**VLSI系統設計**

**-期末報告**

|  |  |  |
| --- | --- | --- |
| **隊名: ICC97** | | |
| **姓名** | **學號** | **授課教師** |
| 王柏智 | E24076611 | Prof. KJLEE |
| 王駿瀚 | E24076750 |
| 陳亮州 | E24076297 |
| 陳俊瑋 | E24076603 |
| 黃子青 | E24076938 |
| 蕭又瑜 | E24076459 |

1. **系統簡介**
   1. 指令集格式

本次使用的指令集為RISC-V指令集，其所用之指令分為以下數種不同類別：

1. I-type

Immediate data type，這種類別的指令會將立即值與一來源暫存器內所暫存之值進行算術運算或邏輯運算，並將其結果儲存至目標暫存器中。

1. R-type

Register data type，這種類別的指令會將兩來源暫存器中的值取出，對其進行算術運算或邏輯運算，最後將運算之結果儲存回目標暫存器。

1. Mem-type

進行Load、Store指令時使用，此指令會包含目標的基底 (base)，以及相對於基底的偏移值 (offset) ，及一暫存器位址，對於基底加上偏移值之位置進行暫存器值之取值或存值。

1. Branch-type

進行各種branch指令，其中會包含要跳至位置之offset值，以及要比較之兩暫存器之位址，依照兩暫存器內之值來進行是否跳值之依據。

* 1. 指令集格式欄位的名稱、長度、說明
     1. 指令集長度、格式
        1. I-type

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 指令 | [31:20] | [19:15] | [14:12] | [11:7] | [6:0] |
|  | Immediate | RS1 | FUNCT3 | RD | OP-IMM |
| ADDI |  |  | 000 |  | 0000000 |
| SLTI |  |  | 001 |  | 0000000 |
| ANDI |  |  | 100 |  | 0000000 |
| ORI |  |  | 101 |  | 0000000 |
| XORI |  |  | 010 |  | 0000000 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 指令 | [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] |
|  | FUNCT7 | shamt | RS1 | FUNCT3 | RD | OP-IMM |
| SLLI | 0000000 |  |  | 010 |  | 0000000 |
| SRLI | 0000000 |  |  | 011 |  | 0000000 |
| SRAI | 0100000 |  |  | 011 |  | 0000000 |

* + - 1. R-type

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 指令 | [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] |
|  | FUNCT7 | RS2 | RS1 | FUNCT3 | RD | OP-REG |
| ADD | 0000000 |  |  | 000 |  | 0000001 |
| SLT | 0000000 |  |  | 001 |  | 000001 |
| AND | 0000000 |  |  | 100 |  | 000001 |
| OR | 0000000 |  |  | 101 |  | 000001 |
| XOR | 0000000 |  |  | 110 |  | 000001 |
| SLL | 0000000 |  |  | 010 |  | 000001 |
| SRL | 0000000 |  |  | 011 |  | 000001 |
| SUB | 0100000 |  |  | 001 |  | 000001 |
| SRA | 0100000 |  |  | 011 |  | 000001 |

* + - 1. Mem-type

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 指令 | [31:20] | [19:15] | [14:12] | [11:7] | [6:0] |
|  | OFFSET | BASE | FUNCT3 | RD | OP-LW |
| LW |  |  | 000 |  | 1000000 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 指令 | [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] |
|  | OFFSET | SRC | BASE | FUNCT3 | OFFSET | OP-SW |
| SW |  |  |  | 000 |  | 1000001 |

* + - 1. Branch-type

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 指令 | [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] |
|  | OFFSET | RS2 | RS1 | FUNCT3 | OFFSET | OP-BRM |
| BEQ |  |  |  | 111 |  | 0000010 |
| BNE |  |  |  | 101 |  | 0000010 |
| BLT |  |  |  | 001 |  | 0000010 |
| BGE |  |  |  | 011 |  | 0000010 |

* + 1. 指令集說明
       1. I-type

|  |  |
| --- | --- |
| 指令 | 說明 |
| ADDI | 將立即值與RS1位址內之值進行加法運算後存入RD位址 |
| SLTI | 若RS1內之值小於立即值，將RD之值設為1，反之為0 |
| ANDI | 將立即值與RS1位址內之值進行AND運算後存入RD位址 |
| ORI | 將立即值與RS1位址內之值進行OR運算後存入RD位址 |
| XORI | 將立即值與RS1位址內之值進行XOR運算後存入RD位址 |
| SLLI | 將RS1內之值向左做邏輯位移存入RD |
| SRLI | 將RS1內之值向右做邏輯位移存入RD |
| SRAI | 將RS1內之值向右做算術位移存入RD |

* + - 1. R-type

|  |  |
| --- | --- |
| 指令 | 說明 |
| ADD | 將RS1內之值與RS2內之值進行加法運算後存入RD位址 |
| SLT | 若RS1內之值小於RS2內之值，將RD之值設為1，反之為0 |
| AND | 將RS1內之值與RS2內之值進行AND運算後存入RD位址 |
| OR | 將RS1內之值與RS2內之值進行OR運算後存入RD位址 |
| XOR | 將RS1內之值與RS2內之值進行XOR運算後存入RD位址 |
| SLL | 將RS1內之值向左做邏輯位移存入RD |
| SRLI | 將RS1內之值向右做邏輯位移存入RD |
| SUB | 將RS1內之值與RS2內之值進行減法運算後存入RD位址 |
| SRA | 將RS1內之值向右做算術位移存入RD |

* + - 1. Mem-type

|  |  |
| --- | --- |
| 指令 | 說明 |
| LW | 將記憶體中BASE (reg) 加上OFFSET位置之值存入RD |
| SW | 將SRC暫存器之值存放至BASE加上OFFSET之記憶體位置 |

* + - 1. Branch-type

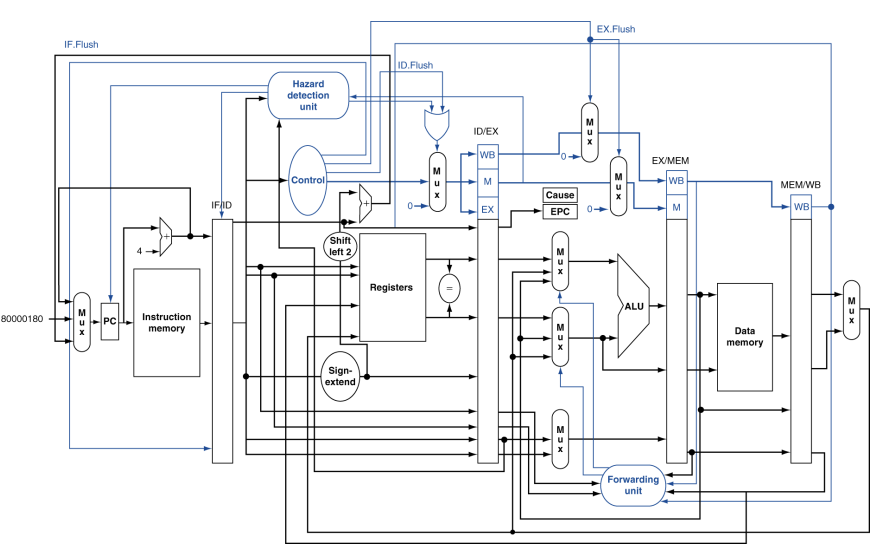
|  |  |
| --- | --- |
| 指令 | 說明 |
| BEQ | 當RS1與RS2內之值相等，跳轉至下一指令加OFFSET之位址 |
| BNE | 當RS1與RS2內之值不相等，跳轉至下一指令加OFFSET之位址 |
| BLT | 當RS1小於RS2內之值，跳轉至下一指令加OFFSET之位址 |
| BGE | 當RS1大於等於RS2內之值，跳轉至下一指令加OFFSET之位址 |

* 1. Branch指令與Jump指令的定址

我們所製作之架構使用Word addressing，每次Program counter計數的單位皆為一個Word，因此我們在設計Branch指令時也是依此設計，依照條件成立與否決定是否跳轉，若條件成立，則跳轉至下一指令位址加上Offset之位址，而我們此次並無特別設計Jump指令，此為BEQ R0 R0之pseudo instruction。

* 1. 架構
     1. 基本零件

我們以大二計算機組織時所學的MIPS五級管線化CPU作為我們參考的方向，並以下圖開始設計所需要的基本零件。



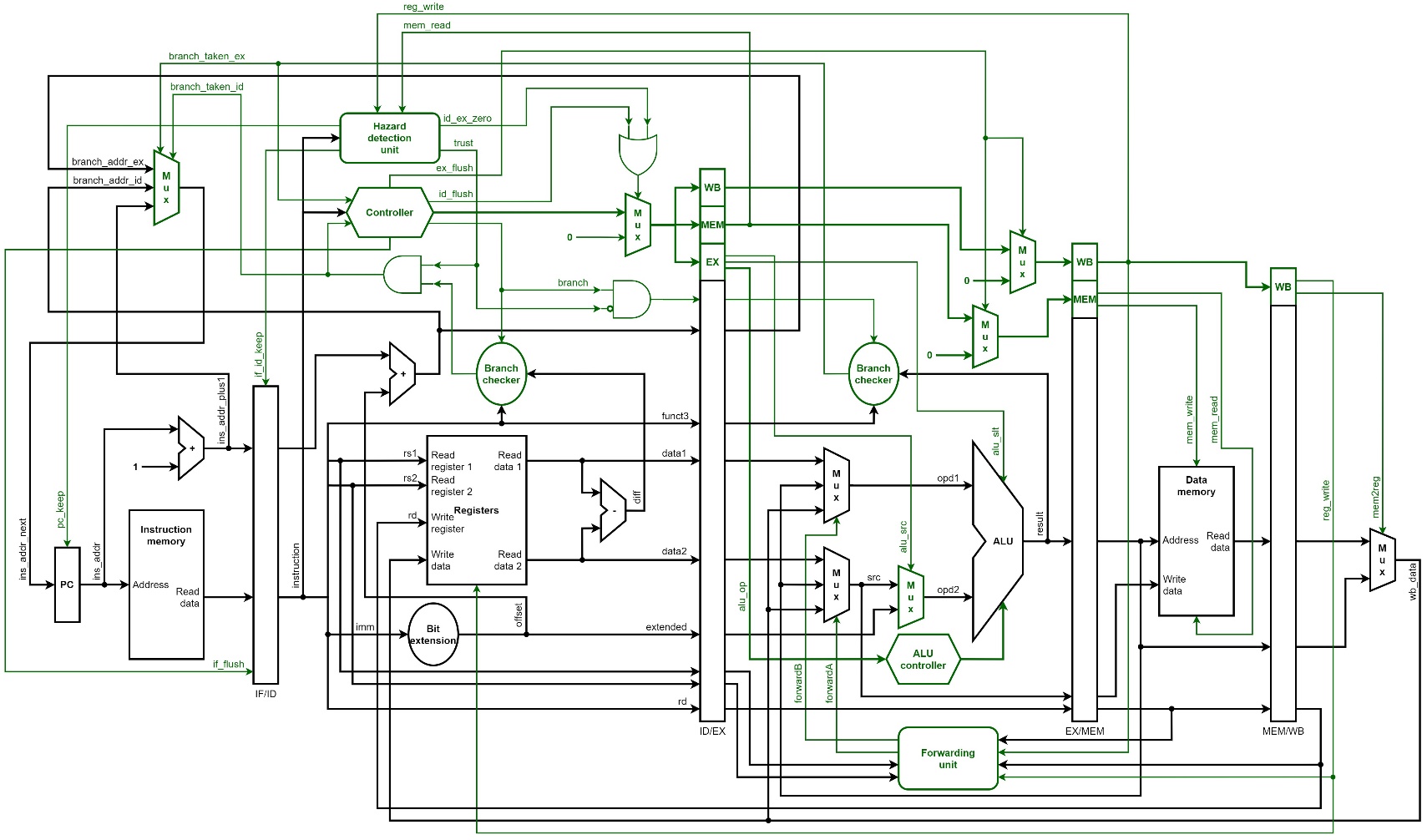
圖一、5 stage MIPS pipelined CPU

在cpu\_top\_module.v下的sub module在IF stage有：program counter (pc.v)；而在ID stage則有暫存器 (reg\_file.v)、用來檢branch結果的branch checker (branch\_checker.v)，與將immediate data或shift amount 或offset 作sign/unsign extension的 bit extension (bit\_extension.v)；到了EX stage則有ALU 的控制器 (alu\_controller.v) 與ALU本人 (alu.v)，而每一級與每一級中間都各有一個stage register (if\_id\_reg.v、id\_ex\_reg.v、ex\_mem\_reg.v、mem\_wb\_reg.v) 用來傳遞pipeline的資料或是控制訊；而在這5級pipeline之外的控制單元有：用來偵測危障的危障偵測單元 (hazard\_detection\_unit.v)、產生pipeline控制訊號的控制器 (controller.v)、與負責選擇forwarding data的forwarding unit (forwarding\_unit.v)。

