332 Advanced Computer Architecture

Unassessed tutorial exercise 1

Exercise 1.1. Instruction set issues in pipeline design

During the gruelling interview for the post of Deputy Technical President, IC Inside™ Computers Incorporated, you have been asked to investigate a new addressing mode for a MIPS-like architecture. The idea is to allow one of the operands of an ALU instruction to be in memory. To offset this increase in complexity, you restrict all memory addressing (in Load, Store and ALU instructions) to be register-indirect only (i.e. no displacement addressing).

For example, suppose `R1` is an index into an array starting at address 100, and we need to multiply the `R1`$^{\text{th}}$ element by 123. On MIPS you could write:

```
LD   R2, 100(R1)
MULI R3, R2, 123
```

Here we load `R2` with the element of the array, using displacement addressing. With the modified architecture, you would have to write

```
ADDI R2, R1, 100
MULI R3, (R2), 123
```

Here, the first instruction does the displacement calculation explicitly, leaving a pointer to the array element in `R2`. The second instruction then reads the value from memory.

1. Propose a change to the 5-stage MIPS pipeline (as shown in the notes) to accommodate this addressing mode. Draw the new pipeline and explain the changes (your new design should be just as simple as the original).
2. What new data hazards are created by this addressing mode using your modified pipeline? Give an example of each of the hazards you find. Explain where in the pipeline the hazard occurs.
3. List all changes or additions which must be made to the hardware to accommodate this new addressing mode.
4. List all the questions which must be answered to estimate the impact of this change on overall performance. Do you think the new design is better?

Exercise 1.2. Assessing a vector processing enhancement

Background theory: Amdahl's Law

This exercise concerns Amdahl's Law, which is discussed at length in the textbook. It captures a favourite architecture slogan: "make the common case fast, and make the fast case common":

$$\text{speedup}_{\text{overall}} = \frac{1}{(1 - \text{fractionenhanced}) + \frac{\text{fractionenhanced}}{\text{speedup}_{\text{enhanced}}}}$$

Background architecture: vector pipeline architectures

A vector pipeline processor is an extension to a conventional CPU, which provides registers which can hold an entire vector of floating-point operands (e.g. 64 words). In addition, the machine would have instructions which manipulate vectors. For example, to implement the following loop:

```
for i := 1 to 64 do
  for j := 1 to 64 do
    Y[k,j] := a * X[i,j] + Y[k,j];
```

This could be programmed as follows:

```
R1 := 0
R2 := 0
R3 := k*N

for i := 1 to 64 do
    LV      V2,X(R1)   ; load row of X (R1 is i*N, where N is row length)
    MULTSV  V3,a,V2    ; multiply each element of the row vector by scalar 'a'
    LV      V1,Y(R2)   ; load k'th row of Y, where R2 is k*N)
    ADDV    V4,V3,V1   ; add Y[i] and a*X[i]
    SV      V4,Y(R3)    ; store the vector into array Y

    R1 := R1+N
    R2 := R2+N
end do
```

The classic vector pipeline machine is the Cray 1, and there are many later designs from Cray and others (eg the NEC SX-6 used in the Earth Simulator). The main advantages of vector registers and vector instructions are that pipelined floating point arithmetic can be organised very efficiently, and the maximum throughput of a highly-interleaved memory system can be exploited.

We will study vector processors in more detail shortly. No further details are needed for this exercise.

Exercise 1.2.1. The benefit of a vector mode

Assume that we are considering enhancing a machine by adding a vector mode to it. When a computation is run in vector mode, it is 20 times faster than the normal mode of execution. We call the percentage of time that could be spent using vector mode the *percentage of vectorisation*.

1. What percentage of vectorisation is needed to achieve a speedup of 2?
2. What percentage vectorisation is needed to achieve one-half the maximum speedup available from using vector mode?
3. Suppose you have measured the percentage of vectorisation for programs to be 70%. The hardware design group says they can double the the speed of the vector mode with a significant engineering investment. You wonder whether the compiler crew could increase the use of vector mode as another approach to increasing performance. How much of an increase in the percentage of vectorisation (relatitive to current usage) would you need to obtain the same performance gain? Which investment would you recommend?
4. Draw (sketch!) a graph which plots the speedup as a percentage of the computation performed in vector mode. Label the y axis “Net Speedup”, and label the x axis “Percent Vectorisation”.

(each part should take only a few minutes; you do not really need a calculator to uncover the basic point of this exercise – back-of-the-envelope estimates are quite adequate).

Exercise 1.2.2. Before and After

Assume that we make an enhancement to a computer that improves some mode of execution by a factor of 10. Enhanced mode is used 50% of the time, measured as a percentage of the execution time when the enhanced mode is in use *(rather than as defined in the notes, where the percentage of the running time* **without** *the enhancement is used).*

1. What is the speedup we have obtained from fast mode?
2. What percentage of the original execution time has been converted to fast mode?

