VLSI System Design

(Graduate Level)

Fall 2021

HW1 report

Must do self-checking before submission:

Compress all files described in the problem into one tar

All System Verilog files can be compiled under SoC Lab environment

All port declarations comply with I/O port specifications

Organize files according to File Hierarchy Requirement

No any waveform files in deliverables

Student name: 鄭丞祥

Student ID: N26094891

# Problem Description

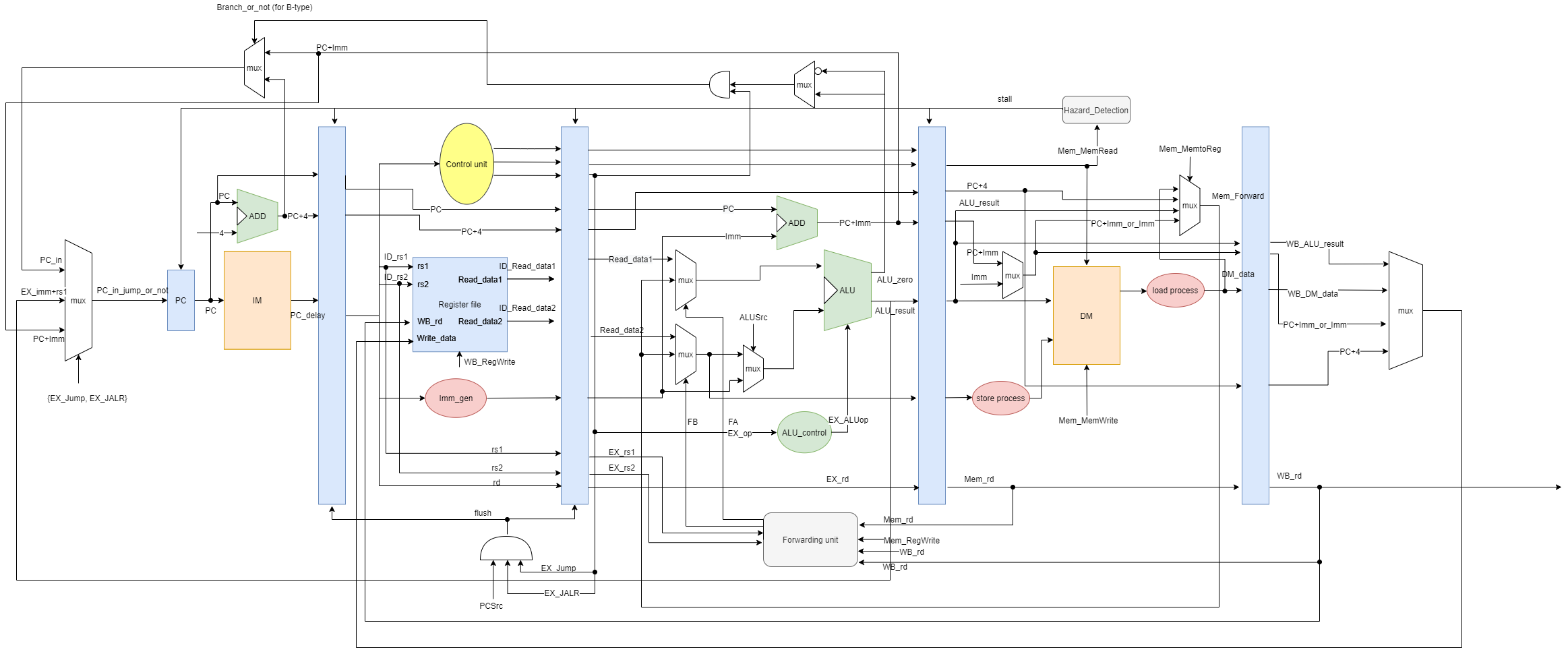
Implement a RISC-V pipelined CPU with the following requirements:

* The RISC-V ISA with the specified 37 instructions
* Pipeline stage is 5
* Register file size: 32 x 32-bits
* Solve data hazard, control hazard, and structural hazard
* Instruction memory size: 64KB
* Data memory size: 64KB

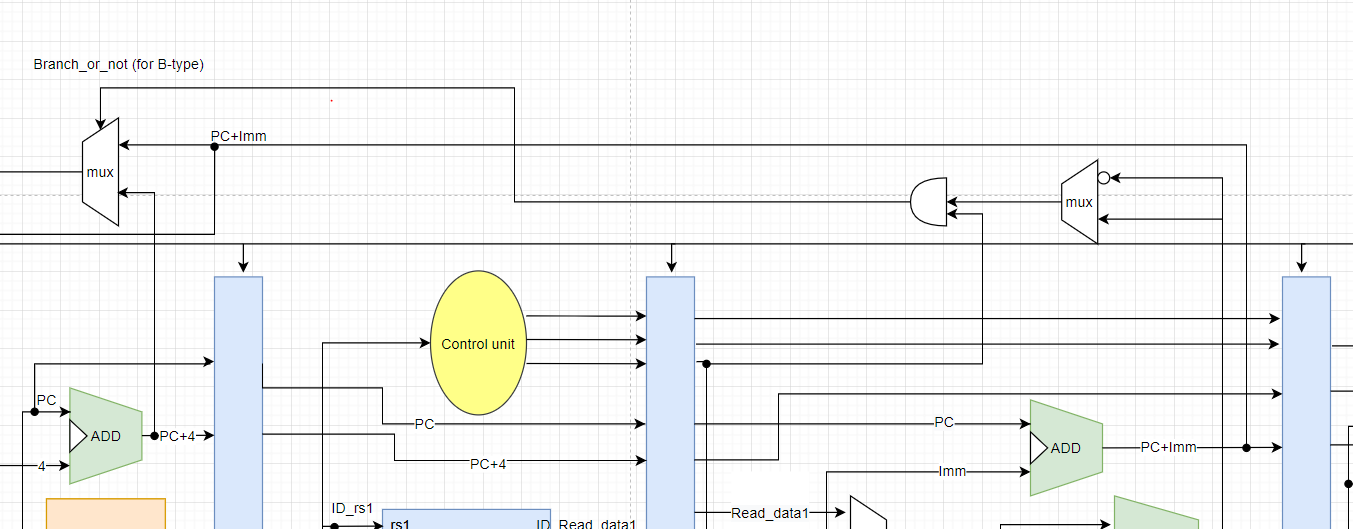
# Check List

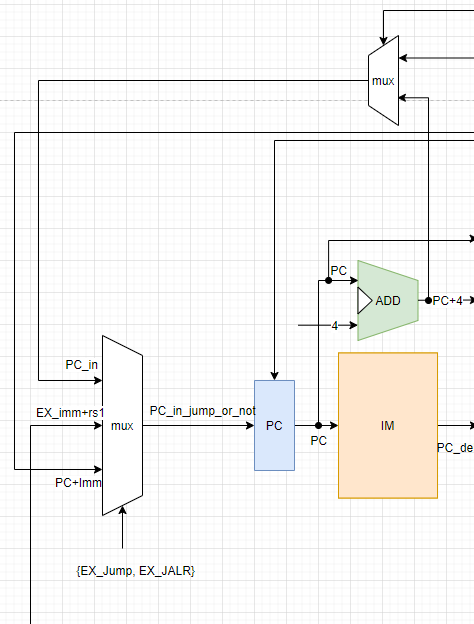
|  |  |  |  |
| --- | --- | --- | --- |
| RTL\_code | | Synthesis | |
| Prog0 | Pass | Syn0 | Pass |
| Prog1 | Pass | Syn1 | Pass |
| Prog2 | Pass | Syn2 | Pass |
| Prog3 | Pass | Syn3 | pass |
| Superlint | 100% | | |

# Block Diagram

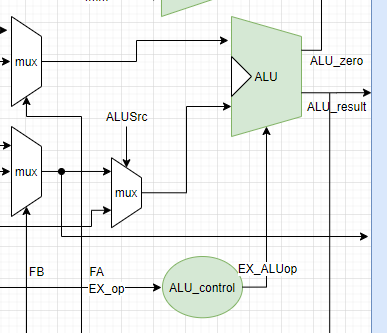
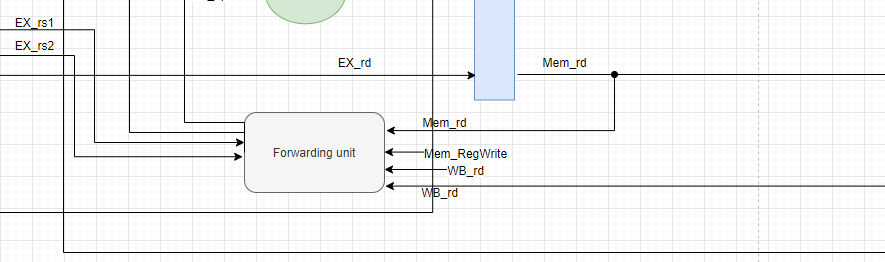
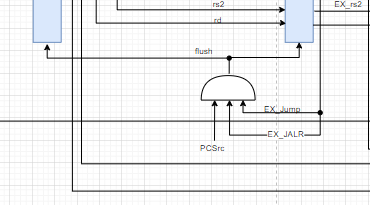
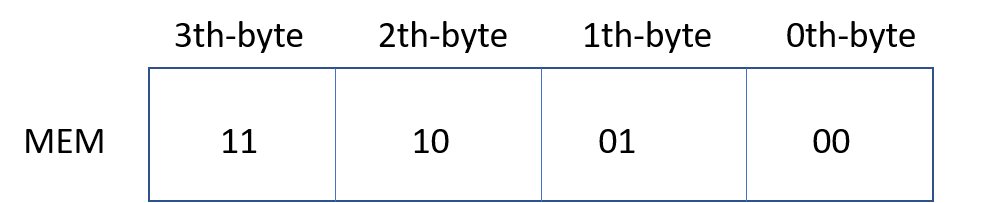
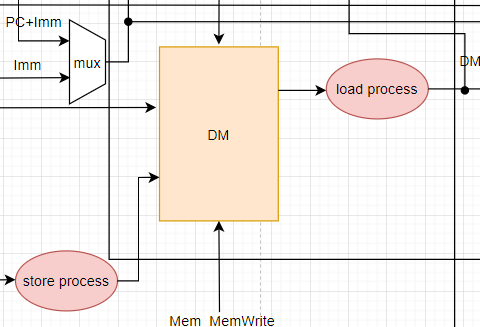


這次的HW1的主題是要設計一個5-stage pipelined的RSIC-V CPU，而在這個作業當中比較特別的地方是不僅需要對於pipeline的運算瞭若指掌，同時還需要針對因為pipeline化而產生的hazard問題，想辦法加以解決，以下開始各級pipeline的運作介紹:

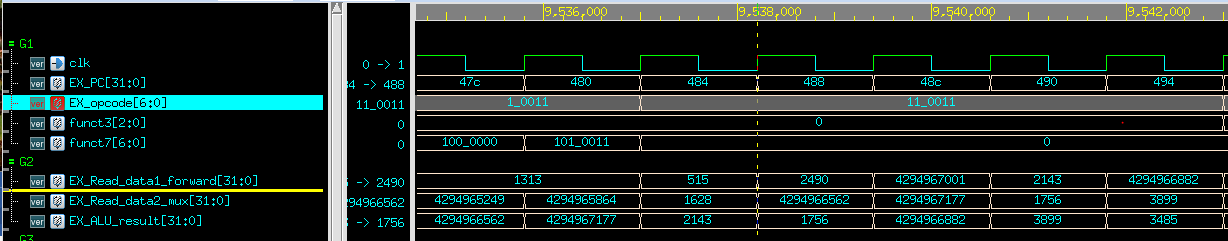
1. 首先是IF-stage階段，在這個階段首先必須要決定Point-Counter(PC)到底是多少，然而PC的數值為多少必須要先經過前面的MUX選擇才可決定，原因在於實作的指令中存在著B-type、JALR、J-type這三種指令，這些指令一旦被執行的話都會進而更動到PC的數值。
   1. 因為B-type指令而產生的PC數值是透過EX\_Branch這根訊號線和程式碼中被命名為condition\_sel的訊號線共同決定是否採用，所以究竟會跳與否，是端看兩根訊號線做AND後的結果，名為PCSrc的訊號，其會去選擇MUX輸出(PC\_in)是PC+Imm或PC+4，如圖所示:



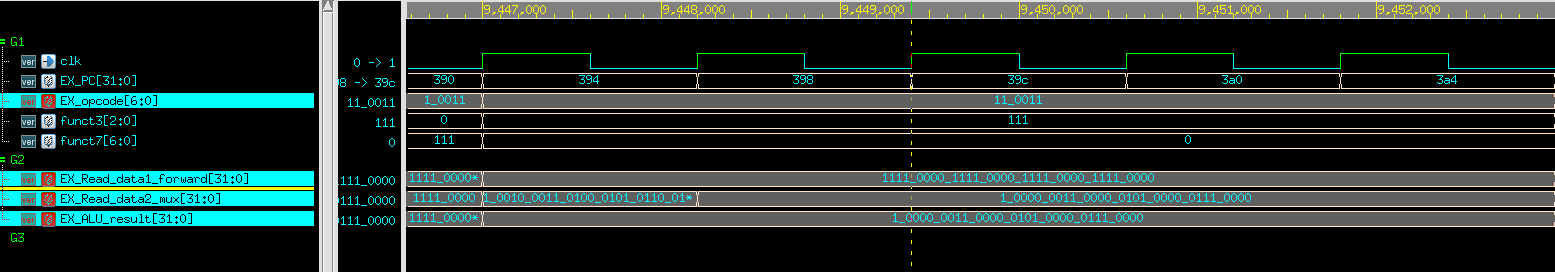
* 1. 判斷完Branch後，接下來還有JALR以及JUMP這個指令要判斷是否會跳，而這兩個指令的判斷方式也一樣是透過MUX來去選擇，當今天MUX的訊號為00、01、10時會各自對應到PC\_in、Imm+rs1(JALR)、PC+imm(J-type)這三項輸出，以得到下一個應該得到的正確PC值。區塊圖如右圖所示。
  2. 當電路產生出正確的PC以後，接下來只要將PC作為addr送入到IM，就可以得到正確的instruction，這裡唯一需要特別注意的地方在於因為IM為非理想的SRAM，所以當它接收到addr input，它的output instruction是在下一個clk才會送出!

1. 再來，經過第一級的pipeline後，指令會被送往ID-stage來去做解碼，透過RISC-V本身規定的格式，可以將32bit長的指令拆為rs1、rs2、rd、Imm、opcode等資訊。
   1. 這一級中的Register file(RF)會將被input進rs1、rs2這兩個addr，進而將這兩個addr中所存放的資料給output出去，成為Read\_data，同時間它也會負責處理在第五級pipeline中所write back回來的data和addr，將此data正確的寫入RF相對應的addr。
   2. 而同樣存在於ID-stage的Control unit負責處理的事情就是將instruction去做解碼，判斷這個指令是屬於哪一個type的指令，並且會透過opcode、funct3、funct7來決定說接下來此指令會需要執行甚麼功能，以決定要傳遞下去的控制訊號線該怎麼賦值。
   3. 由於並不是每個指令所用到的Imm都相同，因此在這級中的Imm\_gen負責的行為就是將Imm的應該要有的格式根據不同的指令去做正確的解讀，以便輸出正確的Imm，也就是處理Imm格式問題。
2. 在這個時間點，資料經過了IF，ID，以及兩級pipeline後來到了EXE階段，在這個階段，會完成整個CPU主要的運算，像是加法，移位，OR等計算，而主要實現這些運算的功能單元為ALU，但要讓ALU知道它此時該做甚麼運算前，會需要先利用ALU control這個單元得到指令相關的詳細資訊才可以進行動作。
   1. ALU control主要會需要從Control那邊先得知目前正在進行的指令是哪種type，得知以後會再根據各個type，依照它們的funct3、funct7來做編碼，也就是當此編碼後的3bit為多少應該做甚麼事情，這個編碼會作為輸出傳遞給ALU，讓ALU接收到這邊編碼後在內部解碼以得知當前指令它應該需要負責甚麼運算。至於會利用這種兩段式的控制方式，而不是直接利用ID的control給ALU對應的訊號線的原因是在於為了減低control控制器的控制線複雜度。概念如右圖所示。
   2. 而在這一級有個重要的功能單元名為Forwarding unit，這個單元的目的是為了解決因為pipeline化後而產生的data hazard問題，當今天輸入的某個指令與它前一個(兩個)指令所以使用的暫存器具有相依性，且具有立即使用的需求時，就會需要透過Forwarding unit將前一個(兩個)已經算出來的或者是讀取出來的資料即時的回饋到此指令當前的stage。以我的架構來說，電路會判斷EXE stage中ALU的兩個來源暫存器(rs1、rs2)是否與MEM stage、WB stage的目的暫存器一樣，如果一樣的話就表示若我要正確執行當前在EXE階段的這個指令，就必須先行把MEM、WB的資料拿回來當作ALU的輸入，因為若不透過forwarding把資料先行拿回來當作input的話則EXE階段的指令會以錯誤的數值去進行計算(也就是因為MEM、WB最終的數值還未存入RF裡面，所以EXE階段的指令它從RF存取到錯誤的數值)。概念如下圖所示。
   3. 另外在這一級還會有個重要的判斷就是JALR、JUMP以及Branch指令，當今天指令的執行結果告訴CPU它確定要跳，則會需要將已經進入到IF/ID的指令給flush到，因為這兩個指令不是經過跳躍後需要被執行的指令，不flush掉的話會影響到整個CPU讀值取值的運行。概念如右圖所示。
3. 指令來到了MEM階段，在這個階段中會進行的行為主要是把ALU運算出來的結果存入DM或是把DM中相對應位置的資料給讀取出來，但因為這次作業除了單純的LW、SW以外，還要求要實現LB、SB、LH、SH等行為，因此在進行load、store指令前會先進行預處理，以能正確執行指令。
   1. 舉SB為例，因為一個word是4 byte一組，所以在一個word裡面可以存放一個byte的位置共有四個，而今天ALU運算出來的結果為要存入DM的addr(也就是要存入第幾個byte)，因此要如何能讓它4 byte aligned 的方法就是看ALU result的最後兩個bit，這兩個bit會決定資料該存入1個word中的第幾個byte。SH同理。概念如下圖所示。
   2. 而同樣舉LB為例，它其實所執行的動作也就是只是將1個word中那4個byte的資料挑一個讀取出來，如果今天讀取的資料是00位置的資料，那剩餘左邊的bit數就以sign extension都方式補齊到32bit長即可。如右圖所示。
   3. 而在這一級中還有最後一項動作相當重要，也就是為了解決load-use問題而產生的功能單元hazard detection，也就是stall operation。在我的架構設計中，我是MEM stage判斷stall訊號是否該為1來去執行對應行為，當今天hazard detection接收到從control unit那邊而來的MemRead訊號為1時，表示當前的指令是lw，這時CPU就會讓stall為1，使的IF/ID、ID/EXE、EXE/MEM這三個pipeline以及PC這個flip-flop都保持原狀，會需要這樣的原因在於因為DM不是理想的，當LW剛來到MEM stage時它的addr剛餵進去DM來去找尋對應的資料，而相對應的資料則會需要一個clk的時間，也就是在下一個clk的時候才會吐出，因此才會需要stall住所有的pipeline和PC，讓大家都保持原狀一個clk以等待正確的值從DM讀出後並forwarding回去，來達到正確的運行結果。
4. 最終，在WB stage的行為是要把data寫回去RF裡面，而要寫回去的資料會有四種情況，分別是ALU result、DM data、PC+4 & PC+Imm，所以這裡會利用一個4 to 1 Mux來選擇要write back回去的資料究竟該要哪一個才正確。

# Program Test

* Prog 0:
  + R-type:
    1. 測試ADD功能:

可以看到當前EXE階段的opcode=11\_0011，確認為R-type，而funct3和funct7皆為0，因此為ADD指令，而ALU的兩個Input為515和1628，兩者相加後的結果為2143，結果正確。

* + 1. 測試AND功能:

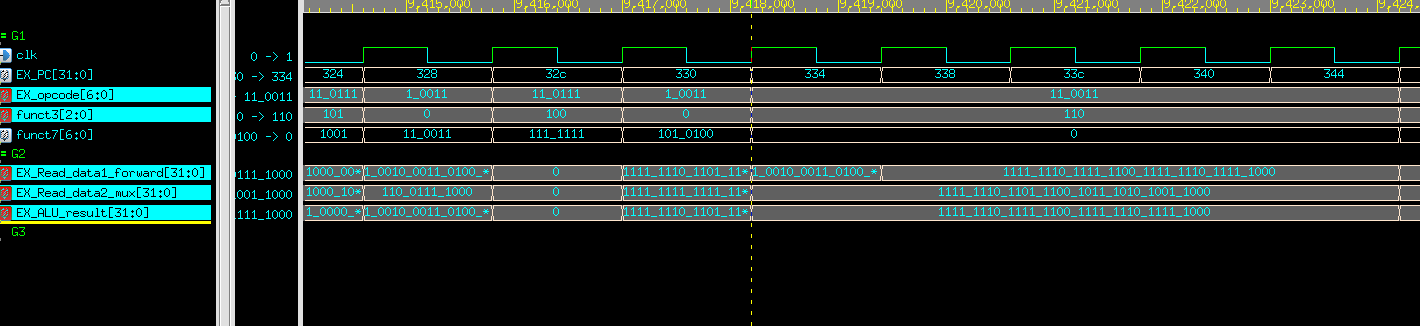
當前的EXE opcode=110011，且funct3=111，因此為R-type的AND指令，而ALU的兩個input如下

ALU\_in1: 1111\_0000\_1111\_0000\_1111\_0000\_1111\_0000

ALU\_in2: 0001\_0000\_0011\_0000\_0101\_0000\_0111\_0000

ALU\_result: 0001\_0000\_0011\_0000\_0101\_0000\_0111\_0000

結果正確。

* + 1. 測試OR功能:

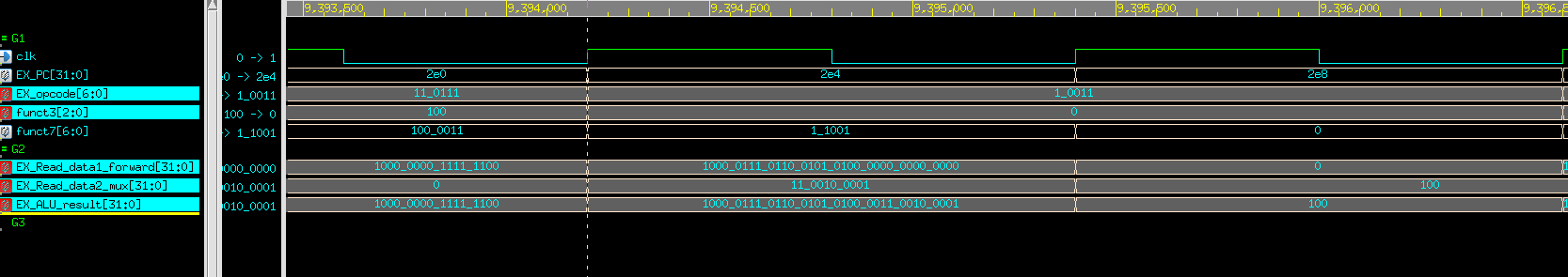
當前指令的opcode = 110011，funct3 = 110，funct7 = 0，所以為OR指令，而ALU的兩個input為

