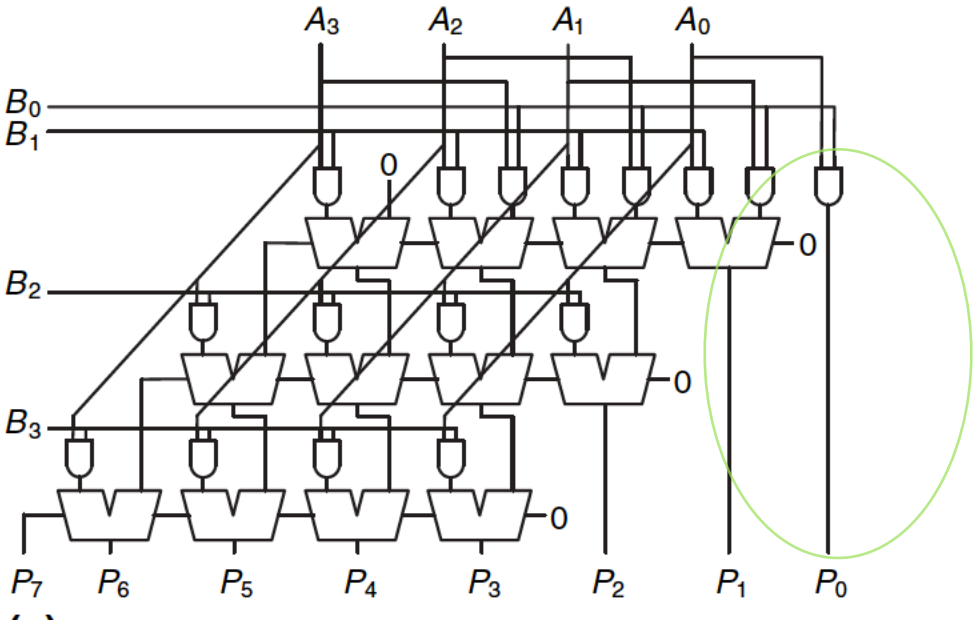
James Johnston
Final CS6133 spring 2017

Part 1

Consider the 4 x 4 multiplier described in Digital Design and Computer Architecture, chapter 5 section 5.2.6 and answer the following questions:

Redesign the multiplier such that it is synchronize to a 1Ghz clock source.

Observing the the 4 x 4 multiplier in chapter 5 section 5.2.6 we see that the multipliers is clearly not synchronized. The logic gates are executed in a way that the logic gate on the far right of the diagram, just an AND gate, will excite far sooner and faster than that of the logic gates on the far left, which include 3 ALUs and an AND gate.

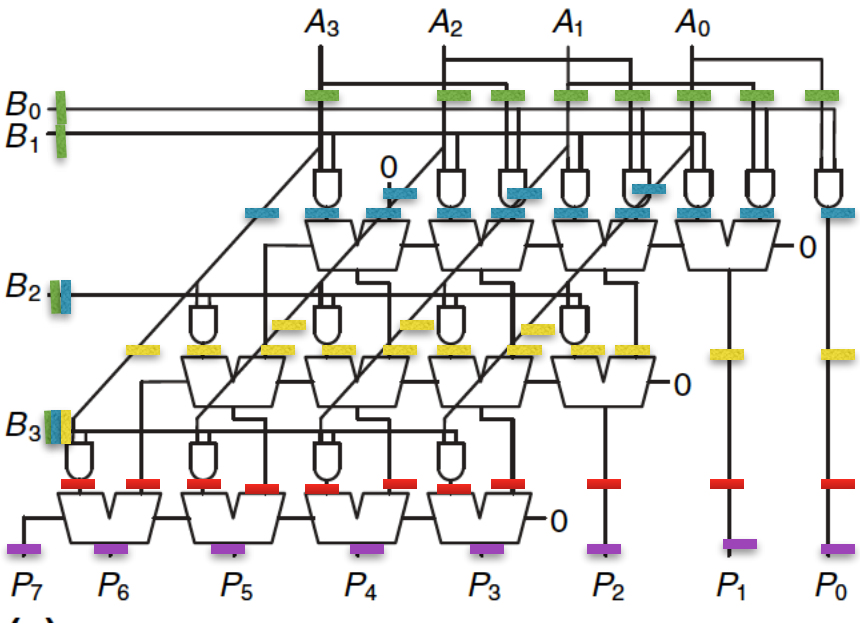


The result of P0 will execute sooner than P6 and therefore be out of sync. To synchronize the multiplier we need to redesign it with the addition of registers to ensure synchronization.

