reservation stations | | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
| Add1 | *N* | `ADDD` | F0 | F0 | | | #1 |
| Add2 | Y | `SUBD` | *F4* | F2 | | | #2 |
| Add3 | *Y* | *ADDD* | *F6* | *F8* | | | *#5* |
| Mult1 | *Y* | *MULTD* | *F6* | *F8* | | | *#6* |
| Mult2 | | | | | | | |
| Int1 | *N* | *ADDI* | *R2* | *1* | | | *#3* |
| Int2 | *N* | *ADDI* | *R3* | *1* | | | *#4* |
| Int3 | | | | | | | |

| Reorder buffer | | | | | |
|---|---|---|---|---|---|
| Entry | Busy | Instruction | State | Destination | Value |
| 1 | *N* | `ADDD F4, F0, F0` | *Commit* | F4 | *F0 + F0* |
| 2 | Y | `SUBD F4, F4, F2` | *Execute* | F4 | |
| 3 | *Y* | *ADDI R2, R2, 1* | *Write* | *R2* | *R2 + 1* |
| 4 | *Y* | *ADDI R3, R3, 1* | *Write* | *R3* | *R3 + 1* |
| 5 | *Y* | *ADDD F2, F6, F8* | *Execute* | *F2* | |
| 6 | *Y* | *MULTD F0, F6, F8* | *Issue* | *F0* | |
| 7 | | | | | |
| 8 | | | | | |

| FP register status | | | | | | | |
|---|---|---|---|---|---|---|---|
| Field | F0 | F2 | F4 | F6 | F8 | ... | F30 |
| Reorder # | *6* | *5* | 2 | | | ... | |
| Busy | *Y* | *Y* | Y | | | ... | |

## Question 3: Vector vs DSP Showdown

Examine the two architectures below.

The first architecture is a 25 Mhz 3-stage dsp processor. A block diagram showing some of the fully-bypassed datapath is shown below. The three stages are fetch, decode (where branches

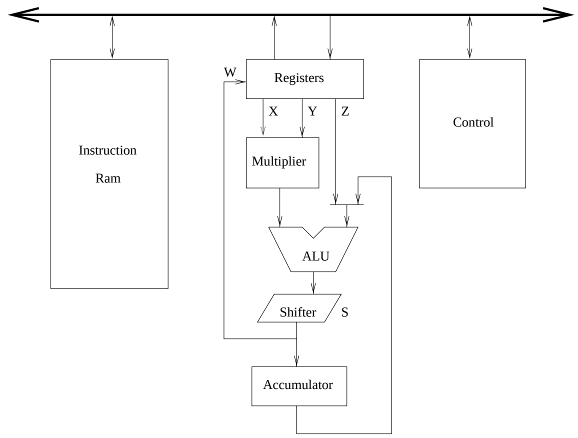


Figure 1: The DSP block diagram

are evaluated and the PC updated), and execute (where memory and register writes also occur). The processor is able to multiply, accumulate, and shift during its execute stage. It has the same load, store, and branch instructions as DLX. It also includes a `LT` instruction, which loads a value into a register from memory, and decrements the base register to the next element. The arithmetic operations are slightly different:

- Register 0 always contains the value 0
- Register 1 always contains the value 1
- The result from the shifter is always written to the accumulator on arithmetic operations
- Operations can be specified as `MAC W, X, Y, Z, S`, where `W` is the register to be written; `X` and `Y` are registers that go to the multiplier; `Z` is the register that goes to the alu; and `S` specifies the amount to right shift the result.
- Operations can also be specified as `MACA W, X, Y, S`, where `W` is the register to be written; `X` and `Y` are registers that go to the multiplier; the accumulator goes to the alu; and `S` specifies the amount to right shift the result.

The second architecture is a 100 Mhz vector processor with a MVL of 64 elements. It has one FP add/subtract FU, one FP multiply/divide FU, and a single memory FU. The startup overhead is 5 cycle for add, subtract, multiply, and divide instructions, and 10 cycles for memory instructions. It supports flexible chaining but not tailgating.

## Question 3 (continued)

Here is the code for the dsp:

```
LP: LT     R2, 0(R5)          # Load R2 with new value
    MAC    R0, R10, R2, R0, 0 # Perform the calculation
    MACA   R0, R11, R2, 0
    MACA   R2, R12, R2, 0
    BNEZ   R5, LP
    SW     R2, -4(R5)          # Delayed branch
```

a) (2 points) What is the peak performance, in results per second, of the above three-tap filter?