ALU\_in1: 1111\_1110\_1111\_1100\_1111\_1110\_1111\_0000

ALU\_in2: 1111\_1110\_1101\_1100\_1011\_1010\_1001\_1000

ALU\_result: 1111\_1110\_1111\_1100\_1111\_1110\_1111\_1000

結果正確。

* + I-type:
    1. 測試ADDI功能:

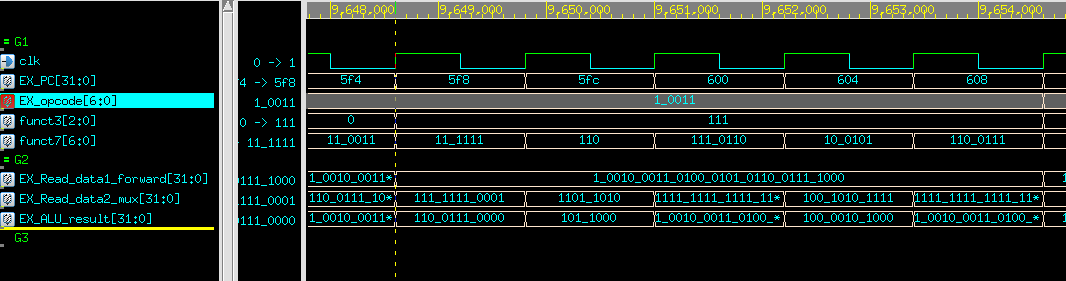
當前的EXE opcode=10011，且funct3=0，因此為I-type的ADDI指令，而ALU的兩個input如下

ALU\_in1: 1000\_0111\_0110\_0101\_0100\_0000\_0000\_0000

ALU\_in2: 0000\_0000\_0000\_0000\_0000\_0011\_0010\_0001

ALU\_result: 1000\_0111\_0110\_0101\_0100\_0011\_0010\_0001

結果正確。

* + 1. 測試ANDI功能:

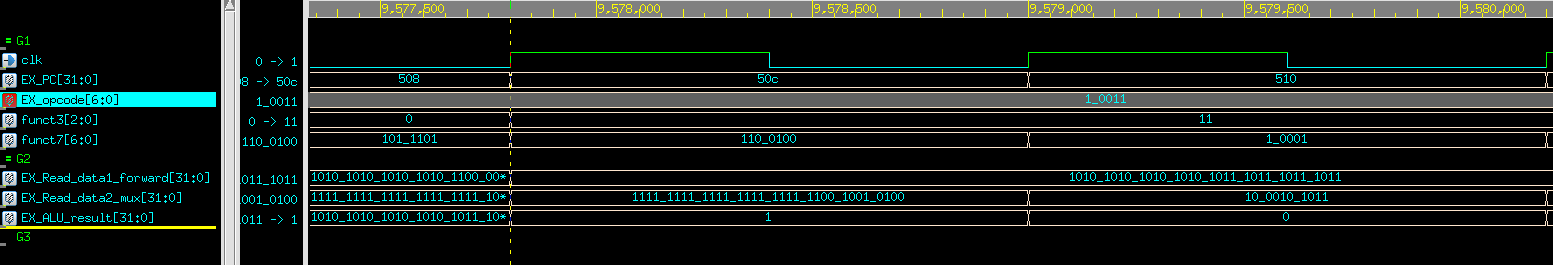
當前的EXE opcode=10011，且funct3=111，因此為I-type的ANDI指令，而ALU的兩個input如下

ALU\_in1: 0001\_0010\_0011\_0100\_0101\_0110\_0111\_1000

ALU\_in2: 0000\_0000\_0000\_0000\_0000\_0111\_1111\_0001

ALU\_result: 0000\_0000\_0000\_0000\_0000\_0110\_0111\_0000

結果正確。

* + 1. 測試SLTIU功能:

當前的EXE opcode=10011，且funct3=11，因此為I-type的SLTIU指令，而ALU的兩個input如下

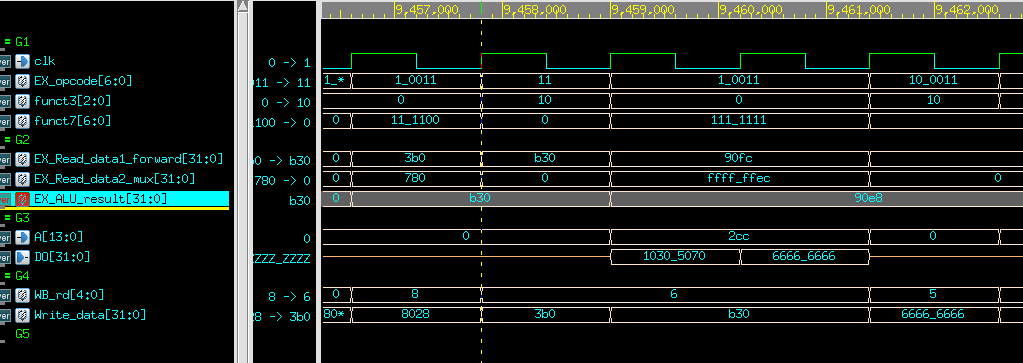
ALU\_in1: 1010\_1010\_1010\_1010\_1011\_1011\_1011\_1011

ALU\_in2: 1111\_1111\_1111\_1111\_1111\_1100\_1001\_0100

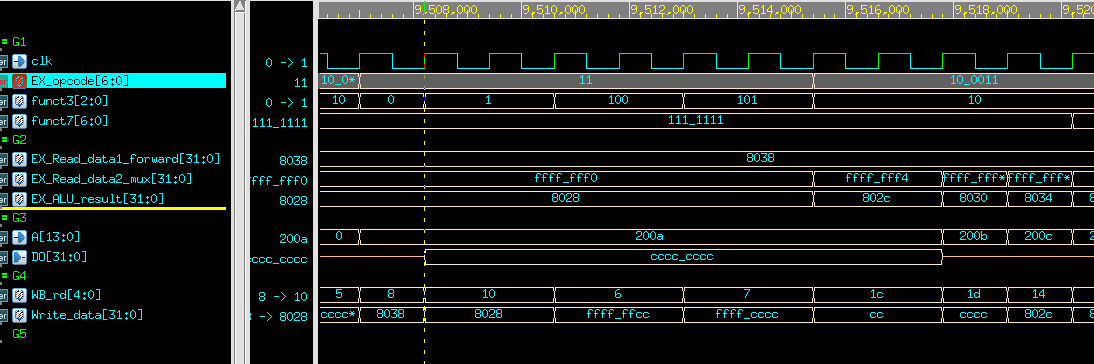
因為ALU\_in1 < ALU\_in2，所以結果為1

ALU\_result: 1

結果正確。

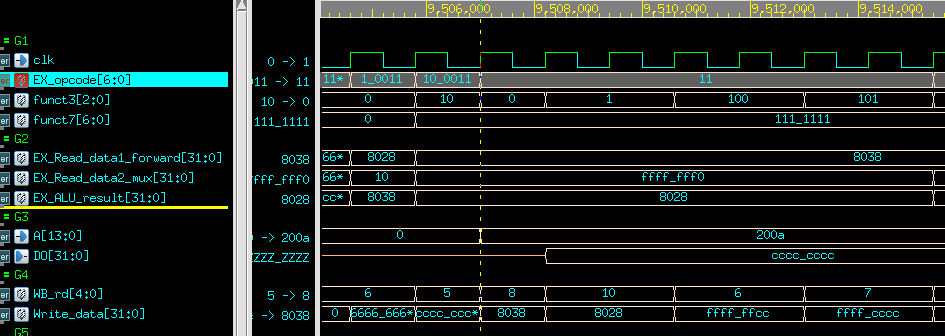
* + LW、LH、LB:
    1. 測試LW功能:

當前的EXE opcode=11，且funct3=10，因此為I-type的LW指令，而ALU result的答案會在下個cycle時跑到MEM stage並除以4作為DM的addr輸入，也就是M[ALU\_result/4]，而此位置的data會再相隔一個cycle吐出，要注意的是因為LW在MEM stage會有stall狀況產生，因此LW會給的addr會在MEM stage維持兩個cycle。

* + 1. 測試LH功能:

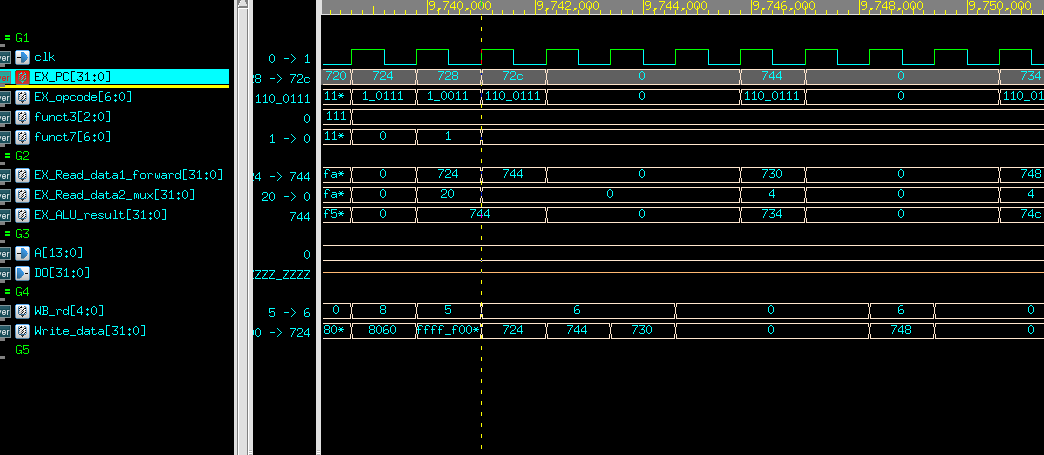
當前的EXE opcode=11，且funct3=1，因此為I-type的LH指令，而ALU result的答案會在下個cycle時跑到MEM stage並除以4作為DM的addr輸入，也就是M[ALU\_result/4]，而此位置的data會再相隔一個cycle吐出，要注意的是因為LH在MEM stage會有stall狀況產生，因此LH會給的addr會在MEM stage維持兩個cycle。

其中特別要注意的是因為這是LH指令，空缺的bit數會用sign extension來補齊。

* + 1. 測試LB功能:

當前的EXE opcode=11，且funct3=0，因此為I-type的LB指令，而ALU result的答案會在下個cycle時跑到MEM stage並除以4作為DM的addr輸入，也就是M[ALU\_result/4]，而此位置的data會再相隔一個cycle吐出，要注意的是因為LB在MEM stage會有stall狀況產生，因此LB會給的addr會在MEM stage維持兩個cycle。

