**Ace21064 Microprocessor**

Architecture Specification

**r0p1**

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Description** | **Author** |
| 0.1 | 10/13/2016 | Initial version | Yijun LI |
|  |  |  |  |

**Content**

[1 Ace21064 Core Basics 8](#_Toc495264679)

[1.1 Overview 8](#_Toc495264680)

[1.2 Features 8](#_Toc495264681)

[1.3 Pipeline architecture of Ace21064 10](#_Toc495264682)

[2 Level 1 Instruction Cache (iCache) 11](#_Toc495264683)

[2.1 Overview 11](#_Toc495264684)

[2.1.1 Introduction 11](#_Toc495264685)

[2.1.2 Architecture 12](#_Toc495264686)

[2.1.3 Signals 12](#_Toc495264687)

[2.1.4 Submodules 13](#_Toc495264688)

[2.1.5 Timing 13](#_Toc495264689)

[2.2 Instruction Alignment Unit 13](#_Toc495264690)

[2.2.1 Introduction 13](#_Toc495264691)

[2.2.2 Architecture 13](#_Toc495264692)

[2.2.3 Signals 13](#_Toc495264693)

[2.2.4 Submodules 13](#_Toc495264694)

[2.2.5 Timing 13](#_Toc495264695)

[3 Instruction Fetch Unit (IFU) 14](#_Toc495264696)

[3.1 Overview 14](#_Toc495264697)

[3.1.1 Introduction 14](#_Toc495264698)

[3.1.2 Architecture 14](#_Toc495264699)

[3.1.3 Ports 16](#_Toc495264700)

[3.1.4 Submodules 17](#_Toc495264701)

[3.2 Branch decode Unit (BrDEC) 19](#_Toc495264702)

[3.2.1 Introduction 19](#_Toc495264703)

[3.2.2 Architecture 19](#_Toc495264704)

[3.2.3 Ports 19](#_Toc495264705)

[3.2.4 Branch Decoder Ways(brdec\_way) 20](#_Toc495264706)

[3.2.4.1 Introduction 20](#_Toc495264707)

[3.2.4.2 Ports 20](#_Toc495264708)

[3.3 Branch Prediction Unit (BPD) 22](#_Toc495264709)

[3.3.1 Introduction 22](#_Toc495264710)

[3.3.2 Architecture 22](#_Toc495264711)

[3.3.3 Ports 23](#_Toc495264712)

[3.3.4 Submodules 24](#_Toc495264713)

[3.3.5 Branch History Table (BHT) 25](#_Toc495264714)

[3.3.6 Pattern History Table (PHT) 26](#_Toc495264715)

[3.3.7 27](#_Toc495264716)

[3.4 Return Address Stack (RAS) 27](#_Toc495264717)

[3.4.1 Introduction 27](#_Toc495264718)

[3.4.2 Architecture 27](#_Toc495264719)

[3.4.3 Ports 27](#_Toc495264720)

[3.5 Branch Target Buffer (BTB) 28](#_Toc495264721)

[3.5.1 Introduction 28](#_Toc495264722)

[3.5.1 Ports 29](#_Toc495264723)

[3.5.2 BTB Way 30](#_Toc495264724)

[3.5.3 Ports 31](#_Toc495264725)

[3.6 Branch Ordering Buffer (BOB) 31](#_Toc495264726)

[3.6.1 Introduction 31](#_Toc495264727)

[3.6.2 Architecture 32](#_Toc495264728)

[3.6.3 Ports 32](#_Toc495264729)

[4 Instruction Decoder Unit(IDU) 33](#_Toc495264730)

[4.1 Overview 33](#_Toc495264731)

[4.1.1 Introduction 33](#_Toc495264732)

[4.1.2 Submodules 33](#_Toc495264733)

[4.2 Instruction Buffer 33](#_Toc495264734)

[4.2.1 Introduction 33](#_Toc495264735)

[4.2.2 Architecture 34](#_Toc495264736)

[4.2.3 Ports 34](#_Toc495264737)

[4.3 Instruction Decoder 34](#_Toc495264738)

[4.3.1 Introduction 34](#_Toc495264739)

[4.3.2 Architecture 34](#_Toc495264740)

[5 Register Renaming Unit (RRU) 35](#_Toc495264741)

[5.1 Overview 35](#_Toc495264742)

[5.1.1 Introduction 35](#_Toc495264743)

[5.1.2 Signals 36](#_Toc495264744)

[5.1.3 Submodules 36](#_Toc495264745)

[5.2 Register Alias Table (RAT) 37](#_Toc495264746)

[5.3 Register Free List (RFL) 37](#_Toc495264747)

[5.3.1 Ports 38](#_Toc495264748)

[5.4 Data dependency Checker 39](#_Toc495264749)

[5.5 Memory dependency predictor 39](#_Toc495264750)

[5.5.1 Introduction 39](#_Toc495264751)

[5.5.2 Architecture 39](#_Toc495264752)

[6 Instruction Schedule Unit (ISU) 40](#_Toc495264753)

[6.1 Overview 40](#_Toc495264754)

[6.1.1 Introduction 40](#_Toc495264755)

[6.1.2 Features 41](#_Toc495264756)

[6.2 Reservation Station 41](#_Toc495264757)

[6.2.1 Signals 42](#_Toc495264758)

[6.2.2 Submodules 42](#_Toc495264759)

[7 Physical Register File (PRF) 43](#_Toc495264760)

[7.1 Overview 43](#_Toc495264761)

[7.1.1 Introduction 43](#_Toc495264762)

[8 Execution Unit 44](#_Toc495264763)

[8.1 Overview 44](#_Toc495264764)

[8.1.1 Introduction 44](#_Toc495264765)

[9 Load Store Unit (LSU) 45](#_Toc495264766)

[9.1.1 Overview 45](#_Toc495264767)

[9.1.2 Introduction 45](#_Toc495264768)

[10 Level 1 data Cache (dCache) 46](#_Toc495264769)

[10.1.1 Overview 46](#_Toc495264770)

[10.1.2 Introduction 46](#_Toc495264771)

[11 Retire Unit 47](#_Toc495264772)

[11.1 Overview 47](#_Toc495264773)

[11.1.1 Introduction 47](#_Toc495264774)

[11.2 Interrupt 48](#_Toc495264775)

[11.3 Branch Misprediction Recovery 48](#_Toc495264776)

[12 Coprocessor 50](#_Toc495264777)

[12.1 Overview 50](#_Toc495264778)

[12.2 Program Counter Generator (PCGen) 50](#_Toc495264779)

[12.2.1 Introduction 50](#_Toc495264780)

[12.2.2 Architecture 50](#_Toc495264781)