To emulate the propagation delay of the logic gates and ALU I will need to add several register to ensure that at each level the clocks are in sync.

I will insert color coded registers to help with organization. The first register is green, i added this in the beginning to ensure that all of the values get there at the same time. The second is blue, the third yellow, the fourth is red and the last is purple.

Below is my redesigned multiplier such that it is synchronize to a 1Ghz clock source.

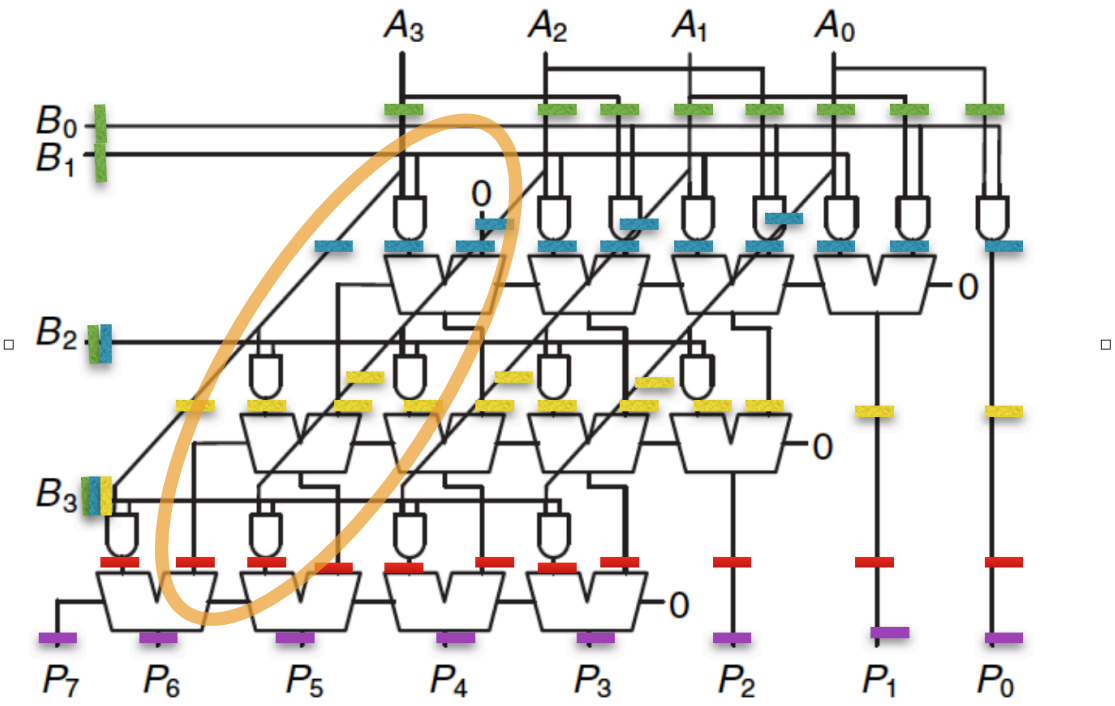


Prove, by calculating the worse case combinatorial propagation delays, that your synchronize multiplier works reliably over 0 C to 100 C temperature range.

Assume the following propagation delays:

L.E. / Temp.	Gate	ALU
100 °C	310ps ±10ps	600ps ±10ps
50 °C	260ps ±10ps	500ps ±10ps
0 °C	210ps ±10ps	400ps ±10ps

The worse case combinatorial propagation delay is at 100 degree Celsius and the path with the most logic gates is indicated below.



