# Midterm I

**SOLUTIONS**  
March 30\(^{th}\), 2011  
CS252 Graduate Computer Architecture

<table>
<thead>
<tr>
<th>Your Name:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SID Number:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
<th>Possible</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total</th>
<th>100</th>
</tr>
</thead>
</table>
[ This page left for π ]

3.141592653589793238462643383279502884197169399375105820974944
Question #1: Short Answer [20pts]

Problem 1a[3pts]: Explain the difference between implicit and explicit register renaming. Can an architecture with Tomasulo scheduling use explicit register renaming? Explain.

Implicit register renaming occurs as a consequence of scheduling, as evidenced by the original Tomasulo algorithm for the 360/91. It changes programmer-visible registers into internal tags or values. Explicit register renaming replaces user-visible register names with internal, physical register names. Often explicit register renaming is performed in a separate stage between the fetch stage and decoding / scheduling stages. Yes, an architecture with Tomasulo scheduling could use explicit register renaming on the entrance to the pipeline (to remove WAR/WAW hazards), then use reservation-stations to schedule instructions (removing RAW hazards); the scheduling would work with physical register names instead of tags.

Problem 1b[3pts]: What are precise exceptions? Explain how precise exceptions are achieved in an architecture with explicit register renaming (such as the R10000 or 21264, both discussed in class)?

Precise Exceptions are interruptions in the flow of instructions with a single identifiable instruction such that (1) all instructions prior to that instruction have committed their state and (2) neither the identified instruction nor any following instructions have altered the state of the machine. Machines with explicit register renaming can restore the state of the machine to a precise exception point by restoring the register rename mapping table (and freelist) to the value it had at the precise exception point. (i.e. such machines do not have to restore actual register values, since the correct values are still in the physical register file).

Problem 1c[3pts]: What is simultaneous multithreading? What is the basic argument for using simultaneous multithreading?

Simultaneous Multithreading is a multithreading technique that mixes instructions from multiple threads into a single superscalar pipeline, thereby getting higher utilization of the functional units. Instructions from different threads can be issued in the same cycle to the pipeline (the “simultaneous” part of SMT). The basic justification for simultaneous multithreading is that high-issue-rate processors typically have a lot of wasted issue slots.

Problem 1d[2pts]: What was one of the main insights of the RISC work? Hint: simply reducing instructions was a consequence, not an insight.

One of the most important lessons of the RISC work was that computer designers should consider the whole picture (including both software and hardware layers) rather than micro-optimizing any one layer (such as the hardware). RISC also placed a strong focus on measurement and benchmarks as providing objective mechanisms for evaluating performance.
Problem 1e[3pts]: Yeh and Patt described two different types of branch predictors: those that exploit correlation between successive branches (global history) and those that exploit the past behavior of each branch in isolation (local history). Does it make sense to combine these ideas in some way? Why or why not? How would you combine them? Explain with a block diagram.

Yes, it makes sense to combine both GA and PA styles of branch prediction; we read at least two papers that showed that different branches are best handled by different algorithms (because they have different behavior). The easiest way to combine them is to use a tournament branch predictor that uses one predictor to determine the type of prediction to use for each branch, then uses two other predictors to perform each type of prediction. A generic diagram for a tournament predictor is as follows:

![Tournament Predictor Diagram]

Problem 1f[3pts]: Why does it make sense to use trace scheduling with a VLIW? What is the structure of the resulting code? Is there any relationship between trace scheduling and branch prediction?

Since a VLIW is statically scheduled, every operation that is put into a single instruction must be able to actually be run in parallel (no stalling). Unfortunately, with typical programs, there are very few instructions in each block of code from which to choose a set of operations to run in parallel (typical code has a branch every 5 instructions). Trace scheduling picks a “high-probability” trace through the code and schedules all of the instructions along the trace as if they were in the same block (i.e. no branches). The resulting code has a lot of instructions running in parallel, with many potential exit points (where the original branches might disagree with the selected trace). The resulting code executes blocks of instructions followed by “fixup code” to handle undoing the result of incorrectly executed instructions along the trace. Yes, there is a relationship between trace scheduling and branch prediction: the trace represents a group of co-predicted branches (they are predicted to follow the trace).