雖然上圖中有指令記憶體與資料記憶體，但是我們認為這兩個零件並不屬於cpu之中，所以我們是將它們放在cpu之外 (ins\_mem.v、data\_mem.v) ，在testbench時再將其連接到cpu的I/O port來傳輸資料。

而圖中一些簡單的邏輯閘、加法器或是多工器，在撰寫程式時是以assign的方式作出來的，並沒有特別寫成一個module，到時候在合成時便會被自動合出適合的零件。

* + 1. 架構圖



圖二、本次實作出cpu之架構

1. **系統目前可執行之指令**

下表為我們設計之組合語言的格式

|  |
| --- |
| I type:  ADDI RD RS1 imm EX: ADDI R1 R0 1 把R0+1=0+1=1的結果寫入R1  SLTI RD RS1 imm EX: SLT1 R2 R1 87 如果R1<87，就把1寫入R2  ANDI RD RS1 imm EX: ANDI R3 R2 1 R2 跟1 AND完後寫入R3  ORI lRD RS1 imm EX: ORI R4 R3 2 R3 跟2 OR 完後寫入R4  XORI RD RS1 imm EX: XORI R5 R4 2 R4跟2 XOR完後寫入R5  SLLI RD RS1 imm EX: SLLI R6 R5 3 把R5邏輯左移3次後寫入R6  SRLI RD RS1 imm EX: SRLI R7 R6 1 把R6邏輯右移1次後寫入R7  SRAI lRD RS1 imm EX: SRAI R8 R6 1 把R6算術右移1次後寫入R8 |
| R type:  ADD RD RS1 RS2 EX: ADD R1 R1 R2 把R1+R2的結果放到R1  SLT lRD RS1 RS2 EX: SLT R2 R4 R1 如果R4小於R1就把R2設成1  AND RD RS1 RS2 EX: AND R3 R3 R2 把R3 AND R2的結果放到R3  OR lRD RS1 RS2 EX: OR R4 R3 R2 把R3 OR R2的結果放到R4  XOR RD RS1 RS2 EX: XOR R5 R1 R1 把R1 XOR R1的結果放到R5  SLL lRD RS1 RS2 EX: SLL R6 R1 R1 把R1邏輯左移R1次後寫入R6  SRL lRD RS1 RS2 EX: SRL R7 R1 R1 把R1邏輯右移R1次後寫入R7  SRA RD RS1 RS2 EX: SRA R9 R8 R1 把R8算術右移R1次後寫入R9  SUB RD RS1 RS2 EX: SUB R8 R1 R6 把R1 - R6的結果放到R8 |
| Branch type:  BEQ LABEL RS1 RS2 EX: BEQ LOOP R9 R0 如果R9 = R0就跳到LOOP  BGE LABEL RS1 RS2 EX: BGE LP R0 R0 如果R0大於等於R0就跳到LP  BNE LABEL RS1 RS2 EX: BNE LP R8 R0 如果R8不等於R0就跳到LP  BLT LABEL RS1 RS2 EX: BLT LP R0 R6 如果R0小於R6就跳到LP |
| Mem type:  LW RD BASE OFFSET EX: LW R1 R0 1 把記憶體位址(R0+1)的值放到R1  SW RS1 BASE OFFSET EX: SW R9 R0 1 把R9的值存回記憶體位址(R0+1) |
| 假指令:  J LABEL EX: 會被我們的assembler組譯成BEQ LABEL R0 R0  MV RD RS1 EX: 會被我們的assembler組譯成ADDI RD RS1 0  NOP EX: 會被我們的assembler組譯成ADDI R0 R0 0相當於不做事 |

1. **系統驗證方法與結果分析**
   1. 驗證方法

全部的指令都跑一次、氣泡排序法、含負數的氣泡排序、斐波納契數列，使用design vision確認無timing violation轉成Layout後有使用RC、LVS進行驗證

* + 1. Timing check:

Graphical user interface, application, Word

Description automatically generated

圖三、Timing check

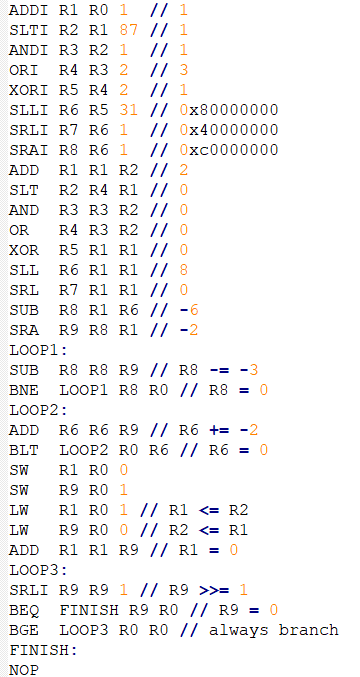
* + 1. LVS:

Graphical user interface, text

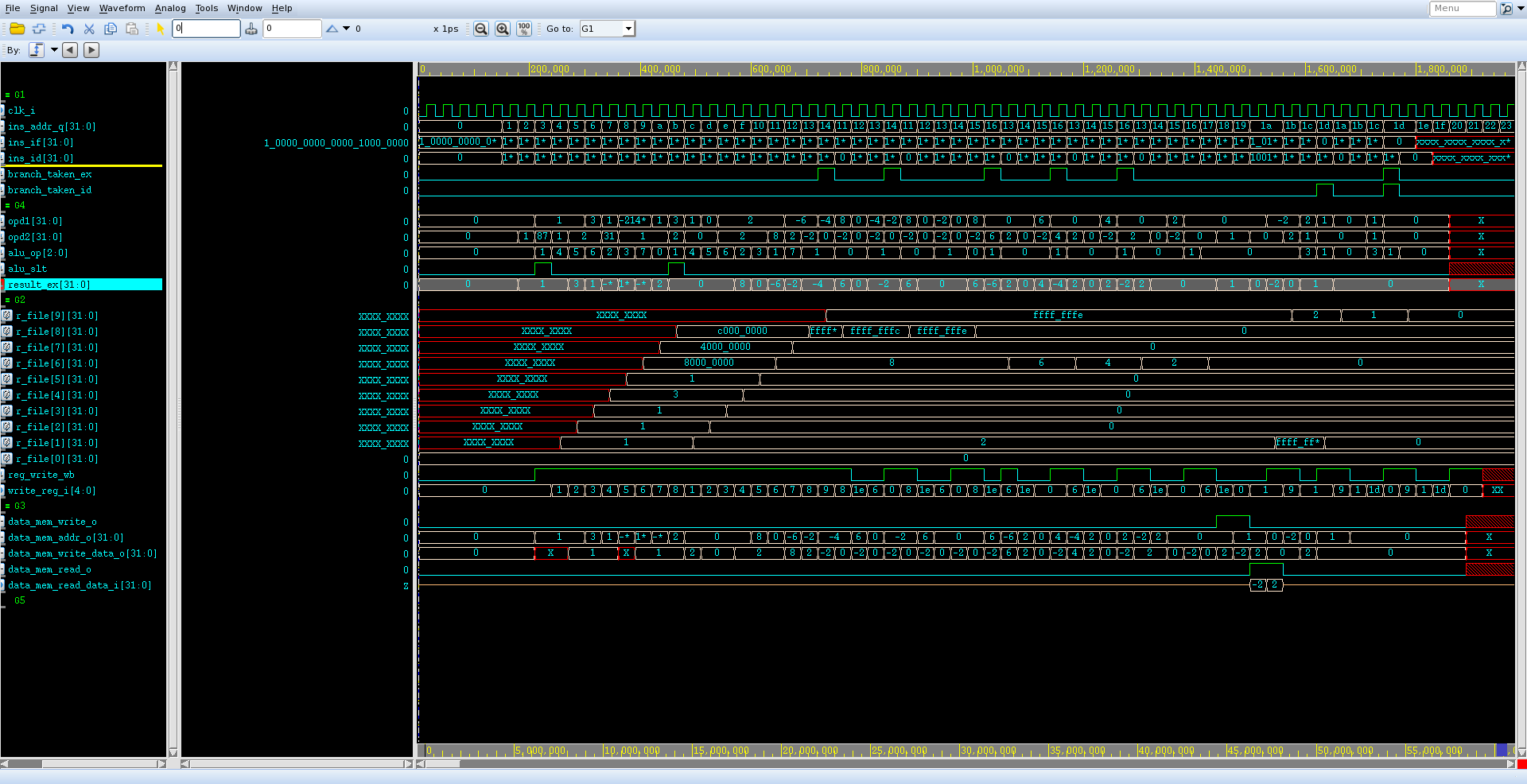
Description automatically generated

圖四、LVS驗證

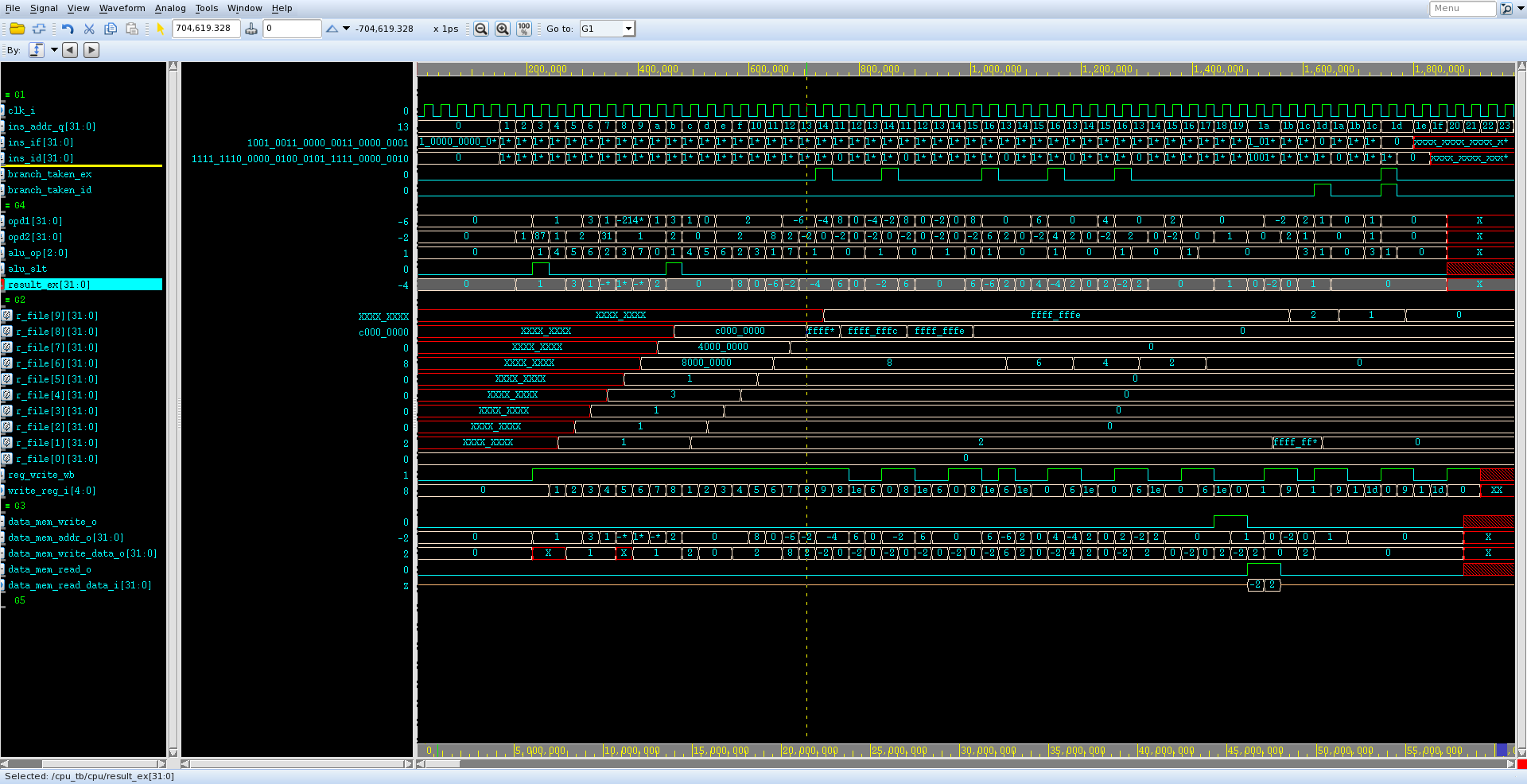
* 1. 結果分析
     1. “單一” 指令正確性
        1. all\_ins.s



