Dual-Issue In-order RISC-V Processor

Jiun Hao Chen
dept. Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan
duncan11514@gmail.com

Pin Hsuan Shih
dept. Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan
vivian900509@gmail.com

Chun Hao Chang
dept. Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan
peter263658@gmail.com

I. Introduction

First, an in-order 6-stage pipeline dual-issue processor is designed with the support of RV32IM. The architecture of the processor is shown in Fig. 4.

The processor is a 32-bit dual-issue processor, which means the length of data the IF stage fetches doubled from 32 bits to 64 bits, and it can execute 2 instructions parallelly. In addition, we also implement a memory subsystem including I-cache and D-cache.

II. RTL Design

The 6-stage pipeline in the processor is divided into program counter (PC) stage, instruction fetch (IF) stage, instruction decode (ID) stage, issue solve (IS) stage, execute (EX) stage, write back (WB) stage, which show in Fig. 5. Moreover, the processor has control and status register (CSR), but it lacks the support of some instructions (ECALL, EBREAK, FENCE, FENCEI) and interrupts.

A. PC stage

The PC stage control the behavior of the program counter. Normally, the program counter (PC) should be PC+8 if the issue be dispatched ideally. In the real case, the control flow instructions are frequent; branch is one of them and we know that when the branch occur, the flush penalty will greatly influence the performance of the processor. Therefore, making a high accurate branch prediction is essential.

Here, we use a branch hit table (2-bit saturation predictor) with a branch target buffer to implement the branch prediction. The idea is that, when the branch occur, we will update the branch hit table and record the source and destination PC in branch target buffer. When the branch hit table predict the branch will be taken at the current PC, the next PC will be the target PC in branch target buffer instead of PC+8. The whole structure is shown in Fig. 1.

B. IF stage

IF stage is responsible for the instruction fetch. It needs to communicate with the I-cache to fetch the instruction successively. The most important part in this stage is to control the time to fetch the next instruction. Here, we implement a skid buffer Fig. 2 to block and control the fetch request. We fetch two instructions in one packet in one cycle, but the execution unit may not execute them simultaneously; hence, we need to stall the instruction fetch in some cases.

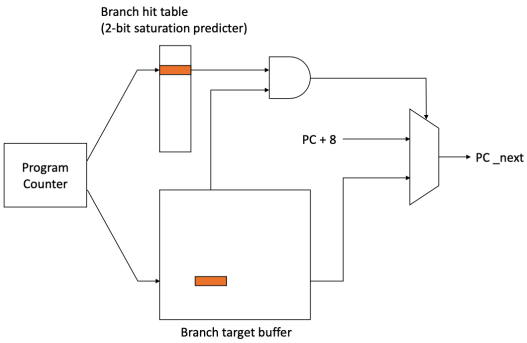

Fig. 1: Branch Prediction

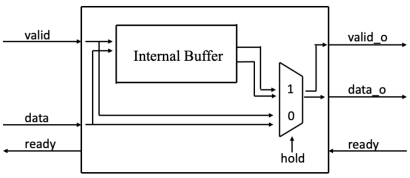

Fig. 2: Skid buffer

C. ID stage

In this stage, we decode the instruction to distinguish the execution unit (ALU, LSU, MUL, DIV, CSR) usage. After decoding, the PC, instructions and execution unit information will be pushed into an instruction queue. The instruction queue can store 4 instructions (2 packets), ensuring the maximum usage of execution units. The instruction queue is shown in Fig. 3.

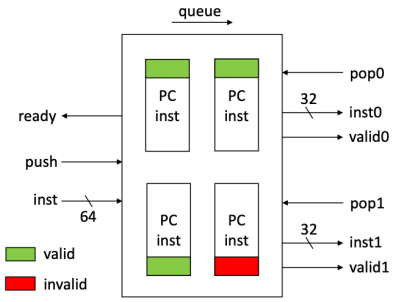

Fig. 3: Instruction queue

D. IS stage

To solve the data hazard and data forwarding within dual issue processor, we need to dispatch the instruction to corresponding execution units. Note that there are only one load save unit (LSU), multiply unit (MUL), control status register (CSR) and divide unit (DIV) in our design; therefore, we may not execute two instructions in one packet simultaneously.

The register file is included in this stage before dispatching the instruction to the execution units. It contains 4 read ports and 2 write ports with priority write and parallel read.