Problem 1g[3pts]: Why is it advantageous to speculate on memory dependencies between load and store instructions? Describe two techniques for performing such speculation.

Most of the time, loads do not depend on stores that are in the pipeline at the same time. Thus, a conservative strategy that never launches loads until all dependencies are computed (i.e. addresses are known) can potentially lose performance (especially since we would like to start loads early to deal with cache misses). Assuming that one has a mechanism for recovering from mis-speculation (i.e. ROB), two possibilities are: (1) guess that loads with unknown dependencies never depend on prior stores (“naive speculation”) or (2) try to learn the dependencies between loads and stores by starting with naive speculation, then keeping extra information when such speculation fails (using extra bits in the cache or “store-sets” to track which stores a given load depends on).
Problem #2: Out-of-Order Superscalar Processor [30pts]

In this problem, we will explore a three-way superscalar processor. As shown in the above figure, this processor has two execution pipelines – each of which is slightly different. All three pipelines can execute integer operations in a single cycle (such as arithmetic ops or branch conditions) and can execute Galois Field operations in 3 cycles. In addition, Pipeline 1 can execute floating point operations: addf (2 cycles), multf (3 cycles), divf (5 cycles). Further, Pipeline 3 can execute memory operations in 4 cycles, the first cycle of which computes the address, the second two cycles of which perform the actual data cache operation, and the fourth cycle of which performs a “Tag Check” to match the current memory operation against the data cache. All operations are fully pipelined. Thus, for instance, multiple load operations can be in different stages of Pipe 3 at the same time.

The earlier part of the pipeline, namely instruction management consists of 6 stages:

- Stages $F_1$ and $F_2$ handle fetching up to three instructions at a time from the instruction cache and are completely pipelined (the instruction cache takes two cycles to fetch instructions). A Branch Target Buffer (BTB) helps to compute the next address.
- Stage D begins decoding the instructions and computing target addresses.
- Stage R explicitly renames registers for three instructions at a time. This stage also performs an initial instruction decode and checks the instruction cache tags.
- Stage ROB is a Reorder buffer and Ready Instruction Selection stage. The ROB stage can select up to three instructions to pass onto the $O_p$ stage. The ROB stage must make sure that the instructions that it selects are ready to go (i.e. will have all arguments by the time they exit the $O_p$ stage) and that they can all execute simultaneously (e.g. it cannot select two floating point instructions in same cycle).
- Stage $O_p$ fetches operands for three instructions from the register file and/or other stages of the processor. It further decodes the instructions.

Problem 2a(2pts): What is the minimum number of instructions that we must flush on a branch misprediction? Explain, assuming that branches are completed in the EX$_1$ stage.

*If we assume that the branch is the latest instruction in its group of three fetched instructions (i.e. no instructions were fetched after the branch during the cycle in which we fetched the branch), then the answer is 18: if the branch makes it through execution in the fastest possible number of cycles, then it will be done in 7 cycles (from fetch to execution). This 7 cycles includes $6 \times 3 = 18$ instructions after the branch.*
Problem 2b[2pts]: What is the earliest stage of instruction execution (F₁, F₂, D, R, …) that we can start a branch prediction operation? Explain. How would the BTB help to make up for this?

The earliest that we can start a branch prediction is during the F₁ stage, since we only need the instruction address to be available to start looking up in the history table (trick question?). However, since most interesting branch prediction algorithms take multiple cycles, and since the target address of the branch will not be available until the R stage, the BTB will help us by making a preliminary prediction during the F₁ stage allowing a response to a branch to occur in the cycle after we start fetching the branch instruction (before knowing that it is even a branch!).

Problem 2c[3pts]: Why does the presence of explicit register renaming simplify the process of feeding values from instructions that have finished execution (but not yet committed) to instructions that are about to start execution (i.e. simplify the scheduling/RAW hazard elimination process)? Hint: how many places can hold an operand for a new instruction? Do instruction operands have to be fetched form the ROB? Why or why not?