The path is AND -> ALU -> ALU -> ALU

At 0 degrees Celsius:
propagation delay
1 gate x 220ps + 3 ALU x 410ps = 1450ps

1 Ghz = 10^9 cycles per 1 second
1 second = 10^{12} picoseconds
1 Ghz = 10^9 cycles per 10^{12} picoseconds
1 Ghz = .001 cycles per 1 picosecond

thus without the introduction of registers the CPU at 1GHz completes in1.45 cycles.

then with the registers and their hold time will take one clock cycle

so we have:

1 gate x 220ps + 3 ALU x 410ps + 5 registers = 5000ps
and the 5 clock cycle will be synced at 1GHz

At 100 degrees Celsius:
propagation delay
1gate x 320ps + 3 ALU x 610ps = 2150ps

thus without the introduction of registers the CPU at 1GHz completes in 2.15 cycles.

then with the registers and their hold time will be one clock cycle
we have

1 gate x 220ps + 3 ALU x 410ps + 5 registers = 5000ps
so the clock will be synced

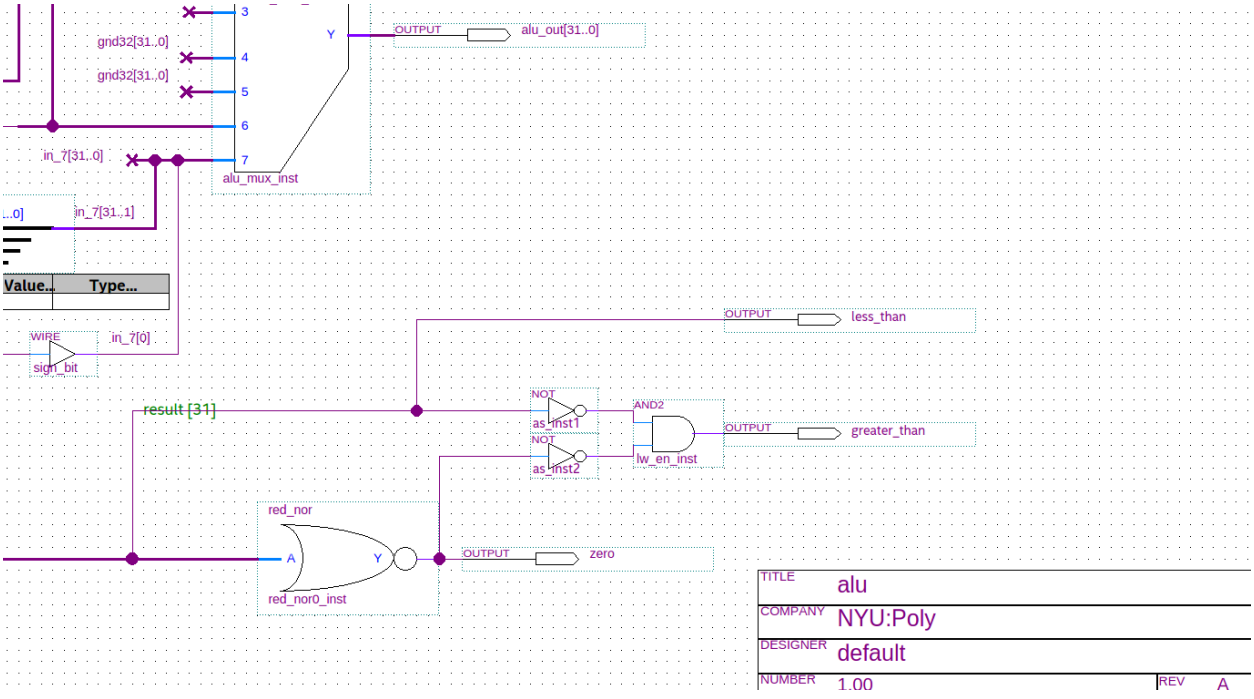
Part 2
Considering the MIPS SS v2 (Simple CPU) and its ISA for this part.
How does the simple CPU handle illegal instructions?
Describe, with details, what Simple CPU does when an undefined opcode and/or function code is presented.

