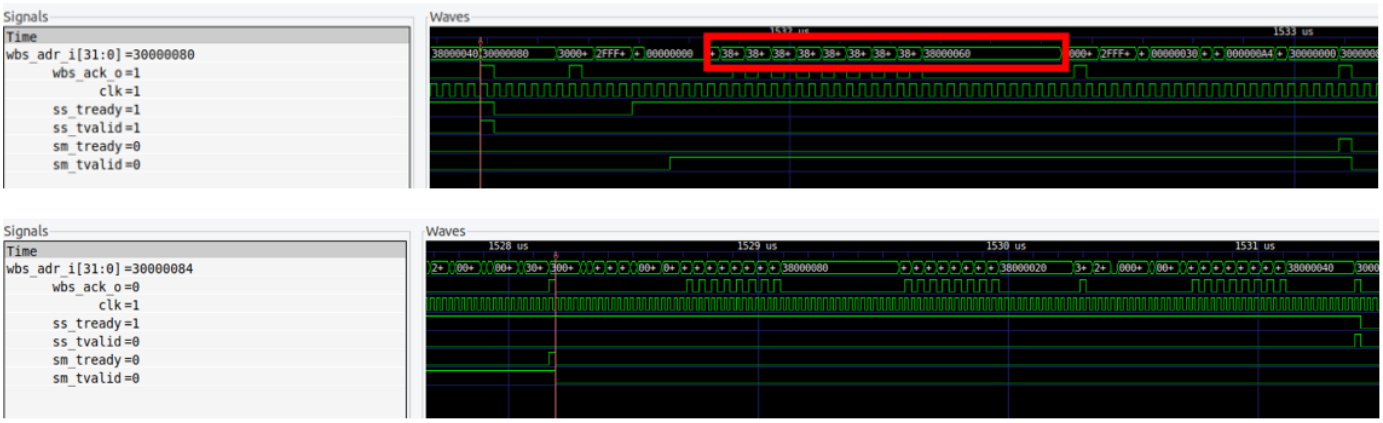
Lab4-2 優化過程

我們先從未優化前的 waveform 分析!



