

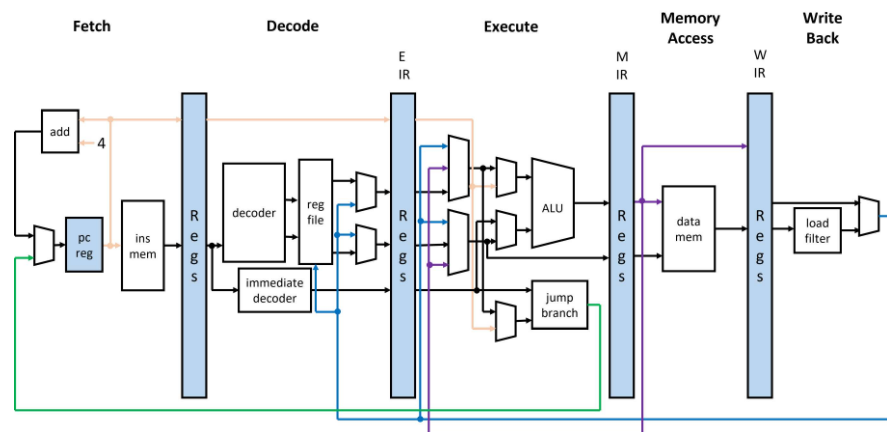
4. Hardware design project

GitHub URL: <https://github.com/Chia-Yu-Kuo/Hardware-design>

(1) RISC-V Special HW Pipeline-CPU with APR and Code Coverage

甲、摘要

設計一個五級的 pipeline CPU 電路，附加上之浮點數運算、Branch prediction、快速乘法器設計、除法器等功能，並用 RISC-V 組合語言來驗證系統正確度 (單一指令正確性、merge sort、Fibonacci sequence) 與 ICC 檢查，最後利用 APR layout 出電路並完成 LVS 驗證。



圖一、基本 Pipeline-CPU 架構

乙、想法

i. CPU 內部設計

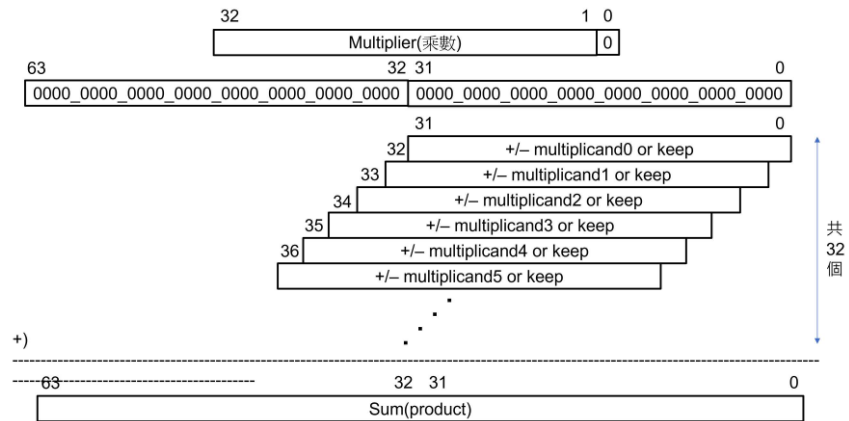
管線化設計: 用五個 stage 劃分，IF stage 讀取指令，ID stage 指令解碼，EX stage 執行，MEM stage 記憶體存取，WB stage 寫回暫存器。

