Assignment-2

JASHWANTH REDDY GANGULA

jgangula@syr.edu

SUID: 646254141

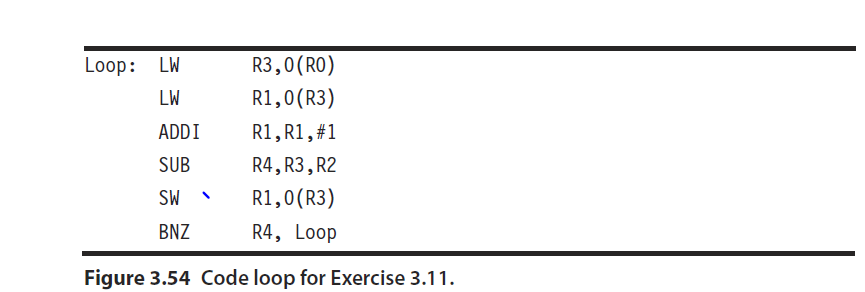
[10/10/10] <3.3> Assume a five-stage single-pipeline microarchitecture (fetch,

decode, execute, memory, write-back) and the code in Figure 3.54 . All ops are

one cycle except LW and SW , which are 1 + 2 cycles, and branches, which are 1 + 1

cycles. There is no forwarding. Show the phases of each instruction per clock

cycle for one iteration of the loop.



Solution:

F - > Fetch

D -> Decode

E - > Execute

M - >Memory Access

W -> Write back to register file

For the 5 stages of instruction execution, the Load and store have a 1+2 cycle latency.

Assuming no forwarding, the below table fills in the pipeline stages of all the instructions for a single iteration of the loop, with the notation mentioned above. The \* mark in the table implies the stalling of pipeline stage to wait for the result of previous instruction data dependency hazards etc.

The load instruction takes 2 more cycles at memory access stage due to latency. Same applies for store instruction too. The execution of any instruction depends on the availability of operands and hence the execution stage of instructions will wait until the operands are ready or available. The branch prediction happens after the execution phase of branch instruction. So all the cycles of the next instruction will have to wait for the prediction status of branch instruction. The branch instruction BNZ completes the Execution stage after 16 clock cycle because of the previous store instruction memory latency. The next instruction of branch cannot start until the execution phase of branch is completed. Hence the next instruction here starts at clock cycle 17.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ClockCycle Count | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| LW R3,0(R0) | F | D | E | M | \* | \* | W |  |  |  |  |  |  |  |  |  |  |  |  |  |
| LW R1,0(R3) |  | F | D | \* | \* | \* | E | M | \* | \* | W |  |  |  |  |  |  |  |  |  |
| ADDI R1,R1,#1 |  |  | F | \* | \* | \* | D | \* | \* | \* | E | M | W |  |  |  |  |  |  |  |
| SUB R4,R3,R2 |  |  |  |  |  |  | F | \* | \* | \* | D | E | M | W |  |  |  |  |  |  |
| SW R1,0(R3) |  |  |  |  |  |  |  |  |  |  | F | D | E | M | \* | \* | W |  |  |  |
| BNZ R4, Loop |  |  |  |  |  |  |  |  |  |  |  | F | D | E | \* | \* | M | W |  |  |
| LW R3,0(R0) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | F |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

a. [10] <3.3> How many clock cycles per loop iteration are lost to branch overhead?

In the above table the Fetch stage of LW R3, 0(R0) is expected to begin at the clock cycle 13 after the branch instruction but begins at the 17th clock cycle.

Hence branch overhead = 17 - 13 = 4 clock cycles.

b. [10] <3.3> Assume a static branch predictor, capable of recognizing a backwards

branch in the Decode stage. Now how many clock cycles are wasted on branch overhead?