圖一:waveform

我們可以發現整個 throughput 為 197cycle，但可以發現我們硬體做計算時只有 14cycle 就可以做完，所以問體是卡在 cpu 上，我們可以發現 wb_addr 會跑去 3800x0000 的位置從 bram 讀取 code，這也導致軟體的部分，也就是 x->y(對應到 c code 中 reg_fir_x = i 到 outputsignal[i] = reg_fir_y)花費的 cycle 數高達 68 cycle 以及 y->x(對應到 c code 中 outputsignal[i] = reg_fir_y 迴圈至 reg_fir_x = i)花費的 cycle 數高達 129 cycle。所以我們要想辦法減少這兩部分。剛剛說到我們發現 cpu 會跑去做其他事情，所以我們打開 assembly code 去看有沒有哪部分可以優化。

```
#include <stdint.h>
#include <stdbool.h>

#define N 64

int outputsignal[N];
// AP control
#define reg_fir_control (*(volatile uint32_t*)0x30000000)

// FIR input X, FIR output Y
#define reg_fir_x (*(volatile uint32_t*)0x30000080)
#define reg_fir_y (*(volatile uint32_t*)0x30000084)



int* __attribute__ ( ( section ( ".mprjram" ) ) ) fir(){

                uint8_t  register i=0;
                reg_fir_control = 1; //set ap_start, bit[0] = 1

                while(i<N){

                    while((reg_fir_control >> 4) & 1 !=1); // external signal x[n] ready, wait until bit[4] = 1
                            reg_fir_x = i;

                    while((reg_fir_control >> 5) & 1!=1); // external signal y[n] ready, wait until bit[5] = 1
                                outputsignal[i] = reg_fir_y;
                                i=i+1;
                }
                while((reg_fir_control >> 1) & 1 != 1); // read ap_done, bit[1] = 1

                return &outputsignal[63];
}
```

```
38000000 <fir>:
38000000:   ff010113    addi    sp,sp,-16
38000004:   00812623    sw      s0,12(sp)
38000008:   00912423    sw      s1,8(sp)
3800000c:   01010413    addi    s0,sp,16
38000010:   00000493    li      s1,0
38000014:   300007b7    lui     a5,0x30000
38000018:   00100713    li      a4,1
3800001c:   00e7a023    sw      a4,0(a5) # 30000000 <_erodata+0x1ffffd10>
38000020:   0580006f    j       38000078 <fir+0x78>
38000024:   00000013    nop
38000028:   300007b7    lui     a5,0x30000
3800002c:   0007a783    lw      a5,0(a5) # 30000000 <_erodata+0x1ffffd10>
38000030:   300007b7    lui     a5,0x30000
38000034:   08078793    addi    a5,a5,128 # 30000080 <_erodata+0x1ffffd90>
38000038:   00048713    mv      a4,s1
3800003c:   00e7a023    sw      a4,0(a5)
38000040:   00000013    nop
38000044:   300007b7    lui     a5,0x30000
38000048:   0007a783    lw      a5,0(a5) # 30000000 <_erodata+0x1ffffd10>
3800004c:   300007b7    lui     a5,0x30000
38000050:   08478793    addi    a5,a5,132 # 30000084 <_erodata+0x1ffffd94>
38000054:   0007a783    lw      a5,0(a5)
38000058:   00048613    mv      a2,s1
3800005c:   00078693    mv      a3,a5
38000060:   00400713    li      a4,4
38000064:   00261793    slli    a5,a2,0x2
38000068:   00f707b3    add     a5,a4,a5
3800006c:   00d7a023    sw      a3,0(a5)
38000070:   00148793    addi    a5,s1,1
38000074:   0ff7f493    zext.b  s1,a5
38000078:   03f00793    li      a5,63
3800007c:   fa97f4e3    bgeu    a5,s1,38000024 <fir+0x24>
38000080:   00000013    nop
38000084:   300007b7    lui     a5,0x30000
38000088:   0007a783    lw      a5,0(a5) # 30000000 <_erodata+0x1ffffd10>
3800008c:   10000793    li      a5,256
38000090:   00078513    mv      a0,a5
38000094:   00c12403    lw      s0,12(sp)
38000098:   00812483    lw      s1,8(sp)
3800009c:   01010113    addi    sp,sp,16
380000a0:   00008067    ret
```

圖 2:未優化的 assembly code

我們可以發現 assembly code 中 while 迴圈的部分十分的攏長，，也正因為如此他才要再去 3800 讀

取 code。 所以我們利用 -O1(當然也可以用-O2、-O3、-Os 或-Ofast)這些指令來讓編譯器幫我們優化 assembly code(在 run_sim 中修改如下)

```
rm -f counter_la_fir.hex

riscv32-unknown-elf-gcc -Wl,--no-warn-rwx-segments -g \
        --save-temps \
        -Xlinker -Map=output.map \
        -I../../firmware \
        -march=rv32i -mabi=ilp32 -D__vexriscv__ \
        -Wl,-Bstatic,-T,../../firmware/sections.lds,--strip-discarded \
        -ffreestanding -nostartfiles -O1 -o counter_la_fir.elf ../../firmware/crt0_vex.S ../../firmware/isr.c fir.c counter_la_fir.c
# -nostartfiles
riscv32-unknown-elf-objcopy -O verilog counter_la_fir.elf counter_la_fir.hex
riscv32-unknown-elf-objdump -D counter_la_fir.elf > counter_la_fir.out

# to fix flash base address
sed -ie 's/@10/@00/g' counter_la_fir.hex

iverilog -Ttyp -DFUNCTIONAL -DSIM -DUNIT_DELAY=#1 \
        -f./include.rtl.list -o counter_la_fir.vvp counter_la_fir_tb.v

vvp counter_la_fir.vvp
rm -f counter_la_fir.vvp counter_la_fir.elf counter_la_fir.hexe
```