E. EX stage

This two stages contain all of the execution units including two ALU, one LSU, one DIV, one MUL and CSR unit. Their CPI (cycle per instruction) is different at all. How to control the pipeline is crucial in this part. We use a simple method to do that. Do not do anything except all of the execution unit finish their task. This method may be inefficient but it is easier to implement. The following list the function our execution units have.

1) ALU: Basic arithmetic logic unit, it perform all type of the arithmetic instruction in RV32I.
2) LSU: Load save unit, responsible for communicating with D-cache, loading or storing data.
3) DIV: This sub-module executes instructions DIV, DIVU, REM, REMU. It performs long division to return results of quotient or remainder. The number of total cycle is dynamic, which is the number of bits of the dividend.
4) MUL: This sub-module executes instructions MUL, MULH, MULHSU, MULHU. It only needs one cycle to get the result, but the timing will become longer.
5) CSR: Control and status register, this unit is responsible for control the CPU status, for instance, enable interrupt or not. In this works, we do not implement the exception and interrupt.

F. WB stage

This stage hold all data that will be written back to general purpose register file and CSR register.

G. Data Bus

Since there are many peripherals in our design like ROM, RAM, UART. How to easier transfer data crossing module is what we need to think. Here, we implement a simple data bus, using the 4-bit MSB to select where the data should be passed.

H. Sub-Memory System

In order to simulate the actual CPU operation. Our CPU memory is composed of the instruction and data cache inside of the CPU and the RAM, ROM, UART outside of the CPU. The structure of the I/D-cache is shown in Fig. 6, and our memory layout is shown in Fig. 7.

1) ROM: we assign 32KB to ROM, storing the instruction.
2) RAM: we assign 16KB to RAM, storing the data, lma data, bss data, stack data, etc.
3) ICACHE: ICACHE is 2-way set associative, 64 blocks each index, 8 words each block, 4byte per word, then the size of ICACHE is 4byte/word * 8 words/block * 64 blocks/way * 2 ways = 4KB.
4) DCACHE: DCACHE is 2-way set associative, 64 blocks each index, 8 words each block, 4byte per word, then the size of DCACHE is 4byte/word * 8 words/block * 64 blocks/way * 2 ways = 4KB.
5) LRU: When missing occurring, we reload the data from ROM or RAM. Because we have multi-entries, we use LRU (Least Recently Used) to decide which way should be flushed and reload new data.

I. Synthesis

All RTL modules are synthesizable, and we have two option can select when synthesizing.

1) BRANCH_PREDICTION: 0/1
2) SUPPORT_MULDIV: 0/1

Here, we use synthesis timing = 8.2(ns)

MULDIV \ PREDICTION	Disable	Enable
Disable	131646	172533
Enable	143677	184494

TABLE I: Synthesis Area Table (μm^2)

MULDIV \ PREDICTION	Disable	Enable
Disable	2.959	3.625
Enable	3.049	3.717

TABLE II: Synthesis Power Table (mW)

III. Verification

To verify the functionality of the processor, we perform riscv-tests to do a simple instruction-level tests including instruction logic, source and destination and bypassing.

The above test belongs to compliance test; in other words, different from verification. Normally, random instruction testing is a better way to verify the functionality of the processor. Google has a random instruction generator riscv-dv which can generate lots of instruction and golden pattern with the instruction set simulator(ISS) like Spike. However, riscv-dv works with the "vcs", we can only run the generator on the workstation. Some problem raise that the workstation lacks of some libraries when run the riscv-dv script. Therefore, we cannot verify our processor with that. Instead of random instruction verification, we run some tests generated from C code. The circumstances including arithmetic operation, memory pressure test and performance test. Trying to increase the testing coverage.

IV. Software

Assembly code acts as the bridge from hardware to software. To make the processor run, we need to know how to use it. There are 32 general purpose registers (x0∼x31) in RV32I.