(These exercises are taken from Hennessy and Patterson, exercises 1.1, 1.2 and from Chapter 1 of the first edition).

Paul Kelly, Imperial College, 2005

Notes on the solutions to unassessed exercise 1

1. Propose a change to the 5-stage MIPS pipeline (as shown in the notes) to accommodate this addressing mode. Draw the new pipeline and explain the changes.

   Notice that the changes do not force us to have two ALU stages: arithmetic instructions need an ALU to do the arithmetic, but do not need to do any effective address calculation (this is why it was so important to disallow displacement addressing).

   However, in processing an arithmetic instruction with a memory operand, it is necessary to access memory *before* performing the arithmetic, so we must have the MEM cycle before the EX cycle.

   So it looks like a solution is simply to switch the MEM and EX stages in the MIPS pipeline, giving a picture like this (it's a good idea to draw it this way to show up the relative timings):

Clock:	1	2	3	4	5	6	7	8	9
Instruction 1	IF	ID	MEM	EX	WB				
Instruction 2		IF	ID	MEM	EX	WB			
Instruction 3			IF	ID	MEM	EX	WB		
Instruction 4				IF	ID	MEM	EX	WB	
Instruction 5					IF	ID	MEM	EX	WB

   The question is, can this handle all the instructions in the instruction set. For example, notice that displacement addressing has to be disallowed not only in arithmetic instructions, but also in loads and stores. *(Why?)*

2. What new data hazards are created by this addressing mode using your modified pipeline? Give an example of each of the hazards you find. Explain where in the pipeline the hazard occurs.

   With the pipeline above, there is essentially one hazard, although you have to consider two cases. Consider initially the following:

```
ADD R1, R2, R3
NOP               ; (no operation)
ADD R4, (R1), R5
```

   The MEM stage of the last instruction must use the value generated at the end of the EX stage two instructions previously. This problem can be dealt with using the *forwarding* (sometimes called *bypassing*) technique: special hardware catches the case, and routes the operand direct from EX to MEM without going through the register file. Thus a stall is avoided.

   Now consider the case where `R1` is used immediately after the instruction in which it is computed:

```
ADD R1, R2, R3
ADD R4, (R1), R5
```

Again, the MEM stage of the last instruction must use the value generated by the first one, but if we refer to the timing we see that the MEM stage for the second instruction is supposed to start in the same cycle as the EX stage in which `R1` is calculated. Thus there is no alternative but to stall.

Referring to the textbook (pg. 264), observe that both of these hazards are read-after-write (RAW) hazards: a stage tried to read a value before it was written. A write-after-read (WAR) hazard occurs if a register or memory cell should be overwritten *after* it is read, but because of the pipeline, it could be overwritten before it is read. A write-after-write (WAW) hazard occurs if the pipeline changes the order in which writes occur, thereby leaving the wrong values in the end.

Why aren't there any WAR or WAW hazards in our pipeline?

3. List all changes or additions which must be made to the hardware to accommodate this new addressing mode.

   The number of memory ports needed is not changed. Similarly, the number of simultaneous accesses to the register file is unchanged (how many *do* we need to handle?). We do need to modify the data path so addresses can be routed direct from the register file to the memory address register (MAR).

   We need new hardware to handle forwarding: essentially a multiplexer on the MAR so that data can be routed to it either from the register file or from the ALU. We also need additional control hardware to handle this, and to organise stalling on the data hazard noted above.

4. List all the questions which must be answered to estimate the impact of this change on overall performance.

   Impact on clock time?

   Percentage of loads and stores which use displacement addressing (since each of these would now involve an additional instruction to compute the effective address).

   Percentage of loads which load a value which is only used once by an ALU operation — since it is in this case that the new addressing mode saves us something.

   After the compilers have been changed so that displacement addressing is calculated in a register, what percentage of instructions that calculate an address are immediately followed by a load instruction which uses it? (since this is the case where a stall would occur).

Exercise 1.2.1

This exercise concerns Amdahl's Law, which is introduced in the notes and discussed at length in the textbook:

$$\text{speedup}_{\text{overall}} = \frac{1}{(1 - \text{fractionvectorised}) + \frac{\text{fractionvectorised}}{\text{speedup}_{\text{enhanced}}}}$$

1. For net speedup of 2, we need

$$\frac{1}{(1 - \text{fractionvectorised}) + \text{fractionvectorised}/20} = 2$$

Rearranging Amdahl's Law gives:

$$\text{fractionvectorised} = \frac{\text{speedup}_{\text{overall}} \times \text{speedup}_{\text{enhanced}} - \text{speedup}_{\text{enhanced}}}{\text{speedup}_{\text{overall}} \times \text{speedup}_{\text{enhanced}} - \text{speedup}_{\text{overall}}}$$

So

$$\text{fractionvectorised} = \frac{2 \times 20 - 20}{2 \times 20 - 2} = \frac{20}{38} = 52.6\%$$

2. With the original vector mode design, 70% vectorisation yields a net speedup of $\frac{1}{0.3+0.7/20} = 2.985$. Increasing the vector mode speedup to a factor of 40 would increase this to $\frac{1}{0.3+0.7/40} = 3.15$.