圖五、執行所有指令之組合語言



圖六、程式在cpu上執行之結果



圖七、branch指令之執行結果

我們使用的是Risc-V的架構，R0永遠為0，pc一開始從0每個cycle往上+1，ins\_if根據ins\_addr\_q[31:0]讀取instruction memory的資料，ins\_addr\_q[31:0] = 0時ins\_if讀到ADDI R1 R0 1 的 machine code，ins\_addr\_q[31:0] = 1時ins\_if讀到SLTI R2 R1 87的 machine code，而ADDI R1 R0 1進到ID stage。

ins\_addr\_q[31:0] = 2時ins\_if讀到ANDI R3 R2 1 的 machine code，而SLTI R2 R1 87進到ID stage，而ADDI R1 R0 1進到EX stage…………如此pipeline下去。

可從波形看出當ins\_addr\_q[31:0]顯示為2時，ADDI R1 R0 1指令已經執行到ex stage，result\_ex = 0+1 = 1，此值將在ins\_addr\_q[31:0]顯示為4時執行wb stage，將1放到R1。

all\_ins.s的第1行為SLTI R2 R1 87，可從波形看出當ins\_addr\_q[31:0]顯示為3時，指令已經執行到ex stage，因為R1 = 1被forwarding過來，R1 = 1會小於立即數87所以result\_ex = 1，此值將在ins\_addr\_q[31:0]顯示為5時執行wb stage，將1寫入R2。

all\_ins.s的第2行為ANDI R3 R2 1，可從波形看出當ins\_addr\_q[31:0]顯示為4時，指令已經執行到ex stage，因為R2 = 1被forwarding過來，R2 = 1與立即數1做AND邏輯運算所以result\_ex = 1，此值將在ins\_addr\_q[31:0]顯示為6時執行wb stage，將1寫入R3。

all\_ins.s的第3行為ORI R4 R3 2，可從波形看出當ins\_addr\_q[31:0]顯示為5時，指令已經執行到ex stage，因為R3 = 1被forwarding過來，R3 = 1與立即數2做OR邏輯運算所以result\_ex = 3，此值將在ins\_addr\_q[31:0]顯示為7時執行wb stage，將3寫入R4。

all\_ins.s的第4行為XORI R5 R4 2，可從波形看出當ins\_addr\_q[31:0]顯示為6時，指令已經執行到ex stage，因為R4 = 3被forwarding過來，R4 = 3與立即數2做XOR邏輯運算所以result\_ex = 1，此值將在ins\_addr\_q[31:0]顯示為8時執行wb stage，將1寫入R5。

all\_ins.s的第5行為SLLI R6 R5 31，可從波形看出當ins\_addr\_q[31:0]顯示為7時，指令已經執行到ex stage，因為R5 = 1被forwarding過來，R5 = 1與立即數31做SLL運算所以result\_ex = 0x8000\_0000，此值將在ins\_addr\_q[31:0]顯示為9時執行wb stage，將0x8000\_0000寫入R6。

all\_ins.s的第6行為SRLI R7 R6 1，可從波形看出當ins\_addr\_q[31:0]顯示為8時，指令已經執行到ex stage，因為R6被forwarding過來，R6 = 0x8000\_0000與立即數1做SRL運算所以result\_ex = 0x4000\_0000，此值將在ins\_addr\_q[31:0]顯示為10時執行wb stage，將0x4000\_0000寫入R7。

all\_ins.s的第7行為SRAI R8 R6 1，可從波形看出當ins\_addr\_q[31:0]顯示為9時，指令已經執行到ex stage，R6 = 0x8000\_0000與立即數1做SRA運算所以result\_ex = 0xc000\_0000，此值將在ins\_addr\_q[31:0]顯示為11時執行wb stage，將0xc000\_0000寫入R8。

all\_ins.s的第8行為ADD R1 R1 R2，可從波形看出當ins\_addr\_q[31:0]顯示為a時，指令已經執行到ex stage，R1 = 1與R2 = 1做ADD運算所以result\_ex = 2，此值將在ins\_addr\_q[31:0]顯示為c時執行wb stage，將2寫入R1。

all\_ins.s的第9行為SLT R2 R4 R1，可從波形看出當ins\_addr\_q[31:0]顯示為b時，指令已經執行到ex stage，R1被forwarding過來，R4 = 3大於R1 = 1做SLT運算所以result\_ex = 0，此值將在ins\_addr\_q[31:0]顯示為d時執行wb stage，將0寫入R2。

all\_ins.s的第10行為AND R3 R3 R2，可從波形看出當ins\_addr\_q[31:0]顯示為c時，指令已經執行到ex stage，R2被forwarding過來，R3 = 1與R2 = 0做AND邏輯運算所以result\_ex = 0，此值將在ins\_addr\_q[31:0]顯示為e時執行wb stage，將0寫入R3。

all\_ins.s的第11行為OR R4 R3 R2，可從波形看出當ins\_addr\_q[31:0]顯示為d時，指令已經執行到ex stage，R3被forwarding過來，R3 = 0與R2 = 0做OR邏輯運算所以result\_ex = 0，此值將在ins\_addr\_q[31:0]顯示為f時執行wb stage，將0寫入R4。

all\_ins.s的第12行為XOR R5 R1 R1，可從波形看出當ins\_addr\_q[31:0]顯示為e時，指令已經執行到ex stage，R1 = 2與R1 = 2做XOR邏輯運算所以result\_ex = 0，此值將在ins\_addr\_q[31:0]顯示為10時執行wb stage，將0寫入R5。

all\_ins.s的第13行為SLL R6 R1 R1，可從波形看出當ins\_addr\_q[31:0]顯示為f時，指令已經執行到ex stage，R1 = 2與R1 = 2做SLL運算所以result\_ex = 8，此值將在ins\_addr\_q[31:0]顯示為11時執行wb stage，將8寫入R6。

all\_ins.s的第14行為SRL R7 R1 R1，可從波形看出當ins\_addr\_q[31:0]顯示為10時，指令已經執行到ex stage，R1 = 2與R1 = 2做SRL運算所以result\_ex = 0，此值將在ins\_addr\_q[31:0]顯示為12時執行wb stage，將0寫入R7。

all\_ins.s的第15行為SUB R8 R1 R6，可從波形看出當ins\_addr\_q[31:0]顯示為11時，指令已經執行到ex stage，R6被forwarding過來，R1 = 2與R6 = 8做SUB運算所以result\_ex = -6，此值將在ins\_addr\_q[31:0]顯示為13時執行wb stage，將0xffff\_fffa寫入R8。

all\_ins.s的第16行為SRA R9 R8 R1，可從波形看出當ins\_addr\_q[31:0]顯示為12時，指令已經執行到ex stage，R8被forwarding過來，R8 = 0xffff\_fffa與R1 = 2做SRA運算所以result\_ex = -2，此值將在ins\_addr\_q[31:0]顯示為14時執行wb stage，將0xffff\_fffe寫入R9。

all\_ins.s的第17行為LOOP1:SUB R8 R8 R9，可從波形看出當ins\_addr\_q[31:0]顯示為第一個13時，指令已經執行到ex stage，R8與R9被forwarding過來，R8 = -6與R9 = -2做SUB運算所以result\_ex = -4，此值將在ins\_addr\_q[31:0]顯示為15時執行wb stage，將0xffff\_fffc寫入R8。

all\_ins.s的第18行為BNE LOOP1 R8 R0，可從波形看出當ins\_addr\_q[31:0]顯示為第一個13時，ins\_id為BNE LOOP1 R8 R0，ins\_if為ADD R6 R6 R9，但此時R8的值還在ex stage計算，所以要等到ins\_addr\_q[31:0]顯示為第一個14時，R8 = -4與R0做BNE運算所以branch\_taken\_ex = 1，下一個指令將會回到LOOP1，而ADD R6 R6 R9與BLT LOOP2 R0 R6將會被洗掉。

當ins\_addr\_q[31:0]顯示為第三個14時，R8 = 0與R0做BNE運算所以branch\_taken\_ex = 0。

all\_ins.s的第20行為BEQ LOOP2 R0 R6，可從波形看出當ins\_addr\_q[31:0]顯示為第一個15時，ins\_id為BEQ LOOP2 R0 R6，ins\_if為SW R1 R0 0，但此時R6的值還在ex stage計算，所以要等到ins\_addr\_q[31:0]顯示為第一個16時，R6 = 6與R0做BEQ運算所以branch\_taken\_ex = 1，下一個指令將會回到LOOP2，而SW R1 R0 0與SW R9 R0 1將會被洗掉。

當ins\_addr\_q[31:0]顯示為第四個16時，R6 = 0與R0做BEQ運算所以branch\_taken\_ex = 0。

all\_ins.s的第21行為SW R1 R0 0，可從波形看出當ins\_addr\_q[31:0]顯示為17時，指令已經執行到ex stage，R0與立即值0相加所以result\_ex = 0，此值將在ins\_addr\_q[31:0]顯示為18時執行mem stage，dara\_mem\_write\_o = 1、data\_mem\_addr\_o = 0以及data\_mem\_write\_data = R1 = 2。

all\_ins.s的第22行為SW R9 R0 1，可從波形看出當ins\_addr\_q[31:0]顯示為18時，指令已經執行到ex stage，R0與立即值1相加所以result\_ex = 1，此值將在ins\_addr\_q[31:0]顯示為19時執行mem stage，dara\_mem\_write\_o = 1、data\_mem\_addr\_o = 1以及data\_mem\_write\_data = R9 = -2。

all\_ins.s的第23行為LW R1 R0 1，可從波形看出當ins\_addr\_q[31:0]顯示為19時，指令已經執行到ex stage，R0與立即值1相加所以result\_ex = 1，此值將在ins\_addr\_q[31:0]顯示為1a時執行mem stage，dara\_mem\_read\_o = 1、data\_mem\_addr\_o = 1以及data\_mem\_read\_data = -2，在ins\_addr\_q[31:0]顯示為1b時執行wb stage，將0xffff\_fffe寫入R1。

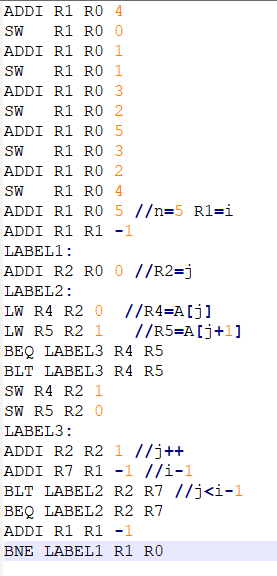
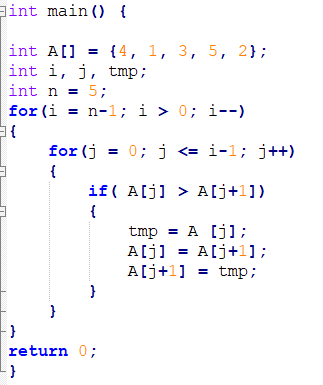
all\_ins.s的第24行為LW R9 R0 0，可從波形看出當ins\_addr\_q[31:0]顯示為1a時，指令已經執行到ex stage，R0與立即值0相加所以result\_ex = 0，此值將在ins\_addr\_q[31:0]顯示為1a時執行mem stage，dara\_mem\_read\_o = 1、data\_mem\_addr\_o = 1以及data\_mem\_read\_data = 2，在ins\_addr\_q[31:0]顯示為1b時執行wb stage，將2寫入R1。

all\_ins.s的第25行為ADD R1 R1 R9，但會發生Load use的危障，所以ins\_addr\_q[31:0]維持在1a兩個cycle。

all\_ins.s的第28行為BGE LOOP3 R0 R0，可從波形看出當ins\_addr\_q[31:0]顯示為第一個1d時，ins\_id為BGE LOOP3 R0 R0， R0與R0做BGE運算所以branch\_taken\_id = 1，下一個指令將會回到LOOP3 。

