## Technical Background

The competition is to build a "HACK computer" compatible with its ISA, with the best performance when loaded to an [#FPGA](https://www.linkedin.com/feed/hashtag/?keywords=fpga&highlightedUpdateUrns=urn%3Ali%3Aactivity%3A6927335125860954112).  
Measuring the performance is done by the time it takes the "program", which is pre-determined, to finish. The program has visual feedback that can be seen on a VGA connection from the FPGA.  
The Hack Computer is a theoretical computer design created by Noam Nisan and Shimon Schocken and described in their book, "The Elements of Computing Systems: Building a Modern Computer from First Principles."  
- <https://lnkd.in/ekFbnZ_R> "Wikipedia - [#Hack\_computer](https://www.linkedin.com/feed/hashtag/?keywords=hack_computer&highlightedUpdateUrns=urn%3Ali%3Aactivity%3A6927335125860954112)"  
- <https://lnkd.in/efkzWjtw> "[#nand2tetris](https://www.linkedin.com/feed/hashtag/?keywords=nand2tetris&highlightedUpdateUrns=urn%3Ali%3Aactivity%3A6927335125860954112)"

### The starting point of the competition

We got a SystemVerilog code of the most basic CPU that supports the HACK ISA - very similar to the example from the book by Noam Nisan and Shimon Schocken.  
Additionally, we got a ROM with the program and a RAM used by the program for its registers, stack pointer, and more.  
The memory that the VGA will use to display the result of the run can access that same RAM.  
Other blocks: VGA graphic controller & performance counters  
You can see the starting point source code here:  
- <https://lnkd.in/eY5jfdMs>  
  
I will note that due to the constraint of synchronous memory of the FPGA, this basic architecture allows executing a new Instruction fetch only once every two cycles. This specific implementation can be thought of as a multi-cycle with an IPC of ~0.5.  
The code's runtime with the initial solution we got is ~28 seconds.

### Performance enhancement

The first and most obvious step to improve the processor's performance is running the CPU clock at a higher frequency.  
By adding a simple PLL with a clock ratio of ~2, you can cut the runtime in half!  
  
After several attempts on the FPGA to change the PLL frequency ratio, I achieved a runtime of ~12 seconds.  
I will note that when trying to raise the clock frequency too much, the logic no longer converges, the visual feedback in VGA is wrong, and in some cases, the run never ends.

## Pipeline architecture

A common way to improve the CPU IPC is to switch to a pipeline architecture. Traditional CPU pipeline stages:  
Fetch->Decode->Execute->Mem\_Access->Write\_back

### Known Pipeline Hazards

1) Data Hazard: READ After WRITE will cause HAZARD because the WRITE data is in the execute/write\_back stage, and the READ is in the decode stage.  
2) Load Hazard: Reading from memory (load operation), the DATA is not in the CPU pipe, so performing a simple forwarding unit is impossible. (bubble insertion is needed)  
3) Ctrl Hazard: Jump/branch commands are determined in the execute stage. The PC has speculatively advanced & sent a redundant instruction fetch, and this will cause incorrect instructions to go down the pipe.

### HACK ISA background & new issues

The Hack instruction can manipulate three 16-bit registers:  
D- Data Register.  
A- Address register.  
M- Selected data memory register.  
Hack instructions have a self-explanatory syntax.  
Example: D = M, M = D + 1, D = 0, A = M + D and so on.  
Thus, the HACK ISA can execute arithmetic commands on RAM Data (M=M+D). This type of command is not easily compatible with classical architecture.

### Solutions

\*Data Hazard\* - Forward solutions can be implemented for registers A and D using Hazard Detection logic for a particular register & forwarding the latest data to the ALU.  
\*Ctrl Hazards\* - When identifying a "branch-condition-met," update the PC to the new address and perform a "flush" to PIPE. There is a penalty of two cycles in each jump/branch.  
\*Load hazard\* - I decided to make a perceptual change in the architecture and treat RAM as a large register file.  
The RAM is accessed for reading in the decode stage, so the data is accessible in the execute stage.  
Note: I will need dual access memory, with one port for reading (decode) and one for writing (write\_back).  
  
Because we run our model on FPGA, we are limited to specific types of memory. Luckily, dual Access memory exists in the DE10-Light portfolio.  
  
To allow VGA to read the memory and display the result, we duplicated the RAM. Both RAMs will get the same write\_back data, but the VGA reads & Decode reads are on different memories.  
See the diagram in the attached image.