Explicit register renaming ⇒ every new value is assigned a unique physical register name. Consequently, the operands for a new instruction are relatively easy to find: either they are coming from an instruction that is still in the pipeline, in which case there is a unique place in the pipeline holding the result (identified by the register name), or they are in the register file. Because of explicit renaming, we do not need to put values in the ROB: Uncommitted values are simply placed in the register file; we restore values at a precise exception point simply by restoring register mappings.

Problem 2d[2pts]: Although we could write back results to the register file as soon as they are completed (e.g., write the result of an integer operation after EX₁) the above pipeline shows results passed down their respective pipelines until they hit the end before they are written back to the register file. What are the pros and cons of this approach?

The pros of this approach is that we only need as many write ports in the register file as number of pipeline stages (since values go from the end of the pipeline into the register file). If we wrote immediately to the register file as soon as a value is ready, we might need many more simultaneous writes to the register file. The cons of holding values in the pipelines is increased complexity for the bypass/forwarding network.

Problem 2e[3pts]: Explain what the Op stage does (possibly with a block diagram). How does it decide which values go to which instruction operands? Can the decision about which values will be used come from the ROB/Scheduler stage? Why or why not?

The Op stage figures out where to get values for operands of instructions that have just been released for execution. Because of explicit renaming, these values can come from only two possible places: either the register file (if the producing instruction is completed) or from one place in the execution pipelines. Thus, the Op stage must have $3 \times 2 = 6$ register read ports (two for each incoming instruction) and 6 bypass paths from the execution pipelines, each of which can relay values from every pipeline stage of every pipeline.

Assuming that instructions already in the execution pipelines cannot stall, then the timing of each instruction execution is completely set after it leaves the ROB. Thus, the ROB stage can decide exactly where the Op stage should get operands from.
Problem 2f[2pts]: Can you come up with two reasons why we might not want to stall instructions once they have passed the ROB stage (i.e. have entered the decode stage and/or one of the execution pipelines). *Hint: one of your reasons may be related to (2e).*

Some reasons include: (1) stalling an instruction can be difficult, since you need to stall all instructions prior to the instruction in the pipeline and produce bubbles in the pipeline afterwards. In particular, the decision to stall often happens at the end of the cycle and must be propagated quickly up to prior stages; thus, stalling can impact the cycle-time of the overall pipeline. (2) Stalling an instruction in one pipeline may require stalling dependent instructions in other pipelines that have already been released by the ROB stage. This requires dependency analysis that is not really present in the pipeline; the simplest solution would be to stall all pipelines, but that is clearly undesirable from a performance standpoint [see second part of answer to 2f].

*I was pretty lenient with this question, as long as your answers seemed reasonable.*

Problem 2g[4pts]: Suppose that we do not provide an ability to stall the execution pipelines (as discussed in 2d). How could we structure the instruction Selection logic of the ROB so that a cache miss never causes us to flush the ROB? What is the resulting penalty in number of cycles between a load and a dependent instruction? How could we use prediction to reduce this penalty (*Hint: the Alpha 21264 did this.*)

*Make sure that we never launch instructions from the ROB that are dependent on loads until after we have guaranteed that they will not cache miss (i.e. passed the TC stage). If we do this, then we can simply wait until a load is completely committed (either with or without a cache miss) before we start dependent instructions. The result of this policy is that (in the best case of cache hit), we need 4 cycles between a load and dependent instruction (when load passes TC, we will have dependent instruction in ROB). If we are willing to make a guess on whether or not we will hit in the cache (and are thus willing to flush the ROB on mistake), we could start dependent instructions early, thus having a dependent instruction in the the Op stage when our load is in the Mem2 stage (reducing penalty to 2 cycles).*

Problem 2h[3pts]: Note that the TagC stage is after the two memory stages. Explain how store instructions could function in this pipeline without incorrectly overwriting cache lines in the first-level cache during a cache miss. Do you have to do anything special when there are two back-to-back stores which both miss in the cache (considering the non-stall philosophy of 2g)?