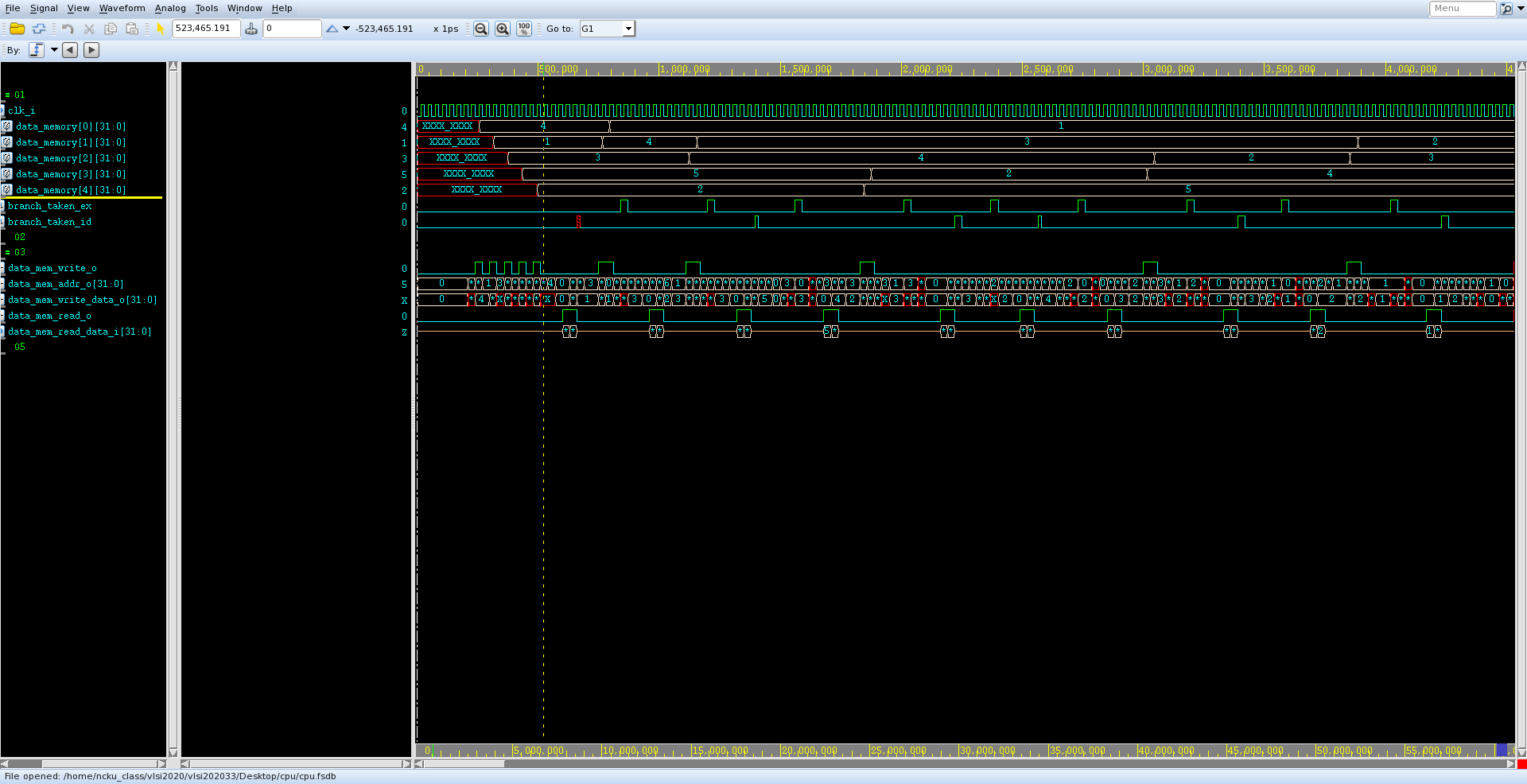
all\_ins.s的第27行為BEQ FINISH R9 R0，可從波形看出當ins\_addr\_q[31:0]顯示為第二個1d時，ins\_ex為BEQ FINISH R9 R0， R9 = 0與R0做BEQ運算所以branch\_taken\_ex = 1，下一個指令將會到FINISH。

* + 1. 程式正確性
       1. 排序



圖八、氣泡排序測試

(左圖為氣泡排序的c程式，右圖為我們compile過後的組合語言)



圖九、氣泡排序測試前的波形圖

一開始程式會透過ADDI跟SW指令分別把4、1、3、5、2 寫到主記憶體中的data\_memory[0]、data\_memory[1]、data\_memory[2]、data\_memory[3]、data\_memory[4]，再把R1當成i設成array的total number - 1 後進入兩層for loop ，再把R2當作j先設成0，每次進到LABEL1都會將R2歸零，每次離開內迴圈都會將R1的值減1，只有在R1 等於0時會離開外迴圈否則又進到內迴圈。

R1 = 4時:

4因為大於1，所以4、1、3、5、2變成1、4、3、5、2

4因為大於3，所以1、4、3、5、2變成1、3、4、5、2

5因為大於4，所以1、3、4、5、2維持1、3、4、5、2

5因為大於2，所以1、3、4、5、2變成1、3、4、2、5

R1 = 3時:

3因為大於1，所以1、3、4、2、5維持1、3、4、2、5

4因為大於3，所以1、3、4、2、5維持1、3、4、2、5

4因為大於2，所以1、3、4、2、5變成1、3、2、4、5

R1 = 2時:

3因為大於1，所以1、3、2、4、5維持1、3、2、4、5

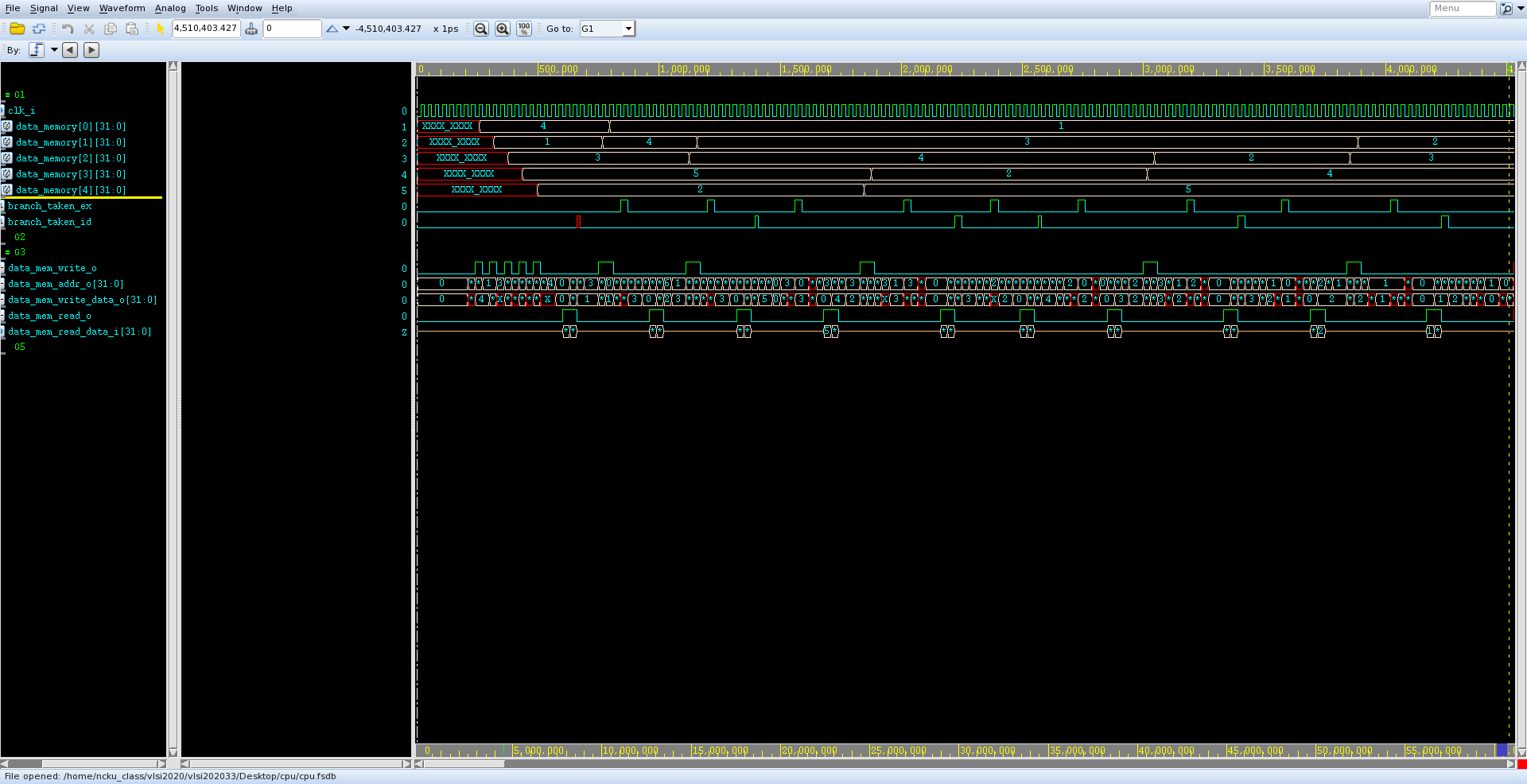
3因為大於2，所以1、3、2、4、5維持1、2、3、4、5

R1 = 1時:

3因為大於1，所以1、2、3、4、5維持1、2、3、4、5

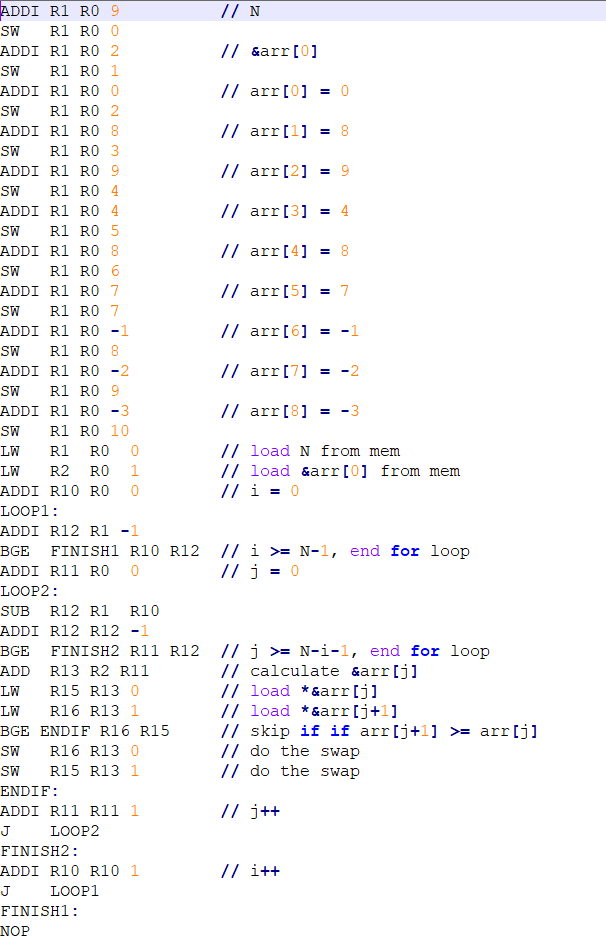
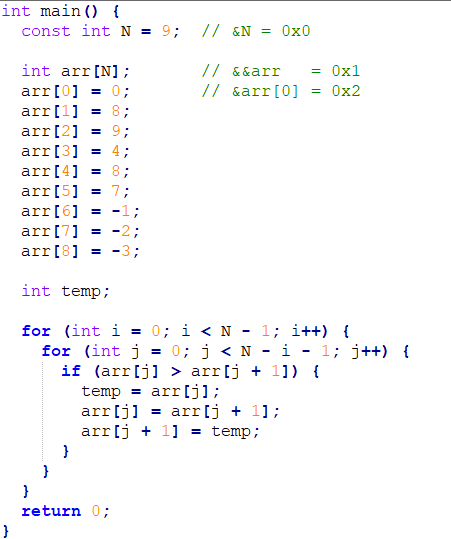
R1 = 0時跳出迴圈程式結束。

觀察下圖，data\_memory[0]、data\_memory[1]、data\_memory[2]、data\_memory[3]、data\_memory[4]最後存的值為1、2、3、4、5。