[12.2.3 Ports 50](#_Toc495264782)

[12.3 Power Management (PowerM) 51](#_Toc495264783)

[12.3.1 Introduction 51](#_Toc495264784)

[12.3.2 Architecture 51](#_Toc495264785)

[12.3.3 Ports 51](#_Toc495264786)

**Table of Figures**

[Figure 1‑1 Block Diagram of Ace21064 7](#_Toc495264787)

[Figure 1‑2 Ace21064 Integer Pipeline 9](#_Toc495264788)

[Figure 2‑1 Instruction Cache Overview 10](#_Toc495264789)

[Figure 2‑2 ace\_icache block diagram 11](#_Toc495264790)

[Figure 3‑1 Instruction Fetch Unit block diagram 13](#_Toc495264791)

[Figure 3‑2 PC geneerator 15](#_Toc495264792)

[Figure 3‑3 Tournament predictor functional diagram 21](#_Toc495264793)

[Figure 3‑4 pipelined tournament predictor 22](#_Toc495264794)

[Figure 3‑5 branch target buffer microarchitecture 28](#_Toc495264795)

[Figure 5‑1 Register Renaming Unit 35](#_Toc495264796)

[Figure 10‑1 Ace21064 level 1 data cache overview 45](#_Toc495264797)

[Figure 12‑1 PC generate logic 49](#_Toc495264798)

**Table of Tables**

[Table 3‑1 *ace\_fetch* unit port signals 15](#_Toc495264799)

[Table 3‑2 *ace\_fetch* sub-modules 16](#_Toc495264800)

[Table 3‑3 *brdec* module port signals 18](#_Toc495264801)

[Table 3‑4 *brdec\_way* module port signals 19](#_Toc495264802)

[Table 3‑5 branch prediction unit ports 23](#_Toc495264803)

[Table 3‑6 *bpd* sub-modules 23](#_Toc495264804)

[Table 3‑7 BHT Port List 24](#_Toc495264805)

[Table 3‑8 BHT Port List 25](#_Toc495264806)

[Table 3‑9 branch prediction unit ports 28](#_Toc495264807)

[Table 4‑1 ace\_decode sub-modules 32](#_Toc495264808)

[Table 4‑2 instruction buffer ports 33](#_Toc495264809)

[Table 5‑1 RRU port signals 35](#_Toc495264810)

[Table 5‑2 ace\_rename sub-modules 35](#_Toc495264811)

[Table 6‑1 reservation stationh port signals 41](#_Toc495264812)

[Table 6‑2 ace\_rename sub-modules 41](#_Toc495264813)

# Ace21064 Core Basics

## Overview

ACE21064 is a RISCV ISA based hex-issue superscalar processor core, which implemented with 12-stage integer pipeline, here is the block diagram:



Figure ‑ Block Diagram of Ace21064

## Features

* hex-issue out-of-order 12-stage integer pipeline
* 64bit virtual address, 40bits physical address (can be extern to 64 bits)
* 16KB Virtually indexed physically tagged 2-way set associative instruction cache
* iCache with LRU replace strategy
* 40-entry full associative instruction TLB, 32 entries for 4K page, and 8 entries for 2M page
* G-Share and Pap Tournament Hybird branch predictor
* 8-entry return address stack(RAS)
* 32-entry instruction buffer
* 4-instruction can be decoded in one CPU cycle (machine width 4)
* 4-wide rename from 80 physical registers file, which is 65-bits each, renaming in unified method
* the physical register file can be accessed from 11 read port or 7 write port
* speculative and architectural rename maps, with respective RAT and FREELIST
* ROB-Based mechanism for branch misprediction recovery (weak but simple)
* Store Set based memory dependence prediction
* 32-entry scheduler used in issue stage, up to 6 instructions can be selected for execution (issue width 6)
* speculative wakeup mechanism
* instruction replay
* centralized and distributed combined reservation station is used for efficiency
* Non-data-capture architecture is used in register read phase
* the reservation station is designed in compressed mode
* 2 simple ALU is used for add sub shift etc.
* 1 complex ALU for multiply and divide operation
* 1 branch unit designed for branch instructions
* 2 address generation unit, designed for load and store operation
* 16-entry load and store queue, to keep data before data cache access
* 32KB Virtually indexed physically tagged 2-way set associative dual-port level 1 data cache
* level 1 data cache is designed with 8 interleaved banks each way, 2KB each bank
* Data cache write strategy: hit: write back; miss: write allocate; with LRU replace strategy
* Two full associative data TLBs for each cache port.
* Data TLB have 40-entry, 32 entries for 4K page, and 8 entries for 2M page
* Pipelined data cache write operation.16-entry write buffer
* 64-entry reorder buffer is used to support 8 wide retirement

## Pipeline architecture of Ace21064

Ace21064 pipeline architecture is detailed here, there are 12 pipeline stages:

* Stage0: get the pc\_f0 from PC generator, and use this PC to access iCache, iTLB, BTB, RAS and BPD simultaneously.
* Stage1:



Figure ‑ Ace21064 Integer Pipeline

# Level 1 Instruction Cache (iCache)

## Overview

Level 1 instruction cache of Ace21064 is a two way-associative, virtually indexed and physically tagged cache, each data block has 32 bytes, 128 cache lines in each way, 40bits physical address is designed for temporary, so 28bits higher address range is kept in tag array of each cache line, Total size of l1icache is 16KB, and memory is managed in 4KB page size.

### Introduction

In order to reduce the hit time of level 1 cache, virtually indexed, physically tagged cache architecture is used, as widely used in modern processor. The overview of instruction cache system of Ace21064 is showing below:



Figure ‑ Instruction Cache Overview

Ace21064 is 64bits architecture, and physical address is remained to 40bits for current design. Totally 1TB address range can be accessed, the memory system is managed in 4KB page size, so the low 12bits of virtual address is used as page offset, due to the two-way associative cache organization the total cache capability is 16KB large. To support the 4-issue pipeline, the fetch stage is designed in 8 instructions per-fetch, so the instruction cache is designed to 8 words per-entry for design convenient. Totally 32bytes needs 5 bits to index each byte in a cache entry, named block offset. The width of index segment can be calculate following the formula:

Or following the page width, 7 bits are left for cache line index, so there are 128 cache line in each way.

According the width of physical address, 28bits need to be kept in tag array of icache, which will be compared with the hit-value of iTLB to determine the fetch operation is hit or not.