3 results per loop, 6 instructions per loop, 25 Mhz
12.5M results per second

b) (2 points) What would be the peak performance, in results per second, of the above code, if it was a five-tap filter?

5 results per loop, 8 instructions per loop, 25 Mhz
15.625M results per second

## Question 3 (continued)

c) (3 points) Translate the following DLX code to code that will operate on this DSP. Assume that all floating point calculations below can be done in fixed-point on the DSP. Do not worry about round-off error from converting between floating point and fixed point. Assume that for DLX code, F0 contains 0, F2 contains 0.5, and F4 contains 2.0. Assume that for the DSP code you will write, R2 contains the value 2. Assume for both that register R5 contains the correct initial loop count.

```
LP: LW      R3, 0(R5)    # Load R3 with the new value
    MOVI2FP F6, R3       # Move the value from R3 to F6
    CVTI2F  F6, F6       # Convert integer value to floating point
    MULTF   F8, F4, F6   # Multiply by 2
    ADDD    F10, F8, F0  # Add accumulator to value
    MULTF   F0, F10, F2  # Divide value by 2
    CVTF2I  F12, F0      # Convert to integer representation
    MOVFP2I R3, F12      # Move it to integer registers
    SW      R3, 0(R5)    # Store value back to mem
    ADDI    R5, -4       # Point to next element
    BNEZ    R5, LP       # If not done, branch back
```

```
    MAC  R0, R0, R0, R0, 0 # Clear accumulator
LP: LT   R3, 0(R5)         # Load value
    MACA R3, R3, R2, 1     # R3 <- ((R3*2)+acc)/2
    BNEZ R5, LP            # Branch if done
    SW   R3, -4(R5)        # Delayed branch slot
```

## Question 3 (continued)

Here is the code for the vector machine:

```
LP:  LV      V1, R5      # Load V1 with new value
     MULTSV  V2, F0, V1  # Perform the calculation
-----------------------------------------------
     MULTSV  V3, F1, V1
     ADDV    V2, V2, V3
-----------------------------------------------
     MULTSV  V3, F2, V1
     ADDV    V2, V2, V3
     SV      V2, R5
     SUBI    R5, R5, 8
     BNEZ    R5, LP
```

d) (3 points) Show the convoys of vector instructions for the above code. Follow the timing examples in the book. Draw lines to show the convoys on the existing code shown above.

(Shown above with lines)

e) (4 points) Show the execution time in clock cycles of this loop with $n$ elements ($T_n$); assume $T_{\text{loop}} = 15$. **Show the equation, and give the value of execution time for $n$=64.**

$T_n = \lceil \frac{n}{64} \rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$

$T_{64} = \left\lceil \frac{64}{64} \right\rceil \times (T_{loop} + \mathtt{LV}_{start} + \mathtt{MULTSV}_{start} + \mathtt{MULTSV}_{start} + \mathtt{ADDV}_{start} + \mathtt{MULTSV}_{start} + \mathtt{ADDV}_{start} + \mathtt{SV}_{start}) + 64 \times 3$

$T_{64} = \left\lceil \frac{64}{64} \right\rceil \times (15 + 10 + 5 + 5 + 5 + 5 + 5 + 10) + 64 \times 3$

$T_{64} = 252$

f) (3 points) What is $R_\infty$ for this loop?

$$R_\infty = \frac{\text{Operations per iteration} \times \text{Clock rate}}{\lim_{n\to\infty} \text{Clock cycles per iteration}}$$

$$\lim_{n\to\infty} \text{Clock cycles per iteration} = \lim_{n\to\infty} \left(\frac{T_n}{n}\right) = \lim_{n\to\infty} \left(\frac{3n + (60/64)n}{n}\right) = 3.9375$$

$$R_\infty = \frac{5 \times 100}{3.9375} = 127\text{MFLOPS}$$

## Question 3 (continued)

g) (3 points) List 6 characteristics of DSP instruction set architectures that differ from general purpose microprocessors.

For (g), (h), (i), and (j), there are many possibly answers besides what is listed here.

- Autoincrement addressing
- Circular addressing
- Bit reverse addressing
- FFT specific addressing
- Saturating overflow
- Fast multiply-add
- Narrow data
- Fast loops

h) (1 point) Which of those characteristics are supported in vector architectures?

- Autoincrement addressing
- Multiply-add (with chaining)
- Fast loops

i) (1 point) Which of the unsupported characteristics could be handled in software?

- Circular addressing

j) (2 points) What changes to the hardware would you make to handle the remaining characteristics?

- Saturating overflow
- Narrow data support
- FFT support