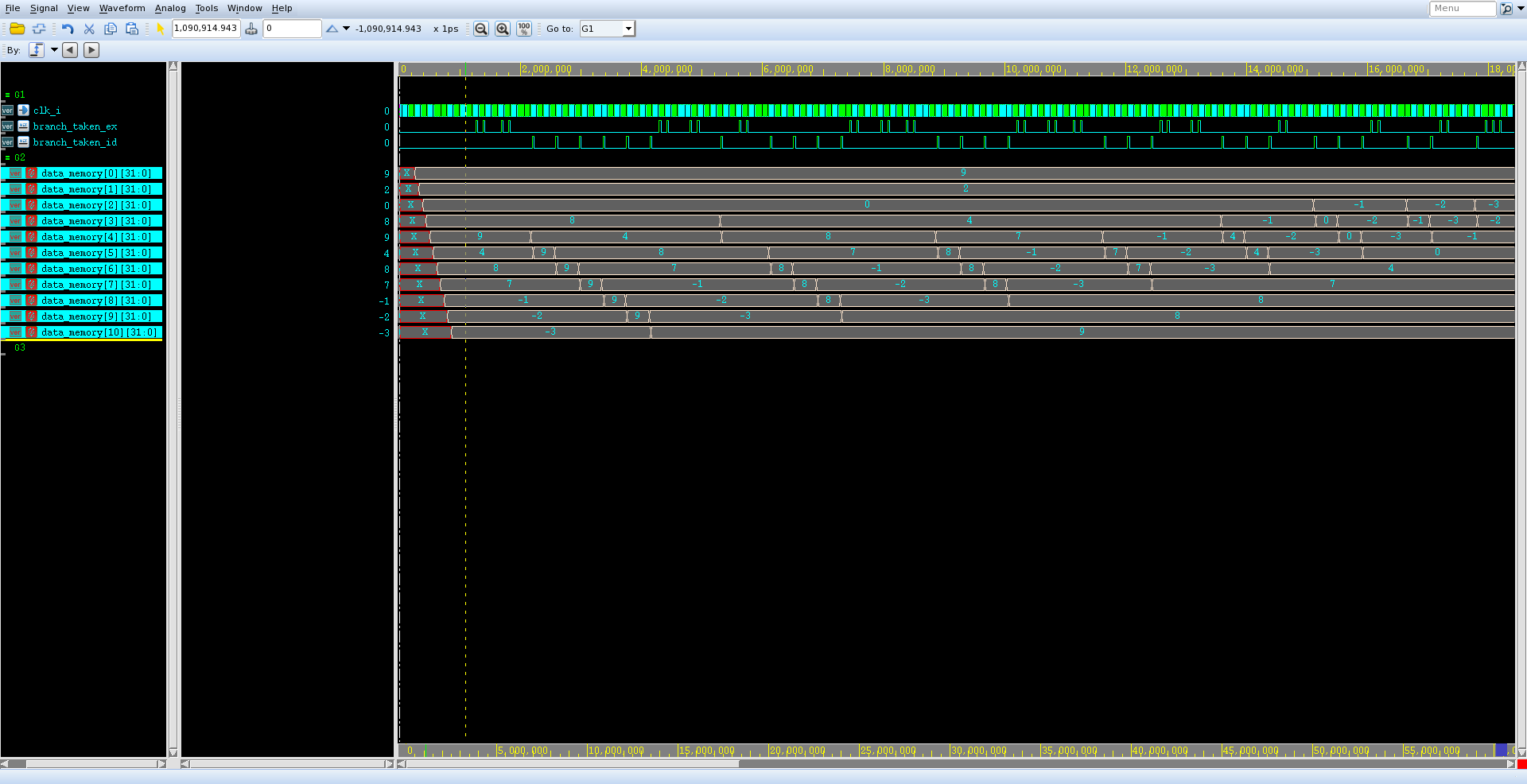
圖十、氣泡排序測試後的波形圖

接下來測試數列有含負數的氣泡排序程式



圖十一、含負數的氣泡排序測試

(左圖為氣泡排序的c程式，右圖為我們compile過後的組合語言)



圖十二、含負數的氣泡排序測試模擬結果

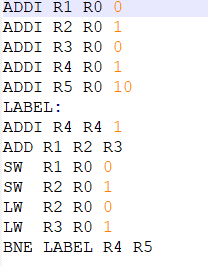
可從上圖看出，一開始程式透過ADDI跟SW把9寫到data\_memory[0]，代表總共有9個數字要排序，接著把2寫到data\_memory[1]，代表要排序的資料會從data\_memory[2]開始擺放，即data\_memory[2]~ data\_memory[10]會存最後排序的結果。

再分別把0、8、9、4、8、7、-1、-2、-3 寫到主記憶體中的data\_memory[2]、data\_memory[3]、data\_memory[4]、data\_memory[5]、data\_memory[6] 、data\_memory[7]、data\_memory[8]、data\_memory[9]、data\_memory[10]。

之後進行氣泡排序，最後可看出主記憶中data\_memory[2]~ data\_memory[10]存的值為: -3、-2、-1、0、4、7、8、8、9。

* + - 1. 斐波納契數

以下為我們斐波納契數的組合語言與測試模擬結果



圖十三、斐波納契數的組合語言



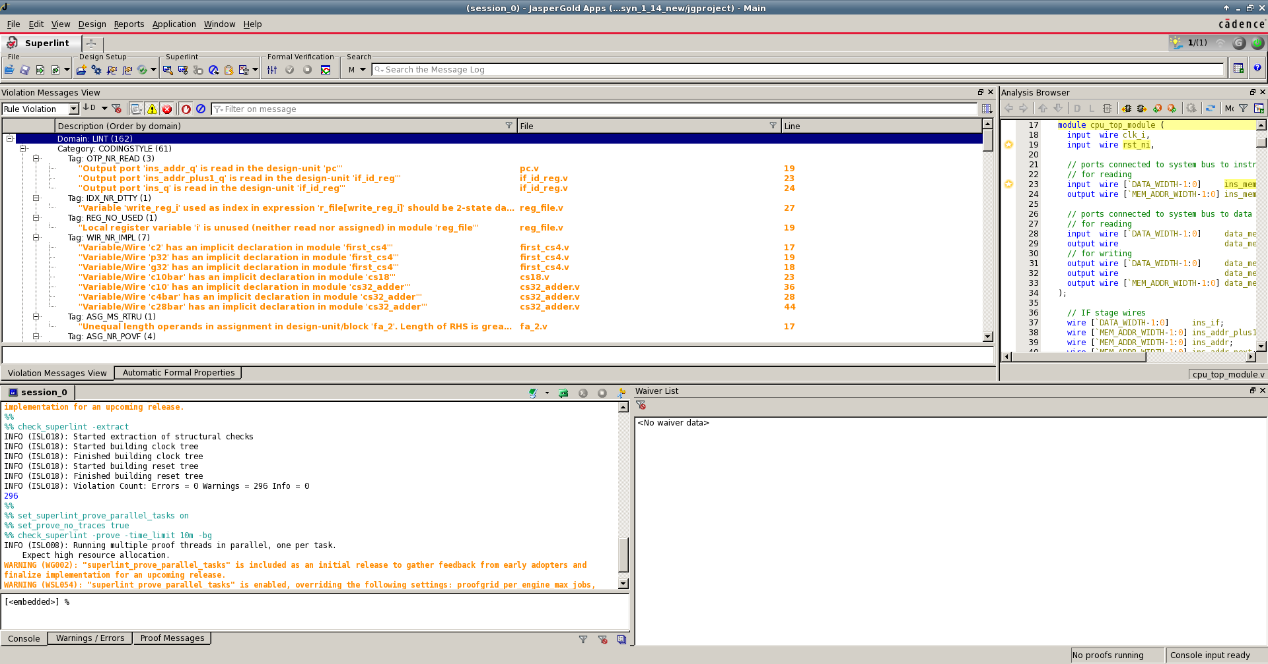
圖十四、斐波納契數測試模擬結果

F0 = 0 F1 = 1 F2 = 1 F3 = 2 F4 = 3 F5 = 5 F6 = 8 F7 = 13 F8 = 21 F9 = 34 F10 = 55

R1為Fn、R2為Fn-1，初始值為1、R3為Fn-2，初始值為0，R4為n，初始值為1，剛開始進LABEL後會先+1，所以n是從2開始，F2 = F1+F0 = 1+0 = 1，即R1 = R2+R3 = 1，把R1跟R2的值分別store 到主記憶體中data\_memomry[0]、data\_memomry[1]，再把data\_memomry[0]、data\_memomry[1]的值load到R2、R3，F3 = F2+F1 = 1+1 = 2，即R1 = R2+R3 = 2，把R1跟R2的值分別store 到data\_memomry[0]、data\_memomry[1]，再把data\_memomry[0]、data\_memomry[1]的值load到R2、R3，如此反覆運算直到n = 10，跳出迴圈程式結束，可從波形看出，data\_memory[0]儲存了F2到F10的值：

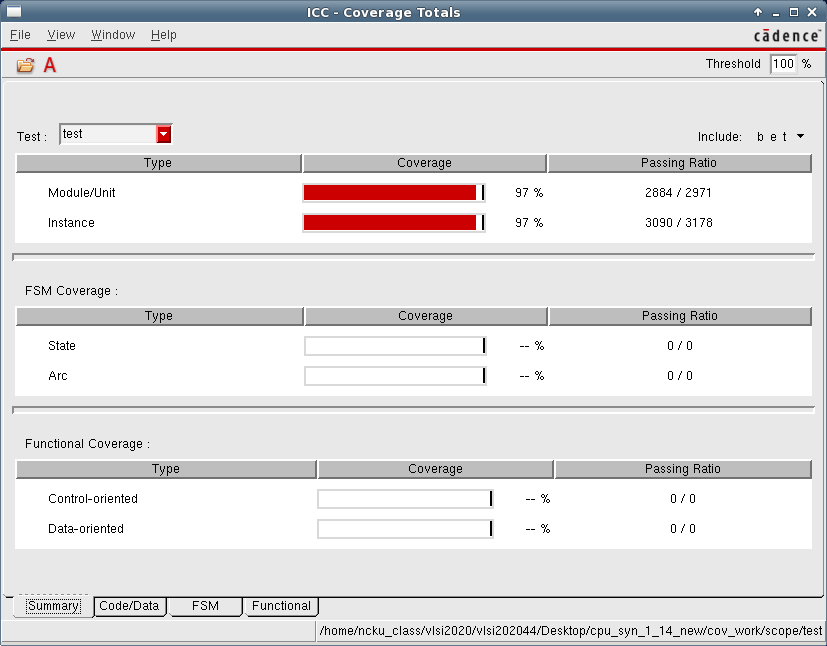
F2 = 1 F3 = 2 F4 = 3 F5 = 5 F6 = 8 F7 = 13 F8 = 21 F9 = 34 F10 = 55

1. **SuperLint與ICC檢查結果**
   1. SuperLint檢查結果



圖十五、檢查結果：Error Count = 0

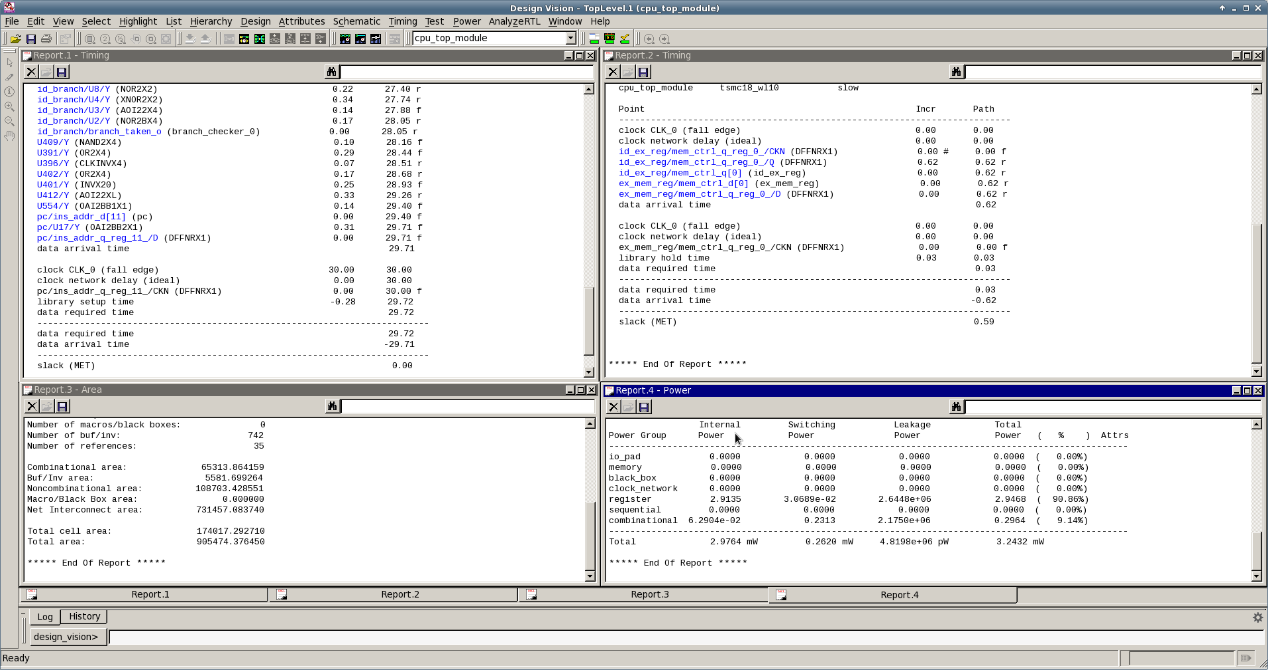
* 1. ICC檢查結果



圖十六、檢查結果：Total Coverage = 97%

1. **效能分析**

Design Vision Results



圖十七、design vision分析

* 1. 速度

Clock period: 30 ns

Setup time: 29.72 ns

Hold time: 0.83 ns

Slack (max/min): 0.0 / 0.59

* 1. 面積

Total Cell Area: 174017.29 μm2

Total Area: 905474.37 μm­2

* 1. 功耗

Total Power: 3.2432 mW

1. **管線化**
   1. 說明

我們的管線化分為5個階段，包含IF(instruction fetch)、ID(instruction decode)、EX(execute)、MEM(memory)、WB(write back)