### Architecture

Figure ‑ ace\_icache block diagram

### Signals

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| Common Interface | | | |
| clock | input | 1 | system clock |
| reset\_n | input | 1 | system reset low active |
| pc\_s0\_i | input | 64 | program counter in pipeline stage 0 |
| pc\_s1\_i | input | 64 | program counter in pipeline stage 1 |
| Interface Between MMU and l1icache | | | |
| itlb\_phystag0\_i | input | 28 | physical tag from itlb0 |
| itlb\_phystag1\_i | input | 28 | physical tag from itlb1 |
| itlb\_miss0\_i | input | 1 | itlb miss flag |
| itlb\_miss1\_i | input | 1 | itlb miss flag |
| l1icache output | | | |
| ic\_stall\_s0\_o | output | 1 | l1icache stall flag |
| ic\_inst\_s1\_o | output | 256 | l1icache data out |
|  | | | |
|  |  |  |  |

### Submodules

### Timing

## Instruction Alignment Unit

### Introduction

### Architecture

### Signals

### Submodules

### Timing

# Instruction Fetch Unit (IFU)

## Overview

Instruction fetch is responsible for providing a continuous instruction stream to the rest of the pipeline, fetch stage achieves a fetch bandwidth of 8 instructions from level1 instruction cache. And a dynamic branch predictor to speculate on the outcome of a branch instruction. The branch prediction mechanism is composed of four major hardware structures: branch predictor (BPD), branch target buffer (BTB), return address stack (RAS), and branch ordering buffer (BOB) etc.

### Introduction

The IFU is one of the most performance related part in whole design, in current implementation 2 pipeline stages are designed to improve the system frequency.

### Architecture

The basic function of the fetch unit is to produce an address every cycle to fetch instructions from the instruction cache and to fetch instructions from the instruction cache to feed them to the rest of pipeline. And the instruction cache is accessed in parallel with the next PC being generated. The problem is that we need to know which instructions being fetched from the I-cache are branches and their target addresses. To produce the PC of the next fetch block in the next cycle, so as not to delay fetching the next fetch block.

Here is the functional diagram of fetch stage



Figure ‑ Instruction Fetch Unit block diagram

As we have shown in figure 3-1, our design has a pipelined fetch, which has two stages to reach the performance requirement.

Fetch Stage 0 consists of the branch prediction logic (partly), branch target buffer, and return address stack, along with the logic for fetching instructions from the instruction cache (I-cache). I-cache and branch prediction logic in Fetch Stage 0 are shown separately since they will be pipelined separately. Fetch Stage 1 has the logic for extracting the fetch block from the instructions supplied by the I-cache as well as the logic for recovering from a BTB miss.

The BTB is the table that stores the branch PC with its taken target and its type (unconditional jump, call, return, or conditional branch). If a PC hits in the BTB, it will prove that the corresponding instruction is a branch and will provide its taken target and its type. Considering our prediction logic is pipelined in two fetch stages, so we can’t get the prediction of each conditional branch until stage 1 of fetch unit, we attached a 2bit saturate counter to get the branch direction result in fetch cycle 0, If the more complex and accurate predictor gets the same direction, no flush operation will take place, or else the next PC will be changed .All this information service the next PC generation.

The RAS is the stack that used to predict the targets of returns. A routine can be called from many call sites and the branch prediction logic has to produce the return site corresponding to the call. Calls and returns follow a simple last-in-first-out (LIFO) behavior. Thus, we employ a small stack structure named as return address stack (RAS). Calls pushes the address of the return site onto the stack and returns pop them out. When the BTB signals that the instruction being fetched is a return, it takes the target provided by the RAS

Fetch Stage 1 has the logic for extracting the fetch block from the instructions supplied by the I-cache named BrDEC, as well as the logic for recovering from a BTB miss. The instructions that are fetched and aligned in this cycle are partial decoded. Partial decode is used to detect branches and to calculate their target addresses. In case the BTB missed on the first predicted-taken branch, if any, among the fetched instructions, then a recovery signal is sent to Fetch Stage 0 to restart from the missed branch’s taken-target. Fetch Stage 1 also maintains a FIFO Branch ordering Buffer (BOB), which has all the in-flight branches in the processor for updating the BPD non-speculatively in program order and rolling back BPD state such as the branch history register (BHR) used by the branch prediction algorithm. It also maintains a checkpointed copy of the Return Address Stack (RAS). It is required to correct the RAS when there is a BTB miss on a call or return. It sends the required instruction block down the pipeline.

Here is the simplified PC generator diagram, more details of PC generator can be found in *chapter coprocessor*



Figure ‑ PC geneerator

### Ports

Table ‑ *ace\_fetch* unit port signals

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| clock | input | 1 | system clock |
| reset\_n | input | 1 | system reset low active |
| pc\_f0 | input | 64 | program counter in fetch stage 0 |
| pc\_f1 | input | 64 | program counter in fetch stage 1 |
| signals from retire stage | | | |
| flush\_rt\_i | input | 1 | flush signal from retire stage |
| flush\_pc\_rt\_i | input | 64 | flush PC from retire stage |
| brcond\_vld\_rt\_i | input | 1 | condition branch retired flag |
| brindir\_vld\_rt\_i | input | 1 | indirect branch retired flag |
| brdir\_rt\_i | input | 1 | branch direction from retire stage |
| Signals from iCache | | | |
| inst\_align\_i | input | 256 | instruction fetch group1 |
| icache\_stall\_i | input | 1 | indicate the stall status of icache (icache miss) |
| Temporary | | | |
| instbuf\_full\_i | input | 1 | come from instruction buffer between fetch stage1 and decode stage |
| Output signal for pc generator | | | |
| branch\_pc\_o | output | 64 | branch target pc from BTB or RAS (fetch0 stage) |
| override\_pc\_o | output | 64 | override pc value from fetch1 stage |
| override\_vld\_o | output | 1 | override valid flag from fetch1 stage |
|  |  |  |  |
| Output signal for instruction buffer | | | |
| inst0\_vld\_d0\_o | output | 1 | instruction0 valid signal registered from fetch stage1 |
| inst1\_vld\_d0\_o | output | 1 | instruction1 valid signal registered from fetch stage1 |
| inst2\_vld\_d0\_o | output | 1 | instruction2 valid signal registered from fetch stage1 |
| inst3\_vld\_d0\_o | output | 1 | instruction3 valid signal registered from fetch stage1 |
| inst4\_vld\_d0\_o | output | 1 | instruction4 valid signal registered from fetch stage1 |
| inst5\_vld\_d0\_o | output | 1 | instruction5 valid signal registered from fetch stage1 |
| inst6\_vld\_d0\_o | output | 1 | instruction6 valid signal registered from fetch stage1 |
| inst7\_vld\_d0\_o | output | 1 | instruction7 valid signal registered from fetch stage1 |
| inst0\_d0\_o | output | 32 | aligned instruction0 registered from fetch stage1 |
| inst1\_d0\_o | output | 32 | aligned instruction1 registered from fetch stage1 |
| inst2\_d0\_o | output | 32 | aligned instruction2 registered from fetch stage1 |
| inst3\_d0\_o | output | 32 | aligned instruction3 registered from fetch stage1 |
| inst4\_d0\_o | output | 32 | aligned instruction4 registered from fetch stage1 |
| inst5\_d0\_o | output | 32 | aligned instruction5 registered from fetch stage1 |
| inst6\_d0\_o | output | 32 | aligned instruction6 registered from fetch stage1 |
| inst7\_d0\_o | output | 32 | aligned instruction7 registered from fetch stage1 |

