

CSE502 Lecture 15 - Tue 3Nov09

Review: MidTerm Thu 5Nov09 - Outline of Major Topics

Computing system: performance, speedup, performance/cost

Origins and benefits of scalar instruction pipelines and caches

Pipeline Hazards

Structural – need more HW to wait less

Data Dependence – **RAW** (WAR & WAW resolvable)

ByPass HW lessens **RAW** delays even if in-order Xeq

Dynamic HW support out-of-order Xeq & speculation

Control – where fetch after jumps or branches

Instruction Level Parallelism (ILP) => pipelines

Hazards limit fast pipelining

Ways to lessen impact of hazards

More memory and register ports

Branch/Jump prediction

Out-of-order execution

Speculative execution

Code rescheduling (moving)

Static – by compiler SW
Loop Unrolling
(Software pipelining)

Dynamic – by Tomasulo-style HW
Advantages of Tomasulo
Disadvantages of Tomasulo

Superscalar (2-8 instructions per cycle) vs VLIW
Advantages and disadvantages of each
Which better? Why?
Fast CPUs use both – how? Why?

Thread-level support

Cray/Tera MTA-1 128 threads * 256 CPUs, NO caches
Multi-CPU (2-8 cores) on a chip instead of speculation
Less power per core from lower voltage, less logic

Vector Processing

Loop unrolling

Loop execution time

Without chaining

With chaining

****OPEN BOOK****

****OPEN NOTES****

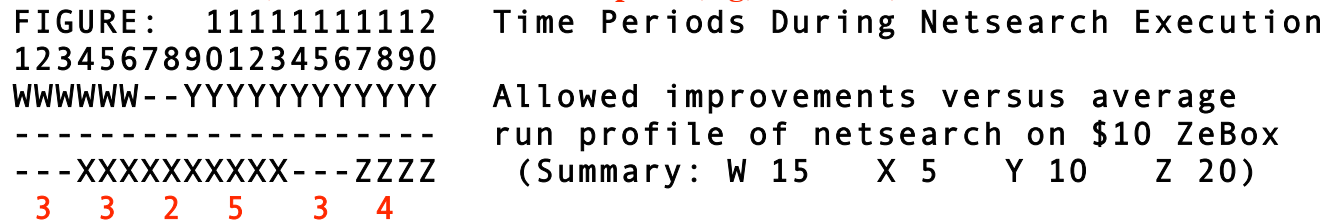
****CLOSED FRIENDS****

You have 75 minutes for 3 questions with 8 parts on 3 pages = 100 points. Read all questions before you start. Some easy questions may be later. Show your math work for partial credit. Note any special assumptions that you make for any question. Raise your hand for help. Write your answers on fronts and backs of this exam, and hand it in directly to me. PLEASE PUT YOUR SUNY ID WHERE I MAY SEE IT DURING THE EXAM.

1) (40 pts) Your NETalker company is designing a new wireless ZeBox, that will run your patented netsearch code 95% of its working time. You must make a critical hardware choice affecting both the cost and speed of ZeBox. The old Box costs \$36 and takes an average of 40 seconds per search. A new ZeBox that takes 40 seconds per search costs only \$10. Four optional hardware improvements to ZeBox are feasible: **W, X, Y, Z**. Each speeds up only parts of netsearch execution, as shown in the figure.

For an average run of netsearch divided into 20 equal time periods, only **W** can speed up 3 periods; **W** or **X** (but not both together) can speed the next 3 periods; only **X** for 2 periods; **Y** or **X** for 5; only **Y** for 3, and **Y** or **Z** for 4. Each improvement has a different speedup factor for its sections of the run and an additional cost. **W** is 15 times faster for \$15 more. **X** is 5 times faster for \$5. **Y** is 10x for \$10 more. **Z** is 20 times for \$20 more. If two improvements are possible during a time period, only the faster one has any effect. The question is **what WXYZ combination gives the best performance-to-total cost ratio**. For example, if ZeBox has all four optional improvements **W+X+Y+Z**, total_cost for each new ZeBox will be \$60. If only **X**, \$15. If **Y+Z**, \$40.

(Hardware cost for each options, eg, \$10 for Y, is same whether it is used 1/20 of time or 12/20ths)



1A) (15 pts) **What is the maximum speedup possible for netsearch?** (If none of WXYZ used, speedup is 1.)