1. 從記憶體中讀取第一個指令。
2. 解碼第一個指令

記憶體儲器中讀取第二個指令

1. 執行第一個指令

解碼第二個指令

從記憶體中讀取第三個指令

1. 將第一個指令的運算結果存入記憶體或到記憶體取資料

執行第二個指令

解碼第三個指令

從記憶體中讀取第四個指令

1. 把第一個指令的結果寫回暫存器

將第二個指令的運算結果存入記憶體或到記憶體取資料

執行第三個指令

解碼第四個指令

從記憶體中讀取第五個指令

1. 第一個指令執行完畢

把第二個指令的結果寫回暫存器

將第三個指令的運算結果存入記憶體或到記憶體取資料

執行第四個指令

解碼第五個指令

1. 從記憶體中讀取第六個指令

以此類推，不斷讀取新的指令

* 1. 階段的劃分

IF級: PC會從前面的多工器拿到下一個指令的位址，接著進入instruction memory抓取下一個指令，並將指令送入IF/ID 暫存器。

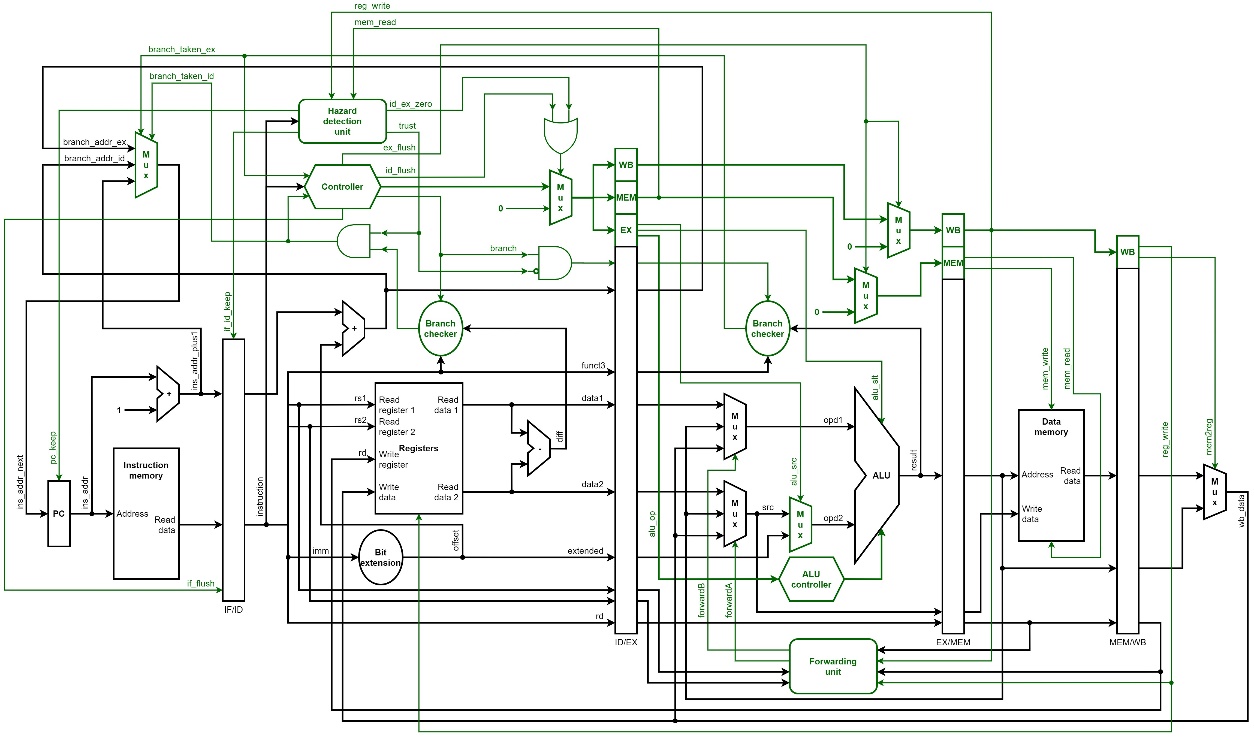
ID級: 解碼從IF/ID 暫存器中取得的指令，並依據不同的指令將訊號線分別傳到不同的地方。如若是R-type的指令中的ADD、SLT、SUB指令，就會把指令中[24:20][19:15]的bit，也就是RS2、RS1的訊號線分別送到register區的read\_register\_1及read\_register\_2。而指令中[11:7]的bit，也就是RD的訊號線則會送到register區的write\_register。若是branch-type，會把[31:25]的offset部分與PC+1送到一個多工器去決定下一個要執行的指令的位址。最後會把下一級所需要用到的資訊由Register區，送到ID/EX暫存器，以供下一級的運算。

EX級: 執行進入ALU的各種運算，資料在進入ALU前會先經過多工器，由forwarding controller選擇要進入ALU的訊號，並由ALU controller決定執行何種運算。而運算完的結果(位址或資料)放到EX/MEM暫存器。

MEM級: 從EX/MEM暫存器取得運算完address或data，並存入data memory。或是根據讀到的address把data取出來，並放入MEM/WB暫存器。

WB級:把上一級取得的data寫回register memory。

* 1. 管線化的危障
     1. Forwarding (data hazard)



MUX\_B

forwardB

result

MUX\_A

WB\_rd

WB\_write

MEM\_rd

MEM\_write

rs1

rs2

forwardA

圖十八、forwarding之說明

1. EX/MEM forward

如果MEM\_write為1(前一個指令要寫入register file)、MEM\_rd不為0(寫入的位置不為R0)，且rs1等於MEM\_rd (前一指令寫入的位置為後一指令讀取的位置)，則將result 拉回MUX\_A。並將forwardA設為2'b10。

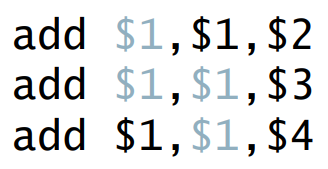
如果MEM\_write為1(前一個指令要寫入register file)、MEM\_rd不為0(寫入的位置不為R0)，且rs2等於MEM\_rd (前一指令寫入的位置為後一指令讀取的位置)，則將result 拉回MUX\_B。並將forwardB設為2'b10。

1. MEM/WB forward

如果WB\_write為1(前一個指令要寫入register file)、WB\_rd不為0(寫入的位置不為R0)，且rs1等於WB\_rd (前一指令寫入的位置為後一指令讀取的位置)，則將result 拉回MUX\_A。並將forwardA設為2'b01。

如果WB\_write為1(前一個指令要寫入register file)、WB\_rd不為0(寫入的位置不為R0)，且rs2等於WB\_rd (前一指令寫入的位置為後一指令讀取的位置)，則將result 拉回MUX\_B。並將forwardB設為2'b01。

* + 1. Double data hazard



MEM/WB forward

EX/MEM forward

EX/MEM forward

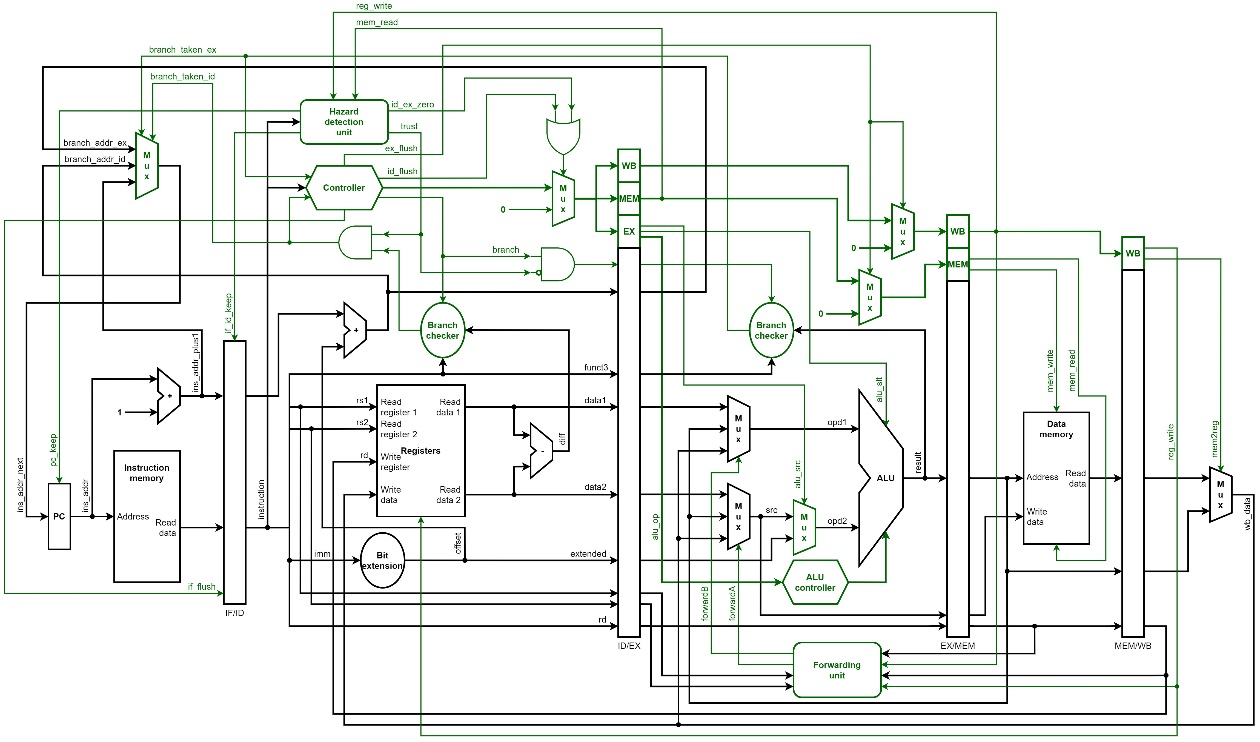
如上方的例子，由於$1在指令1及指令2中皆為寫入的位置，而指令3若要進行forwarding，是要forward指令2的result，並非指令1的result。因此剛剛提到的MEM/WB forward的進行，前提是沒有發生EX/MEM forward，換句話說，上方的例子，指令3要進行的是EX/MEM forward，而指令2也是進行EX/MEM forward。

* + 1. Load-use hazard

id\_ex\_zero

if\_id\_keep

pc\_keep



EX\_rd

EX\_mem\_read

rs1

rs2

圖十九、load-use hazard之說渡明

如果EX\_mem\_read為1(前一指令為LW)，且rs1或rs2等於EX\_rd(前一指令LW讀取後寫入的位置為後一指令讀取的位置)，即發生load-use hazard，必須stall一個bubble，此時:

pc\_keep設為1(program counter保持原值一個cycle)

if\_id\_keep設為1(此register保持原值一個cycle)

id\_ex\_zero設為1(清空ID/EX reg裡的值，製造一個bubble)

1. **特殊設計**
   1. ALU

ALU中的加法器，是採用低功耗且高效能的32位元進為跳躍加法器，共切割成4-18-6-4位元的四個加法器，除了LSB為一般的全加器外，剩餘的部分都會產生generate signal和propagate signal，利用這兩個輸出去減少carry在加法器中經過的gate數量，可以加速延遲時間，而我們可以看到在不同的block間，有一些carry是反相的，原因是在此電路裡頭用到許多AOI (And-Or-Inverter)以及OAI (Or-And-Inverter)，在CMOS製成裡，這兩種邏輯夾可以減少靜態功率，讓整個加法器的效能可以變得更好。

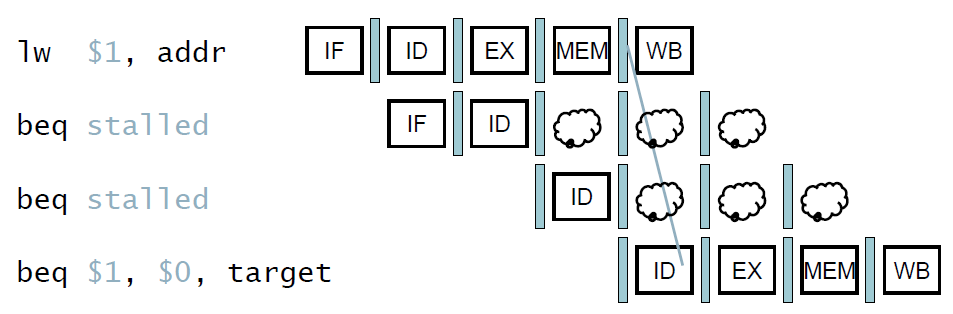
A picture containing logo

Description automatically generated

圖二十、block diagram of static low-power, high-performance 32-bit carry skip adder

利用此種加法器與a + b所合成出來的adder相比，速度上的確有顯著的提升，但是面積並沒有大上多少，功耗也比較低，但如果是與carry lookahead adder相比，由於後者面積太大且速度也差不多所以採用此種架構。

* 1. Two stage branch outcome determination



圖二十一、load and branch

我們所參考之MIPS的五級架構若是遇到mem access的指後接著馬上是branch的話，會需要如上圖一樣讓branch指令等兩個clk cycle，這是因為在id級就作branch的判斷，而id級並沒有forwarding的機制，一定得等到lw的結果跑到wb級，寫回register後判斷的結果才會是正確的。