### Submodules

Table ‑ *ace\_fetch* sub-modules

|  |  |  |
| --- | --- | --- |
| Num | Module | Description |
| 1 | **brdec** | pre-decode logic, used to get branch information in instruction word for branch prediction |
| 2 | **bpd** | branch prediction(more details see *branch prediction unit* section) |
| 4 | **btb** | branch target buffer, used to get the branch target address if the branch is predicted to be taken |
| 5 | **bob** | branch order buffer, |
| 6 | **ras** | return address stack, used to get the return address quickly after a function call |
|  |  |  |

## Branch decode Unit (BrDEC)

### Introduction

Branch decode logic partially decode the fetched instruction, get the branch related information for branch prediction logic, including branch direction prediction and branch target prediction.

The decoded information including:

* branch flag: indicate this instruction is a branch instruction.
* branch type: indicate which kind of branch this instruction is.
* branch target: if the branch is a PC and immediate relative branch instruction.
* branch class: indicates the branch is function call relative, and details the RAS operation type.

### Architecture

In current implementation, fetch width is 8 words (instructions), so we need decode 8 instructions in one cycle (worst case), to get the prediction needed information. In current design we instance 8 *brdec\_way* modue to decode each instruction concurrently.

the location of this module is located in fetch stage1, because of the cache data is fetched in fetch stage0, after the instruction word flopped out, instruction alignment also located in current stage, so this path will be the most critical path, should pay more attention.

### Ports

Table ‑ *brdec* module port signals

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| pc\_f1\_i | input | 64 | program counter in fetch stage 1 |
| inst0\_i | input | 32 | instruction 0 from alignment |
| inst1\_i | input | 32 | instruction 0 from alignment |
| inst2\_i | input | 32 | instruction 0 from alignment |
| inst3\_i | input | 32 | instruction 0 from alignment |
| inst4\_i | input | 32 | instruction 0 from alignment |
| inst5\_i | input | 32 | instruction 0 from alignment |
| inst6\_i | input | 32 | instruction 0 from alignment |
| inst7\_i | input | 32 | instruction 0 from alignment |
| ras\_data\_i | input | 64 | target address of ***xRET*** instruction from RAS |
| flush\_vld\_i | input | 1 | instruction flush control signal in fetch stage1 |
|  | | | |
| ras\_data\_o | output | 64 | the pc value need to be kept into RAS for fast return |
| ras\_ctrl\_o | output | 2 | current branch instruction’s RAS operation (more details about RAS operation see *Return Address Stack* section) |
| btb\_we\_o | output | 1 | has branch instruction in current instruction fetch group |
| btb\_br\_pos\_o | output | 3 | branch instruction position in current instruction fetch group |
| btb\_br\_typ\_o | output | 2 | branch instruction type: conditional/indir\_pc/indir\_rs/indir\_ras |
| btb\_br\_tar\_o | output | 64 | branch instruction target address (if we can get it this stage) |
| inst\_valid\_o | output | 8 | indicate which is the valid branch instruction (only one branch is supported in one fetch group) |

### Branch Decoder Ways(brdec\_way)

#### Introduction

brdec\_way receives one instruction from icache alignment module, current implementation follows the encoding of RV32/64IM instruction set according to specification “riscv-spec-2.2.pdf” and “riscv-privileged-v1.10.pdf”.

#### Ports

Table ‑ *brdec\_way* module port signals

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| inst\_i | input | 32 | instruction word from alignment |
| pc\_f1\_i | input | 64 | program counter in fetch stage 1 |
| ras\_data\_i | input | 64 | RAS data for instruction URET |
| rs1\_data\_i | input | 64 | source register data |
| br\_flag\_o | output | 1 | branch instruction exist flag |
| br\_typ\_o | output | 2 | branch type flag |
| br\_tar\_o | output | 64 | branch target address if address is valid |
| ras\_ctl\_o | output | 2 | RAS operation of current instruction |
| rs1\_idx\_o | output | 5 | architecture register file index of source register1 in JALR |
| rs1\_req\_o | output | 0 | source register read request |
|  |  |  |  |

Branch decode information and encodings.

* branch type: (br\_typ)
  + BR\_COND: conditional branch. e.g. BEQ
  + BR\_UNCOND: unconditional branch. e.g. JAL
  + BR\_INDIR: indirect branch. e.g. JALR
  + BR\_INDIRRET: indirect branch (function return). e.g. URET
* RAS control: (ras\_ctl)
  + RAS\_NOACT: RAS no action needed
  + RAS\_PUSHPC: RAS push command
  + RAS\_POPPC: RAS pop command
  + RAS\_PUSHPOP: RAS push and RAS pop enable

## Branch Prediction Unit (BPD)

### Introduction

Ace 21064 implements a Tournament predictor, hybrid with gShare (Global) and Pap (Local) (named by Yeh and Patt[1993]) predictor. The hybrid predictor similar to the one we used was implemented in the Compaq Alpha21264 processor [Kessler,1999] the local history component used a 1024-entry BHT with 10-bit per-branch histories. This 10-bit history is then used to index into a single 1024-entry PHT (pht\_inst0). The global history component uses a 12-bit history that indexes into a 4096-entry PHT (pht\_inst1) of 2-bit counters. The meta-predictor also uses a 1024-entry table of counters (pht\_inst2) index by the BHR value hashed with PC.

Here is the functional diagram of tournament predictor



Figure ‑ Tournament predictor functional diagram

### Architecture

Current implementation is aimed to achieve high performance design, as introduced previous, fetch stage is divided into two pipeline stages to enable greater increases to the processor clock frequency. A faster clock speed means that there is less time to perform a branch prediction. To achieve a single cycle prediction, the sizes of the branch predictor tables, such as the PHT, must be reduced. Smaller branch prediction structures lead to more capacity and conflict aliasing and, therefore, to more branch mispredictions. The branch misprediction penalty has also increased because the number of pipe stages has increased. So here is a tradeoff between the fetch efficiency and the clock speed and pipeline depth.

