HW1: Real-time Debugging of a HW-SW Platform

學號:0711282 邱頎霖

INTRODUCTION

此次作業在原始 Dhrystone 不做任何修改下所得到的 DMIPS 為 0.72, 此篇 report 中將會先透過反組譯所得到的組合語言、Aquila 架構與 Vivado waveform 進行分析, 找出當前的缺點後, 以組合語言嵌入 Dhrystone 原始碼中實現優化的方法。

ANALYZE

反組譯後可以獲得 CODE. 1. 組合語言內容,搭配 waveform 可以清楚且快速的知道哪些時候發生 stall, 觀察整個 *strcpy function* 後, 可以知道主要發生 stall 的指令為 Fig. 1. 用紅線框選出來的兩個區塊。

stall data fetch

Fig. 1. 中第一個 stall 區塊發生 *stall_data_fetch*, 而發生主要原因為 *core_top.v* 中的 finite state machine 中的

$$dS\_nxt == d\_WAIT$$

往回追朔最終在 *decode.v* 中找到導致 *stall_data_fetch* 發生的情況為:

```
assign re = rv32_load | rv32_amo;
assign we = rv32_store | rv32_fencei;
```

decode.v 會往後不斷傳遞給下一個 pipeline stage, 在 execute stage 將訊號輸入到 memory stage 就會發生 stall。 總結, 以此次作業情況, 當要使用 *load* 或是 *store* 會 stall 所有 pipeline stage 1 cycle, 因此如果需要使用到這兩個指令時, stall 1 cycle 無法避免。

stall data hazard

第二個 stall 區塊除了發生上述的 *stall_data_fetch*, 還發生 *stall_data_hazard*, data hazard 的產生源自 pipeline 中某一指令使用到之前階段指令尚未產生的結果, 解決方法可以分為軟體方法與硬體方法。
軟體方法分為:

1. 重新排序程式碼使其沒有資料相依
2. 暫停 pipeline 使需要的結果產生後再執行

硬體方法為:

1. forwarding: 通過拉線的硬體方式,將需要的資料提早送到前面的pipeline stage

需要特別注意的是使用 forwarding 的方式如果遇到 load-use instruction, 必須要 stall 1 cycle, 如 Fig. 2. 當要load 資料時最少要等到 memory stage 。 根據 Aquila 原始碼中可以找到 *stall_data_hazard* 發生的源頭為 *decode.v* 為 CODE. 2. 中的內容, *re_o* 表示上一筆指令是否為load 指令, *rd_addr_o* 表示上一筆指令的目的暫存器,

FIG. 1. waveform after executing strcpy in Dhrystone

```
00001ea0 <strcpy>:
    1ea0: 0005c783 lbu   a5,0(a1)
    1ea4: 00050713 mv    a4,a0
    1ea8: 00078c63 beqz  a5,1ec0 <strcpy+0x20>
    1eac: 00170713 addi  a4,a4,1
    1eb0: 00158593 addi  a1,a1,1
    1eb4: fef70fa3 sb    a5,-1(a4)
    1eb8: 0005c783 lbu   a5,0(a1)
    1ebc: fe0798e3 bnez  a5,1eac <strcpy+0xc>
    1ec0: 00070023 sb    zero,0(a4)
    1ec4: 00008067 ret
```

CODE. 1. strcpy assembly code

當上一筆指令為load 指令且當前指令的來源暫存器與上一筆指令的目的暫存器相同時, 即為 load-use instruction, 因此需要 stall 1 cycle。 此處僅 stall program counter 與 fetch stage, 其他 pipeline stage 仍要繼續執行。 根據 FIG. 1. 我們可以驗證 CODE. 1. 中的 1ebc 這條指令在decode stage 時的確顯示 data hazard

根據 strcmp 的 waveform Fig. 2. 後可以發現其缺點與 strcpy 近乎相同, 都是含有 *stall_data_fetch* 與 *stall_data_hazard*, 因此我的想法是將優化的方法同時套用到這兩個 function 上理論上就可以比原始碼獲得更高的DMIPS.

根據上面的分析, 可以知道 load-use instruction 會導致額外 stall 1 cycle, 然而不如 *stall_data_fetch* 無法避免, 透過重新排列指令的方法可以將原本含有 load-use instruction 給消除, 因此重新排列指令將會是本次實現優化的目標。

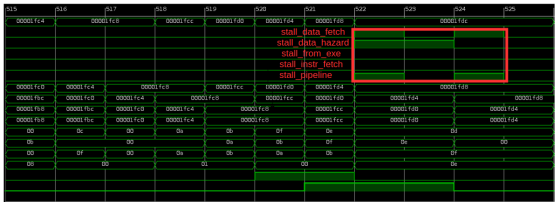

FIG. 2. waveform after executing strcmp in Dhrystone

```
wire is_rs1_rd_same = (rs1_addr_o == rd_addr_o)
    && (is_r_type || is_s_type || is_b_type
        || is_i_type || is_fence || is_csr_type);
wire is_rs2_rd_same = (rs2_addr_o == rd_addr_o)
    && (is_r_type || is_s_type ||

assign is_load_hazard_o =
    (is_rs1_rd_same | is_rs2_rd_same) && re_o;
```