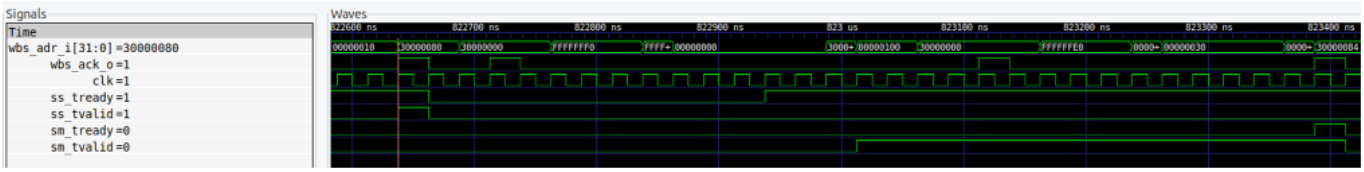
圖 3:run_sim 修改位置

```
#include <stdint.h>
#include <stdbool.h>

#define N 64

int outputsignal[N];
// AP control
#define reg_fir_control (*(volatile uint32_t*)0x30000000)

// FIR input X, FIR output Y
#define reg_fir_x (*(volatile uint32_t*)0x30000080)
#define reg_fir_y (*(volatile uint32_t*)0x30000084)



int* __attribute__ ( ( section ( ".mprjram" ) ) ) fir(){

        uint8_t  register i=0;
        reg_fir_control = 1; //set ap_start, bit[0] = 1

        while(i<N){

            while((reg_fir_control >> 4) & 1 !=1); // external signal x[n] ready, wait until bit[4] = 1
                    reg_fir_x = i;

            while((reg_fir_control >> 5) & 1!=1); // external signal y[n] ready, wait until bit[5] = 1
                        outputsignal[i] = reg_fir_y;
                        i=i+1;
        }
        while((reg_fir_control >> 1) & 1 != 1); // read ap_done, bit[1] = 1

        return &outputsignal[63];
}
```

```
38000000 <fir>:
38000000:   300007b7        lui     a5,0x30000
38000004:   00100713        li      a4,1
38000008:   00e7a023        sw      a4,0(a5) # 30000000 <_erodata+0x1ffffe08>
3800000c:   00400693        li      a3,4
38000010:   00000713        li      a4,0
38000014:   04000593        li      a1,64
38000018:   0007a603        lw      a2,0(a5)
3800001c:   08e7a023        sw      a4,128(a5)
38000020:   0007a603        lw      a2,0(a5)
38000024:   0847a603        lw      a2,132(a5)
38000028:   00c6a023        sw      a2,0(a3)
3800002c:   00170713        addi    a4,a4,1 # 20001 <_fstack+0x1fa01>
38000030:   00468693        addi    a3,a3,4
38000034:   feb712e3        bne     a4,a1,38000018 <fir+0x18>
38000038:   300007b7        lui     a5,0x30000
3800003c:   0007a783        lw      a5,0(a5) # 30000000 <_erodata+0x1ffffe08>
38000040:   10000513        li      a0,256
38000044:   00008067        ret
```

圖 4: 優化後 assembly code

我們可以發現優化後 assembly code ，也變得更加簡潔。這時再重新觀察 waveform



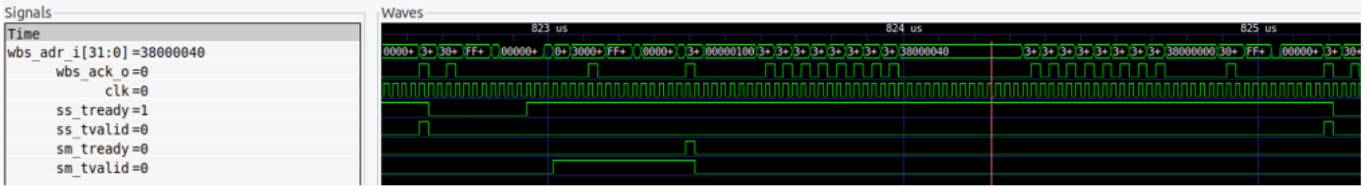


圖 5: 第一次優化後的 waveform:

經由第一次優化後的 x->y 花費的 cycle 數為 29 cycle 以及 y->x 花費的 cycle 數為 72 cycle，整體 throughput 為 101cycle。在這可以明顯的發現 cpu 有些時候不會再跑去 bram 重新讀指令，後面會證明是由 cache 來運作。除此之外，我們發現 3000x0000 在送 y 值之前讀了兩次，對應到了 while 迴圈判斷是否可以接收 y，這造成不必要的 cycle 再跑一次至 3000x0000，但我們發現其實可以直接送進去交給硬體判斷，上圖可以發現都是 smtvalid 先拉至 1 後等待 smtready 完成傳遞，但如果我不利用 while 迴圈來判斷，而是直接先給 smtready 拉至 1，然後等待 smtvalid 完成傳遞，這樣可能會花更少的 cycle，故我們需要修改 fir.c 來進行第二次優化:

```
int* __attribute__ ( ( section ( ".mprjram" ) ) ) fir(){

        uint8_t  register i=0;
        reg_fir_control = 1; //set ap_start, bit[0] = 1

        while(i<N){

                reg_fir_x = i;
                outputsignal[i] = reg_fir_y;
                i=i+1;

        }

        while((reg_fir_control >> 1) & 1 != 1); // read ap_done, bit[1] = 1

        return &outputsignal[63];
}
```

```
177 38000000 <fir>:
178 38000000:       300007b7        lui     a5,0x30000
179 38000004:       00100713        li      a4,1
180 38000008:       00e7a023        sw      a4,0(a5) # 30000000 <_erodata+0x1ffffe08>
181 3800000c:       00400713        li      a4,4
182 38000010:       00000793        li      a5,0
183 38000014:       300006b7        lui     a3,0x30000
184 38000018:       04000593        li      a1,64
185 3800001c:       08f6a023        sw      a5,128(a3) # 30000080 <_erodata+0x1ffffe88>
186 38000020:       0846a603        lw      a2,132(a3)
187 38000024:       00c72023        sw      a2,0(a4) # 20000 <_fstack+0x1fa00>
188 38000028:       00178793        addi    a5,a5,1
189 3800002c:       00470713        addi    a4,a4,4
190 38000030:       feb796e3        bne     a5,a1,3800001c <fir+0x1c>
191 38000034:       300007b7        lui     a5,0x30000
192 38000038:       0007a783        lw      a5,0(a5) # 30000000 <_erodata+0x1ffffe08>
193 3800003c:       10000513        li      a0,256
194 38000040:       00008067        ret
```