The overriding organization is used in current design to rectify this situation. first cycle, uses the cache line/way prediction result as the initial target prediction, and instruction fetch uses this prediction to start fetch operation. At the same time the tournament predictor starts its prediction lookup, but this prediction will not be available for next cycle. In cycle 2, while waiting for the second predictor's prediction, the first predictor provides another prediction so that the instruction cache can continue fetching more instructions. A lookup in the second predictor for this branch is also started, and therefore the second predictor must be pipelined. In the end of cycle 2, the come out of tournament predictor is compared to the original prediction. If the predictions match, then the first predictor was correct (with respect to the second predictor) and fetch can continue. If the predictions do not match, then the second predictor overrides the first prediction. Any further fetches that have been initiated in the meantime are flushed from the pipeline, and the first predictor and instruction cache are reset to the target of the overridden branch.

Here is the pipelined tournament predictor and overriding prediction mechanism:



Figure ‑ pipelined tournament predictor

### Ports

Branch prediction logic is designed to take two pipeline stages

Table ‑ branch prediction unit ports

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| clock | input | 1 |  |
| reset\_n | input | 1 |  |
| pc\_f0\_i | input | 64 |  |
| pc\_f1\_i | input | 64 |  |
| pipctl\_flush\_rt\_i | input | 1 |  |
| pipctl\_fill\_f1\_i | input | 1 |  |
|  | | | |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  | | | |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

### Submodules

Table ‑ *bpd* sub-modules

|  |  |  |
| --- | --- | --- |
| Num | Module | Description |
| 1 | **bpd0** | branch prediction(more details see *branch prediction unit* section) |
| 2 | **bpd1** |  |
| 3 | **bht** |  |
| 4 | **pht** |  |

### Branch History Table (BHT)

BHT in Ace21064 has 10 bits wide, which can record ten branch taken history. And has 1024 entries for different branches.

Read BHT :

Read Address: (Read Index) there are 1K entries in design, so the read index width should be 10 bits from current PC. As we know the RISCV ISA is 32bit width, so the bits [1:0] is reserved. bht\_rd\_index\_i[9:0] comes from cur\_pc\_i[11:2].

Read Data: when we find the entry which the cur\_pc has indexed, read the data out for PHT index. We rename it as bht\_br\_hist\_o[9:0].

Write BHT:

Write Address: (Write Index) we update BHT when the branch instruction is committed (after execute stage, the branch direction is confirmed) we named reference pc as confirmed pc. So bht\_wt\_index\_i[9:0] comes from cm\_pc\_i[11:2]

Write Data: As the definition of BHT (a shifter), the write data is the branch direction of the confirmed pc instruction, (bht\_cm\_brdir\_i)

Write enable: except address and data we need to know when to update the BHT entry, so write enable signal is needed. As we know the BHT entry can only be update when the branch is confirmed for the correct branch history record. (bht\_cm\_brdir\_se\_i) (fixme: when to update the BHT [SuperScalar RISC Processor Design p118])

Table ‑ BHT Port List

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| Global interface |  |  |  |
| clk | Input | 1 | Clock |
| rst\_b | Input | 1 | Reset |
| BHT Read Interface | | | |
| bht\_rd\_index\_i | Input | 10 | BHT read index, part of current pc |
| bht\_br\_hist\_o | Output | 10 | Branch history, BHT read data |
| BHT Write Interface | | | |
| bht\_wt\_index\_i | Input | 10 | BHT write index, part of confirmed branch instruction’PC |
| bht\_cm\_brdir\_i | Input | 1 | confirmed branch instruction’s direction |
| bht\_cm\_brdir\_se\_i | Input | 1 | confirmed branch instruction direction shift enable |

### Pattern History Table (PHT)

BHT in Ace21064 has 10 bits wide, which can record ten branch taken history. And has 1024 entries for different branches.

Read BHT :

Read Address: (Read Index) there are 1K entries in design, so the read index width should be 10 bits from current PC. As we know the RISCV ISA is 32bit width, so the bits [1:0] is reserved. bht\_rd\_index\_i[9:0] comes from cur\_pc\_i[11:2].

Read Data: when we find the entry which the cur\_pc has indexed, read the data out for PHT index. We rename it as bht\_br\_hist\_o[9:0].

Write BHT:

Write Address: (Write Index) we update BHT when the branch instruction is committed (after execute stage, the branch direction is confirmed) we named reference pc as confirmed pc. So bht\_wt\_index\_i[9:0] comes from cm\_pc\_i[11:2]

Write Data: As the definition of BHT (a shifter), the write data is the branch direction of the confirmed pc instruction, (bht\_cm\_brdir\_i)

Write enable: except address and data we need to know when to update the BHT entry, so write enable signal is needed. As we know the BHT entry can only be update when the branch is confirmed for the correct branch history record. (bht\_cm\_brdir\_se\_i) (fixme: when to update the BHT [SuperScalar RISC Processor Design p118])

Table ‑ BHT Port List

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| Global interface |  |  |  |
| clk | Input | 1 | Clock |
| rst\_b | Input | 1 | Reset |
| BHT Read Interface | | | |
| bht\_rd\_index\_i | Input | 10 | BHT read index, part of current pc |
| bht\_br\_hist\_o | Output | 10 | Branch history, BHT read data |
| BHT Write Interface | | | |
| bht\_wt\_index\_i | Input | 10 | BHT write index, part of confirmed branch instruction’PC |
| bht\_cm\_brdir\_i | Input | 1 | confirmed branch instruction’s direction |
| bht\_cm\_brdir\_se\_i | Input | 1 | confirmed branch instruction direction shift enable |

### 

## Return Address Stack (RAS)

### Introduction

The RAS is the stack that used to predict the targets of returns. A routine can be called from many call sites and the branch prediction logic has to produce the return site corresponding to the call. Calls and returns follow a simple last-in-first-out (LIFO) behavior. Thus, we employ a small stack (16 entries in current implementation) structure named as return address stack (RAS). Calls push the address of the return site onto the stack and returns pop them out. When the BTB signals that the instruction being fetched is a return, it takes the target provided by the RAS

### Architecture

RAS operation:

* no operation:
  + Current PC and previous PC (override RAS) has no relationship about function call and return.
* push :
  + RAS write: BTB hit and the branch type kept in BTB shown as BR\_UNCOND/BR\_INDIR (in order to avoid the critical path, the RAS operation is kept in BTB entry directory). So RAS write operation should follow the BTB hit and RAS operation is push or poppush.
  + RAS override: When BTB miss on a call jump, BPD prediction is taken on a ras\_ctl is push or pushpop from BrDEC.
* pop:
  + BTB hit, and RAS operation kept in BTB entry shows a pop operation (branch type is xRET)
  + BTB miss, BrDEC detect a xRET instruction in fetch stage 1(this is an unconditional branch, so no need to check the BPD prediction)
* poppush:
  + This is a merge of pop operation on BTB miss (first cycle), and cascade a push operation on BTB hit(second cycle).