因此我們這裡用了一個比較特殊的設計，就是讓branch指令不只能夠在id級做，在ex級也可以做，這樣的好處是由於forwarding的機制都是做在ex級，像是假設branch的前一個指令是addi，當addi跑到mem級時，運算結果個以forward回ex級，讓現在在ex級的branch指令得到正確的source，而得到正確的結果；又或是當branch的前一個指令是lw時，這個時候我們的pipeline會先因為load-use hazard讓branch指令在if級stall 一個cycle，下一個cycle時lw跑到mem級拿資料，這時在id級的branch並不能正確判斷branch結果，可是當下一個cycle時，lw跑到wb級；branch跑到ex級，這時wb的資料就以被forward到ex級，所以branch的結果是個以相信的。

實作這種設計的方法是讓hazard detection unit在decode時幫忙判斷id級的branch結果是不是可以相信的，而這是依據這時在id級的branch指令的rs1與rs2是不是ex級或是mem級的rd，若有一個相同，則代表這時從register讀出來的值還不是正確的，所以不能相信，必需等下一個clk cycle branch指令跑到ex級時，有forward回來的正確值，計算出的branch結果才會是可以相信的。

這樣會再延伸出一個問題，就是當兩個連續的branch指令來時，第一個由於在id級不可信，所以到了ex級才做判斷，而這時下一個branch指令已進到id級，若是這時ex級的判斷為branch taken，且id級也判斷為可信的branch taken，那這時到底要跳到哪一個address？這個問題可以用以下的真值表來解釋：

|  |  |  |
| --- | --- | --- |
| branch\_taken\_id | branch\_taken\_ex | branch\_address |
| 0 | 0 | ins\_addr\_plus1 |
| 0 | 1 | branch\_addrss\_ex |
| 1 | 0 | branch\_addrss\_ed |
| 1 | 1 | branch\_addrss\_ex |

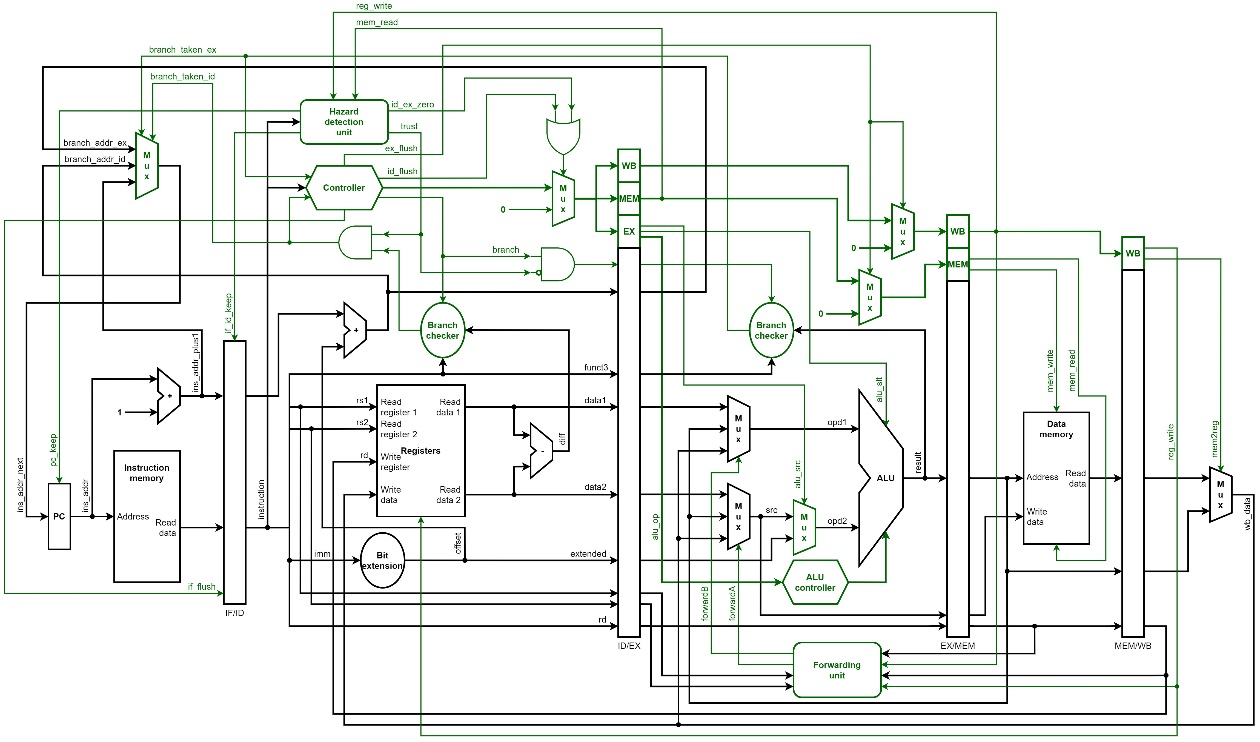
上表為id級的branch taken與ex級的branch taken可能發生的排列組合，由於ex級的branch指令是比較早進來的，所以當兩級的branch都判斷為要跳時，必需以ex級的指令為優先，跳到ex級branch應該跳到的位址。

這樣的設計方法若是可以搭配branch prediction，可望再提升效能，減少不必要的stall cycle。

1. **問題與討論**
   1. clk\_generator

在設計pipeline的時候我們參考了老師投影片中two phase clk的設計，所以參照講義中的code寫了一份我們的clk\_generator.v，結果發現老師的code是沒辦法合成的，所以我們改用有限狀態機來實作：用兩個state，01與10，只要clk\_master來的時候就換到另一個state，來達到兩個phase的效果，這樣的作法在pre-sim是沒有問題的，可是合成後的post-sim與預期的不一樣，我們問了助教後才發現，這樣的作法沒有考慮到flip-flop的delay，導致整個系統變成是由兩個不一樣的clk驅動，而造成time violation。

* 1. Use-branch/Load-branch hazard



trust

MEM\_writeMEM\_rd

EX\_write

EX\_rd

rs1

rs2

圖二十三、use-branch/load-branch說明

如果EX\_write為1 (前一個指令要寫入register file)，且rd等於rs1或rs2 (前一指令寫入的位置等於後一個指令讀取的位置)，trust訊號為0，代表如果後一個訊號為branch，那麼此時在ID級所判斷是否要branch的結果不可信。若是前一個指令為R-type，此即為use-branch hazard，若是為LW，即為load-branch hazard。

如果MEM\_write為1 (前兩個指令要寫入register file)，且rd等於rs1或rs2(前兩指令寫入的位置等於後兩個指令讀取的位置)，trust訊號為0，代表如果後兩個訊號為branch，那麼此時在ID級所判斷是否要branch的結果不可信。若是前一個指令為R-type，此即為use-branch hazard，若是為LW，即為load-branch hazard。

* 1. Two stage branch outcome determination pitfalls

在做兩級branch指令判斷時，還有一個陷阱：若是當id級作完branch的判斷，且為可信時，無論有無branch taken，到了下一級一定要被洗掉，變成NOP，否則到了ex級又會被做一次，造成錯誤的branch結果。

1. **心得**
   1. 王柏智

這次的final project跟以前的作業比起來要複雜得多，而且因為是以小組的形式來做，因此很多事情必須事前先溝通好，例如要統一I/O port 。我們首先先決定要實做的23個指令，接著參考網路上CPU的架構，再一起研究完架構後再分配工作，然後一起決定好I/O port，最後才開始撰寫每人各自被分配到的部分。我的工作是完成IF級的撰寫，由於我的撰寫速度較慢，在確定好我負責的小module都沒問題後，我就跟著大家一起幫忙controller的debug以及用Superlint檢查coding style。大部分的問題都出在hazard的偵測以及該要怎麼去forward還沒write back的資料，每個問題幾乎都要從錯誤的地方開始一個一個往上找，由於架構龐大，所以都要找了很多層才能找到錯誤的地方並修正。其中有一個問題找了很久還是找不到，請助教來也花了很長一段時間也找不到根本的原因，最後終於發現我們把clock分成兩個的phase，並且都是正緣觸發，而這樣會導致兩個clock其實會有一點點誤差。在最一開始我們其實就討論過要用2個phase的clock還是直接把一個clock用正負緣觸發就好，當初我們是覺得兩種是同樣的意思，但沒想到我們選了會有問題的方案，真的非常感謝助教發現這個大問題，也糾正了我們觀念上的大錯誤。

另外，這次final project在合成完要進行layout，用EDA tool自動佈線，由於我們開始layout的時間離EDA cloud關閉已經所剩不多，因此一遇到步驟有稍微不懂的就沒有細想馬上衝去問助教，結果常常都獲得很簡單的答案，是我們自己沒注意到的小問題，十分感謝助教在半夜還不厭其煩地為我們解決問題，在最後layout完看到笑臉，此時後面負責ICC的組員也傳出了ICC 97%的喜訊，大家都非常開心。

這次的project讓我學習到如何以小組形式完成一個大project，從必須共同規範的部分再到個別完成自己工作，每個人都把自己負責的部分做到最好，再加上我們這組的溝通是十分良好的，終於完成了這個大工程**。**

* 1. 王駿瀚

在這次的Final project裡，我主要負責EX stage的部分，由於這塊基本上都是組合邏輯的電路，所以並不需要注意時序的問題，但必須要能夠做出能夠快速運算的ALU，於是我一直思考與實驗，看什麼樣子的方式能夠把速度提升，並且又不會增加太多的面積，後來發現把會需要用到加法器的運算和邏輯的運算分開，可以優化電路到最少的延遲，而且延遲最久的永遠都是加法器，所以我開始嘗試不同的加法器架構，一開始想說先用carry lookahead adder的架構，後來發現面積實在是有點大，後來又多加了一些carry select adder在上面減少面積，但還是相當的大，最後才決定來參考這篇論文，但論文裡又有很多我看不懂的地方，有些我甚至覺得他根本就接錯線了，於是又花了很多時間去研究，運用迪摩根的方式來一步一步地了解正確的接線方法應該長怎樣，又要思考圖裡哪裡還需要再多加一個inverter，整個過程用到了許多很基礎的觀念，我覺得這是我在做final project裡，運用到最多以前覺得沒意義的東西，透過這個final project讓我對那些知識有了不一樣的看法。

除了在ALU及ALU Controller的設計以外，我也學到了很多團隊合作的重要性，也很慶幸我的團隊夥伴們都很厲害，以及相當積極的態度，其實我們正式開始的時間，大概在報告的前一個禮拜而已，但從正式開始的那天起，我們每天都在CIC待好待滿，甚至回到家裡也是在做final project，沒日沒夜地寫Verilog，但也因為大家的團隊分工及努力，才讓我們能在期限內完成。

在整個final project進行的過程中，我覺得難度相較於以往的homework來說，難度是不同地層次，畢竟這是我第一次要做一整個系統，對於整個系統的分工以及團隊的溝通都是一個全新的挑戰，而且我們也碰到了很多以前計算機組織沒有遇到過的問題，像是use-branch之類的hazard要怎麼處理，或是合成後的波型模擬與合成前的不同，這些都依靠著大家的力量一一克服了，此外我們在轉layout的時候也一直卡關，在那最後的關頭心情真的是很忐忑不安，畢竟失敗了可能也沒辦法再重新寫一份，所以當看到LVS驗證的笑臉時，CIC一瞬間充滿了大家的歡呼聲，我覺得那種感覺真的很棒，也感謝隊友積極地想做好這次的final project，和助教們在我們遇到疑問時能夠給予協助。