圖 6:第二次優化的 fir.c 及對應的 assembly code

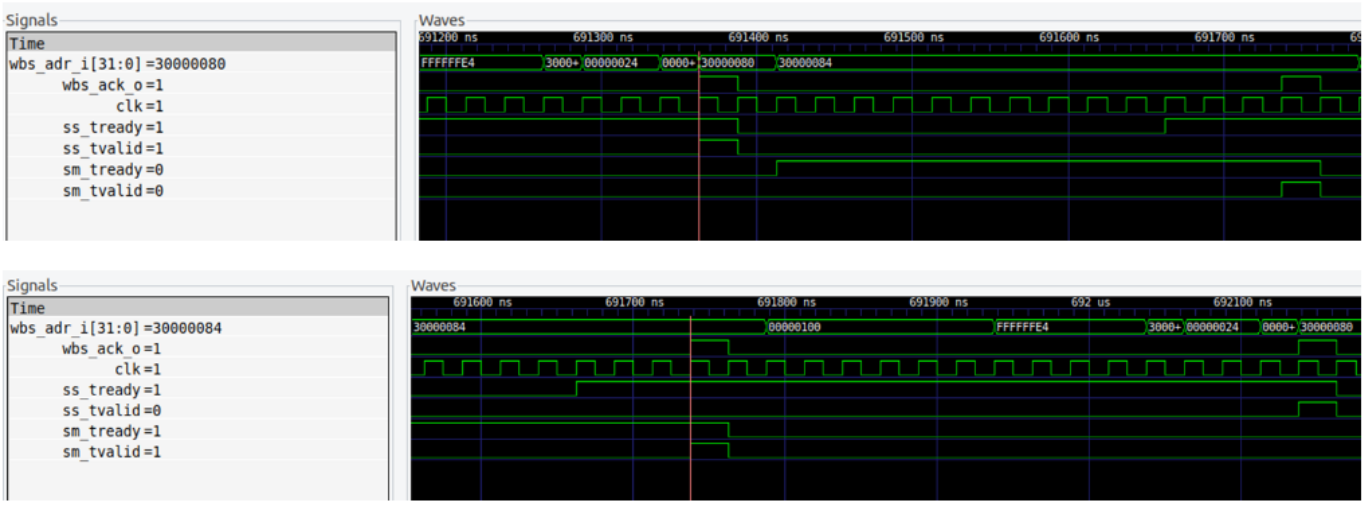


圖 7:第二次優化後的 waveform

經由第二次優化後的 x->y 花費的 cycle 數為 15 cycle 以及 y->x 花費的 cycle 數為 16 cycle，整體 throughput 為 31cycle。而且可以發現 cpu 完全不用至 bram 重新讀取指令。然而經由老師的指點，我們可以藉由調整 assembly code 的位置來優化。順帶一題我們可以發現其實 x->y 已經無法再優化主要是受限我們自己的硬體設計，所以我們主要優化 y->x 這段。由老師給我們的想法我們簡單的修改了 fir.c 至使 assembly code 的順序有些改變。