### Ports

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| clock | input | 1 |  |
| reset\_n | input | 1 |  |
| flush\_rt\_i | input | 64 |  |
| invalid\_f1\_i | input | 64 |  |
| signals from module BrDEC | | | |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  | | | |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  | | | |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## Branch Target Buffer (BTB)

### Introduction

4-Way associative branch target buffer (BTB ) is realized in Ace 21064 processor, also named branch target address cache (BTAC) in some systems. 256 entries located in one BTB way. Each entry contains tag array and data array as a general cache organization. But there are some distinguishes between original caches:

The tag array is used to keep the higher part of branch instruction’s PC, also named branch instruction address (BIA) in some computer systems. As detailed in figure below, the BTB tag width is 18 in current design. Which is hashed each 18-bit of 54-bit of original BTB tag data, to reduce area cost, in balance of cost and performance.



Figure ‑ branch target buffer microarchitecture

Data array is used to keep the branch target address of the indexed branch instructions, and some branch related information, like BTB entry valid, RAS control signal, branch instruction position in fetch group, and branch type. More details see btb\_way section.

### Ports

Branch prediction logic is designed to take two pipeline stages

Table ‑ branch prediction unit ports

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| clock | input | 1 |  |
| reset\_n | input | 1 |  |
| pc\_f0\_i | input | 64 |  |
| pc\_f1\_i | input | 64 |  |
| signals from module BrDEC | | | |
| brdec\_brext\_f1\_i | input | 1 | current bundle branch instruction exist flag |
| brdec\_brpos\_f1\_i | input | 3 | branch instruction position in current fetch group |
| brdec\_brtyp\_f1\_i | input | 2 | branch type |
| brdec\_brtar\_f1\_i | input | 64 | branch target address |
| brdec\_rasctl\_f1\_i | input | 2 | RAS operation type |
| signals from retire stage, which used to update BTB when branch confirmed | | | |
| brcond\_vld\_rt\_i | input | 1 | retired branch is a conditional branch |
| brindir\_vld\_rt\_i | input | 1 | retired branch is an indirect branch |
| brdir\_rt\_i | input | 1 | branch direction (taken or not taken) |
| brtar\_rt\_i | input | 64 | branch target address |
| brpc\_rt\_i | input | 64 | branch instruction PC |
| output signals from BTB | | | |
| btb\_hit\_f0\_o | output | 1 | BTB request hit signal |
| btb\_brpos\_f0\_o | output | 3 | branch instruction position in fetch group |
| btb\_brtyp\_f0\_o | output | 2 | branch instruction type |
| btb\_brtar\_f0\_o | output | 64 | branch instruction target address |
| btb\_rasctl\_f0\_o | output | 2 | RAS operation type |
| btb\_brdir\_f0\_o | output | 1 | branch direction (taken or not taken) |
|  |  |  |  |

### BTB Way

BTB data entry format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 73 | 72 71 | 70 69 | 68 67 | 66 64 | 63 0 |
| Entry format: | btb\_valid | btb\_cnt | btb\_ras\_ctl | btb\_br\_typ | btb\_br\_pos | brb\_br\_tar |
|  | 1 | 2 | 2 | 2 | 3 | 64 |

* btb\_valid : btb entry valid flag
* btb\_cnt: a branch saturate counter to recode the instruction’s directory (this can be found in stage f0)
* btb\_ras\_ctl: return address stack operation control
* btb\_br\_typ: instruction branch type
* btb\_br\_pos: branch instruction position in fetch group
* btb\_br\_tar: branch target address

BTB write strategy:

Write the predicted taken branch information only.

BTB miss operation: continue execution (performance preferred)

When a branch is predicted to be taken, but the BTB miss occurred, Ace21064 choose to fetch instructions from continued instructions. If the branch is computed, and if it is not equal with the continued address (this situation is quite often), the instructions in the pipeline after the branch instruction will be flushed, and re-fetch from the computed branch target address. And if the branch is an indirect branch, there may be need a lot cycles to get the target address. But if a branch direction miss prediction occurs, the “continued execution” strategy is quite smarter than stall the whole pipeline for a lot cycles.

### Ports

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| clock | input | 1 | system clock |
| reset\_n | input | 1 | system reset low active |
| pc\_f0\_i | input | 64 | program counter of fetch stage 0 |
|  | | | |
| btb\_sp\_we\_i | input | 1 |  |
| btb\_sp\_brpos\_i | input | 3 |  |
| btb\_sp\_brtyp\_i | input | 2 |  |
| btb\_sp\_brpc\_i | input | 64 |  |
| btb\_sp\_brtar\_i | input | 64 |  |
| btb\_rt\_we\_i | input | 1 |  |
| btb\_rt\_brdir\_i | input | 1 |  |
| btb\_rt\_brpc\_i | input | 64 |  |
| taken\_addr\_i | input | 64 |  |
| ras\_ctl\_i | input | 2 |  |
| btb\_ras\_ctl\_o | output | 2 |  |
| btb\_br\_pos\_o | output | 3 |  |
| btb\_br\_typ\_o | output | 2 |  |
| btb\_br\_tar\_o | output | 64 |  |
| btb\_br\_dir\_o | output | 1 |  |
| btb\_hit\_f0\_o | output | 1 |  |
| btb\_hit\_f1\_o | output | 1 |  |

## Branch Ordering Buffer (BOB)

### Introduction

Branch ordering Buffer (BOB) is a FIFO like buffer, which has all the in-flight branches in the processor for updating the BPD non-speculatively in program order and rolling back BPD state such as the branch history register (BHR) used by the branch prediction algorithm. It also maintains a checkpointed copy of the Return Address Stack (RAS). It is required to correct the RAS when there is a BTB miss on a call or return. It sends the required instruction block down the pipeline.

### Architecture

### Ports

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| clock | input | 1 |  |
| reset\_n | input | 1 |  |
| flush\_rt\_i | input | 64 |  |
| invalid\_f1\_i | input | 64 |  |
|  | | | |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Instruction Decoder Unit(IDU)

## Overview

### Introduction

Decode stage in Ace21064 processor contains two physical pipeline stages (two cycles), first is a 32-entry instruction buffer, which receives up to 8 instructions from IFU, these instructions are write into a circular buffer with tail pointer, and only 4 instructions are read from the head pointer to the second stage of instruction decoder, feed 4 wide decoder, 4 instructions can be decoded in one cycle.

In the decode stage, every branch instruction should be assigned with a code, and only one branch instruction can be decoded in one clock cycle.