CODE. 2. detect load-use instruction

IMPLEMENTATION

rescheduling

根據 CODE. 1. 可以先觀察是否可以實現指令重排。 初始化會先將要複製的字元 load 到暫存器 *a5*, 判斷是否為空字串, 是的話 branch 到 1ec0 處。
否的話執行下面直到字串完全複製結束:

1. 移動要複製的字串指針 *a4* 指向下一個要填入複製字元的再後面一個位置
2. 移動被複製的字串指針 *a1*,指向下一個要複製的字元
3. 將 *a5* 內容存到 *a4* 指向位置的前一個位置
4. 將下一個要複製的字元 load 到 *a5*
5. 利用 *a5* 內容判斷字串是否到尾端, 否的話繼續執行此迴圈

透過上面我們可以知道會發生 load-use instruction 是因為第五步驟每次在判斷字串是否複製結束前, 必須先將下一個要複製的字元給 load 出來, 因此不要讓 load-use instruction 出現的方法就是 每次先做複製後在更新兩個字串的指標, CODE. 3. 為改動後的組合語言

對於 strcmp 也使用同樣的概念進行優化。 原先 strcmp function 的步驟如下:

1. 初始化: 分別將兩個指標往前移動一個位置
2. 比對過程: 將兩個指標往後移動一個位置, 並分別 load 到暫存器, 進行比對
3. 若非相同, return 相對應的數值;否則繼續第二步驟進行比對

原始碼中產生 load-use instruction 的原因出現在比對過程中將字元 load 到暫存器後馬上接著比對, 因此優化方式為先指標指向的字元 load 到暫存器中, 更新指標使兩個指標分別指向字串下一個位置, 再將 load 到暫存器的字元比對後, 即可避免 load-use instruction 的產生。 優化過後的內容如 CODE. 4. 所顯示, 也避免 load-use instruction 的產生。

```
00001ea0 <strcpy>:
    1ea0:       0005c783 lbu a5,0(a1)
    1ea4:       00050713 mv     a4,a0
    1ea8:       00078c63 beqz a5,1ec0 <copy_end>

00001eac <copy>:
    1eac:       00f70023 sb     a5,0(a4)
    1eb0:       00158593 addi a1,a1,1
    1eb4:       0005c783 lbu a5,0(a1)
    1eb8:       00170713 addi a4,a4,1
    1ebc:       fe0798e3 bnez a5,1eac <copy>

00001ec0 <copy_end>:
    1ec0:       00070023 sb     zero,0(a4)
    1ec4:       00008067 ret
```

CODE. 3. optimized strcpy

```
00001fb8 <strcmp>:
    1fb8:       0080006f j 1fc0 <compare>

00001fbc <str_end>:
    1fbc:       02078663 beqz   a5,1fe8 <ret_z>

00001fc0 <compare>:
    1fc0:       00054783 lbu    a5,0(a0)
    1fc4:       0005c703 lbu    a4,0(a1)
    1fc8:       00150513 addi   a0,a0,1
    1fcc:       00158593 addi   a1,a1,1
    1fd0:       fee786e3 beq    a5,a4,1fbc <str_end>
    1fd4:       fff00513 li     a0,-1
    1fd8:       00e7f463 bleu   a4,a5,1fe0 <ret_o>
    1fdc:       00008067 ret

00001fe0 <ret_o>:
    1fe0:       00100513 li     a0,1
    1fe4:       00008067 ret

00001fe8 <ret_z>:
    1fe8:       00000513 li     a0,0
    1fec:       00008067 ret
```

CODE. 4. optimized strcmp

unroll

根據觀察, Dhrystone 中 string copy 每次都是 copy 長度為30 的字串, 因此可以利用將 loop 展開的方式以增高效能, 避免了判斷是否字串已經複製完、branch 等指令, 當然在這裡寫死複製長度後 string copy 在其他非字串長度為 30 的地方將會出現錯誤, 也不符合作業要求但是因為 DMIPS 有較為提昇因此在此提及。 將 string copy unroll 的過程如下:

1. 初始化, 包含使用另外一個指針進行複製的動作與第 1 次複製
2. 重複 29 次複製
3. 補 0 與 return

SUMMARY

根據我們的分析, load-use instruction 為此次可以優化的大方向, 我們實現重新排成指令的優化想法後將 DMIPS 從 0.72 上升到 0.77, 驗證了我們優化的方向與結果正確。 再多考慮 unroll 的情況下, 可以減省 branch prediction 錯誤的情況與 branch 的發生, 儘管不符合作業要求, DMIPS 最多可以上升至 0.81, 在 trace Aquila 的原始碼過程中, 除了 report 中提及的 forwarding, stall 型態與判斷外, 還有發現 Aquila 還有實做 branch prediction 的部份, 在以往計算機組織時 branch prediction 往往為理論上的知識, 甚至有些教授的作業內容也不包含 pipeline 的實做, 因此透過作業一也算是複習了計算機組織的內容,
最後較為奇怪的發現是加上

(mark_debug = "true" *) wire signal*

仍然有些 port 無法被顯示出來, 為此次作業中發現較為怪異的點。