The simple CPU handles illegal instructions by making a default operation instead of executing the undefined opcode and/or function code presented. Even though the opcode isn't recognized a value is still sent to sb goes through and is sent to wa-register. The alusrc is 1 so the value of sa_out and the last 16 bits will be sent to the ALU. The signal from CU will be 00 and the alu_ctrl last 6 bit of the instruction alu_op is 010 and thus will perform a default addi. sa_out will not got to memory and instead goes through the right most MUX and writes to wa_reg in the register file.

Redesign Simple CPU's ALU to support the following instructions:

ble rx, ry, offset: pc += (rx < ry) ? (offset + 4) : 4
bge rx, ry, offset: pc += (rx > ry) ? (offset + 4) : 4

Note: You are only required to show modification to Simple CPU's ALU for question 2.



Above is my modification of the Simple CPU's ALU. As you can see I added a less_than and a greater_than output. I simply take the most significant bit from the 32 bit bus connected to the branch equal and make it my less_than. If it is a 1 then my output signal is 1 because the number is negative and thus is less than. For the greater_than

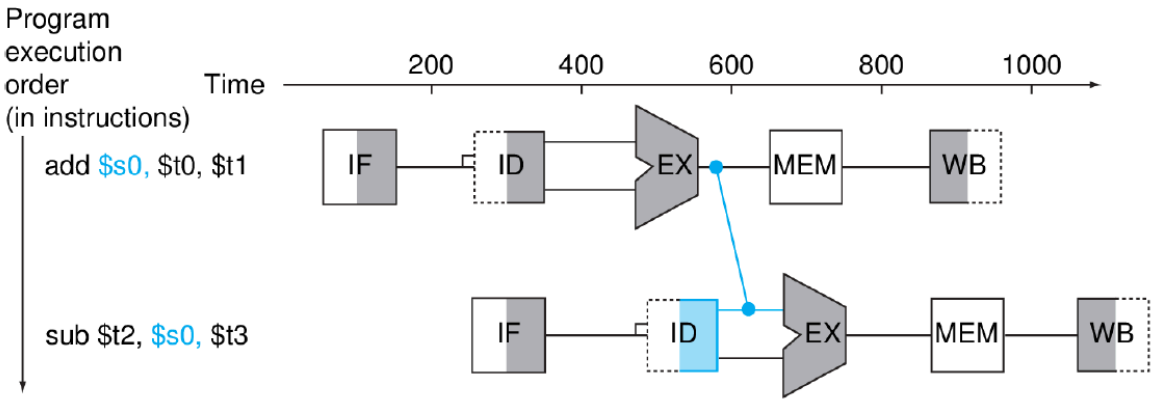
branch, I take the significant bit and negate it. It is then pumped into an AND gate with the result of the branch equals. If the significant bit is not 1 the result of the subtraction was positive and there for the number is greater than. The results from the AND gate is then piped to the greater_than output.

Part 3
Considering the MIPS SS v2 and its ISA for this part.

Assume the Simple CPU is pipelined, as described in lecture, what type of hazards would you encounter in Simple CPU. Remember you only have to consider instructions supported by Simple CPU.

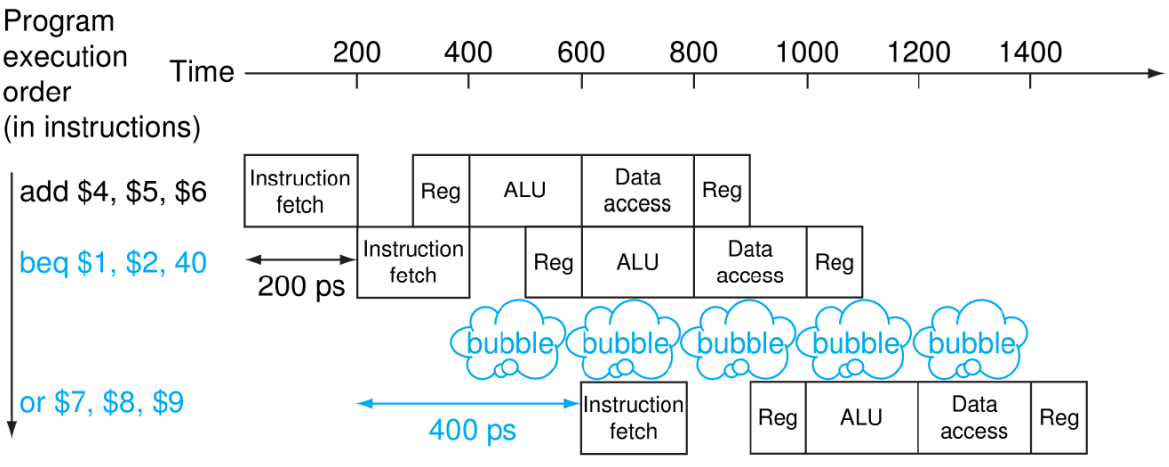
The hazards that the Simple CPU can encounter are data, and control hazards, but no structural hazards. The structural hazards arise due to the due to the resource constraints in our Simple CPU that are not fully pipelined. There is two memories, one memory for instruction and the other memory data. Thus we don't have to worry about having to share one memory. Secondly there is a limited number of instructions and there is no doubling of the components that might occur, say a multiply and an add instruction conflict.