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock cycle	1	2	3	4	5	6	7

圖二、管線化架構

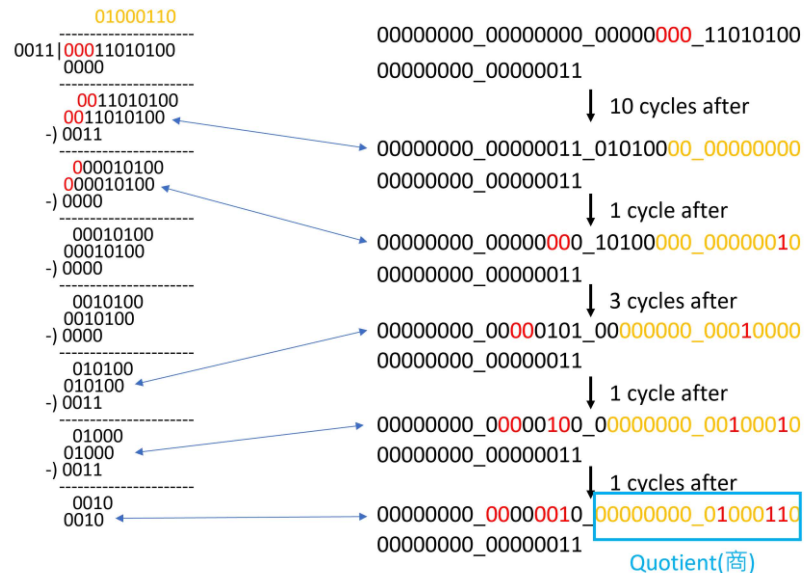
Hazard 處理: 將 memory 切成 IM 與 DM 解決 structure hazard; 利用將指令改為 nop 處理跳轉指令所產生的 control hazard, 但每次 stall 會拖慢速度, 因此有利用 branch prediction 來加速; 利用 forward 技術來解決 data hazard。

Booths multiplier: 利用 3bits 大小的 kernel 來掃乘數, 並查詢 booths encoding 後的表來做加減運算可得乘積。



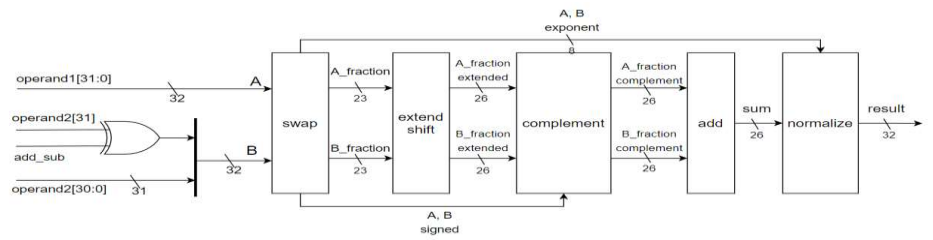
圖三、Booths multiplier algorithm

Divider: 用多級來拆分 32bits 的除法, 把除式中的被除數一直向左移而除數固定, 即可得到商及無止盡的餘數。



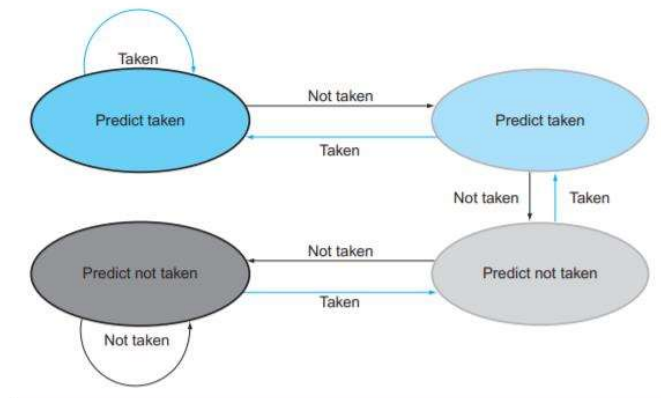
圖四、除法器演算法

Floating point subtractor, adder: 根據 floating point format 來設計加減法電路。先判斷正負, 並根據 exponent 把 operand1, operand2 fraction shift 對齊並做加減運算, 最後 normalization 後輸出。