### Submodules

Table ‑ ace\_decode sub-modules

|  |  |
| --- | --- |
| Module Name | Description |
| inst\_buf | instruction buffer, the fetch stage will fetch 8 instructions per cycle in best case, but decode stage can only decode 4 instructions one time |
| decoder\_0 | decode unit 0 |
| decoder\_1 | decode unit 1 |
| decoder\_2 | decode unit 2 |
| decoder\_3 | decode unit 3 |

## Instruction Buffer

### Introduction

Instruction buffer decouples the instruction fetching and the rest of frontend pipeline stages, It receives up to 8 instructions (best case) from IFU, these instructions are write into a circular buffer with tail pointer, and always 4 instructions are read from the head pointer to the second stage of instruction decoder.

The instruction buffer allows instruction fetching, even the rest of frontend pipeline is stalled because of resource limitation, and because of the existence of this buffer, the decode, rename, and dispatch stage can always be fed with a fix number of instruction, 4 instruction in current design.

### Architecture

### Ports

Table ‑ instruction buffer ports

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| Global interface |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  | | | |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  | | | |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  | | | |
|  |  |  |  |
|  |  |  |  |

## Instruction Decoder

### Introduction

The instruction decode logic is the most clear module, due to the convenient definition of RISC-V ISA, more information of RISC-V ISA refer to doc[riscv-spec-v2.2.pdf].

Currently, Ace21064 only implements integer instructions, and we intend to extend the design for floating-point instructions in next step.

### Architecture

# Register Renaming Unit (RRU)

## Overview

Register renaming removes the false dependencies among instructions which are artifacts of limited architectural registers. The data dependencies of a dynamic instruction stream can be classed as:

True-dependency, which the source register of a younger instruction depends on the outcome of another older instruction in the instruction stream, this dependency also named read after write dependency (RAW)

Output-dependency, where the destination register of a younger instruction is the same as the destination register of another older instruction in the instruction stream, this dependency also called write after write dependency (WAW).

Anti-dependency, where the destination register of a younger instruction is the same as the source register of another older instruction in the instruction stream, this dependency also named as write after read dependency (WAR).

WAW and WAR sometimes also referred as false dependencies, register renaming eliminates false dependencies by mapping architectural destination register of each in-flight instruction to a unique physical register, but WAW dependency should be more attention in dependency check (more details shown below).

### Introduction

Current design implements 4 wide rename from 80 physical registers, both speculative and architectural rename maps maintained, a circular FIFO is implemented as speculative register free list (SpecRFL), contains the unused physical registers. An unused physical register is popped by the SpecRFL to be used as a replacement of the architectural destination register of instruction. A register alias table (RAT) maintains the physical registers to which architectural registers are currently mapped. Accordingly, each architectural source register of the instruction is renamed to a physical source register by looking up RAT.

Checkpoints of SpecRAT and SpecRFL, Architectural Register Alias Table (ArchRAT) are implemented for branch mis-prediction quick recovery. The Architectural state Recovery mechanism updates ArchRAT and ArchRFL when each instruction leaves the pipeline (Retire), when a branch instruction mis-prediction encountered in execute stage, the processor let the pipeline keep moving, until the mis-predicted branch instruction becomes the last instruction in pipeline, recover SpecRAT with ArchRAT (keeps all correct instruction status), and the pipeline will be completely flushed. If a branch resolves correctly the ArchRAT will keep updating and no recovery operation will occur.

Memory dependence prediction using store sets.



Figure ‑ Register Renaming Unit

### Signals

Table ‑ RRU port signals

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| clock | input | 1 | system clock |
| reset\_n | input | 1 | system reset low active |
|  |  |  |  |
|  |  |  |  |
|  | | | |

### Submodules

Table ‑ ace\_rename sub-modules

|  |  |
| --- | --- |
| Module Name | Description |
| spec\_rat | speculative RAT with 12 read ports, 8 write ports |
| spec\_rfl | speculative FreeList with 4 read ports(four-wide renaming per cycle), 8 write ports (eight wide retire per cycle). |
| mdp\_ssit | memory dependence predictor store set ID table |
| mdp\_ssit\_dpd\_chk | identifies intra instruction bundle dependencies for store set IDs |
| mdp\_lfst | last fetched store table for memory dependence prediction |
| mdp\_dpd\_chk | does dependence check between the 4 instructions currently in rename stage0 |
|  |  |
|  |  |
|  |  |

## Register Alias Table (RAT)

In current design the RAT is implemented with SRAM, so the RAT is also called SRAM based RAT (sRAT, contrast with CAM based RAT, cRAT). According to the rename protocol, one instruction needs 2 read ports (source register alias register read), and 1 write port (new alias of destination register, needs to be write into RAT). Considering the speculative execution in branch instruction, if the misprediction occurs, the RAT need to be recover from Architectural RAT, implemented in retire stage.(will be introduced in following chapter).

What’s more, before we update RAT, we should keep the old data in sRAT (which indexed with the logical destination register of current instruction), for two reasons: *First*, when current instruction retire corresponding physical register should be set as free (in RFL); *Second*, if exception triggered or misprediction take place, the pipeline should be flushed, and we need to recover RAT. So another Read port needed for old alias data backup.

Above all, one instructions alias operation needs 3 read ports and 2 write ports. Due to 4-wide renaming, 12 read ports and 4 write ports needed in RAT

SRAM based Register Alias Table is used in current design. The sRAT holds following features:

* 12 read ports
* 4 write ports
* 32 entries (logical register count)
* Recovery Interface with Architectural RAT.

## Register Free List (RFL)

Register free list in current design, used to record the unused physical register, to cooperate with SpecRAT to realize register renaming using unified physical register file. For a 4-wide renaming stage, there are 4 physical registers are needed for 4 instructions in worst case, so 4 read ports are implemented. Because of the retire width is 8, in best case, there will be eight destination register will be set free. In order to achieve 8-wide retire performance the RFL is designed with 8 write port.