In case of static predictor , the end of the loop goes back to the address with negative offset,  
that is continuation of the current loop code. In this case, the immediate next instruction is started after recognizing the backward branch which happens in the decode phase. The PC value of the jump address will be available after the execute phase. In this case the next instruction starts at clock cycle 15. Hence the branch overhead = 15- 13 = 2 clock cycles.

c. [10] <3.3> Assume a dynamic branch predictor. How many cycles are lost on

a correct prediction?

The dynamic branch predictor algorithm works based on branch history. For example, it normally predicts the branch taken if branched the last time, predict the branch not-taken if didn’t branch the last time. In case the prediction turns to be correct no clock cycles are lost. If not, we have to flush out the unnecessary instructions from the pipeline and fetch the correct address.

3.14 [25/25/25] <3.2, 3.7> In this exercise, we look at how software techniques can extract instruction-level parallelism (ILP) in a common vector loop. The following loop is the so-called DAXPY loop (double-precision aX plus Y ) and is the central operation in Gaussian elimination. The following code implements the DAXPY operation, Y = aX + Y , for a vector length 100. Initially, R1 is set to the base address of array X and R2 is set to the base address of Y :

DADDIU R4, R1, #800 ; R1 = upper bound for X

foo: L.D F2,0(R1) ; (F2) = X(i)

MUL.D F4, F2,F0 ; (F4) = a\*X(i)

L.D F6,0(R2) ; (F6) = Y(i)

ADD.D F6, F4, F6 ; (F6) = a\*X(i) + Y(i)

S.D F6, 0(R2) ; Y(i) = a\*X(i) + Y(i)

DADDIU R1,R1,#8 ; increment X index

DADDIU R2,R2,#8 ; increment Y index

DSLTU R3,R1,R4 ; test: continue loop?

BNEZ R3,foo ; loop if needed

Assume the functional unit latencies as shown in the table below. Assume a one cycle

delayed branch that resolves in the ID stage. Assume that results are fully bypassed

Instruction producing

result Instruction using result Latency in clock cycles

FP multiply FP ALU op 6

FP add FP ALU op 4

FP multiply FP store 5

FP add FP store 4

Integer operations and all Any 2  
loads

[25] <3.2> Assume a single-issue pipeline. Show how the loop would look both unscheduled by the compiler and after compiler scheduling for both floating-point operation and branch delays, including any stalls or idle clock cycles. What is the execution time (in cycles) per element of the result vector, Y , unscheduled and scheduled? How much faster must the clock be for processor

hardware alone to match the performance improvement achieved by the scheduling compiler? (Neglect any possible effects of increased clock speed on memory system performance.)

**Solution:**

**Compiler Unscheduled:**

The vector size is 100 . Each float precision is 8 bytes so add 800 offset to R1 to check the end of

Vector operation loop.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Instruction | Operands | Comments | Clock Cycle | Other |
| DAADIU | R4, R1, #800 |  | 1 |  |
| L.D | F2,0(R1) ; | (F2) = X(i) | 2 |  |
| stall |  |  | 3 | Latency of instruction using result. |
| MUL.D | F4, F2,F0 ; | (F4) = a\*X(i) | 4 | Multiplication takes 6 cycles for completion.(wait for F2 result) |
| L.D | F6,0(R2) ; | (F6) = Y(i) | 5 |  |
| Stall |  |  | 6 |  |
| stall |  |  | 7 |  |
| stall |  |  | 8 |  |
| stall |  |  | 9 |  |
| ADD.D | F6, F4, F6 ; | (F6) = a\*X(i) + Y(i) | 10 | F4 result from multiply is available now. So stall previous cycles. |
| stall |  |  | 11 |  |
| stall |  |  | 12 |  |
| stall |  |  | 13 |  |
| S.D | F6, 0(R2) ; | Y(i) = a\*X(i) + Y(i) | 14 | Get the result of F6 now. 4 stalls for ALU op used by store operation. |
| DAADIU | R1, R1, #8 | Base offset of X vector increased to next element | 15 |  |
| DAADIU | R2, R2, #8 | Base offset of Y vector increased to next element | 16 |  |
| stall |  |  | 17 | All integer and load operations stall for any next instruction. Clock cycle at 15 completes execution which 16 stalls here. |
| DSLTU | R3, R1, R4 | test: Continue loop? | 18 | Compare instruction involves integer operation and introduce stall for next cycle to check R3 |
| BNEZ | R3, foo | loop if needed | 19 | Introduces stall for one cycle at ID phase. |
| stall |  |  | 20 |  |
|  |  |  |  |  |

Hence the execution time per element of Y is 20 clock cycles for unscheduled compiled code.