For fastest, use all: W + X + Y + Z Speedup = 20/1.8 = 11.111 times faster

Shortest run time is: 6/15 + 2/5 + 8/10 + 4/20 = 2/5 + 2/5 + 4/5 + 1/5 = 9/5 = 1.8 vs 20.0

1B) (10 pts) **What is the performance versus total_cost ratio for the fastest version of the new ZeBox?**

ZeBox W X Y Z Performance is speedup here = 11.111 times faster

Total cost = \$10 + \$15 + \$5 + \$10 + \$20 = \$60 Ratio = 11.111/60 = 1.852 times faster per \$10

1C) (15 pts) **Which of the 16 possible W,X,Y,Z combinations is best? Give its performance-cost ratio.**

{ Do NOT attempt to evaluate all 16 combinations explicitly unless you have extra time at the end! }

{ Instead, evaluate some carefully chosen combinations and state your reasons for skipping the rest. }

The Z option is most expensive and most rarely usable (4/20ths), so check performance/cost for W+X+Y

RunTimeWXY = 6/15 + 2/5 + 12/10 = 2/5 + 2/5 + 6/5 = 2.0 vs 20 20/2.0 = 10.0 times faster

Cost = \$10+15+5+10 = \$40 RatioWXY = 10.0/40 = 2.50 times faster / \$10 => better than WXYZ

If we eliminate any one other option, 2(X) or 3(W) or 7(Y) of 20 20ths of runtime will NOT be improved so runtime will be nearly 2 or 3 or 7 units longer compared to runtime of 2.0 for WXY. The runtime will be nearly 2 times longer or more, but the cost will not be reduced by 2 or more, so the performance/cost ratio will be smaller. The best performance/cost ratio is for improvements combination W+X+Y.

2) (30 pts) A dual-issue (two pipelines) multiscalar version of the integer DLX 5-stage pipeline has all feasible data forwarding and enough hardware to avoid all structural hazards. However, ALU results and memory loads are available only at stage (EX, MM) end. ALU inputs and memory stores are needed at stage (EX,MM) start.

2A) (5 pts) **What is the instruction penalty for a load memory instruction followed by an ALU instruction using its result? What is the average cycle penalty?** You may use a sketch to justify your answer, if you wish.

Cycle	0	1	2	3	4	5	6
LD 0	IF	ID	EX	MM	WB		
mt 0	IF	ID	EX	MM	WB		
AL 1	IF	ID	==>	EX	MM	WB	
?? 1	IF	ID	==>	EX	MM	WB	
?? 2		IF	==>	ID	EX	MM	WB
...			ALU ok	Load ok			

After Load from memory fetched at Cycle 0, memory value is first known at MM-WB latch and can be forwarded to start of EX stage for an ALU operation fetched at Cyc 2. If a dependent LD-ALU pair occur right after each other with the LD fetched into pipe1 of cycle 0, the dependent ALU will not be issued until cycle 1, when it also goes in pipe1. Its EX stage must stall another cycle. The empty pipe2 and full cycle of stalls, correspond to the loss of pipeline fetch slots for 3 instructions. The **“INSTRUCTION PENALTY” is 3**. The stall is 2 cycles, a **“CYCLE PENALTY” of 2.0** cycles.

Cycle	0	1	2	3	4	5	6
?? 0	IF	ID	EX	MM	WB		
LD 0	IF	ID	EX	MM	WB		
AL 1	IF	ID	==>	EX	MM	WB	
?? 1	IF	ID	==>	EX	MM	WB	
?? 2		IF	==>	ID	EX	MM	WB
...			ALU ok	Load ok			

If a dependent LD-ALU pair occur right after each other with the LD fetched into pipe2 of cycle 0, the dependent ALU will be issued in cycle 1, when it goes in pipe1. Its EX must stall a full cycle, corresponding to the loss of pipeline fetch slots for 2 instructions. The **“INSTRUCTION PENALTY” is 2**. The stall of 1 full cycle gives a **“CYCLE PENALTY” of 1.0** cycles.

Since both cases are equally likely, the average **“INSTRUCTION PENALTY” is 2.5** and the average **“CYCLE PENALTY” is 1.5** cycles.

2B) (10 pts) **Identify** and **give the average cycle penalty** for two **other** (non-zero) DLX data hazards.

(OK if list any two of the following non-zero penalties.)

Integer DLX pipelines have no WAR nor WAW data hazards, so we seek only pairs with a RAW hazard.

Taking a hint from 2D below, the other likely pairs are ALU-ALU, ALU-store, and ALU-load. There is also a tiny ALU-Branch problem, ignored in 2D. The other combinations of linked Load-Store (Load-Load) (redundant unless a bad non-atomic Test&Set Lock or a simulated and costly indirect address for a Store or Load) and Load-Branch (who would keep a loop index or condition in so-slow memory?) are reasonably ignored, as unlikely. See blue line in figures below for cycle penalties after lead ALU of an ALU-xxx pair.