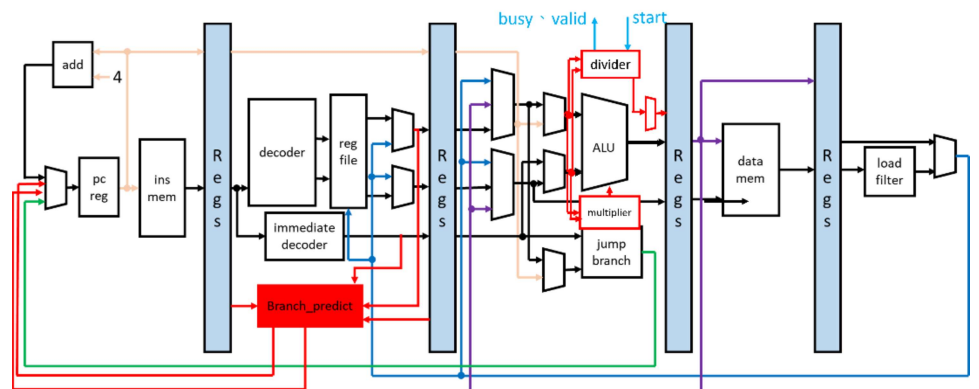


圖五、Floating point 運算電路系統方塊圖

Branch Prediction: 利用過去的跳轉歷史紀錄搭建出 moore machine 和輸出下個 cycle 對應的 PC 位置，並用強弱訊號來加強預測的準確度，降低 CPU 花費再 Control hazard 所浪費的時間。



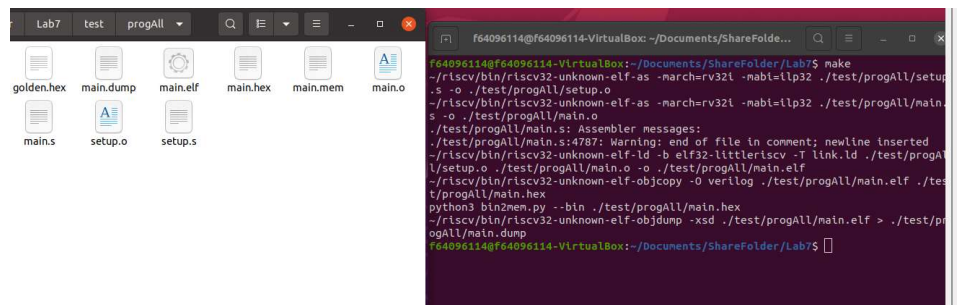
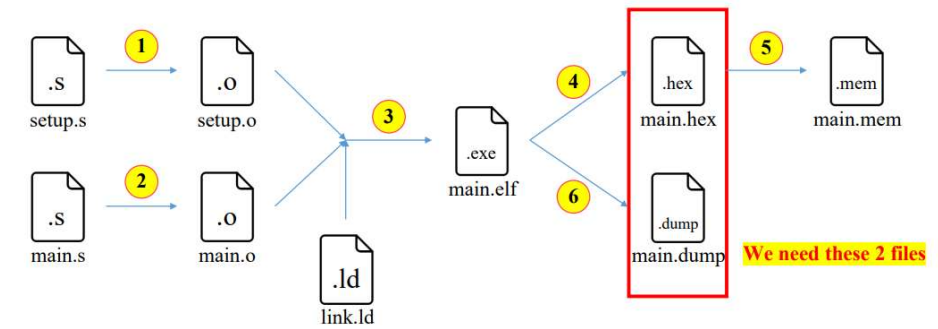
圖六、2bit Branch predictor



圖七、加上特殊硬體後的 Pipeline CPU 架構圖

ii. CPU 驗證

系統正確性: 先專寫 RISC-V 組合語言(基本測試+merge sort+費氏數列)，將寫好的 main.s 與要設定的環境(如 main、result 位置等) setup.s 放入 linux 環境利用 assembler+linker 把他轉為執行檔 main.exe，最後反組譯回我們所要的 IR main.hex 檔及可方便 debug 的 main.dump 檔以及記憶體印象檔。