To achieve this net speedup by improving the compiler, we must increase the percentage vectorisation. We have

$$\frac{1}{(1 - \text{fractionvectorised}) + \text{fractionvectorised}/20} = 3.15$$

Rearranging as before, we get

$$\text{fractionvectorised} = \frac{3.15 \times 20 - 20}{3.15 \times 20 - 3.15} = \frac{43}{59.85} = 71.8\%$$

So the compiler crew only have to do a couple of percent better to outperform the proposed hardware development.

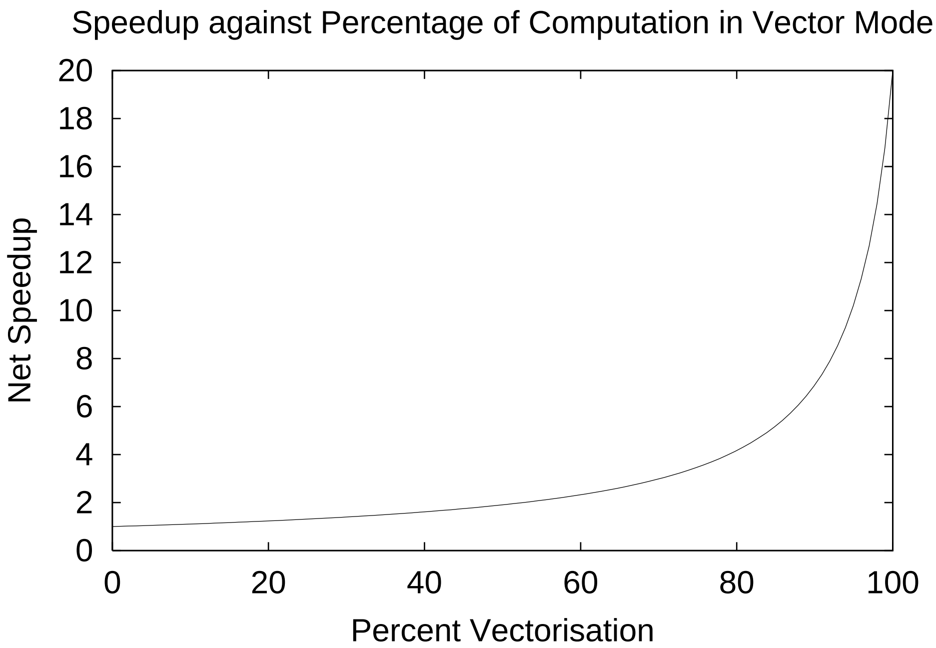
3. To sketch this graph consider a few points, e.g.

   - when the percentage vectorised is 0%, net speedup is 1.
   - when the percentage vectorised is 100%, net speedup is 20.
   - when the percentage vectorised is 50%, net speedup is

   $$\frac{1}{0.50 + 0.50/20} = \frac{1}{0.55} = 1.82.$$

   - when the percentage vectorised is 75%, net speedup is

   $$\frac{1}{0.25 + 0.75/20} = \frac{1}{0.2875} = 3.48.$$



Exercise 1.2

1. To compute the speedup obtained from the fast mode we must work out the execution time without the enhancement. We know that the enhanced execution time consisted of two halves: the unenhanced phase (50%) and the enhanced phase (50%).

   Without the enhancement, the unenhanced phase would have taken just as long (50%), but the enhanced phase would take 10 times as long, i.e. 500%. So the relative execution time without the enhancement would be $50\% + 500\% = 550\%$.

   Thus the overall speedup is

   $$\frac{\text{executiontime}_{\text{unenhanced}}}{\text{executiontime}_{\text{enhanced}}} = \frac{550\%}{100\%} = 5.5$$

2. To find the percentage of the original execution time which was enhanced, we plug these figures into Amdahl's Law again:

   $$\begin{aligned} \text{fractionvectorised} &= \frac{\text{speedup}_{\text{overall}} \times \text{speedup}_{\text{enhanced}} - \text{speedup}_{\text{enhanced}}}{\text{speedup}_{\text{overall}} \times \text{speedup}_{\text{enhanced}} - \text{speedup}_{\text{overall}}} \\ &= \frac{5.5 \times 10 - 10}{5.5 \times 10 - 5.5} \\ &= 45/49.5 = 90.90\% \end{aligned}$$

Paul Kelly, Imperial College, 2005