**Scheduled:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Instruction | Operands | Comments | Clock Cycle | Other |
| DAADIU | R4, R1, #800 | The size of vector is 100 elements | 1 |  |
| L.D | F2,0(R1) ; | (F2) = X(i) | 2 | Load takes 2 cycles. So follow it immediately to load the Y element with no dependency. |
| L.D | F6,0(R2) ; | (F6) = Y(i) | 3 |  |
| MUL.D | F4, F2, F0 | F4 = aX(i) | 4 | F2 result is available now and Mult takes 6 cycles |
| DAADIU | R1, R1, #8 | Increase offset to get X(i+1) | 5 | The current instruction has no dependency with mult. |
| DAADIU | R2, R2, #8 | Increase offset to get Y(i+1) | 6 | The current instruction has no dependency with mult. |
| Stall |  |  | 7 |  |
| Stall |  |  | 8 |  |
| stall |  |  | 9 |  |
| ADD.D | F6, F4, F6 | F6 = aX(i) + Y(i) | 10 | Mult result is available now to use to compute the result. |
| DSLTU | R3, R1, R4 |  | 11 | No data dependency with addition instruction which takes 4 cycles to get the result. |
| stall |  |  | 12 | Integer operation takes 2 cycles to get the R3 result. |
| BNEZ | R3, foo |  | 13 | Branch introduces a new stall .In the meantime store the result to Y vector in the next cycle. |
| S.D | F6, -8(R2) | Store the result in Y(i) | 14 | End of one element execution |

*The scheduled code takes 14 clock cycles for execution of one element of vector operation completion.*

Performance improvement achieved by the scheduling compiler = 20 – 14 = 6 clock cycles.

The processor should be faster by 6 clock cycles. i.e., it should execute 6 extra clock cycles faster in along with the original execution of 14 clock cycles = 6/14 \* 100 = 42.857% faster

**[25] <3.2>** Assume a single-issue pipeline. Unroll the loop as many times as necessary to schedule it without any stalls, collapsing the loop overhead instructions. How many times must the loop be unrolled? Show the instruction schedule. What is the execution time per element of the result?

**Solution:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Instruction | Operands | Comments | Clock Cycle | Other |
| DAADIU | R4, R1, #800 | 100 size vector | 1 |  |
| L.D | F2, 0(R1) | F2 = X(i) | 2 |  |
| L.D | F6, 0(R2) | F6 = Y(i) | 3 |  |
| MUL.D | F4, F2, F0 | F4 = aX(i) | 4 | Takes 6 cycles to execution |
| L.D | F2, 8(R1) | F2 = X(i+1) | 5 |  |
| L.D | F10, 8(R2) | F10 = Y(i+1) | 6 | F10 is later used to store the result. |
| MUL.D | F8, F2, F0 | F8 = aX(i+1) | 7 | To store the mult result we use different register F8 as previous index result is stored in F4. |
| L.D | F2, 16(R1) | F2 = X(i+2) | 8 | Unroll the next iteration in the loop. |
| L.D | F14, 16(R2) | F14 = Y(i+2) | 9 | F14 is later used to store the result. |
| MUL.D | F12, F2, F0 | F12 = aX(i+2) | 10 | Takes 6 cycles to execution |
| ADD.D | F6, F4, F6 | F6 = aX(i) + Y(i) | 11 | No dependency |
| DAADIU | R1, R1, #24 | Next three elements of vector X | 12 |  |
| ADD.D | F10, F8, F10 | F10 = aX(i+1) + Y(i+1) | 13 |  |
| DAADIU | R2, R2, #24 | Next three elements of Vector Y | 14 |  |
| DSLTU | R3, R1, R4 | Compare end of loop | 15 |  |
| ADD.D | F14, F12, F14 | F14 = aX(i+2) + Y(i+2) | 16 |  |
| S.D | F6, -24(R2) | Store the result in Yi | 17 |  |
| S.D | F10, -16(R2) | Store the result F10 in Y(i+1) | 18 |  |
| BNEZ | R3, foo |  | 19 | 1 more stall for branch |
| S.D | F14, -8(R2) | Store F14 in Y(i+2) | 20 | Store the third element in the vector Y |