Data hazards can be encountered. Here data dependencies occur in the pipelined Simple CPU. As described in the below diagram we can have back to back instructions where one register or data is dependent on the first.



In the execution of the above assembly code there is a change in the data in $s0 and the following instruction requires the $s0 result to calculate.

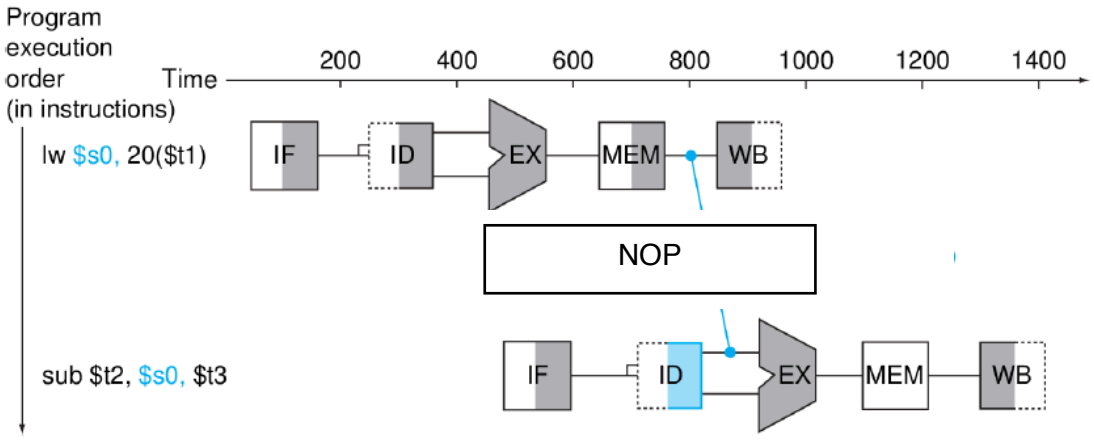
The next hazard is the control hazard. Control hazards occur from the fact that a branch or jump depends on the result of anther instruction. In our Simple CPU our branch



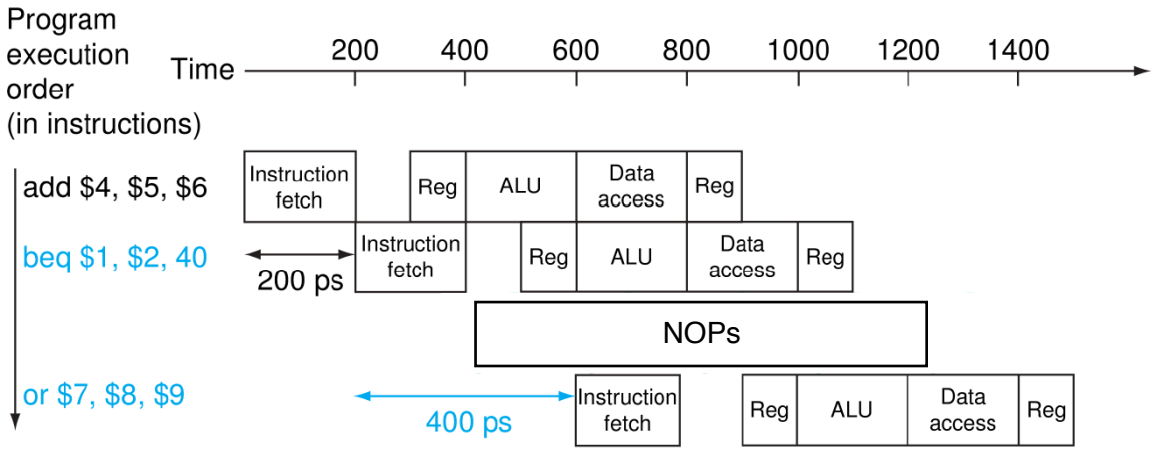
function takes 3 clocks to execute and needs to complete a conditional before executing the branch as you can see in the diagram.