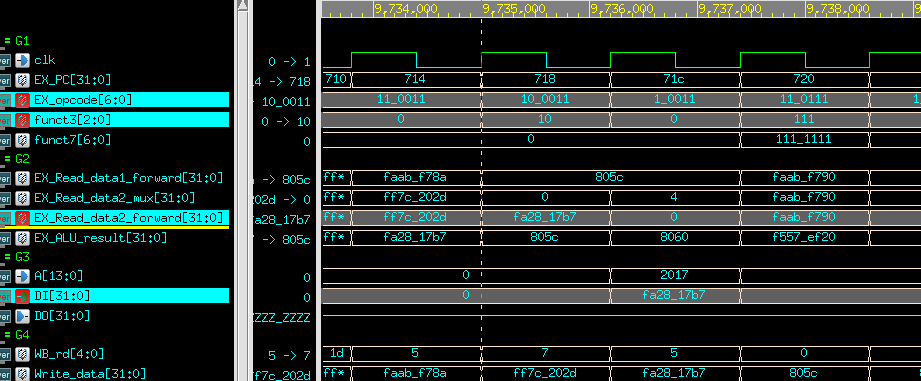
其中特別要注意的是因為這是LB指令，空缺的bit數會用sign extension來補齊。

* + JALR:
    1. 測試JALR功能:

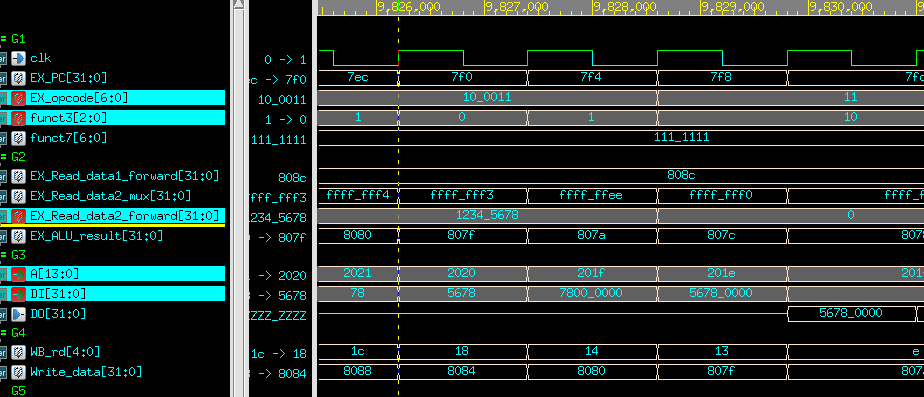
PC+4

當前的EXE opcode=110\_0111，因此為I-type的JALR指令，而ALU的結果為之後要跳躍的目標位置，由圖可知

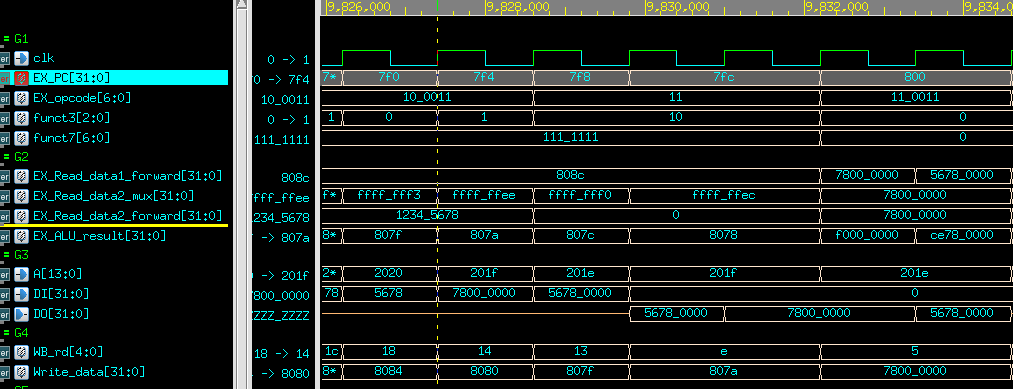
ALU\_result: 744，所以之後PC將會被更新成744這個位置。而當前的PC為72c，其PC:72c+4=730以後會作為新的rd值被寫回去。

* + S-type:
    1. SW功能測試:

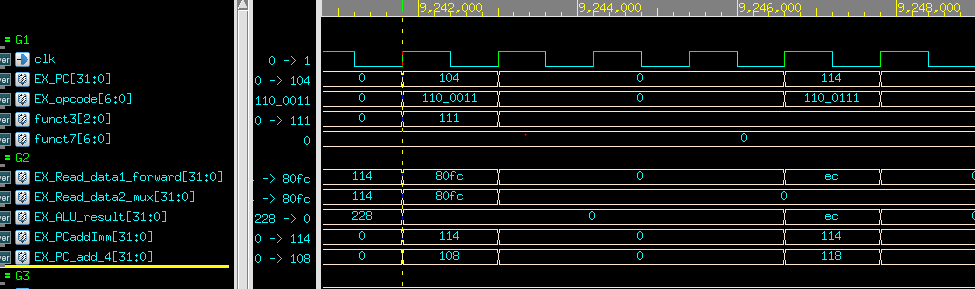
當前的EXE opcode=10\_0011，funct3=10，因此為S-type的SW指令。因為ALU所算出來的答案除以4後會作為DM的addr輸入，所以由圖上可以看到805c / 4 = 2017在下一個cycle也就是sw指令進入到MEM階段時有將2017作為addr輸入給DM，而同時間也會把EX\_Read\_data2\_forward的資料作為input寫入data，因為這是SW，所以會全部完整的寫入。

* + 1. SB功能測試:

當前的EXE opcode=10\_0011，funct3=00，因此為S-type的SB指令。因為ALU所算出來的答案除以4後會作為DM的addr輸入，所以由圖上可以看到807f / 4 = 201f在下一個cycle也就是SB指令進入到MEM階段時有將201f作為addr輸入給DM，而同時間也會把EX\_Read\_data2\_forward的資料作為input寫入data，因為這是SB，且ALU result最後的兩個bit為11，因此會把EX\_Read\_data2\_forward的最後一個byte(value = 78)寫入到最左邊的byte處。

* + 1. SH功能測試:

當前的EXE opcode=10\_0011，funct3=01，因此為S-type的SH指令。因為ALU所算出來的答案除以4後會作為DM的addr輸入，所以由圖上可以看到807a / 4 = 201e在下一個cycle也就是SH指令進入到MEM階段時有將201e作為addr輸入給DM，而同時間也會把EX\_Read\_data2\_forward的資料作為input寫入data，因為這是SH，且ALU result的第二個bit為1，因此會把EX\_Read\_data2\_forward的0~15個bit放到高位元處(31~16 bit)並把剩餘的LSB部分補0。

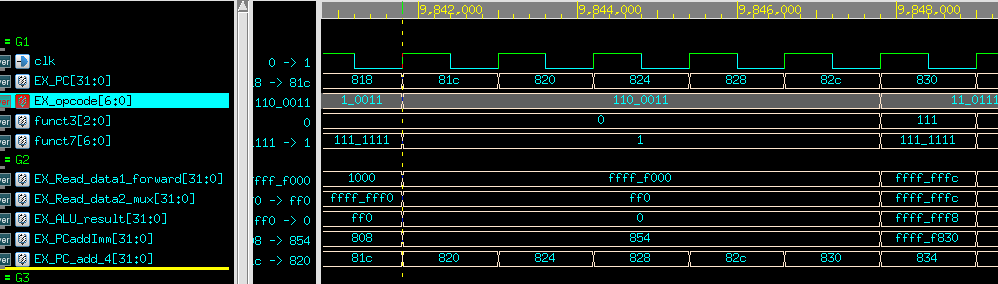
* + B-type:
    1. BGEU功能測試:

當前的EXE opcode=110\_0011，且funct3=11，因此為B-type的BGEU指令，而ALU的兩個input如下

ALU\_in1: 80fc

ALU\_in2: 80fc

因為ALU\_in1 >= ALU\_in2，所以下一個指令的PC應該要為PC+Imm，也就是圖中的114這個數字，而確從圖中也可以看到下一個進來的指令其PC=114。中間為0的部分為被flush掉的指令，兩個cycle為前兩級的指令被flush，第三個cycle的flush是因為要跳的PC餵進去IM後還會需要一個cycle的時間指令才會出來。

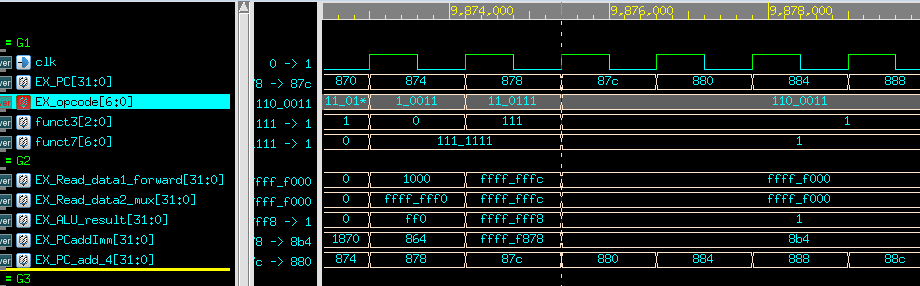
* + 1. BEQ功能測試:

當前的EXE opcode=110\_0011，且funct3=0，因此為B-type的BEQ指令，而ALU的兩個input如下

ALU\_in1: ffff\_f000

ALU\_in2: ff0

因為ALU\_in1 != ALU\_in2，所以說在這次的指令中其不會做跳躍的動作，也不會有flush的行為，下一個PC一樣為PC+4。

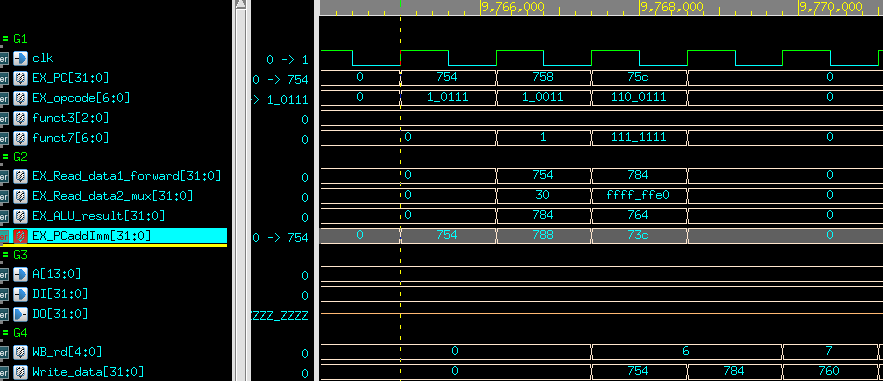
* + 1. BNE功能測試:

當前的EXE opcode=110\_0011，且funct3=1，因此為B-type的BNE指令，而ALU的兩個input如下

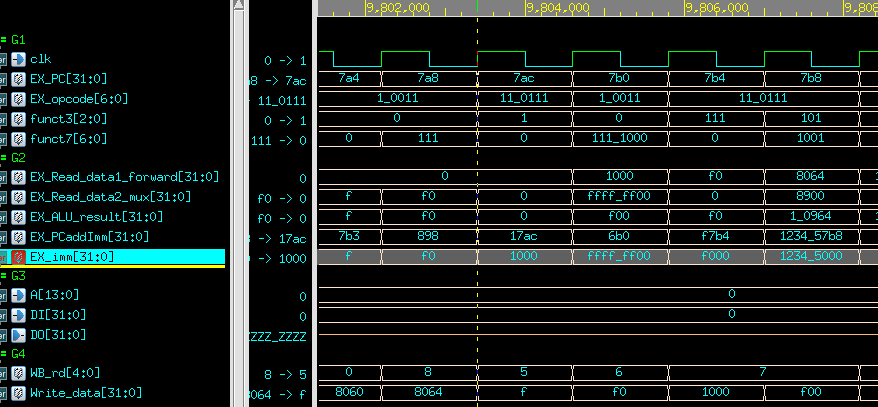
ALU\_in1: ffff\_f000

ALU\_in2: ffff\_f000

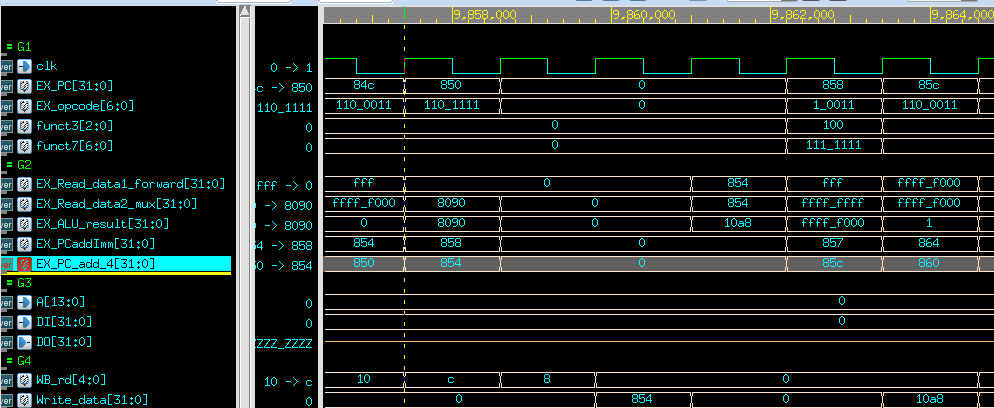
因為ALU的兩個input皆相同，所以說不符合BNE要跳的條件，也就是不會跳，因此下一個PC仍然保持PC+4。

* + U-type:
    1. AUIPC功能測試:

當前的EXE opcode=1\_0111，因此為U-type的AUIPC指令，該指令會把PC+Imm作為要寫回去RF的data，如圖所示。

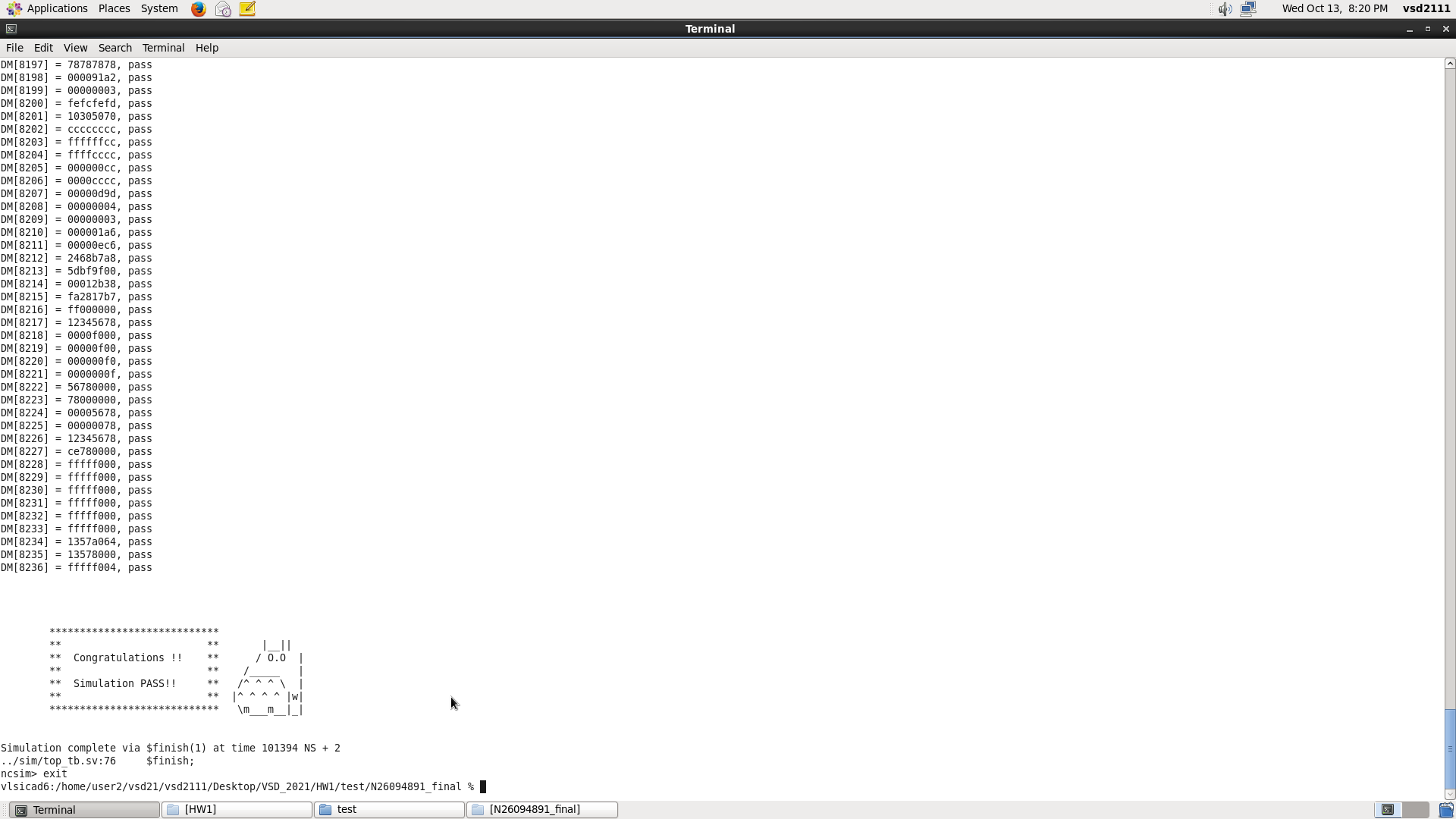
* + 1. LUI 功能測試:

當前的EXE opcode=11\_0111，因此為U-type的LUI指令，該指令會把Imm作為要寫回去RF的data，如圖所示。

* + J-type:
    1. JAL功能測試:

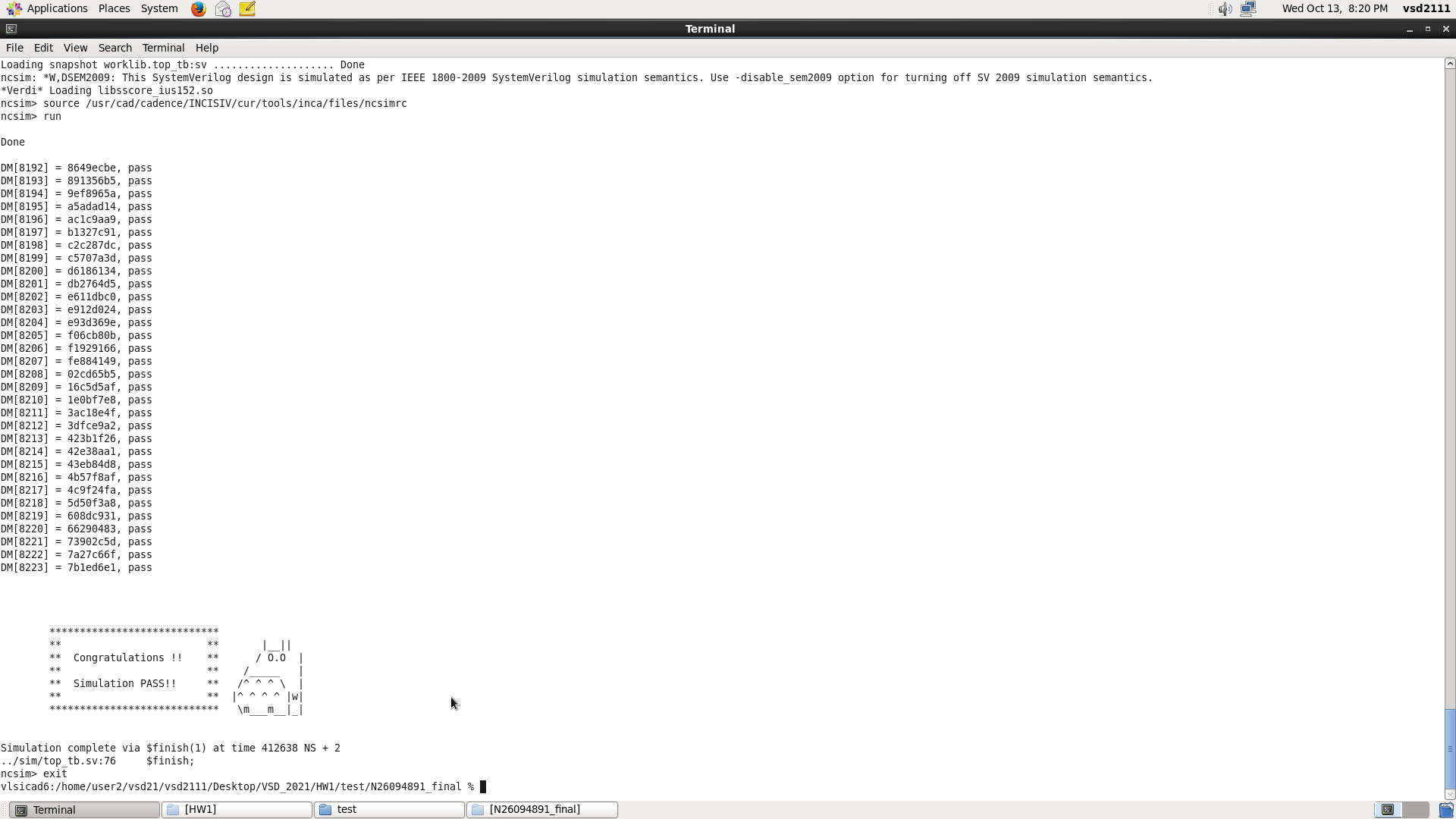
當前的EXE opcode=110\_1111，因此為J-type的JAL指令，該指令會把PC+4作為要寫回去RF的data，而要跳躍的目的PC為PC+Imm，因此可以看到圖中下一個PC的數值為PC+Imm，而write data也為PC+4，又因為此指令有做跳躍的動作，所以會需要將其兩級pipeline中的指令給flush掉，以避免CPU計算錯誤。

* + Prog0測試結果:



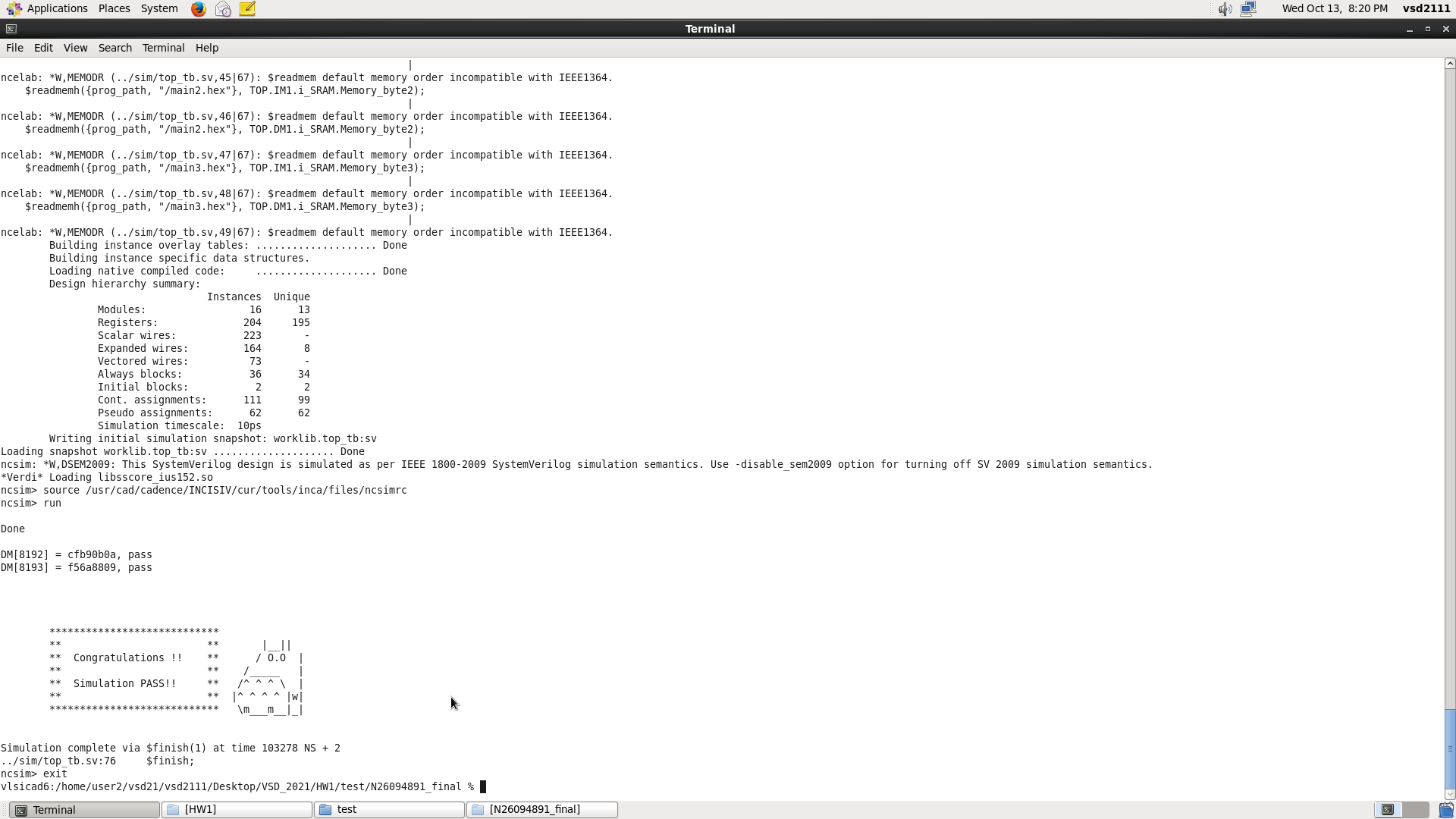
* Prog1: Sort Algorithm

此測試程式可以利用單純的bubble sort完成，基本原理就是不斷地和自己隔壁的數字做比較，當我數字比隔壁的大時就做交換，比較小時就不做任何行為，不斷重複此行為，直到所以的數字都彼此比較過後即完成此演算法。

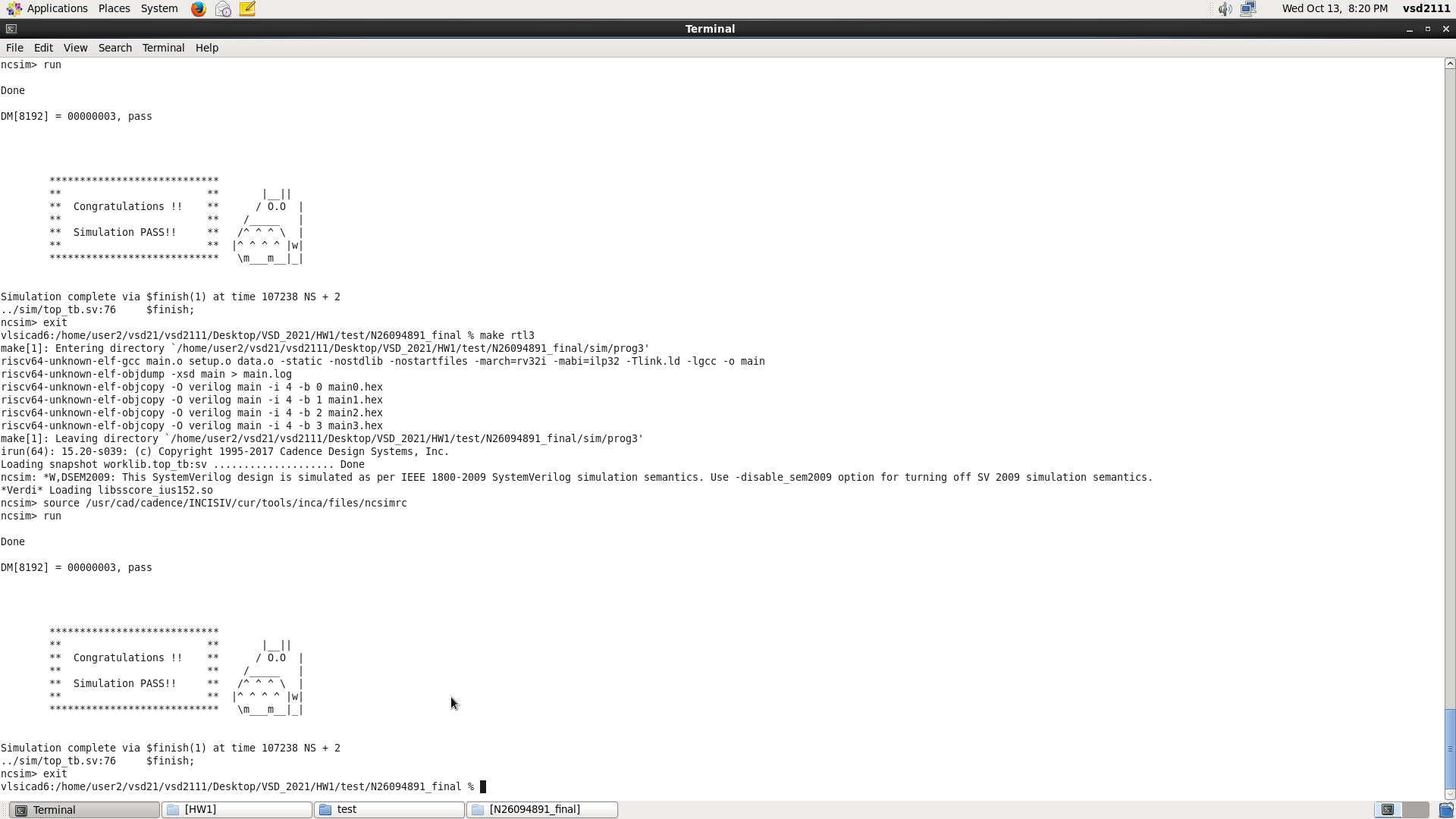


* Prog2: Multiplication

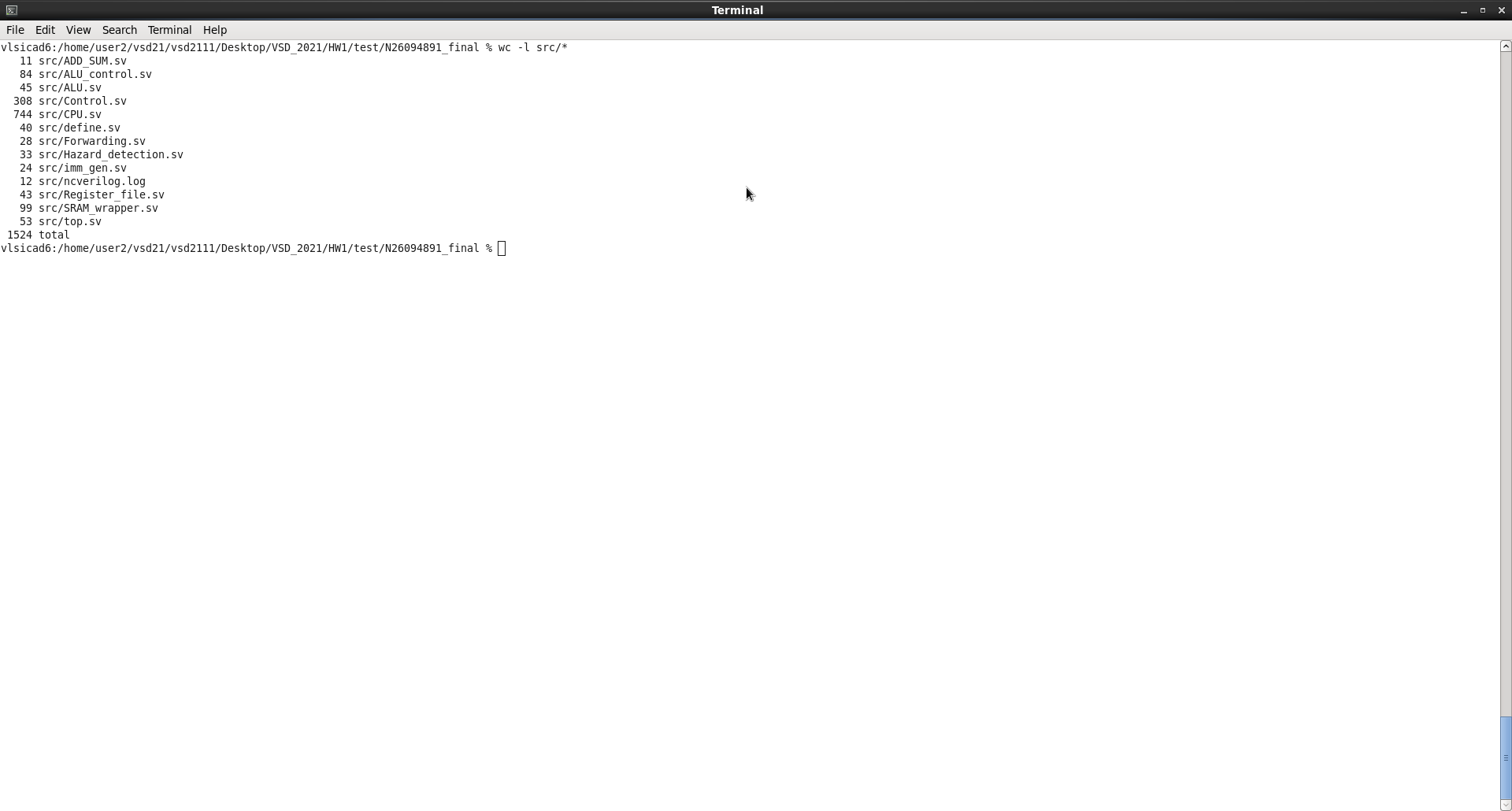
因為此測試程式要求的行為模式為將兩個32bit的數字做相乘，而32bit x 32bit所得到的答案會是64bits 長。 因此相乘後需要將答案分成兩部分存入32bit 的Memory，分成兩部分32bit長的運算可以透過shift(>>)來達成。