*We need to buffer stores so that the current store will check its tag during TC while a previous (checked) store is writing to the cache. We may need to have multiple store results held off the pipeline since the “non-stall” philosophy would allow more than one store to be admitted into the pipeline before we recognize the first cache miss; such a structure would need to be able to respond to later loads for consistency reasons. Note that an alternative would be to “bounce” cache-missed stores back to the ROB or an associate load-store buffer while starting the cache miss process. Such “aborted” stores could be restarted later, after the cache-miss completes.*
Problem 2i[4pts]: Note that a load is not completed until the end of EX₃ and that a store must have its value by the beginning of EX₂. Consider the following common sequence for a memory copy:

```
loop:  ld  r1, 0(r2)   ; load value from source
       st  r1, 0(r3)   ; store value at destination
       add r2, r2, #4 ; increment source pointer
       add r3, r3, #4 ; increment destination pointer
       subi r4, r4, #1 ; decrement length
       bnez r4, loop   ; loop if non-zero
```

Assuming that there is no branch delay slot in the instruction set, assuming that both source and destination memory arrays are in cache, and assuming that we use prediction as suggested in 2g, what would be the average number of instructions per cycle (IPC) achieved by this loop? Explain your reasoning carefully by showing the pairings of instructions that would pass the decode stage for the first three iterations of the loop. How many slots in the ROB would be needed to sustain this rate?

*Every loop has 2 memory ops and 4 instruction ops. Thus, if all is well, we might expect that we could fill up all schedule slots and have an IPC of 3. The following shows that this is true*

Assuming that everything is in the cache and that we predict cache misses ⇒ that we need 2 cycles between load and normal consumer. However, we can get by with 1 cycle between load and consuming store. Further, we have only 1 memory pipeline, so there can be only one memory operation in flight per cycle. Thus, assume that we fetch 3 instructions per cycle continuously, then instructions will leave the ROB as follows (oldest ready instruction first):

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction 1</th>
<th>Instruction 2</th>
<th>Instruction 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ld r1, 0(r2)</td>
<td>add r2, r2, #4</td>
<td>&lt;empty&gt;</td>
</tr>
<tr>
<td>2</td>
<td>add r3, r3, #4</td>
<td>subi r4, r4, #1</td>
<td>bnez r4, loop</td>
</tr>
<tr>
<td>3</td>
<td>st r1, 0(r3)</td>
<td>add r2, r2 #4</td>
<td>add r3, r3, #4</td>
</tr>
<tr>
<td>4</td>
<td>ld r1, 0(r2)</td>
<td>subi r4, r4, #1</td>
<td>bnez r4, loop</td>
</tr>
<tr>
<td>5</td>
<td>ld r1, 0(r2)</td>
<td>add r2, r2, #4</td>
<td>add r3, r3, #4</td>
</tr>
<tr>
<td>6</td>
<td>st r1, 0(r3)</td>
<td>subi r4, r4, #1</td>
<td>bnez r4, loop</td>
</tr>
<tr>
<td>7</td>
<td>st r1, 0(r3)</td>
<td>add r2, r2, #4</td>
<td>add r3, r3, #4</td>
</tr>
<tr>
<td>8</td>
<td>ld r1, 0(r2)</td>
<td>subi r4, f4, #1</td>
<td>bnez r4, loop</td>
</tr>
<tr>
<td>9</td>
<td>ld r1, 0(r2)</td>
<td>add r2, r2, #4</td>
<td>add r3, r3, #4</td>
</tr>
</tbody>
</table>

And, this pattern will repeat... I actually showed 4½ iterations to make the pattern clearer. Thus, the IPC=3 (3 instructions per cycle) after things get started.

To compute the needed size of the ROB, we could start by looking at the 5-stage pipeline (pipeline #1) and say that the ROB needs at least enough instructions to hold (5+1)×3 = 18 instructions (maximum 5-stage pipeline + Op).

To be more accurate: we would notice that a load and its corresponding store are separated by 2 cycles, so when the store completes, we have two cycles of additional fetching. Couple this with the fact that the LD/ST pipeline is shorter by 1 cycle than the 5-stage pipeline, we would say that we need an additional 1 cycle of instructions (3) over the above estimate. Finally, we would note that the actual load (oldest) instruction would have committed as soon as it finishes, so we have 1 less instruction.

