CS252 – Graduate Computer Architecture

University of California
Dept. of Electrical Engineering and Computer Sciences
David E. Culler **Spring 2005**

Last name: ____Solutions First name_______________________

I certify that my answers to this exam are my own work. If I am taking this exam early, I certify that I shall not discuss the exam questions, the exam answers, or the content of the exam with anyone until after the scheduled exam time. If I am taking this exam in scheduled time, I certify that I have not discussed the exam with anyone who took it early.

Signature: ____________________________________

The exam has five problems, which range from concrete to conceptual. Please read the problems and take your time in formulating an answer. The answers are not long. Please show your work and your line of reasoning. There are a total of eight pages, including space for your work. Feel free to write on the backs of sheets, if you need more space.

The exam is open book. You may use your textbook, lecture slides, or a calculator.

Problem	Points	Score
1	10	
2	15	
3	15	
4	25	
5	35	

Problem 1: State and define the hazards presented by instruction level parallelism. For each one, indicate how it can be resolved.

1. *Data Hazards*
    - *RAW (Data dependence) cannot use a value before it is computed. Resolve by forwarding or stalling*
    - *WAW (Output dependence) cannot write a value if a logically preceding instruction might overwrite it Resolve by pipeline design (in-order op-fetch + in-order WB), stalling on potential write to pending register, or renaming*
    - *WAR (anti-dependence) cannot write a value before logically preceding instruction reading the previous value have done so. Resolve by pipeline design (in-order issue with in-order operand fetch), stalling or renaming.*

2. Structural Hazards

Attempt to use the same hardware resource for two different purposes at once. Resolve by adding hardware resources (as design time) or stalling

3. Control Hazards

Cannot determine the control flow until the condition of the branch is resolved. Resolve by stalling. Mitigated by predicting and discarding miss-predicts.

Problem 2. Your current version of ZippyCAD runs through a benchmark design in 43 minutes on your ZIPS10 computer. ZIPS has a new model that they are offering to sell to you. ZIPS30 is a scalar machine like ZIPS10, but 3 times faster. Or you can get the ZIPS1010 vector upgrade that performs vectorized code at 10 times the performance of ZIPS10. You know that ZippyCAD spends a lot of time in its numerical library, so you are intrigued. *How much of ZippyCAD would need to vectorize for the ZIPS1010 to beat the ZIPS30?*

$$SUvector = \frac{Tscalar}{Tvector} = \frac{1}{(1-f)+f/x} = \frac{1}{1-f+f/10} \geq 3$$

$$f \geq 20/27 \approx 74\%$$

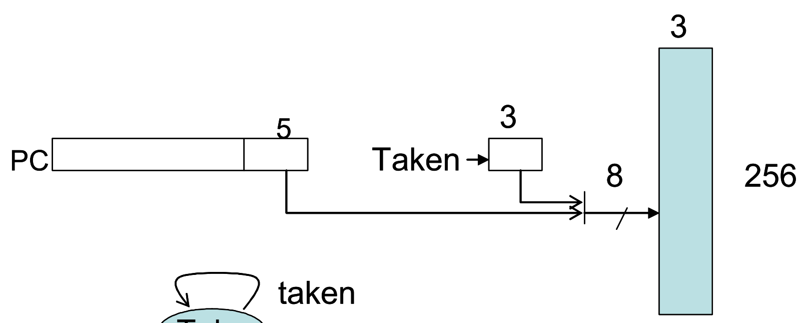
Problem 3: Diagram a correlating branch predictor that uses 3 bits of global information, 5 bits of local information, 3-bit saturating counters. Include the dimensions of all the machine data structures. Diagram the state machine. Give brief pseudo code for how this operates.

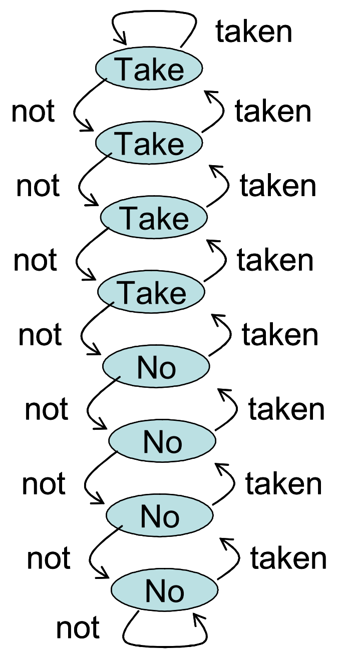
Index a 256 x 3-bit RAM using the low 5 bits of the PC concatenated with 3 bits from the branch history shift register. The shift register has as input the branch direction. The book draws this as a two dimensional table. Same thing.