Could a “NOP” inserter mitigate all hazards you listed above? Support your answer with details.

A “NOP” inserter could migrate the hazards that arise in our pipelined Simple CPU. For our data hazard we simply need to insert a NOP between the instructions of that have dependencies. This allows the first instruction to fully complete and allow the new saved data to become available to the next operation that requires that particular data. An implementation of this insertion can be seen below.



For control hazard we can also insert NOPs after the branch instruction. This will give the branch time to wait for the results of the instruction it needs. The below diagram gives the branch time enough to get the results of the instruction so that we can ensure that it acts correctly.



Part 4

Consider the following code:

```
1l char *a, *b, *c;
2l <allocate 2^29 bytes of memory to a,b>
3l <allocate 2^12 bytes of memory to c>
4l for(i=0; i<=2^29; i++)
5l a[i] = b[i] + c[(i mod 2^12)];
```

Assume the following cache requirements :
DDR memory's smallest transfer size is 512 bits or 64 bytes.

L2 cache is 16MB and is composed of 4096 4KB cache lines. A cache line is the smallest unit that could be transferred between DDR and L2 cache.

The memory subsystem is aligned to 4KB memory boundaries. Assume that memory allocation confirms to 4KB alignment.

Answer the following questions:

Find an optimal cache architecture (direct, n-way associative or fully associative) that would yield the best performance with lowest implementation complexity.

The optimal cache architecture that would yield the best performance with the lowest implementation complexity is the n-way associative method, specifically 3-way associative method. This is because we have three values to store at the same time. Thus we have at least 3 parts with c having the least memory than the other two.

What type of replacement algorithm would yield the lowest miss rate?

LRU (least recently used) is a type of replacement algorithm would yield the lowest miss rate. It chooses the cache line that is not used for the longest time. The algorithm will never affect the c portion because it is small and used all the time. Therefore it will concentrate on replacing a and b. This will increase the hits and lower the probability of misses.

Part 5

Consider Parallel BUS and Serial links described in I/O lecture for this part. State which I/O architecture (serial or parallel) is optimal for each of the following scenarios:

Lots of random 32-bit word transfers between CPU and I/O peripherals

Parallel is optimal in this scenario. Aside from the words being random, and the arrival time of the data parallel is the best option. This is because the a 32 bit word is relatively small and we get the complete data in one clock cycle if we implement parallel. It would take far longer if we have a serial bus to get the complete data.

Large amount contiguous burst transfers occur between I/O peripherals and a few burst transfer between I/O peripherals and CPU

In a situation where there are large amounts of continue burst I/O peripherals and a few burst transfers between I/O peripherals and CPU the best option is serial links. If parallel bus were used it would be much more expensive in terms of hardware. It will lead to a greater cost in manufacturing. Using parallel with burst transfer data it is harder to ensure that the data will arrive at the same time.

Both burst transfers and few 32-bit word transfers between CPU and I/O peripherals. You should support your answers with sufficient details.

The best option here is serial link. This is because we have burst of data that we want to transfer. We can build connections with less resources and cost than parallel. It will also ensure that the data arrives at the same time. If we use parallel we have to build several connection that cost time and resources. Also, with a parallel bus transfer we can't be sure that the data will arrive at the same time. Parallel will also have a great deal more interference than serial.