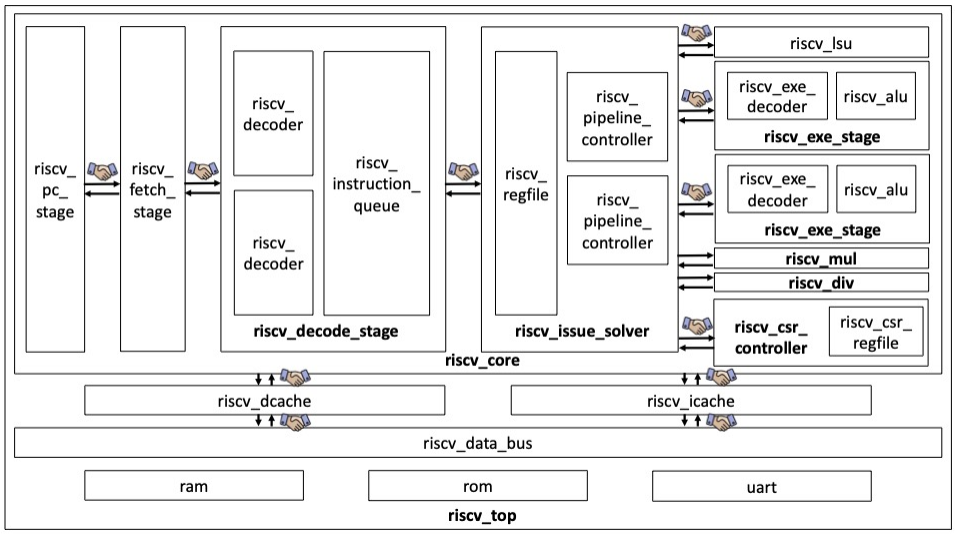

Fig. 4: CPU Architecture

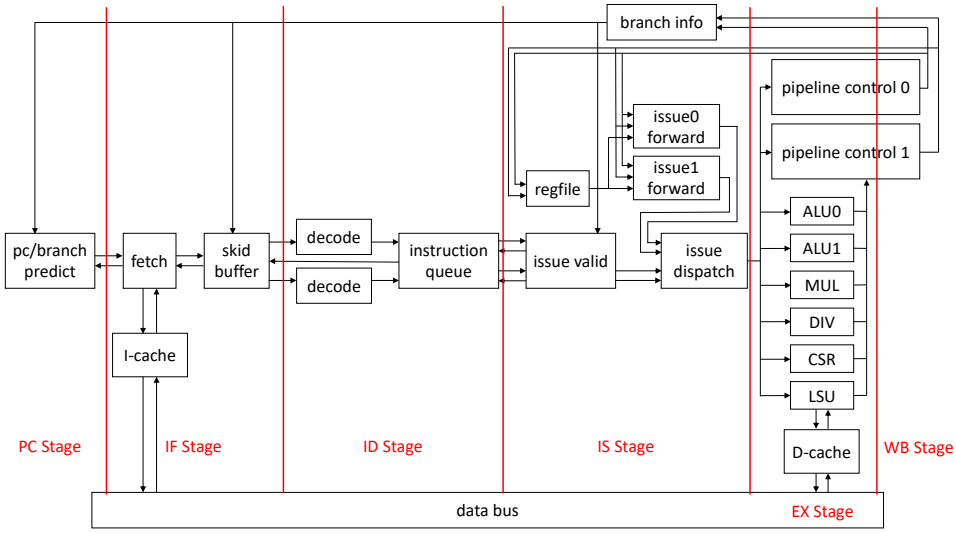

Fig. 5: 6-stage pipeline

In general, we will see its application binary interface (ABI) name, for instance, zero, ra, sp. There are lots of instruction in the RV32I such as **auipc, jal, bge, add**. How to combine them to build a program is a beautiful work.

Here is the segment of our boot up assembly code,

```
    .section .init;
    .globl _start;
    .type _start,@function

_start:
.option push
.option norelax
    la gp, __global_pointer$
.option pop
    la sp, _sp

    call main

loop:
    j loop
```

We can see that there are some code not belong to RISC-V ISA such as **.section, .type**, they belong to the GCC compiler.

Here, we use the Makefile to manage the compile steps, the following is part of our Makefile.

```
RISCV_ARCH := rv32im
RISCV_ABI := ilp32
RISCV_MCMODEL := medlow

TARGET = coremark

CFLAGS += -O2

C_SRCS := main.c

RISCV_PATH := /tools/riscv64-unknown-elf-gcc/
RISCV_GCC  := $(abspath $(RISCV_PATH)/bin/riscv64-unknown-elf-gcc)

LINKER_SCRIPT := link.lds
ASM_OBJS := $(ASM_SRCS:.S=.o)
C_OBJS := $(C_SRCS:.c=.o)

LINK_OBJS += $(ASM_OBJS) $(C_OBJS)
LINK_DEPS += $(LINKER_SCRIPT)

$(TARGET): $(LINK_OBJS) $(LINK_DEPS) Makefile
    $(RISCV_GCC) $(CFLAGS) $(INCLUDES) $(LINK_OBJS) -o $@ $(LDFLAGS)
    $(RISCV_OBJCOPY) -O binary $@ $@.bin

$(C_OBJS): %.o: %.c
    $(RISCV_GCC) $(CFLAGS) $(INCLUDES) -c -o $@ $<
```

