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.

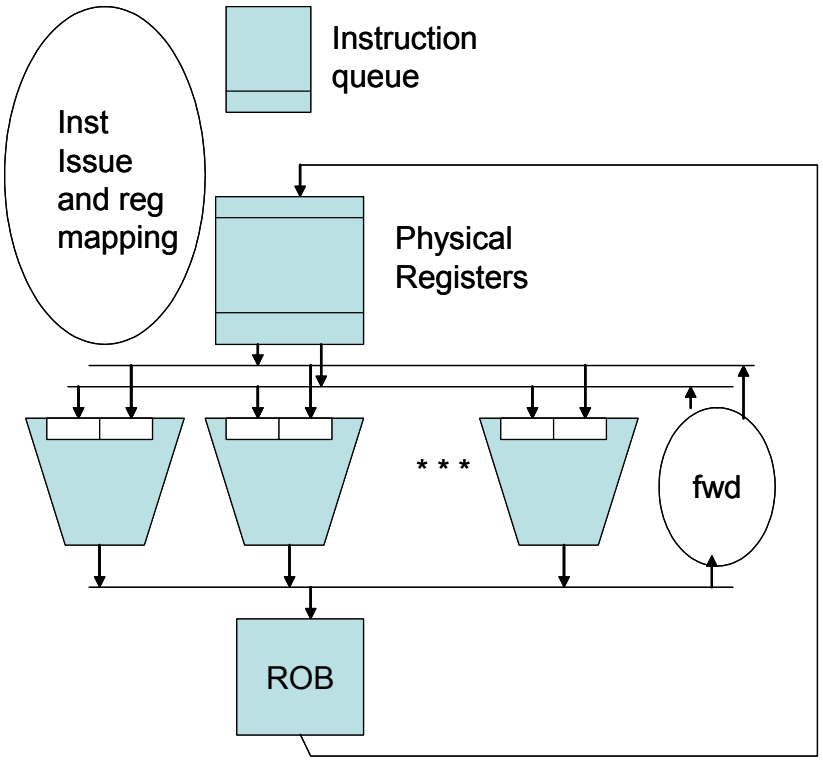
How much stack space is required to execute this loop? Because of this, you don't need to worry about stack overflow/underflow. You do need to deal with limits on the available physical registers and function units.

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.

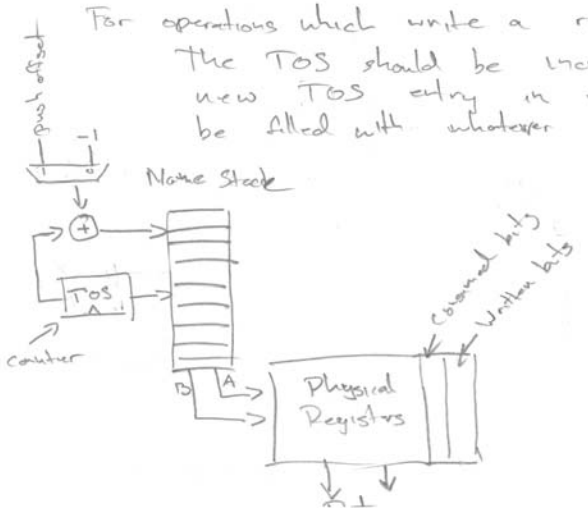
Issue & Operand Fetch:

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 and will be broadcast on the CDB. those not written

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 dat. 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 because the longer fadd instructions delay things, however given a deep enough ROB and a pipelined (or fast) memory unit 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 FUs' / 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 useful.

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