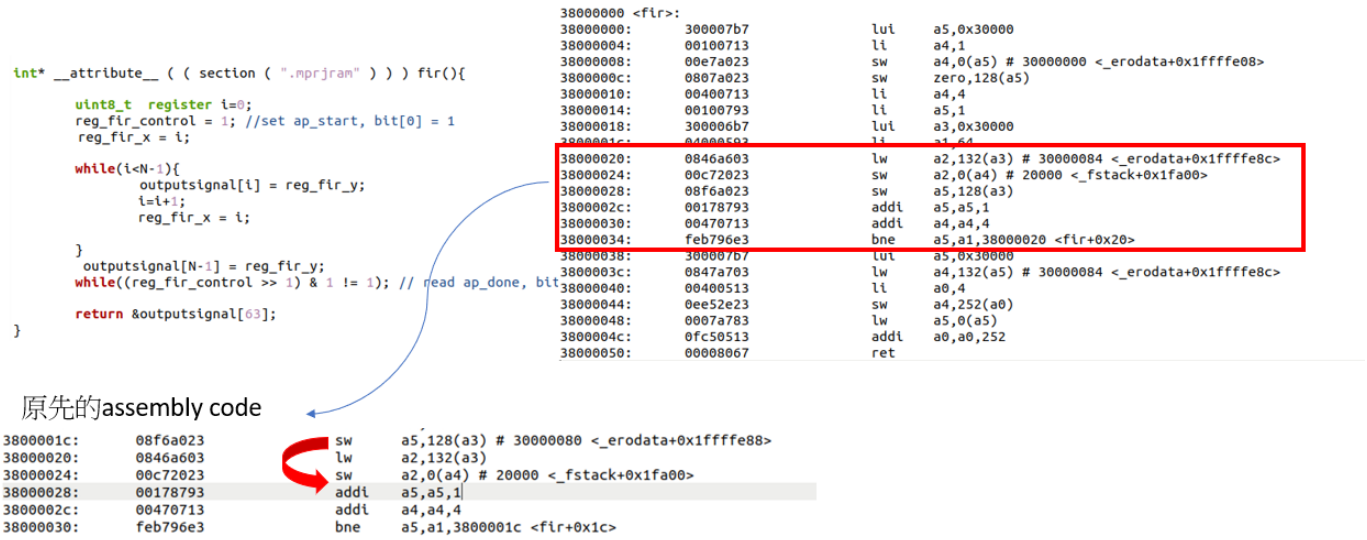


圖 8:第三次優化的 fir.c 及對應的 assembly code 以及對比之前的 assembly code

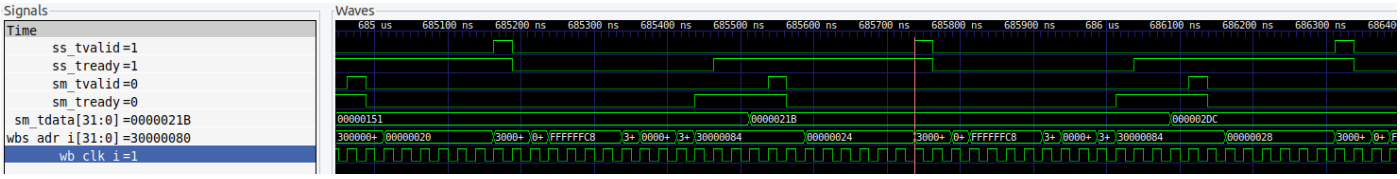


圖 9:第三次優化後的 waveform

改寫後 x->y 花費的 cycle 數為 15 cycle (這部分受限硬體設計關係)以及 y->x 花費的 cycle 數為 8 cycle(明顯比第二次優化時快 2 倍)。然而除了可以藉由更改 fir.c 來變更 assembly code 我們也可以利用.hex 更改對應指令的順序。

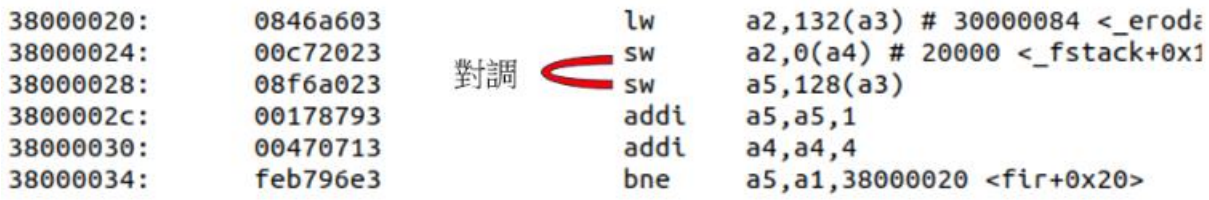


圖 10:assembly code 對調（sw a2, 0(a4) is stalled thus delays the following write x (sw a5, 128 (a3).）