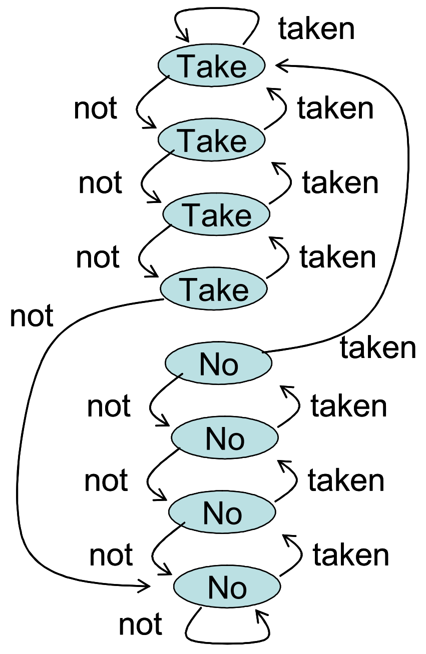
On decode, combine the 8 bits as above, index the prediction table, read the value and predict based on the most significant bit.

Retain prediction table index into the execute stage. At that point, update the table entry according to the saturating counter state-transition-diagram. Write the result to the table. Shift the taken/not-taken bit into the shift history.

There are two reasonable state machine. Both have state encodings going from 000 at the bottom to 111 at the top. The one on the right provides some hysteresis. You don't get stuck toggling between 011 and 100. Even a branch that alternates between taken and not-taken will get predicted right about half the time.







Problem 4. Having fallen in love with the IBM360 early in the course, you've analyzed a dynamic instruction trace of the EGGSELL spreadsheet running your Valentine's Day order list on your MIPS machine in order to consider resurrecting some aspects of the 360. You find that the benchmark executed 1,000,000 instructions in 2,200,000 cycles and that instruction frequencies were:

Arithmetic	50%
Branch	20%
Load	20%
Store	10%

Under more careful analysis you find that 25% of the loads are used to add a value to a single register. Putting these two important discoveries together, you have decided to add a LADD instruction to your old flame MIPS machine. The instruction

```
LADD rt, rs, offset
```

has the RTL semantics

```
REG[rt] := REG[rt] + MEM[REG[rs] + offset]
```

Having also become an expert in micro-architecture, you believe that you can support this instruction without increasing the clock cycle time of your MIPS.

Assuming you can pull this off, how low must be the CPI of the new machine for this enhancement to improve performance of the application?

Give a couple of reasons why you expect the CPI to increase with this enhancement.

Give a brief sketch of how you might modify the microarchitecture of the basic MIPS pipelined datapath to support this instruction without severely impacting the cycle time.

What changes are required to make hazard resolution work properly? What are likely to be the aspects that make it difficult to maintain the cycle time?

Initial CPI = 2.2 LADD eliminates 5% of instructions.
At same CT, new CPI must be < 2.2/.95 ~ 2.3

In a general sense you might imagine that doing the same work in fewer instructions might raise the CPI, but that doesn't answer the question. They might pipeline just as well as the old instructions, maintaining the same CPI and just improving execution time. The reason that almost works is that instructions dependent on the LADD will stall because you cannot forward the value till later. However, such an instruction would otherwise be dependent on the ADD following the load. More accurate is that the load and the add can no longer be separated by independent instructions. The new stall is when the data operand of the LADD itself (not the address operand) is dependent on a previous instruction. Now the LADD will stall even though the LOAD portion could go forward. Also, data misses will be amortized over few instructions.