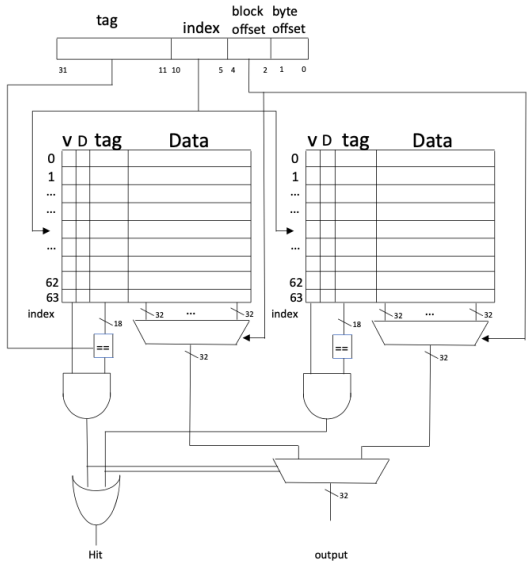

Fig. 6: Cache architecture

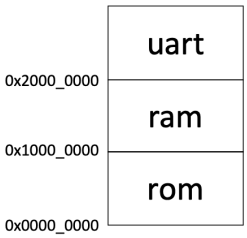

Fig. 7: Memory layout

There is an important file (link.lds) refer in the Makefile, it is a linker script define how to map the memory layout and control information to the output file. The following is part of our linker script,

```
OUTPUT_ARCH( "riscv" )
ENTRY(_start)

MEMORY
{
  flash (xa!ri) : ORIGIN = 0x00000000, LENGTH = 32K
  ram (wxa!ri)  : ORIGIN = 0x10000000, LENGTH = 16K
}

SECTIONS
{
   __stack_size = DEFINED(__stack_size) ? __stack_size : 8K;

  .text           :
  {
    *(.text .text.*)
  } >flash AT>flash

  .stack ORIGIN(ram) + LENGTH(ram) - __stack_size :
  {
    PROVIDE( _heap_end = . );
    . = __stack_size;
    PROVIDE( _sp = . );
  } >ram AT>ram

}
```

We can see that there are two blocks in the linker script. Memory describes the memory such as ROM, RAM, etc. Section describes how to map the input section to the output section. With the Makefile we provided, we can build executable programs run on our processor.

V. Back-end Design

After finishing the front-end design, we began to run APR according to the steps in the lab10 - lab12. First of all, after finishing post layout, there was no violation path in the routing, but there was setup-hold error when running post simulation. In the beginning, we thought it might be a problem with the circuit, because we cannot pass the gate level simulation. Later, by adjusting the fan-out in settings, we successfully passed the gate level simulation test, but the post simulation still the same. We hoped to find the problem through LEC tool. There is no problem with the RTL, and the gate level simulation also passed, so we thought that the RTL design might cause the error. We adjusted the RTL design, add some input buffers and output buffers, and finally passed the post layout simulation. As for the P&R timing, we can see that the target timing is 8ns, but at the end, we get 10ns in P&R. This reason is that the critical path in the whole design is multiplication. A 32-bit $\times$ 32-bits multiplier needs about 8.2ns, which is the limit of our synthesis timing, making the P&R timing worser. We have tried to make the multiplier pipelined. The original design only need one cycle to perform the multiplication. After pipelining, we can get a shorter critical path and better synthesis and P&R time. But this little revise will make the performance drop up to 20%; therefore, we decide to obtain the lower P&R timing to get the better performance.

A. P&R result

Timing (ns)	Core utilization (%)	Total power (mW)	Total area (μm^2)
10	90.097	1.675	204773.184

TABLE III: P&R result table

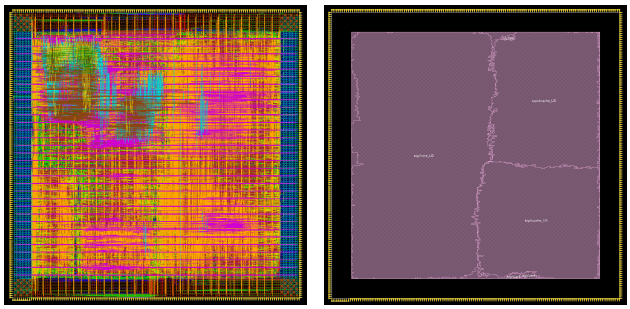

Fig. 8: Cache architecture

B. Power analysis

Before P&R, we do pre-layout power analysis to roughly estimate the power neeeded. After we finish the P&R, we estimate the power by pre-sim waveform, Therefore, we can know the result faster and closer to reality. FIially, we can estimate the power by post-sim waveform to get the most accounts result. The result is