### To summarize

1) Implemented a pipeline to improve the IPC to ~1.  
2) Ctrl, Data, Load Hazard solved.  
3) Duplicated the RAM to allow duel-read & single write every cycle.  
  
📈 At this point, we almost doubled the IPC without compromising the clock frequency. Therefore, we improved the program's running time from ~12 seconds to ~7 seconds.

## Multiple-Issue (superscalar processor)

The next leap in performance will come from fetching multiple instructions per cycle and executing them parallel through the CPU pipe.  
Wiki - <https://lnkd.in/dTjSWawN>

### HACK ISA technical Background

In the HACK ISA, to determine the RAM address for the READ/WRITE operation, we use register A (the "Address register")  
Register A may get new value in two ways:  
1) "A\_type" instruction can be thought of as "load immediate"(li). The hack syntax for this li operation: ```@xxx``` Example:   
@5 //load A with 5.  
2) "C\_type" instruction - gets the output of the ALU. Example:   
A = D+M  
  
Other example:  
@5. // A = 5  
A = M + D // A = M[5] + D

### Simple duel issue example

By fetching multiple instructions from the ROM, we can decode multiple instructions in parallel and look for patterns we support to execute "fusion" into a single operation down the pipe.  
--- Examples: ----  
1) loading an immediate to register D:  
@5  
D = A // D = 5  
2) loading from memory in a specific location to register D:  
@5  
D = M // D = M[5]  
3) Storing to memory in a specific location the data of register D:  
@5  
M = D // M[5] = D

### Multi-Issue

Similarly, we can find patterns of multiple instructions to execute simultaneously.  
In practice, the only real constraint is that we can only WRITE (store) one "word" in a single execution cycle.   
Note: for multiple RAM READs, we can always Duplicate the number of RAM instances.  
--- Examples: ---  
//M[4] = 30000  
@30000  
D = A  
@4  
M = D  
-------  
//M[9] = 2003 - M[1] - M[4]  
@2003  
D = A  
@1  
D = D - M  
@4  
D = D - M  
@9  
M = D  
------  
Note that the number of executed instructions is known only after the decode stage. This means we need to add some complex logic to the "Program counter". Also, we had to implement speculative fetch 2x the number of instructions of the maximum Multi-Issue Decode, so we will have all the instructions we need next cycle.

### To summarize

1) Duplicated the ROM 20 times, supporting 20 instruction fetch every cycle. (This is a simple way - not the optimal way)  
2) Decode up to 10 instructions every cycle and execute them in parallel. Bounded by the memory WRITE bandwidth.  
3) Increment the PC in the correct amount.  
  
📈 At this point, The IPC has more than doubled, but we compromised the clock frequency due to the complex Decode Stage. Therefore, we improved the program's running time from ~7seconds to ~3 seconds.

## Dedicated HW to Accelerate Integer division

The true game-changer in performance will come from adding a dedicated HW block that can divide two 16-bit integers.

### Examining the ROM SW – Benchmark

There are two loops in the ROM Benchmark:  
(1) The outer loop - "LIP"  
I will try and run through the instructions as efficiently as possible (multi-issue, execute ion parallel).   
  
(2) The inner loop - "LOOP"  
When examining the sequence, you see that the algorithm divides 20,000/10 by subtracting 10 from 20,000 and counting number of iteration until M[2] is 0 or less.  
// M[2] initiates with 20,000  
@20000  
D = A  
@2  
M = D  
// M[3] initiates with 10  
@10  
D = A  
@3  
M = D  
// Start Loop  
(LOOP) @1  
M = M + 1  
@2  
D = M  
@3  
D = D - M   
@2  
M = D  
@2  
D = M  
// Loop condition M[2]>0.   
@LOOP  
D ; JGT  
  
With this observation, I have decided to add a dedicated HW block to divide two 16-bit integers.  
The machine implementation is a deterministic 16 cycle divide algorithm.

### triggering the dividing machine

I added an "instruction history recorder" to remember the last 24 instructions to know when to trigger the machine.  
Once the "instruction History" matches the division algorithm, I halt the program counter (PC), send the "Dividend" & "Divisor" to the Dedicated dividing HW, and return the PC after the loop once the division completes.

### To summarize

1) There is a naive algorithm to divide two integers in the inner loop. It takes 2,000 loops to complete - each loop takes 12 cycles.  
2) Added a deterministic HW divider that takes ~20 cycles.  
3) Added an "instruction history buffer" to match the SW divide sequence and trigger the HW divide machine. 

### My final resutls

We improved the program's running time from ~3 seconds to ~0.01 seconds using the HW divider.