Couple all of this together, we get 18+3-1 = 20 slots in the ROB. Needless to say, I accepted anything that was reasonably close to this estimate.
Problem 2j[5pts]: In the figure below, draw all of the bypass paths required to allow this processor to operate at maximum instruction throughput. Label bypass paths with the results that they carry. Here is the key you should use: “I” for integer operations, “G” for Galois Field operations, “A” for floating-point Adds, “M” for floating-point Multiplies, “D” for floating-point Divides and “L” for load results. Hint: Don’t forget bypass paths required for store instructions.

The expected arcs are as follows below. Note that we pass values down each pipeline until they hit the end (to save on number of writeback ports to the register file); consequently, we feed back (bypass) a value from every stage back to the Op stage since there will always be the potential of an integer result. Further, the set of potential values increases as we head further down the pipeline.

To allow for full-speed store operations, we also bypass values that have just completed to the end of the Addr stage (beginning of MEM1 stage). Since values that have already been computed but not yet written back can be bypassed to the end of the Op stage, we only have to do this bypass on the cycle in which computations have finished.
EXTRA CREDIT: Problem 2k[5pts]: Give a rough sketch of how the instruction Scheduler would work in this architecture. Try to be as explicit as possible (keeping in mind that instructions cannot stall after they leave the ROB). How are exceptional cases such as cache misses and divide by zero handled?
[ This page intentionally left blank!]
Problem #3: Software Scheduling [30pts]

For this problem, assume that we have fully pipelined, single-issue, in-order processor with the following number of execution cycles for:

1. Floating-point multiply: **4 cycles**
2. Floating-point divider: **9 cycles**
3. Floating-point adder: **2 cycles**
4. Integer operations: **1 cycle**

Assume that there is one branch delay slot, that there is no delay between integer operations and dependent branch instructions, and that there memory loads and stores require 2 memory cycles (plus the address computation). Assume that all functional units are fully pipelined and bypassed.

Problem 3a[3pts]: Compute the following latencies between instructions to avoid stalls (i.e. how many unrelated instructions must be inserted between two instructions of the following types to avoid stalls)? The first one is given:

- Between a `ldf` and `addf`: 2 Insts
- Between a `ldf` and `multf`: 2 Insts
- Between a `addf` and `divf`: 1 Insts
- Between a `multf` and `addf`: 3 Insts
- Between a `addf` and `stf`: 0 Insts
- Between a `divf` and `addf`: 8 Insts
- Between an integer and `stf`: 0 Insts
- Between an integer and `branch`: 0 Insts

The following code computes a portion of a filter operation. Assume that r1 contains a pointer to the beginning of a window of floating-point numbers and constants. Further, r2 contains a pointer to an accumulation array. Let r3 be the size of the window, f0, and f1 be constants.

```plaintext
filter: ldf F3,0(r1) 2 Stall Cycles
ldf F4,4(r1) 2 Stall Cycles
addf F5,F4,F3 3 Stall Cycles
ldf F6,8(r1) 1 Stall Cycle
multf F7,F6,F0 7 Stall Cycles
addf F8,F7,F5
divf F9,F8,F1
stf 0(r2),F9
addi r1,r1,#12
addi r2,r2,#4
subi r3,r3,#1
bnez r3,filter
nop
```

Problem 3b[2pts]: How many cycles does this loop take per iteration? Indicate stalls in the above code by labeling each of them with a number of cycles of stall:

\[ 15 \text{ stall cycles} + 13 \text{ instructions} = 28 \text{ cycles/iteration} \]
**Problem 3c[4pts]:** Reschedule this code to run with as few cycles per iteration as possible. Do not unroll it or software pipeline it. How many cycles do you get per iteration of the loop now?

```
filter: ldf F6,8(r1)
        ldf F3,0(r1)
        ldf F4,4(r1)
        multf F7,F6,F0
        addf F5,F4,F3
        addf F8,F7,F5
        divf F9,F8,F1
        addi r1,r1,#12
        addi r2,r2,#4
        subi r3,r3,#1
        bnez r3,filter
        addf F5,F4,F3
        addf F15,F14,F13
        addf F8,F7,F5
        addf F18,F17,F15
        divf F9,F8,F1
        divf F19,F18,F1
        addi r1,r1,#24
        addi r2,r2,#8
        subi r3,r3,#2
        stf -4(r2),F9
        stf -4(r2),F19
```