Cycle	0	1	2	3	4	5	6
AL 0	IF	ID	EX	MEM	WB		<==
mt 0	IF	ID	EX	MEM	WB		
AL 1		IF	ID	EX	MEM	WB	
?? 1		IF	ID	EX	MEM	WB	
...				ALU ok	Load ok		

or

Cycle	0	1	2	3	4	5	6
?? 0	IF	ID	EX	MEM	WB		
AL 0	IF	ID	EX	MEM	WB		<==
AL 1		IF	ID	EX	MEM	WB	
?? 1		IF	ID	EX	MEM	WB	
...				ALU ok	Load ok		

ALU-ALU If first ALU fetched at Cycle 0 and issued into pipe1, the following dependent ALU will not be issued until cycle 2 and its EX will start in cycle 3, for a “CYCLE PENALTY” of 1.0 cycle. If the first ALU fetched at Cycle 0 is issued into pipe2, the following dependent ALU will be issued in cycle 1 into pipe2, and there is no (0 cycle) penalty. The average “CYCLE PENALTY” is 0.5 cycle.

(ALU-Store If ALU operation determines the value to be stored in memory, it is not needed until start of MEM, so there is no stall and no penalty.)

ALU-Store If the ALU operation determines a register value used to calculate the effective address of the Store, this is same as ALU-ALU; the “CYCLE PENALTY” is 0.5.

ALU-Load If the ALU operation determines a register value used to calculate the effective address of the Load, this is same as ALU-ALU; the “CYCLE PENALTY” is 0.5.

ALU-Branch:

Cycle 0	1	2	3	4	5	6	
AL 0 IF	ID	EX	MEM	WB		<==	
mt 0 IF	ID	EX	MEM	WB			
BR 1	IF	==>	ID	EX	MEM	WB	
?? 1	IF	==>	ID	EX	MEM	WB	
...			ALU ok				
or							
Cycle 0	1	2	3	4	5	6	
?? 0 IF	ID	EX	MEM	WB			
AL 0 IF	ID	EX	MEM	WB		<==	
BR 1	IF	==>	ID	EX	MEM	WB	
?? 1	IF	==>	ID	EX	MEM	WB	
...			ALU ok				

ALU-Branch If the ALU operation determines the condition register 0/non-0 value tested by the branch at end of its ID stage or, less likely, the register used in mid-ID stage as the address for a jump, the ALU result is needed at the start of the Branch's ID stage. If the lead ALU goes into pipe1 of cycle 0, there is a **"CYCLE PENALTY" of 2.0**. If the lead ALU goes into pipe2 of cycle 0, there is a **"CYCLE PENALTY" of 1.0**. The average **"CYCLE PENALTY" is 1.5**.

If a Load operation determines a register value used to calculate the effective address of a Load or Store,
The delay is the same as for Load-ALU. Load-Load and Load-Store have an average **"CYCLE PENALTY" of 1.5**.

Load-Branch If the Load operation determines the condition register 0/non-0 value tested by the branch at end of its ID stage or, less likely, the register used in mid-ID stage as the address for a jump, the Load result is needed at the start of the Branch ID stage. The stalls are 1 cycle more than for ALU-Branch, so there is an average **"CYCLE PENALTY" of 2.5**.

2C) (5 pts) Choice Q has a Branch Target Buffer (BTB) that is searched in each stage 1 (IF) with the address of the instruction being fetched. Each BTB entry gives the address (the search key) of a recently taken branch or a jump plus its target address for the next instruction to fetch after it. If an instruction address is not in BTB, it is assumed not to change the Program Counter. If it is a control instruction, the actual target address is known at the end of stage 2 (ID) and checked against the prediction. If the prediction was wrong, the BTB is updated, incorrectly fetched instructions are flushed from the pipeline (annulled) and the correct fetch address is used at the middle of stage 3 (EX). If each control instruction that is taken has its address in the BTB 90% of the time and each control instruction with its address in the BTB is taken 80% of the time, **what is the average cycle penalty for each DLX control instruction if choice Q is implemented? (60% of branches/jumps taken.)**

We know: If Branch/Jump is Taken (60%), then it is in BTB (90%) AND it is NOT in BTB (10%); AND If Branch/Jump is in BTB (Y%), then it is Taken (80%) AND it is NOT Taken (20%).

**So Probability(in BTB AND Taken) = $0.8 \times Y/100 = .54 = 0.60 \times 0.90$, which yields $Y = 54/0.8 = 67.5$
If Branch/Jump is in BTB (67.5%), then it is Taken (80%) AND it is NOT Taken (20%).**

Penalty when Branch/Jump in BTB and NOT Taken or When Branch/Jump NOT in BTB and Taken
 $0.675 \times 0.20 = 0.135$ + $0.10 \times 0.60 = 0.06$

P(Wrong) = $0.135+0.06 = 0.195 = P(2.0 \text{ cycle penalty})$ so average cycle penalty per Branch/Jump = 0.39

2D) (10 pts) Assume that BTB choice Q has been implemented and dual-issue DLX instructions run with these frequencies: 53% ALU, 21% Load, 12% Store, and 14% Branch/Jump. Assume these data hazard frequencies:

Closest possible hazard after each ALU instruction - 10% adjacent dependent ALU, 5% 2-away dependent ALU, and 2% 3-away dependent ALU;
10% adjacent dependent store, 5% 2-away dependent store, and 2% 3-away dependent store; and

Closest possible hazard after each memory load instruction - 20% adjacent dependent ALU, 10% 2-away dependent ALU, and 5% 3-away dependent ALU (the compiler eliminates redundant stores).