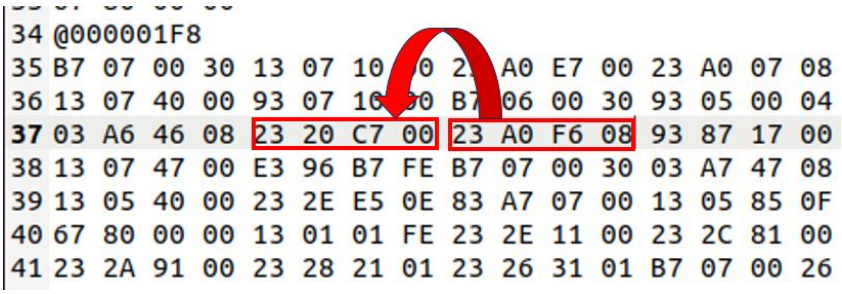


圖 11:hex file 中指令順序位置

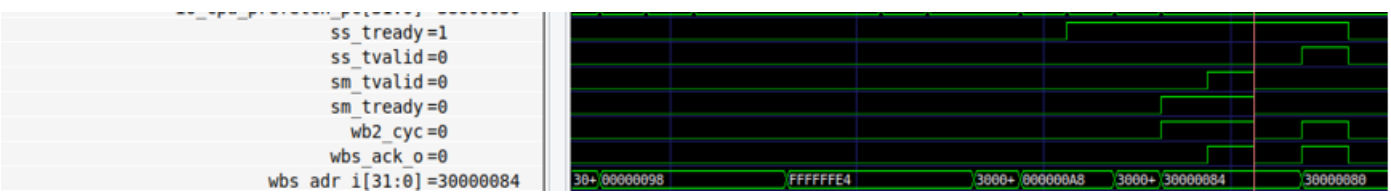


圖 12:第四次優化後的 waveform

所以我們目前可以做到 x->y 為 15 cycles，y->x 為 2 cycles，整體 throughput 為 15 cycle。

因一開始硬體設計上並沒有加一層 FF，所以每次做完 y 都要等到收到 x 才能繼續，這時極限只能做到 2+14 cycles per data 且在此狀況下還要花時間去設計究竟 x->y 還是 y->x 哪段要比較快來配合硬體。不過感謝台大同學的報告，從中發現加了一層 FF 確實可以再進一步優化。有了 FF 後，我們可以再運算的過程中也可以收 x，所以我們現在只要思考如何讓 smtready 之間的距離縮短即可。所以我試著再去縮短 smtready 之間的 cycle 數，最後做出來整體 throughput 為 12 cycle。

```
38000024:       0846a603                lw      a2,132(a3) # 30000084 <_erodata+0x1ffffe8c>
38000028:       00478793                addi    a5,a5,4
3800002c:       08e6a023                sw      a4,128(a3)
38000030:       fec7ae23                sw      a2,-4(a5)
38000034:       00170713                addi    a4,a4,1
38000038:       feb796e3                bne     a5,a1,38000024 <fir+0x24>
```

圖 13:第五次優化後的 while 迴圈對應的 assembly code

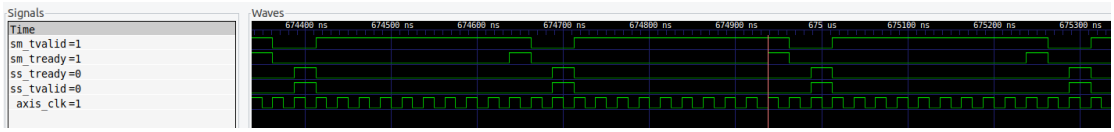


圖 14:第五次優化後的 waveform

補充說明:

1. 剛剛有講到 cpu 沒有再去 bram 抓 code，而我們猜測是由 cache 抓，那我們最後也證實了是由 cache 來運作，如下:

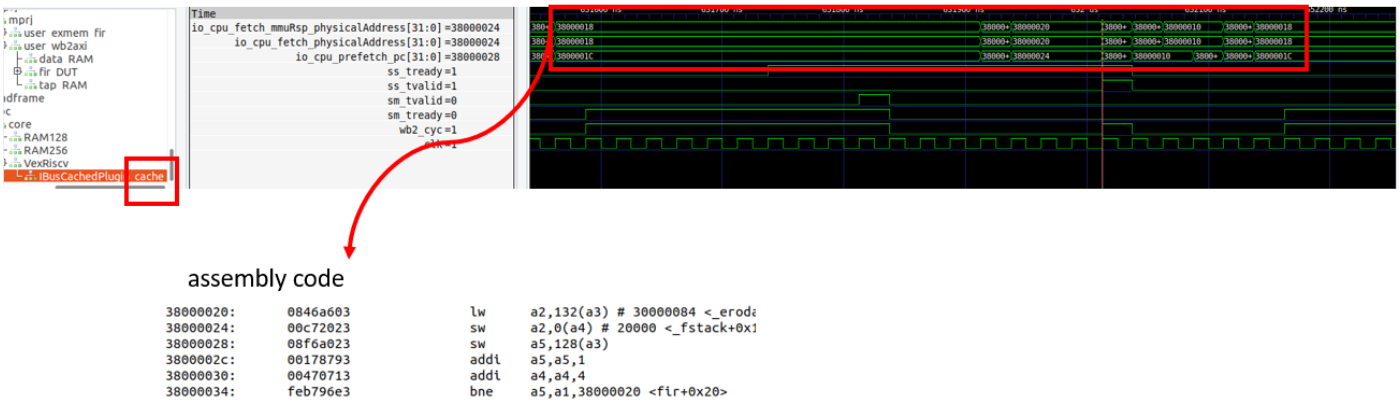


圖 15:cache 的 waveform