Net switching power	Cell internal power	Total power
7×10^{-4}	9.694×10^{-4}	1.675×10^{-3}

TABLE IV: P&R power table (W)

VI. Performance

We use Coremark to measure our performance. Coremark is a benchmark that measures the performance of central processing units (CPU) used in embedded systems. The code is written in C and contains implementations of the following algorithms: list processing (find and sort), matrix manipulation (common matrix operations), state machine (determine if an input stream contains valid numbers), and CRC.

Fig. 9 is our Coremark report with dual issue and branch prediction enable and Table. VI is our comparison table with different issue unit and branch prediction setting.

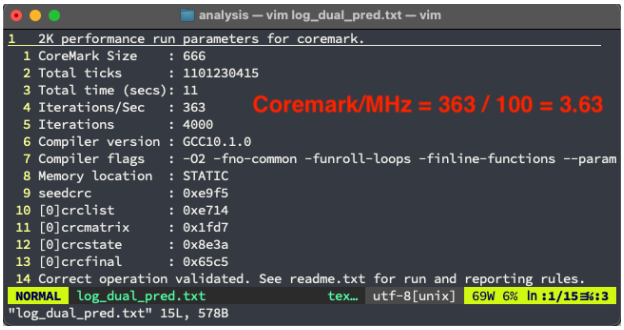

Fig. 9: Dual issue with branch prediction

Issue \ Prediction	Disable	Enable
Single	2.60	3.01
Dual	3.05	3.63

TABLE V: Coremark comparison table

For the number of issue unit, we can see that after enable dual issue, the performance will improve $17 \sim 20\%$

1) Prediction enable: $(3.05 - 2.60)/2.60 = 0.173$
2) Prediction disable: $(3.63 - 3.01)/3.01 = 0.206$

For the branch prediction, we can see that after enable branch prediction, the performance will improve $16 \sim 19\%$

1) Single issue: $(3.05 - 2.60)/2.60 = 0.158$
2) Dual issue: $(3.63 - 3.05)/3.05 = 0.190$

A. Compare with similar work

Processor	Memory sub-system	issue	MHz	CoreMark/MHz
ours	I/D-cache	2	100	3.63
tinyriscv	No	1	50	2.40
biriscv	TCM	2	-	4.01
STM32L562 rev A	No	2	110	4.02
STM32L4R5	No	1	120	3.41
STM32F417IGt6	No	1	168	2.91

TABLE VI: Coremark comparison table

VII. Project Summary

Aspect	Initial plan	Final result
Throughput (MHz)	125MHz	100MHz
Core utilization (%)	$\geq 90\%$	90.097%
P&R Timing	$\leq timing * 1.1$	$10 > 8.2 * 1.1 = 9.02$
Area	$100k\mu m^2$	$205k\mu m^2$

TABLE VII: Project summary

VIII. Work distribution and contribution

1) **Chun Hao Chang: Memory sub-system, Report** About the Memory subsystem, we refer to the structure in the course of Computer Structure, and redesigned the structure to match our function.
2) **Jiun Hao Chen: Riscv-Core, Software, APR, Report** The micro architecture of the processor refer to [1], I redesign lots of module and rearrange the pipeline timing to reduce the critical path. As for software, we use the file structure from [2] and revise some script to make the whole flow can fit our processor design. We design a high speed data bus which can maximize the utilization of the peripheral usage.
3) **Pin Hsuan Shih: Verification, APR, Report** The divider refer to [1], multiplier refer to [1], uart refer to [5], then I redesigned some part to better meet our needs.

References

[1] Ultraembedded. (n.d.). Biriscv. GitHub. https://github.com/ultraembedded/biriscv
[2] Liangkangnan. (n.d.). Tinyriscv. GitHub. https://github.com/liangkangnan/tinyriscv
[3] Yosyshq. (n.d.). Picorv32. GitHub. https://github.com/YosysHQ/picorv32
[4] Prof. onur mutlu. (n.d.). Computer Architecture Lecture 5: Advanced Branch Prediction. https://course.ece.cmu.edu/ ece740/f15/lib/exe/fetch.php?media=18-740-fall15-lecture05-branch-prediction-afterlecture.pdf
[5] Jamieiles. (n.d.). Uart. Github. https://github.com/jamieiles/uart