Now we have 6 stall cycles + 12 instructions = 18 cycles/iteration

**Problem 3d[6pts]:** Unroll the loop once and schedule it to run with as few cycles as possible. Ignore startup code. What is the average number of cycles per iteration of the original loop?  
**Hint:** To make this easier, use the 10s digit in naming registers for the second iteration of the loop, i.e. F3 \(\Rightarrow F13\).

```
filter: ldf F6,8(r1)
        ldf F16,20(r1)
        ldf F3,0(r1)
        ldf F4,4(r1)
        multf F7,F6,F0
        ldf F13,12(r1)
        ldf F14,16(r1)
        multf F17,F16,F0
        ldf F15,F14,F13
        addf F5,F4,F3
        addf F18,F17,F15
        addf F8,F7,F5
        addf F19,F18,F1
        divf F9,F8,F1
        divf F19,F18,F1
        addi r1,r1,#24
        addi r2,r2,#8
        subi r3,r3,#2
        stf -8(r2),F9
        stf -4(r2),F19
```

Here is a reasonable example of how to schedule the code. It is unlikely that you can do better than 3 stall cycles, given the relationship between the divf and stf instructions.

Thus, \# of cycles per loop = \((20 \text{ insts} + 3 \text{ stalls})/2 = 23/2 \text{ cycles/loop} = 11.5 \text{ cycles/loop}\)
Problem 3e[5pts]: Software pipeline this loop to avoid stalls. Use as few instructions as possible. Your code should have no more than one copy of the original instructions. What is the average number of cycles per iteration? Ignore startup and exit code: HINT: Be very careful: the loads are not all equivalent: consider the dataflow graph!

```
filter:  stf  0(r2),F9 ; r1=+0,r2=+0
divf  F9,F8,F1 ; r1=+12,r2=+4
addf  F8,F7,F10 ; r1=+24,r2=+8
multf  F7,F6,F0 ; r1=+36,r2=+12
addf  F5,F4,F3
``
```
ldf  F6,56(r1) ; r1=+48,r2=+16
ldf  F4,52(r1)
ldf  F3,48(r1)
addi  r1,r1,#12
subi  r3,r3,#1
bne  r3,r0,filter
    addi  r2,r2,#4
```

Think of this as a pipeline with the parallelism marked from the end:

```
    ldf  multf
    ldf  addf  divf  stf
    ldf
```

We have 5 unique iterations in play at the same time. Our goal is to put as much space between producers and consumers as possible (making consumers be as many cycles later as possible). We start at the end (stf) and work backwards. The “r1=?, r2=?” annotations at the right show what you have to do to make up for the software pipelining.

Since there are no stalls, there are 12 cycles/iteration.

Problem 3f[3pts]: Assuming that we are allowed to perform loads off the end of the array at r1, show how we can add a small amount of startup code and a very simple type of exit code to make your software pipelined loop a complete loop for any number of iterations > 0. HINT: Assume that you “clean up” the first few iterations at the end.

Basic idea here is that there are no more than 4 bad stores at the beginning of the software pipeline. Save pointers and counts, then execute original code to compute those 4 iterations.

STARTUP CODE:

```
; Save relevant indices for later reexecuting first 4 iterations
filtersc:  add  r10,r1,r0 ;Save r1
        add  r11,r2,r0 ;Save r2
        add  r12,r3,r0 ;Save count
```

EXIT CODE:

```
; Reexecute up to first 4 iterations
filterex:  add  r1,r10,r0 ;restore r1
        add  r2,r11,r0 ;restore r2
        slti  r4,r12,#5 ; Less than 5 iterations?
        bne  r4, finloop ; yes. Use iteration count
            add  r3,r12,r0 ;restore r3
            addi  r3,r0,#4 ; No, peg fixup at 4
        finloop:  <Insert Code for filter from 3a or 3c here>
```
Problem 3g[3pts]: What is the lowest number of cycles per iteration that we might hope to get if we ran the optimized code from (3c) on our superscalar processor from Problem 2? Explain. Hint: use an instruction bandwidth argument. What sort of architectural features might prevent you from achieving this rate?

Looking at (3c), we see that there are 12 instructions:
1. 4 floating point ops (addf/multf/divf)
2. 4 integer ops (addi, subi, bnez)
3. 4 loads/store ops (ldf,stf)

Assuming no other problems, it seems like we could execute one loop in 4 cycles, taking one operation from each of the 3 categories on each cycle – assuming that we can get enough iterations loaded into the machine to remove dependencies….

Things that could get in the way of achieving this: poor branch prediction, cache misses, insufficient ROB resources, insufficient rename registers.

Problem 3h[5pts]: Suppose that we have a vector pipeline in addition to everything else. Assuming that the hardware vector size is at least as big as the number of loop iterations (i.e. value in r3 at entrance), produce a vectorized version of the filter loop. If you are not sure about the name of a particular instruction, use a name that seems reasonable but make sure to put comments in your code to make it obvious what you mean. Your code should take the same three arguments (r1, r2, and r3) and produce the same result.

filter: MVL r3 ; move loop count to vector length
         LVS V3,r1,12 ; Get values stored 0(r1),12(r1)…
         addi r1,r1,#4 ; Advance r1 by 4 (next set of values)
         LVS V4,r1,12 ; Get values originally at 4(r1),16(r1)…
         addi r1,r1,#4 ; Advance r1 by 4 (next set of values)
         LVS V6,r1,12 ; Get values originally at 8(r1),20(r1)…
         ADDV V5,V4,V3 ; Replaces “addf F5,F4,F3”
         MULTV V7,V6,F0 ; Replaces “multf F7,F6,F0”
         ADDV V8,V7,V5 ; Replaces “addf F8,F7,F5”
         DIVV V9,V8,F1 ; Replaces “divf F9,F8,F1”
         SV V9,r2 ; Store result back to memory.

Problem 3i[2pts]: Describe generally (in a couple of sentences) what you would have to do to your code in 3h if the size of a hardware vector is < number of iterations of the loop:

We would have to “Strip mine” the code. We would break it up into chunks, each of which was of the length of the hardware vector. Simple way to do this is to:
1. Compute remainder when iteration count (r3) is divided by hardware vector length
2. Compute one iteration of the strip mine with this remainder count
3. Compute the rest of the iterations at the full vector length.
1. Problem 4: Communication and Networks [20pts]

Problem 4a[2pts]: Define wormhole routing for a multiprocessor network. Why is it desirable?

Wormhole routing allows messages to stretch across the network, from source to destination (like a “worm”). Wormhole routing has a definite advantage over store and forward routing in that it exhibits lower total latency from source to destination.

Problem 4b[2pts]: The mesh network at the right can experience deadlock when messages are wormhole routed. Explain how this can happen.

Any set of messages that form a loop can be deadlocked. For an example, consider the set of messages shown at the right. Each one is blocked from making its desired turn because resources are held by another message.

Problem 4c[3pts]: Give a routing algorithm for an arbitrary k-ary n-cube that will not experience deadlock network (without using virtual channels!). Prove that it is deadlock free and given any conditions that must be true for the network to avoid deadlocking.

We talked about at least two algorithms that fit into this category: Dimension-order routing (or “e-cubed” routing) and one of the other algorithms that selectively removes turns from the network (such as “west-first”).

Here, we will talk about dimension-order routing. This algorithm requires all messages to route their dimensions one at a time, without repeating a dimension or backtracking along a given dimension: in the X direction, then the Y direction, ... (then Z, W, etc). To prove that this algorithm is deadlock free, simply assume that there is a deadlock cycle. Let “Q” be the lowest dimension (i.e. X or Y, etc) in which there is movement. Then, any cycle must route from Q to another dimension and then eventually from another dimension back to Q. This later routing turn is forbidden by our algorithm, so a cycle is impossible.

Important condition: once messages have arrived at their destinations, it must be the case that they will be removed from the network – otherwise, all bets are off with respect to deadlock freedom.