There are two good design solutions. The "Stanford MIPS style" option is to add a sixth stage between MEM and writeback to do that ADD. The data operand, which is carried to MEM for the STORE, must be carried to the ADD stage.
Note that adding a stage does not increase the CPI if no LADDs were used. It simply means that more values will be forwarded, since WB is further delayed. The change required is another level of forwarding. This requires an additional set of data wires along the length of the datapath and widens the forwarding mux. The logic for determining the mux selects is hardly any worse. LADDs cannot forward except from the ADD stage. The increase in cycle time is due to widening each bit slice of the datapath and the additional steering logic on the mux.

The "Berkeley/Sun RISC/Sparc style" would be to split the LADD at the decode stage into three micro ops. The first is essentially a load, the second is a NOP, and the third is the ADD. Observe that the third brings the data operand into the EX stage as the loaded value is produced from the MEM stage. No new wiring is required. We simply feed the loaded value back. Of course, this is no faster than issuing two instruction – and it prevents the compiler from filling the load delay slot, but it does reduce code size. It has no impact on the cycle time.

Problem 5: Now the wide open design problem. The haunting elegance of the stack architecture has stayed in the back of your mind ever since the debate. Now that you have seen superscalar execution, register renaming, forwarding, Tomasulo and all that, you wonder "why can't I apply these techniques to stack machines to find instruction level parallelism there too?" You grab your favorite loop as a test case

```
for (i = 0; i < n; i++) A[i] = A[i] + alpha;
```

which, of course, compiles as

```
for (ptr = A; ptr < &A[n]; ptr++) *ptr += alpha;
```

On entry to this loop there are three values at the Top of Stack:

TOS-8:	Alpha
TOS-4:	ArrayEnd
TOS:	Ptr

The stack code for the loop is as follows.

```
vscal:
     push @0    ; push a copy of Ptr
     load       ; Load the array value (replacing Ptr)
     push @12   ; push a copy of the scale value, alpha
     fadd       ; alpha + *ptr
     push @4    ; push a copy of Ptr
     store      ; *ptr := alpha + *ptr
     pushIm 4   ; pointer increment value
     add        ; ptr++ (update on the stack)
     push @4    ; push ArrayEnd
     push @4    ; push ptr
     sub
     blt vscal
```

The `push @X` instruction pushes the value at offset X from the top of stack. `pushIM X` pushes immediate value X. All other operations pop their operands from the top of stack, remove them, and push a result, if one is generated.

Your starting point for your design is based roughly on the MIPS R10000. It has several function units with a reservation station per function unit and forwarding of results to the function units, as indicated in the diagram below. A large collection of physical registers are provided. They are not in the instruction set architecture. The architected state is the stack and PC. You are to describe how to do the renaming such that you could overlap the execution of multiple iterations of this loop. You will need to invent the mechanism to perform the necessary renaming. You may assume there are enough physical registers to perform one or more iterations of the loop, but not an arbitrary number of overlapping iterations. You may assume there is a mechanism ALLOC that will allocate a free register and provide that register number, if one is available. If none are free, it will indicate a failure. You may, similarly, assume there is an operation FREE(Reg) which frees the specified register.

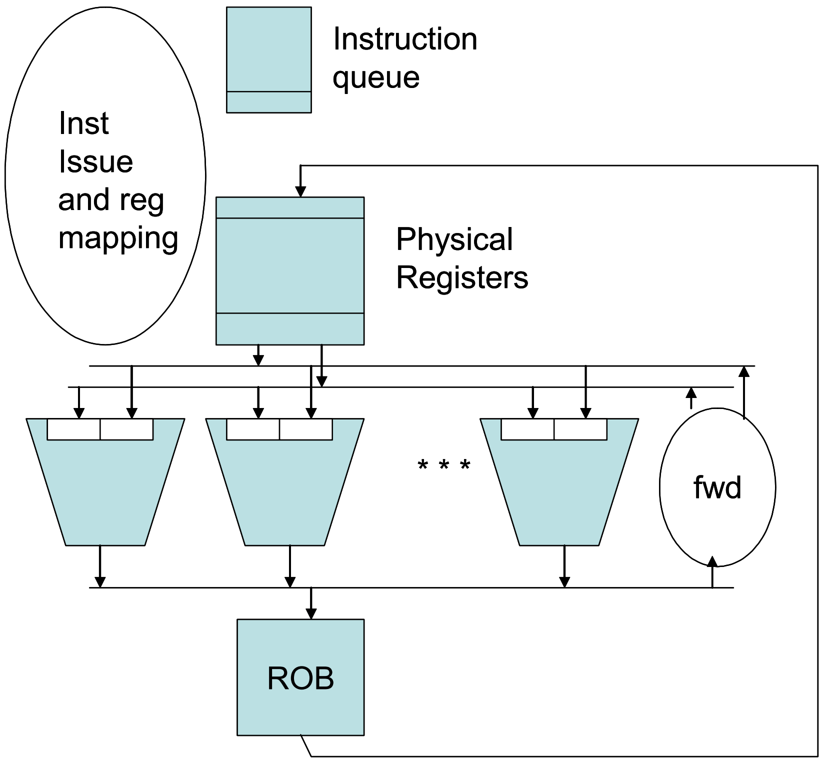
How much stack space is required to execute this loop? Because of this, you don't need to worry about stack overflow/underflow. You do need to deal with limits on the available physical registers and function units.