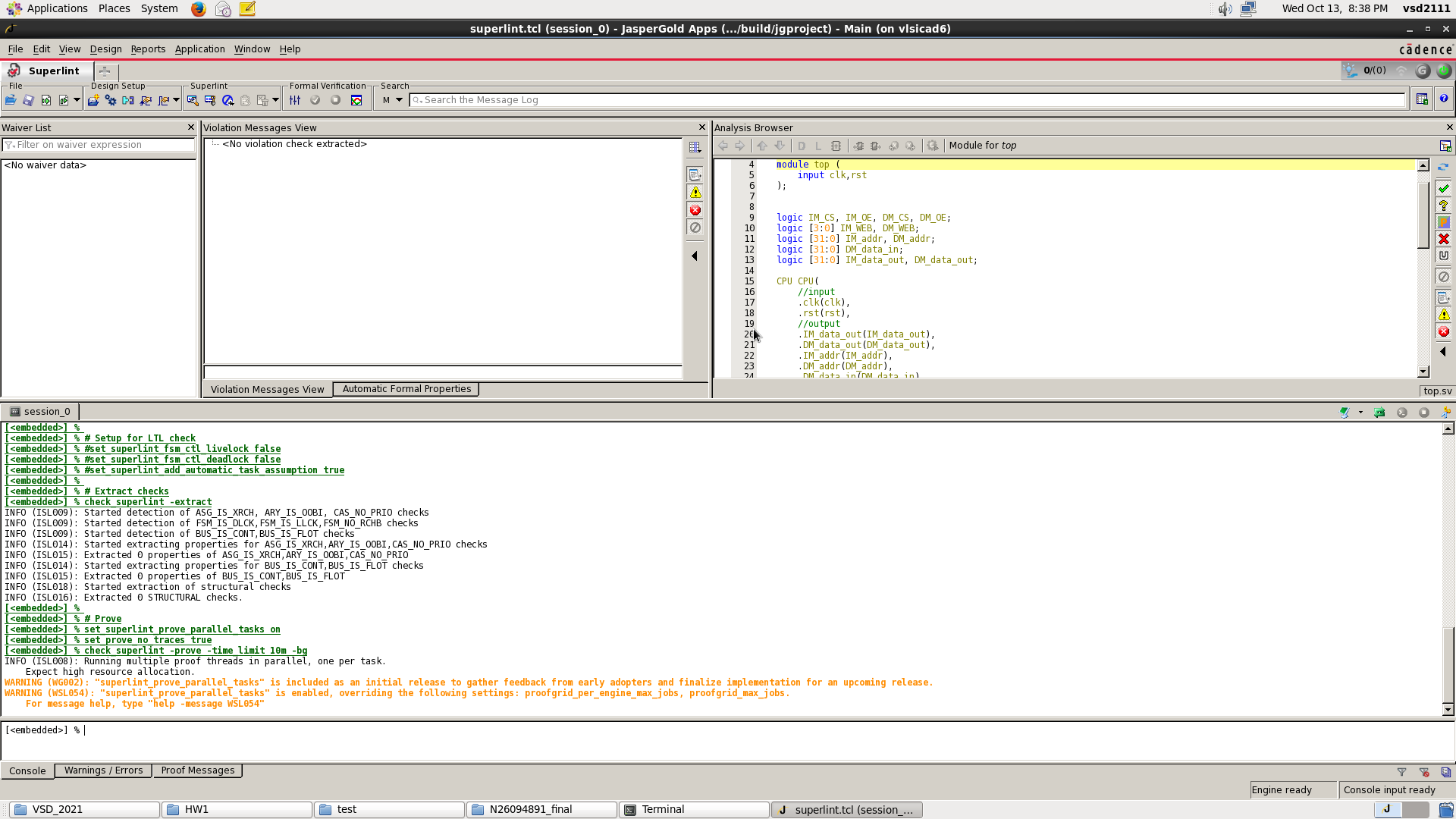
* Prog3: Greatest common divisor (GCD)

此範例程式的要求可以透過利用輾轉相除法來完成，輾轉相除法的概念就是透過將兩數相除後所得到的餘數做為下一次除法的輸入，不斷地執行，直到餘數為0的時候，此時的除數即為最大公因數。

# Number of lines of RLT code: 1524



# Superlint result:



(無任何syntax error或warning)

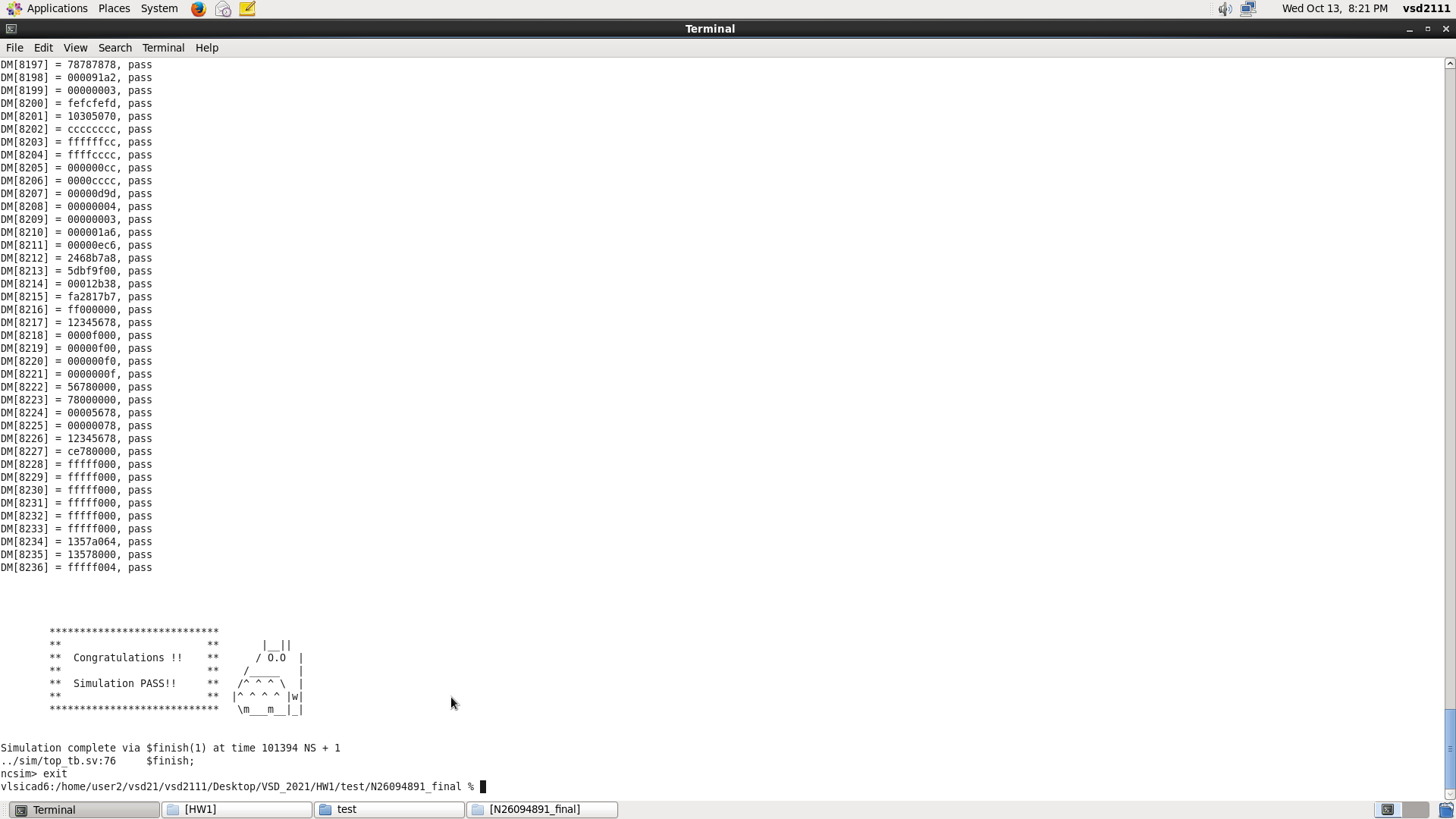
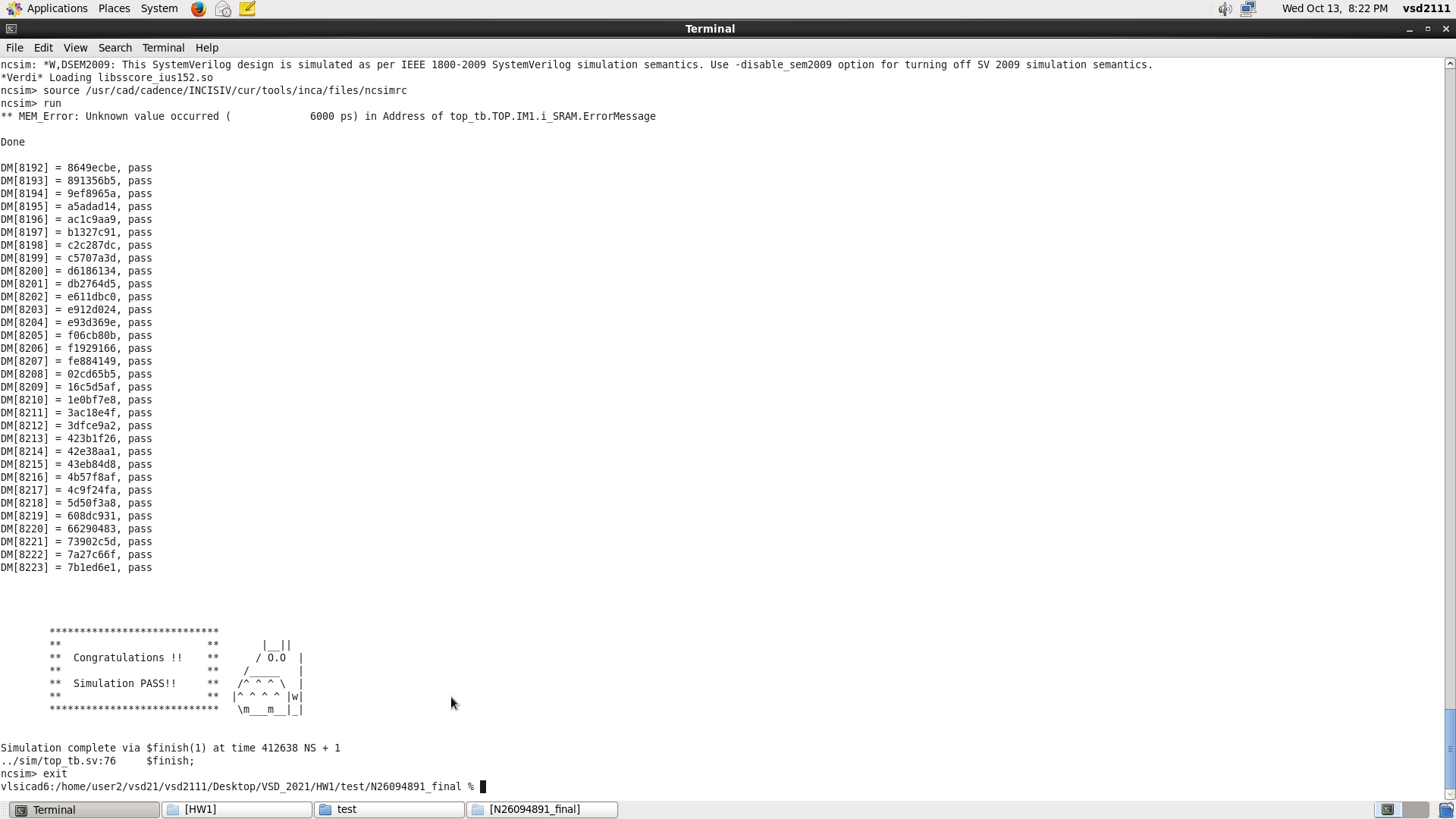
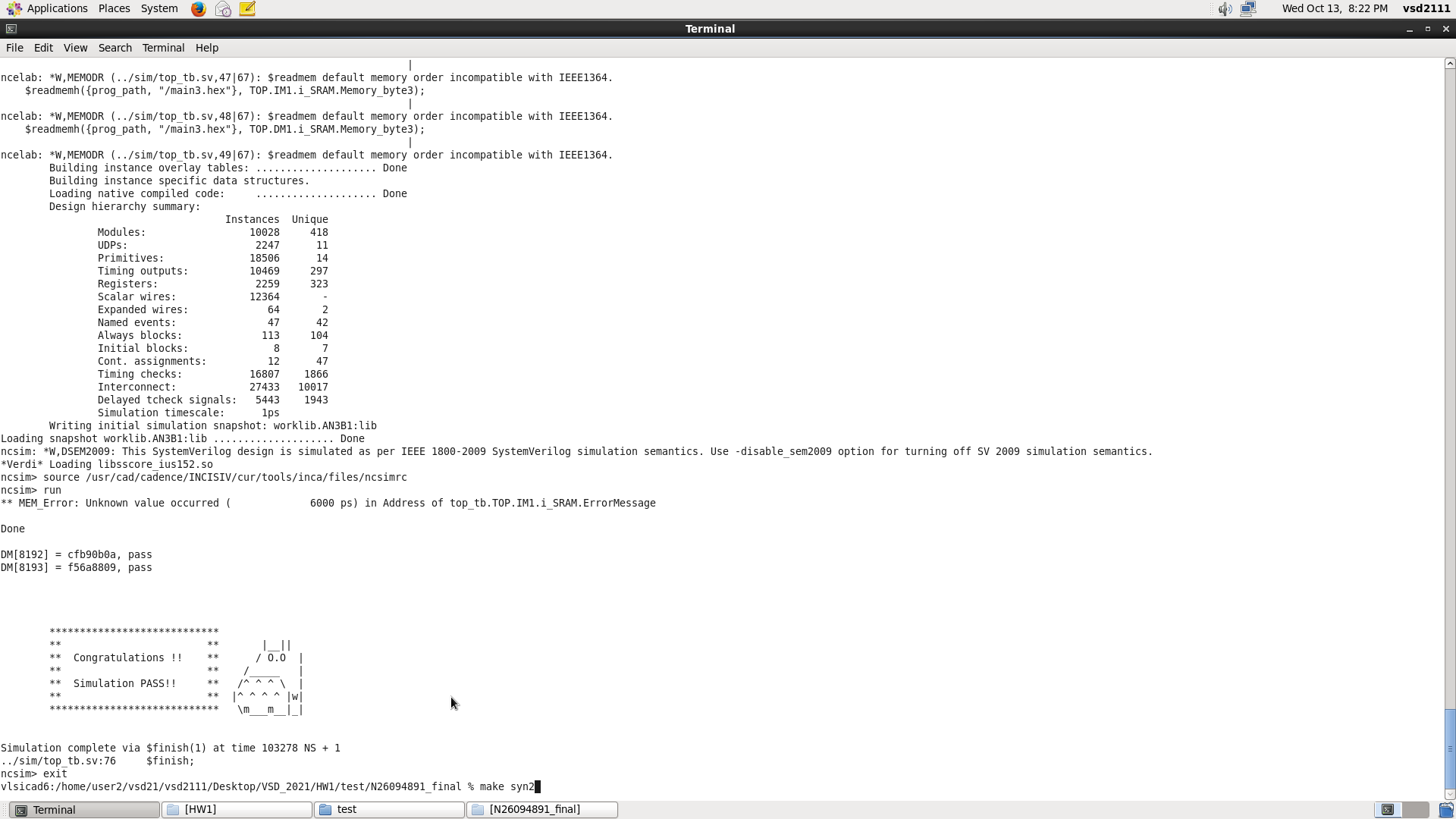
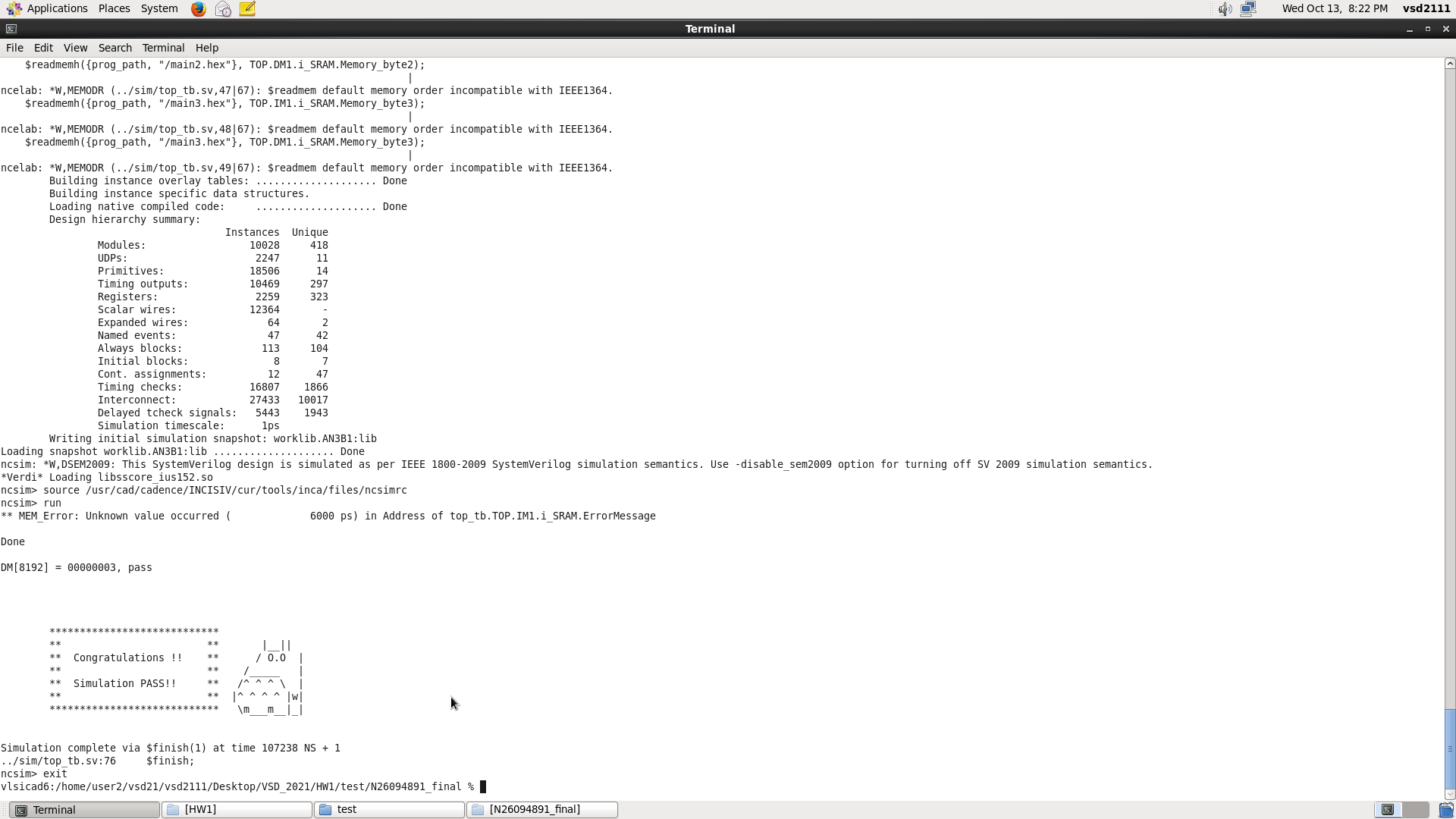
常見的warning有:

1. Case的條件若都寫滿了，其實不需要再寫default，若有寫則會出現warning。
2. 訊號線給值的時候不能只單純寫數值，要寫被賦值對象的bit數，例如assign A = 5’d0; 而不能只寫assign A = 0;
3. 常常會有某兩個接線的module他們彼此之間同一個變數互接線，但兩邊的變數擁有的bit數量不同。

# Synthesis:

* CLK -> 12
* Synthesis Area:

= 5596428.4

* Simulation time:
  + Prog 0: 101394 NS
  + Prog 1: 412638 NS
  + Prog 2: 103278 NS
  + Prog 3: 107238 NS
* PA:
  + (t0 + t1 + t2 + t3) \* Area

= (101394 + 412638 + 103278 + 107238) \* 5596428.4

=

# Problems encountered

在這次的作業當中我遇到了不少的問題，很多問題的產生其實都來自於對CPU的運作不熟悉所產生。

第一個問題首先是當我嘗試將各個子模組兜成一個初版，不含hazard detection和Forwarding的CPU時，我發現我接到一半，對於該怎麼把數值正確的寫進去DM，又獲是如何正確地把資料從DM讀取出來不甚了解，原因在於剛開始我根本不知道那顆SRAM是怎麼運作，不知道當我今天餵給它資料時，它甚麼時候會把資料吐給我，而且要是在甚麼控制訊號線下才會吐資料給我，雖然助教的PPT上面有畫了時序圖，但還是讓我困惑了一陣子。而了解清楚後又遇上不知道LB/SB、LH/SW等指令的行為是怎麼樣，來來回回翻了不少資料以及詢問同學後才搞清楚。

第二個大問題是資料相依的處理方式，雖然在大學時的計算機組織有學過這個概念了，但因為時間久遠一時之間突然對於該如何實作滿是疑惑，為此我因此回去複習了以前計算機組織的課本，重新把為什麼會有forwarding unit、hazard detection這兩個元件存在的必要等相關知識重新補齊後，才好不容易完成了資料相依時的解決辦法。

最後，這次作業中讓我感到最頭痛的問題點莫過於會跳躍的指令配合上Load的問題，會跳躍的指令會需要考量到它是在哪一級跳躍，如果確定有跳躍的話則會需要把前面幾級錯誤的指令給flush掉，而flush指令時又必須要小心因為SRAM為非理想的關係，這一個cycle餵給它addr，它要在下一個cycle才會吐出相對應的指令出來，因此flush需要的次數不像課本上的所講述的那麼直觀，還必須多外加flush一個cycle來處理指令晚出現的狀況。更棘手的是，假如今天跳躍指令的前面一個指令是lw的話，對於我的設計來說更是細微中的bug，因為我的設計是在MEM階段看到lw就會進行stall的動作，所以當MEM階段產生了stall訊號，它會同時讓前面三級的所有訊號都保持原狀，但當EX階段是跳躍相關的指令時，它會想要去更新PC的數值，簡單講，就是同一時間點會有兩個指令想要對PC做更新的動作，但我當初在prog0的時候沒有考量到這種事情，而且prog0也沒有這樣的case出現，因此在prog1卡關了。而prog1波形圖和產生出來的組語指令相當然難閱讀和debug，因為它是排序的測試程式，PC會跳來跳去的，導致我對於這小小的bug，花了巨大的時間成本去找它。

# Lesson learned

在完成這次的作業之前，我在以往會使用verilog的課都只有寫過小小的特殊應用電路而已，行數最多6、700行上下，像這樣完成一顆具有完整功能的CPU在此之前我是完全沒有經驗，從子模組開始刻起，各個子模組之間的溝通透過控制單元傳送控制訊號線來溝通，然後再將彼此合成一個大電路。像這樣了寫法讓我深刻體會到實現一個獨立系統的難度究竟有多高，而且有了block diagram對於設計者來說是多麼一件重要的事情，光是RISC-V CPU這種設計圖存在已久的電路我都可以寫這麼久了，若是今天想要研發一顆前瞻性的晶片，從架構開始打磨所需要花費的時間令我難以想像，而有了block diagram我也才能順利地知道每個模組之間究竟該如何溝通。

經歷了這次作業後，不僅讓我對於架構圖的重要性有了更加的認識，而且為了debug所花費的大量時間也開始讓我對於如何查看波行圖有了不少的sense，整體來說非常有意義!