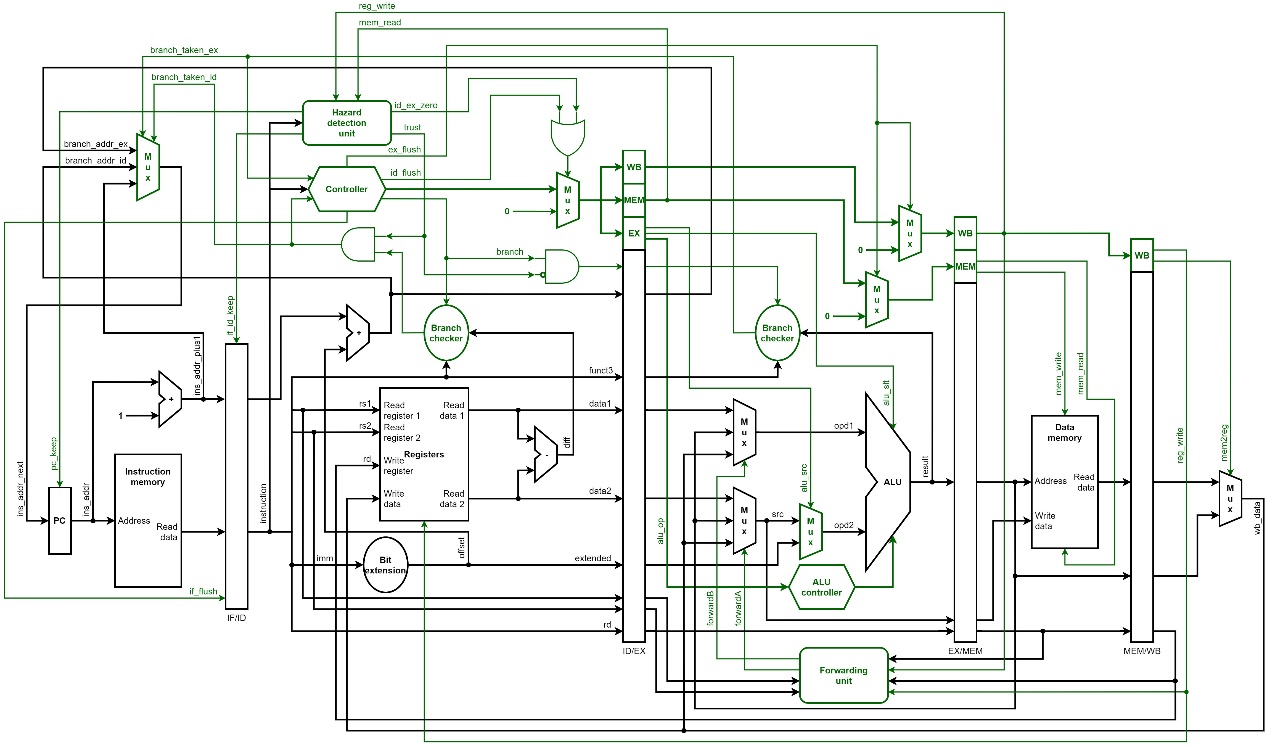
* 1. 陳亮州

這次的final project與之前所做過的作業們真的有很大的不同，這次所要設計的不再是像之前單一或少許功能的小電路，而是需要完成許多指令需求的管線化CPU，因此其中又有許多不同的零件，因此非常看重大家配合的能力，因為大家期末都相當忙碌，因此我們真正開始實作其實只剩大約一個星期的時間了，一開始快速訂出指令集以及我們要使用的架構省去我們很多思考的時間，在這整段過程中，我發現大家妥善的溝通真的是這次可以成功做出CPU的關鍵，從一開始根據大家的需求訂出自己所負責的module需要的IO，到實際在撰寫時臨時需要修改甚麼部分，若是沒有溝通好的話真的會拖延許多不必要的時間。

這次的設計也不是完全依照我們設想的架構下去實作後就可以成功做出來，在依照原本的設計撰寫後，運行模擬程式還是不斷發現新的問題，其中大部分的問題都出在hazard的偵測以及該要怎麼去forward還沒write back的資料，為了一一挑出這些錯誤花了我們非常多，從訊號線一一回推回去錯誤發生的地方，直到我們模擬各種可能出現hazard的情況都正確無誤，到最後我們合成完助教發現我們原本clock使用的方式其實會發生嚴重的timing violation，這段過程真的稱不上有趣，要有足夠的耐心一一的去解析問題所在。

我這次所負責的部分是ID級的電路以及register file的部分，其中包含IF/ID的register、將bit擴展至32bit的電路，由於我做的部分相對hazard detection之類較複雜的電路單純，僅需要與負責controller與hazard detection的同學正確溝通好每個IO的作用其實不用花很長的時間就可以做完了，在等待整合大家的電路的同時，為了節省後續驗證的時間，我花了一點心力在寫assembler以及合成tcl檔的生成程式，雖然花了我一些時間寫它，但是後續省掉許多手刻binary code的時間真的讓我覺得挺值得的。我超感謝這次大家互相都非常配合彼此的需求，而且都在進度上完成自己的工作，真的是非常凱瑞的隊友，另外也很謝謝助教不斷回答我們千奇百怪的問題，就算半夜layout發生問題助教也不厭其煩地協助我們，我們才能如此有效率地做出這次的期末專題。

* 1. 陳俊瑋

****

在這次的pipeline-CPU的final project中，我所負責製作的是上方圖中紅色框框處: MEM/WB register、forwarding unit以及hazard detection。其中，MEM/WB register相對簡單，主要功能是將MEM級的訊號傳至WB級即可，因此這部分的撰寫沒有花我太多時間。

相對而言，forwarding unit和hazard detection unit則是我這段時間一直修改的部分。一開始，我完全按照計算機組織這堂課程中所教的去撰寫，考慮了EX/MEM forwarding、MEM/WB forwarding、double data hazard 以及load-use hazard，原本以為已經萬無一失了，卻在最後驗證階段發現了漏洞。這個漏洞的來源是branch指令，由於我們的架構，讓branch指令在ID級即可決定是否要branch。然而，若是branch指令的上一個指令為R-type或是load-word，且二者的寫入位置為branch的讀取位置，將會發生錯誤。

針對這項漏洞，我所提出的方法是，將forward拉回ID級(上圖中的藍色框框及箭頭)，如此一來，便使得branch指令能夠即時收到正確的訊號。但後來，我們這組的其他組員提出了另一個更有效的方法，就是在hazard detection unit新增一個輸出訊號trust，此輸出訊號若為1，則在ID級中所決定的是否branch結果為可信的，若為0，則為不可信。此部分在書面報告中有詳細描述。

而能夠及時發現這項漏洞，全都要感謝我的組員們的心思縝密，想到branch指令所隱藏的危險，才有最後完美的結果。

以上是code撰寫的部分，然而，在製作這個project的過程中，最讓我覺得驚奇且有趣的地方，就是畫layout的時候，在之前我所修過的超大型積體電路電腦輔助設計概論這堂課中，我們是手動畫出layout，而且電路構造十分簡單。但這次是pipeline的CPU，電路構造相對複雜許多，因此便使用tool來畫。由於第一次使用這個tool，所以很多地方不太理解，還好有助教們的全力相助，讓我們突破重重難關。隨著操作步驟一步步走下去，我們CPU的layout也漸漸成形，每進行一個步驟，就多了一些金屬線，直到最後看到完整的圖，尤其是看到LVS的結果為笑臉時，真的高興壞了!

在這個星期，我們早出晚歸，只為了這項project，過程雖然辛苦，但結果總是好的，尤其是大家一起解決問題後所獲得的成就感，實在無法用言語形容。在整個過程中，大家各司其職，完成自己所負責的部分，最後拼出一個我們心目中理想的CPU，而這顆CPU幕後的最大功臣，當屬蕭又瑜同學，從一開始的架構、指令，到最後發現漏洞、處理bug，幾乎是他一手包辦，可以說是我們這組的核心人物。最後，也要感謝所有的同學們，一起合作將這項project完成，也讓我學到很多事情，受益良多。

* 1. 黃子青

這次的final project真的是件大工程，從指令集跟cpu架構的設計，到RTL code的撰寫，再到組語的測試，再到合成後的電路再跑一次測試，再到最後的layout，每一步都必須小心翼翼地不能出任何一個bug不然就得更改RTL code再重跑一次整個流程，像是我們原本的設計有包含一個two phase clock的generator，它會透過有線狀態機將原本的clk除頻產生兩個相反的clk。

因為pre-sim不會考慮delay，所以跑完測試後結果跟我們預期的一樣我們就以為我們做對了，實際跑合成後，slack都有大於0，也沒有timing violation，但是看了合成後的波形圖後，發現結果會跟pre-sim前的不一樣，我們就整組一起看波形圖debug，de了一個下午後還是找不到原因，於是去問助教，助教一看我們的波形就大聲地驚問你們怎麼會有兩個clock?!助教馬上找出問題的癥結點，跟我們講解我們應該要用同一個clock的正緣跟負緣而不是製造兩個相位相反出來再用他們的正緣，因為合成後的延遲會讓兩個clock的相位有偏移，導致出來的結果會不堪設想，於是我們就聽助教說的把clock改成一個，再跑一次合成後看波形，clock的確是沒有相位偏移的問題了，可是不知為甚麼跑出來的結果還是有問題，後來靠著強力的隊友發現我們的架構hazard有些地方沒有處理掉，才會有如此問題，微調我們的架構後再合成一次，再跑波形結果就正確了，但是距離EDA cloud關閉已經只剩不到15小時，於是我們就在CIC通宵跑layout的部分，途中遇到問題還三更半夜的去打擾助教問他們問題，真的很感謝他們都願意不厭其煩地解決我們遇到的問題，我們才能順利地完成這次project。

我這次所負責的部分是MEM stage的電路以及Instruction Memory的部分，其中包含EX/MEM的register、Data Memory。

Instruction Memory為Read-only memory，透過在testbench readmemb將其寫入指令，cpu只能跟其要資料不能做寫入的動作，Data Memory則是可以讓cpu做寫入的動作及讀取。另外我還撰寫了氣泡排序法、斐波納契數列的組合語言，透過隊友寫的assembler轉成machine code給我們cpu做測試，除此之外我還參與到layout的部分，在經歷了千辛萬苦過後終於看到LVS 的笑臉真的會覺得這一切的努力都值得了，真的很慶幸自己能遇到這麼凱瑞的隊友們，還有這麼強的助教群，才能順利完成這次的期末project。

* 1. 蕭又瑜

我想我們完成了一次十分不簡單的任務，在我二年級修計算機組織的時候我就有想過，這樣一個pipelined的cpu真正做起來會有多複雜？這一次的final project深刻地讓我體會到了。

由於我們組員都是大三的學生，必修的期末考loading太大，讓我們不得已只能在所有科目考完後才開始動工。老實說，在1/8之前，我們真的是0進度，這導致我們的時間變得蠻緊迫的，我們第一天分配完大家要寫的module後，很快系統就慢慢有一個樣子了，大概在動工的48小時後我們就將所有人的module接起來，並寫了一個簡單的testbench，就想來測試了，然而第一次的結果通常是令人常望的，我們的cpu跟本沒有所預期的動作，從這時就是debug噩夢的開始，將所有零件寫出來並不是最困難的，最困難的是找出哪裡寫錯了，這大概花了我們兩天的時間，最令人厭惡的就是編譯可以過，但是執行結果不對，常常是少了一行或接錯線這種低級錯誤就能耗掉我們一個下午的時間，好不容易好像沒有bug了，終於可以拿去合成時，出來的post-sim竟然又跟pre-sim不一樣，這是更令人感到錯折的，好幾次我都在懷疑我們到底做不做得出來，還好這時有助教不厭其煩的幫我們解惑，還幫我們找到想都沒想到的錯誤：clk generator，真的十分感謝。

說到clk generator，這其實是我寫的module，一開始是想說老師課堂上提到的pipelined cpu，是由兩個phase所控制的，我就依樣畫葫蘆，結果發現我的寫法沒辦法合成，所以我就天兵的想了用有限狀態機的方法，沒想到正是這個原因，讓我們一整天都沒有進展，實在是令人哭笑不得。其他我負責的部份還有controller與整個top module，在整個撰寫的過程主要是我開出每個module所需要的I/O，再將大家實作出來的module接起來。而我的工作主要在將系統完成、debug與驗證模擬的結果，若是合成與layout其實都是由我們組員完成的，由其是layout時俊瑋、子青與柏智一步一步照著講義，做到三更半夜跳出笑臉的那一刻，讓大家終於感到努力的一切都是值得的，除此之外，還要感謝亮州寫assembler與tcl的生成程式，讓我們的作業更有效率；而駿瀚則是常常一整天都在努力的研究ALU的架構，做出體積小且相對快的ALU，也非常辛苦，沒有組員之間的合作與付出，就沒有我們的成果。

最後我想說，在選這堂課時其實只是抱著修修看的心態，並不確定自己在這個領域感不感興趣，結果老師授課的內容與我所學到的能力，讓我覺得我修這堂課的選擇實在是太正確了，相信在未來也能運用這學期所學到的東西，並更進一步精進自己的實力。

1. **分工**

|  |  |  |  |
| --- | --- | --- | --- |
| **姓名** | **學號** | **項目** | **貢獻度(%)** |
| 王柏智 | E24076611 | pc.v \*layout | 16 |
| 王駿瀚 | E24076750 | alu\_controller.v alu (advanced adders) id\_ex\_register | 16 |
| 陳亮州 | E24076297 | reg\_file.v bit\_extension.v if\_id\_reg.v synthesis assembler.py SuperLint ICC verification | 16 |
| 陳俊瑋 | E24076603 | forwarding\_unit.v hazard\_detection\_unit.v mem\_wb\_reg.v | 16 |
| 黃子青 | E24076938 | ins\_mem.v data\_mem.v ex\_mem\_reg.v bubblesort.s fibonacci.s layout | 16 |
| 蕭又瑜 | E24076459 | def.v controller.v cpu\_top\_module.v wiring simulation debugging ICC | 20 |

1. **參考資料**
   1. K. Chirca et al., "A static low-power, high-performance 32-bit carry skip adder," Euromicro Symposium on Digital System Design, 2004. DSD 2004., Rennes, France, 2004, pp. 615-619, doi: 10.1109/DSD.2004.1333335.
   2. Computer Orgnization lecture notes
   3. Lecture notes
   4. Celio, Christopher. ucb-bar/riscv-sodor. Regents of the University of California. [12 February 2015].
   5. The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2, Andrew Waterman, Krste Asanovic, CS Division, EECS Department, University of California, Berkeley.