Describe the instruction issue and operand fetch process for different kinds of instructions that appear in the example.

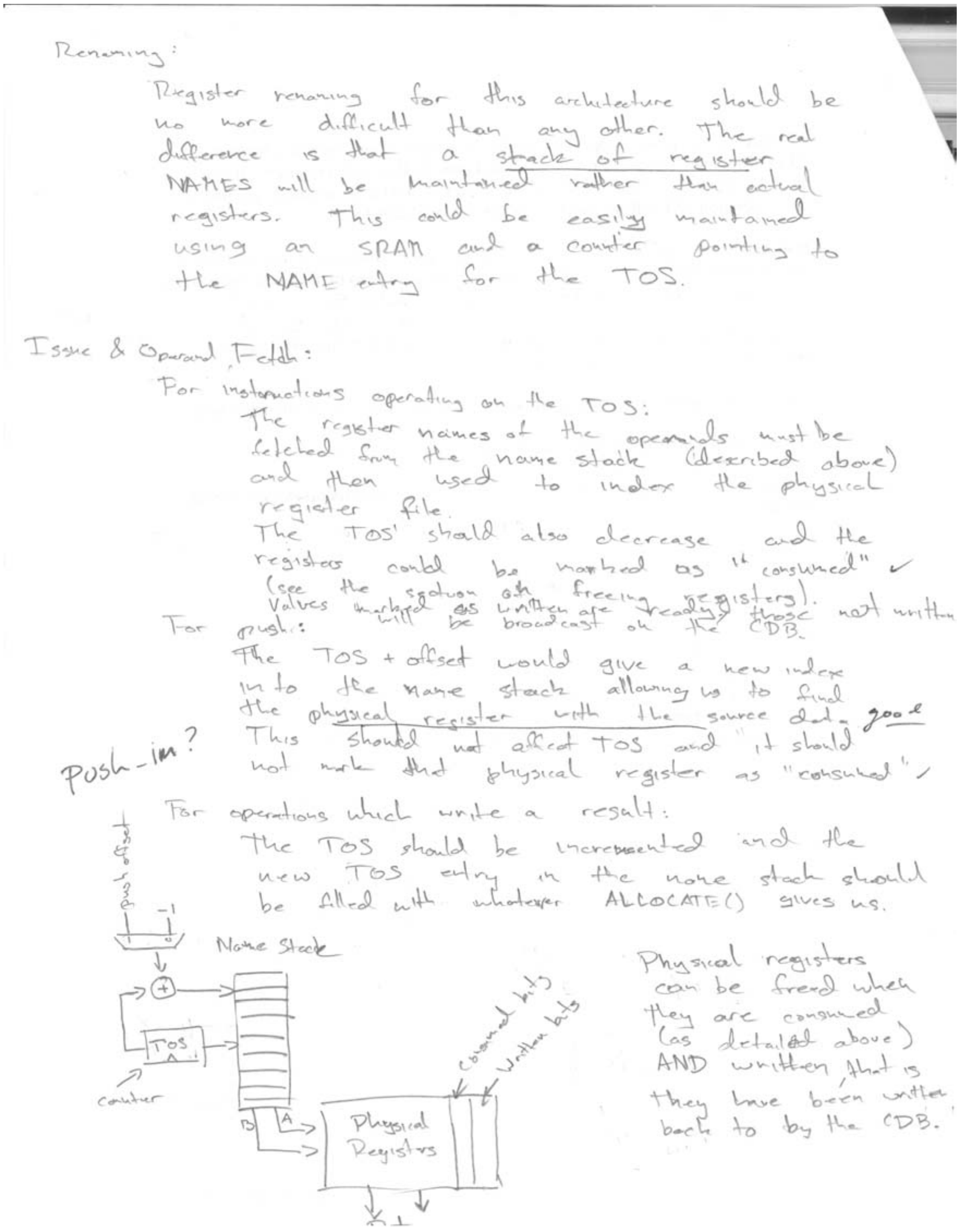
Under what conditions will the machine wait – holding the issue of an instruction?