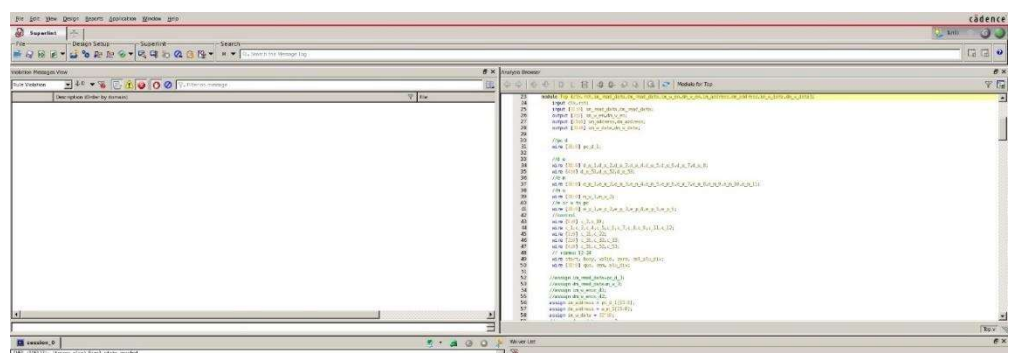


圖八、驗證 pattern 產生過程

ICC 檢查: 為了有效提升 ICC 檢查結果, 除了盡可能增加各種類型的指令使 cpu 每個功能都能確實執行到之外, 發現也可以透過簡化硬體的方法來達成, 在使用的 ISA 架構下, cpu 的 address 傳遞的資料實際上最大就是 16bits, 而當初方便起見傳輸線都設定為 32bits, 也就是說其實有很多地方無法 toggle 到, 因此進行硬體簡化, 將多餘 32bits 的部分簡化成 16bits, ICC 的結果也有顯著提升, 特別是 adder_pc 的部分, 由原本 40%提升到 75%, 將近兩倍之多。

Test	Module	Types	Self Total	Cumulative Total
	adder	16bit	75% (64 / 84)	75% (64 / 84)
	Reg_PC	16bit	100% (34 / 40)	100% (34 / 40)
	Reg_D	16bit	100% (34 / 107)	100% (34 / 107)
	MemPC	16bit	100% (83 / 83)	100% (83 / 83)
	Top	16bit	90% (1038 / 1152)	94% (1781 / 1905)
	Reg_M	16bit	100% (156 / 156)	100% (156 / 156)
	Reg_W	16bit	100% (156 / 156)	100% (156 / 156)
	Decoder	16bit	100% (34 / 56)	100% (34 / 56)
	Controller	16bit	100% (1717 / 2226)	100% (1717 / 2226)
	Reg_E	16bit	100% (124 / 124)	100% (124 / 124)
	Insta_Ed	16bit	100% (78 / 80)	100% (78 / 80)
	ALU	16bit	100% (252 / 256)	100% (252 / 256)
	RegFile	16bit	100% (1122 / 1122)	100% (1122 / 1122)
	Mem_Inst	16bit	100% (134 / 135)	100% (134 / 135)
	JS_Unit	16bit	100% (96 / 97)	100% (96 / 97)
	Mem	16bit	100% (97 / 97)	100% (97 / 97)
	LD_Filter	16bit	100% (74 / 74)	100% (74 / 74)

圖九、ICC 檢查結果



圖十、superlint 檢查結果

丙、結果

i. 硬體正確性:

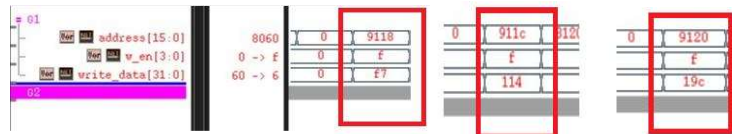
指令:

```
add:
  li t0, 0xffffffff # -1
  li t1, 0xffffffff # -1
  add t0, t0, t1 # t0 = -2
  add t0, t0, t1 # t0 = -3
  add t0, t0, t1 # t0 = -4
  add t0, t0, t1 # t0 = -5
  add t0, t0, t1 # t0 = -6
  li t1, 0xffffffe # -2
  add t0, t1, t0 # t0 = -8
  add t0, t1, t0 # t0 = -10
  add t0, t1, t0 # t0 = -12
  add t0, t1, t0 # t0 = -14
  add t0, t1, t0 # t0 = -16
  sw t0, 0(s0)
  addi s0, s0, 4
```

DM['h9000'] = ffffffff0, pass
 DM['h9004'] = ffffffff8, pass
 DM['h9008'] = 00000008, pass
 DM['h900c'] = 00000001, pass
 DM['h9010'] = 00000001, pass
 DM['h9014'] = 78787878, pass
 DM['h9018'] = 000091a2, pass
 DM['h901c'] = 00000003, pass
 DM['h9020'] = fefcfefd, pass
 DM['h9024'] = 10305070, pass
 DM['h9028'] = cccccccc, pass
 DM['h902c'] = fffffffc, pass

圖十一、以 add 為例，運算後結果與 golden 相同

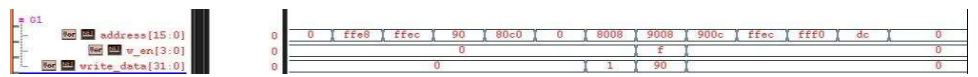
排序:



DM['h9114'] = 000000dd, pass
 DM['h9118'] = 000000f7, pass
 DM['h911c'] = 00000114, pass
 DM['h9120'] = 0000019c, pass
 DM['h9124'] = 000001bb, pass

圖十二、merge sort 後與 golden 相同

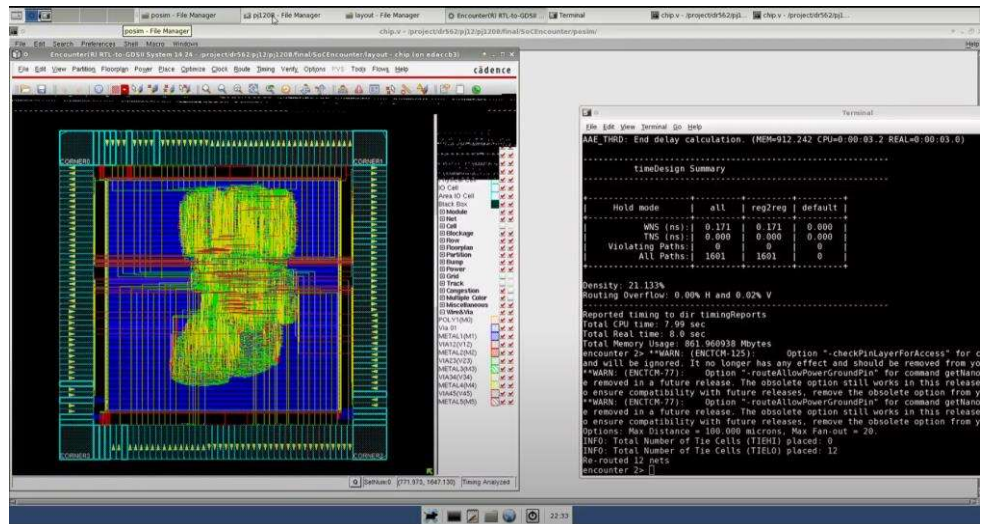
費式數列:



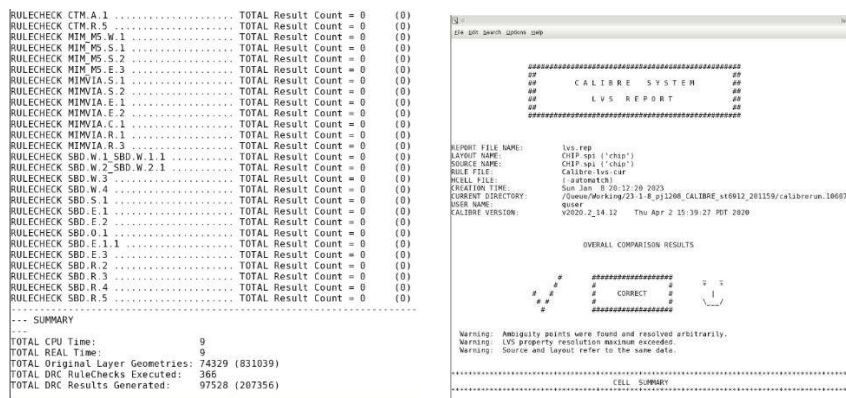
DM['h9000'] = 00000005, pass
 DM['h9004'] = 00000037, pass
 DM['h9008'] = 00000090, pass

圖十三、費式數列後與 golden 相同

ii. APR 結果



圖十四、layout 結果



圖十五、DRC、LVS 結果

丁、問題討論

Q: 關於單一指令需要超過 1 cycle 的指令如何優化?

討論原因:

在 pipeline-CPU 當中如果有一個指令的執行階段超過 1-cycle，CPU 的 fetch 階段以及 execute 階段會 stall (停佇)，使得電腦會有許多單元會進入 idle 的狀態，而且電腦需要經常的使用這種指令的時候，就會造成大量資源和時間的浪費，使得 CPU 的整體效率變的很低。

討論結果:

可以考慮使用”Speculative execution”(預測執行)的方式處理，這個是一種優化的方式，讓其他獨立的指令先行，使的在資源過剩的情況下能夠繼續動作。和 branch prediction 很像，但是不再受限於會跳轉的條件判斷或是只適用於迴圈運算的指令條件。

方法優點：

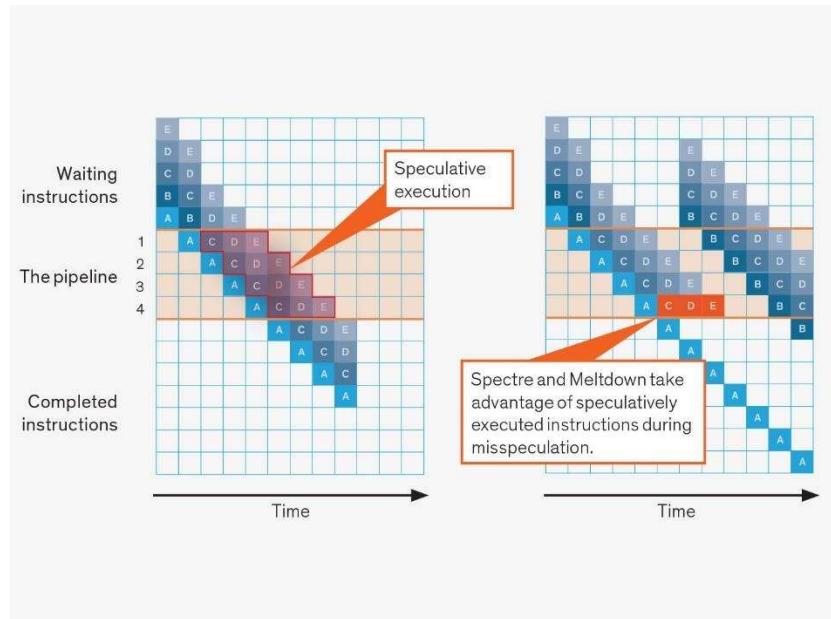
a.執行效率大幅提升，不會受限於執行週期長的指令。

方法缺點：

a. 需要有運算資源被閒置，不會搶奪正在執行指令的資源。

b.增加控制器的複雜度，需要判斷哪些指令沒有資料的相依性。

c.安全性的議題



圖十五、Speculative execution 原理