We ignored the loop overhead by unrolling the loop for every three elements without any stalls created by using the new registers to store the intermediate results.

Total cycles = 20 for 3 elements .

Execution time per element of result = 20/3 = 6.66

**3.17 [20] <3.3>** An (m,n) correlating branch predictor uses the behavior of the most

recent m executed branches to choose from 2m predictors, each of which is an nbit

predictor. A two-level local predictor works in a similar fashion, but only

keeps track of the past behavior of each individual branch to predict future

behavior.

There is a design trade-off involved with such predictors: Correlating predictors

require little memory for history which allows them to maintain 2-bit predictors

for a large number of individual branches (reducing the probability of branch

instructions reusing the same predictor), while local predictors require substantially

more memory to keep history and are thus limited to tracking a relatively

small number of branch instructions. For this exercise, consider a (1,2) correlating

predictor that can track four branches (requiring 16 bits) versus a (1,2) local

predictor that can track two branches using the same amount of memory. For the

following branch outcomes, provide each prediction, the table entry used to make

the prediction, any updates to the table as a result of the prediction, and the final

misprediction rate of each predictor. Assume that all branches up to this point

have been taken. Initialize each predictor to the following:

Correlating predictor

Entry Branch Last outcome Prediction

0 0 T T with one misprediction

1 0 NT NT

2 1 T NT

3 1 NT T

4 2 T T

5 2 NT T

6 3 T NT with one misprediction

7 3 NT NT

Local predictor

Entry Branch Last 2 outcomes (right is most recent) Prediction

0 0 T,T T with one misprediction

1 0 T,NT NT

2 0 NT,T NT

3 0 NT T

4 1 T,T T

5 1 T,NT T with one misprediction

6 1 NT,T NT

7 1 NT,NT NT

Branch PC (word address) Outcome

454 T

543 NT

777 NT

543 NT

777 NT

454 T

777 NT

454 T

543 T

**Solution:**

Correlating predictor

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Branch PC Mod 4 | Entry | Prediction | Outcome | Table Update | Mispredicted |
| 2 | 4 | T | T | - | No |
| 3 | 6 | NT | NT | change to NT | no |
| 1 | 2 | NT | NT | - | no |
| 3 | 7 | NT | NT | - | no |
| 1 | 3 | T | NT | change to T with one misprediction” | yes |
| 2 | 4 | T | T | - | no |
| 1 | 3 | T | NT | change to NT | yes |
| 2 | 4 | T | T | - | no |
| 3 | 7 | NT | T | change to NT with one misprediction | yes |

MisPrediction Rate = 3/9 = 0.33 or 33.33%

Local Predictor

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Branch PC Mod 4 | Entry | Prediction | Outcome | Table Update | Mispredicted |
| 0 | 0 | T | T | T | no |
| 1 | 4 | T | NT | T | yes |
| 1 | 1 | NT | NT | No change | no |
| 1 | 3 | T | NT | T | yes |
| 1 | 3 | T | NT | NT | yes |
| 0 | 0 | T | T | No change | no |
| 1 | 3 | NT | NT | No change | no |
| 0 | 0 | T | T | No change | no |
| 1 | 5 | T | T | T | no |

MisPrediction Rate = 3/9 = 0.33 or 33.33%