Explain when a physical register can be freed.

Explain what limits the number of iterations of the loop that can potentially execute concurrently.



The key idea is to introducing a "renaming stack" of physical register names. A physical register may be freed when its name is popped off the renaming stack AND the instruction that produces it enters the ROB. A single state bit will do. The latter of the two events will free place the physical register back on the free list.
The key optimization is that PUSH just shares the previously allocated physical register. PUSH, POP, and PUSH_IM don't need to enter the execution section at all.
Here's a nice student solution.



Renaming:

Register renaming for this architecture should be no more difficult than any other. The real difference is that a stack of register NAMES will be maintained rather than actual registers. This could be easily maintained using an SRAM and a counter pointing to the NAME entry for the TOS.

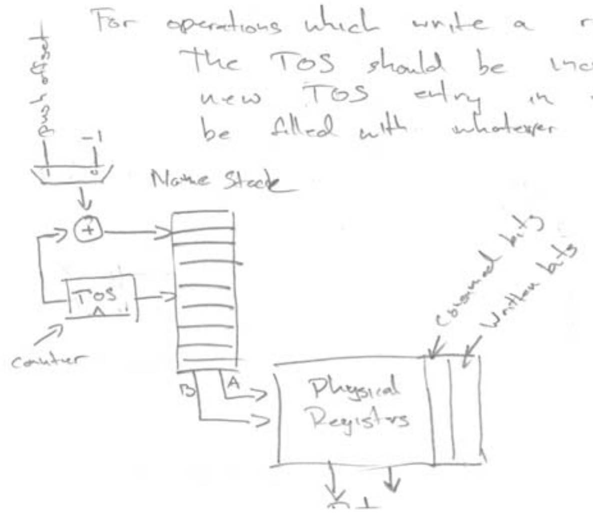
Issue & Operand Fetch:

For instructions operating on the TOS:
The register names of the operands must be fetched from the name stack (described above) and then used to index the physical register file.
The TOS' should also decrease and the registers could be marked as "consumed" ✓ (see the section on freeing registers).
Values marked as written are ready, those not written will be broadcast on the CDB.

For push:
The TOS + offset would give a new index in to the name stack allowing us to find the physical register with the source data. good
This should not affect TOS and it should not mark that physical register as "consumed". ✓

Push-im?

For operations which write a result:
The TOS should be incremented and the new TOS entry in the name stack should be filled with whatever ALLOCATE() gives us.



Physical registers can be freed when they are consumed (as detailed above) AND written, that is they have been written back to by the CDB.

This machine will stall when it runs out of physical registers, the reservation station for the current instruction is busy or the ROB is full, essentially when the instruction has nowhere to go. In the case of this code it is likely the ROB will fill up first as the longer fadd instructions delay things, however given a deep enough ROB and a pipelined (or fast) memory and fadd this code could execute with a CPI of 1.

In this case the number of parallel iterations may be limited by the number of FU's/ reservation stations, especially since I find it unlikely that there are many fadd or memory units.

However the primary limitation in this design is issue rate. Stack architectures tend to have high instruction counts compared to register/register machines like the R10000. This will result in the situation where each microinstruction can actually perform multiple instructions. For example

```
push @4
push @4
sub
```

could easily be handled in one microinstruction, leading me to believe that the issue rate will be the primary bottleneck, and that some kind of superscalar or the ability to translate multiple instructions into one microinstruction will be usefull.

Note: I have assumed branch prediction is in use. Perhaps simply "static taken".