If there are no structural hazards and no memory system stalls, **what is the average number of instructions completed per cycle (IPC) for this dual-issue integer DLX processor?**

Only data and control hazards (penalties in 2A & 2B & 2C) are left to add cycles to the base CPI = 0.5 .

Use cycle penalties: Load-ALU = 1.5 ALU-ALU = 0.50 ALU-store = 0.0 Control = 0.39

CPI = $0.5 + 0.21 \times (0.20 \times 1.5 + 0.10 \times 0.5)$ + $0.53 \times 0.10 \times 0.50$ + 0.14×0.39

= $0.5 + 0.21 \times (0.30 + 0.05)$ + 0.53×0.05 + $0.0546 = 0.5 + 0.0735 + 0.0265 + 0.0546 = 0.6546$

IPC = $1/0.6546 = 1.528$ instructions completed per cycle

NOTE: There originally was a question part 2E) giving a choice R for Branches, a 3-instruction delay slot with a 68% chance of compiler moving one useful instruction into the slot, 11% for 2 in the slot and 1% for 3 useful. The resulting branch penalty is **2.20/2 cycles = 1.10 cycles**. For this choice R, **CPI = $0.60 + 0.154 = 0.754$ and $IPC = 1/0.754 = 1.326$ instructions completed per cycle. BTB speeds runs by $1.528/1.326 = 1.152$, 15% faster.**

3) (20 pts) This is one possible DLX code to calculate a Sum A*X Plus Y inner loop for Gaussian elimination.

```

start:                                     ; assume R1, R2, F2 set correctly before here
saxpy:  LD F2, 0(R1)                        ; load X[j]
          MULTD F4, F0, F2                    ; Multiply a * X[j]
          LD F6, 0(R2)                        ; load Y[j]
          ADDD F6, F6, F4                      ; Add a*X[j] + Y[j]
          SD 0(R2), F6                         ; STORE A * X[l] + Y[l]
          ADDI R1, R1, #8                      ; increment X index
          ADDI R2, R2, #8                      ; increment Y index
          SGTI R3, R1, final_ndx              ; test if done
          BEQZ R3, saxpy                       ; loop if not done
    
```

exit:

Assume this code runs on a single-issue DLX INT+FP pipeline with a one cycle delay slot after each branch. **Unroll this loop twice** and schedule its instructions to **minimize** the number of **stalls** between instructions, using the latencies in the table that follows. Eliminate any unneeded overhead instructions. **Write the resulting code.** If the loop is executed for vectors X[j] and Y[j], each of **size 50**, what is the **total number of stall cycles wasted** between **start** and **exit** using your unrolled code? **What is the value of R1 at exit?**

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
a) FP ALU op	Another FP ALU op	3
b) FP ALU op	Store double	2
c) Load double	FP ALU op	1
d) Load double	Store double	0
e) Integer op	Integer op	0

The left side below shows two copies of the loop code with duplicate overhead instructions combined, registers separated, and LD/SD offsets corrected. The right side shows the code on left rescheduled to eliminate all stalls.

start:
saxpy: LD F2, 0(R1)
LD F8, 8(R1)
MULTD F4, F0, F2
MULTD F10, F0, F8
LD F6, 0(R2)
LD F12, 8(R2)
ADDD F6, F6, F4
ADDD F12, F12, F4
SD 0(R2), F6
SD 8(R2), F12
ADDI R1, R1, #16
ADDI R2, R2, #16
SGTI R3, R1, final_ndx
BEQZ R3, saxpy

exit:

start:
saxpy: LD F2, 0(R1)
LD F8, 8(R1)
MULTD F4, F0, F2
MULTD F10, F0, F8
LD F6, 0(R2)
LD F12, 8(R2)
ADDD F6, F6, F4
ADDD F12, F12, F4
ADDI R1, R1, #16
SGTI R3, R1, final_ndx
SD 0(R2), F6
SD 8(R2), F12
BEQZ R3, saxpy
ADDI R2, R2, #16

exit:

NO stalls left 0 total

$R1 = \text{original } R1 + 25 * 16 = \text{original } R1 + 400$