Problem 4d[3pts]: Higher dimensional networks (e.g. hypercubes) can route messages with fewer hops than lower-dimensional networks. None-the-less, the exploration paper that we read on k-ary n-cubes (Bill Dally) showed that high-dimensional networks were not necessarily the lowest-latency networks. Explain what assumptions lead to this conclusion and reflect when such assumptions are valid.

The important assumption here is that the network must be mapped into two or three dimensions and that the total area consumed by channels across a bisection of the network is limited by physical constraints. A secondary assumption is that the routers are very efficient so that the per-hop cost is not excessive but rather of a similar order to the cost of traversing a wire. As a result, higher-dimensional networks have lower-bandwidth channels (since they have more channels for a fixed budget of cross-section bandwidth). These assumptions are valid whenever bisection bandwidth is actually limited by wiring density, i.e. many systems with a lot of nodes and dense wiring, and when using modern low-latency VLSI routers.
Problem 4e[2pts]: The Linder and Harden paper gave a technique for using $O(2^{n-1})$ virtual channels to allow arbitrary adaptive routing while avoiding deadlock. Can you describe a technique that is less resource intensive (in terms of required number of virtual channels) that still allows arbitrary adaptation and deadlock freedom?

The trick is to divide our virtual channels into two groups, an “adaptive” group and a “deadlock-free” group. Our routing algorithm uses the channels in the first group in anyway desired to route around hot-spots or congestion. Then, if a message gets “stuck” for significantly long, it transitions to the second set of channels. The important point is that once a message transitions to using the deadlock-free channels, then it must route all the way to the destination, without going back to the adaptive channels. The reason that this algorithm is deadlock-free is that messages can never get stuck permanently in the adaptive channels (since we always have the deadlock-free channels as an “out”) and cannot get stuck in the deadlock-free channels since they form a deadlock-free network. One simple instance of this idea would be to put only one virtual channel/physical channel in the deadlock-free group and use dimension-order routing during the second phase.

Problem 4f[3pts]: What was the FO4 metric used in the “Future of Wires” paper? Why is it a useful metric for circuit complexity?

FO4 stands for “Fanout-Of-4” and represents the delay experienced by a single-sized inverter driving an inverter that is 4-times its size (see pictures below). For a given technology, we use the FO4 delay as a normalizing factor for the delays of more complex circuits. It is a useful metric for circuit complexity because the FO4 delay of a given circuit is roughly constant across a wide variety of technologies and thus forms a nice measure of logic complexity (higher $\Rightarrow$ more complex).

Problem 4g[2pts]: The “Future of Wires” paper ultimately concluded that multicore was an important future architectural innovation (as opposed to more complex uniprocessor cores). Can you give two arguments that lead to that conclusion?

Two arguments given in the paper were: (1) with Moore’s-law growth in number of transistors along with the shrinking of transistors, the portion of a chip that is reachable “in one clock cycle” (defined as 8 FO4 delays) has been decreasing rapidly. Consequently, designs that are tolerant of multi-clock-cycle delays in getting to portions of the chip are desirable; multicore fits this constraint by connecting many single-clock domains with a general network. (2) CAD tools are increasingly unable to deal with huge numbers of transistors in designs. By breaking a large chip into a system with many repeated identical components (i.e. multicore), we can present smaller “digestable” chunks to the CAD tool.

Problem 4h[3pts]: What was the basic premise of the Active Messages paper? Why did the paper show such large improvements in the latency (or overhead) to send a message relative to existing message interfaces on the same machine?

The basic premise of the Active Messages paper was that the communications interface (API) used by programmers should better match that of the hardware. The Active Message API had two primary components: (1) that messages should include an address of a handler to run at the destination and (2) that the reception handlers should be short-lived and never block (i.e. it is the job of the handler to integrate incoming communication with an existing computation rather than to start a new computation). It is likely that the paper showed large improvements in the latency (or overhead) of message send because the Active Messages API eliminated a lot of buffer management.
[ Random spare page for scratch ]
[ Random spare page for scratch ]