Register Free List is implemented with following features:

* 4 read ports
* 8 write ports
* 48 entries (physical register count)
* Recovery Interface with Architectural RFL

### Ports

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| System Signals | | | |
| clock | input | 1 | system clock |
| reset\_n | input | 1 | system reset low active |
| signals from retire stage | | | |
| arch\_fl\_rec\_i | input | 1 | Architectural register freelist recover signal |
| arch\_fl\_rec\_data\_i | input | 48\*7 | Architectural register freelist recover data |
| Register request port | | | |
| inst0\_rd\_req\_i | input | 1 | Instruction 0 destination register request signal |
| inst1\_rd\_req\_i | input | 1 | Instruction 1 destination register request signal |
| inst2\_rd\_req\_i | input | 1 | Instruction 2 destination register request signal |
| inst3\_rd\_req\_i | input | 1 | Instruction 3 destination register request signal |
| Register release port | | | |
| retire0\_rls\_rd\_i | input | 7 | retired instruction 0 released destination register |
| retire1\_rls\_rd\_i | input | 7 | retired instruction 1 released destination register |
| retire2\_rls\_rd\_i | input | 7 | retired instruction 2 released destination register |
| retire3\_rls\_rd\_i | input | 7 | retired instruction 3 released destination register |
| retire4\_rls\_rd\_i | input | 7 | retired instruction 4 released destination register |
| retire5\_rls\_rd\_i | input | 7 | retired instruction 5 released destination register |
| retire6\_rls\_rd\_i | input | 7 | retired instruction 6 released destination register |
| retire7\_rls\_rd\_i | input | 7 | retired instruction 7 released destination register |
| retire0\_rls\_rd\_vld\_i | input | 1 | retired instruction 0 released destination register valid |
| retire1\_rls\_rd\_vld\_i | input | 1 | retired instruction 1 released destination register valid |
| retire2\_rls\_rd\_vld\_i | input | 1 | retired instruction 2 released destination register valid |
| retire3\_rls\_rd\_vld\_i | input | 1 | retired instruction 3 released destination register valid |
| retire4\_rls\_rd\_vld\_i | input | 1 | retired instruction 4 released destination register valid |
| retire5\_rls\_rd\_vld\_i | input | 1 | retired instruction 5 released destination register valid |
| retire6\_rls\_rd\_vld\_i | input | 1 | retired instruction 6 released destination register valid |
| retire7\_rls\_rd\_vld\_i | input | 1 | retired instruction 7 released destination register valid |
| Output signal for instruction queue | | | |
| inst0\_vld\_r | output | 1 | instruction0 valid signal registered from fetch stage1 |
| inst1\_vld\_r | output | 1 | instruction1 valid signal registered from fetch stage1 |
| inst2\_vld\_r | output | 1 | instruction2 valid signal registered from fetch stage1 |

## Data dependency Checker

## Memory dependency predictor

### Introduction

### Architecture

# Instruction Schedule Unit (ISU)

## Overview

When the instructions were renamed, the instruction not only keeps in reorder buffer (ROB) but also stored in the reservation station (also named issue queue in many designs). And the operation which write the renamed instruction into ROB, Reservation Station (RS) and Store Queue (SQ) is called *dispatch*. Before dispatch, it is the responsibility of dispatch logic to check for available space of ROB RS or SQ. If the back-end pipeline stages don’t have enough spaces in these resources, the dispatch logic generates stall signal for decode and rename stages. Dispatch is the boundary between in-order instruction processing and out-of-order instruction processing.

In issue stage, the instructions in reservation station which has source operands ready will be delivered into functional unit correspondingly

### Introduction

Instruction schedule unit is the main function unit in issue stage, and is critical to the performance of superscalar microarchitecture. Issue stage buffers the renamed instructions and selects instructions for execution based on the availability of their source operands.

In Ace21064, IFU will fetch 8 instructions in one cycle (in common instruction stream), which we have introduced previously. Decode and rename stage have operation wide of 4 named as “machine width”. Due to the dependence between instructions, scheduler usually can’t issue 4 instructions into function unit in one cycle. In order to get the maximum parallelism of execution, this design use issue width in six, up to 6 instructions can be selected for execution every cycle.

32-entry scheduler.

Two main steps used in issue stage in ISU. They are wakeup and select, the features in wakeup and select stages are shown below:

* Wakeup

Speculative wakeup(In reversion 0 there is no speculative wakeup)

One cycle execute instruction: wakeup and select in one cycle, to make RAW dependency instruction can be executed back to back;

Multi-cycle execute instruction: delayed wakeup strategy, only when their data is actually produced, for example load instruction.

* Select

Instruction replay (replay queue based replay is used)

Several function unit share the same reservation station, combined with centralized and distributed reservation station are used.

Non-data-capture architecture (read source data after instruction was selected in reservation station) is used

Compressed reservation station is used.

Considering the issue width of six, the physical register file should have read port count 2\*(issue width) to support six instructions execution in one cycle.



### Features

Reservation station entry data:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| Instr.Issued | Instr.Valid | Rs1.Valid | Rs1 | Rs2.Valid | Rs2 | Rd |

## Reservation Station

There are two reservation station existed in this design, one for flow control, and simple function ALU like add shift etc. named “RS0 (reservation station0)”, and the other keeps the instructions of load store and complicated calculation, like divide and multiply. Named “RS1 (reservation station1)”. For current design there are 16 entries in RS0 and 16 entries in RS1. In dispatch stage, the dispatch (allocation) logic receives the information of 4 instructions from register renaming stage, write instruction data (source register index, and some control information, see RS entry consistence details) into different RSs by different instruction type.

### Signals

Table ‑ reservation stationh port signals

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| clock | input | 1 | system clock |
| reset\_n | input | 1 | system reset low active |
| inst0\_package[0] | input | 1 | Instruction 0 valid flag |
| Inst0\_package[1] | input | 1 | Instruction0 is simple ALU instruction flag |
| Inst0\_package[2] | input | 1 | Instruction0 is complex ALU instruction flag |
| Inst0\_package[3] | input | 1 | Instruction0 is simple branch instruction flag |
| Inst0\_package[4] | input | 1 | Instruction0 is simple memory access instruction flag |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  | | | |

### Submodules

Table ‑ ace\_rename sub-modules

|  |  |
| --- | --- |
| Module Name | Description |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Physical Register File (PRF)

## Overview

As mentioned before, Non-Data-Capture architecture is used in current design, Physical Register File (PRF) will be accessed after the instruction was issued. In such structure, the PRF read port is twice of issue width. So read port of PRF in current implementation is 6\*2, 12 read port needed.

PRF is the register file which holds all the committed and non-committed instruction results. The source register specifiers of an issued instruction, reads the corresponding values of PRF. At the same time, the source register specifier also compared with the destination register specifier of the previous instruction, to determine if the source operator needs to be captured from the bypass logic.

In current design, PRF is implemented with RAM, due to the 6-wide issue, 12-read and 6-write ports are required

### Introduction

.

# Execution Unit

## Overview

In current version, six functional units has been implemented, each unit executes a different class of integer instructions. Six functional units are:

* Two Simple ALUs

Simple ALU performs simpler arithmetic and logic operations, like: addition, subtraction, logic operation etc. all operations executed in Simple ALU takes a single clock cycle to complete.

* Complex ALU

Complex ALU performs complex arithmetic operation like multiply, divide etc. these operations take multiple cycles to execute, two clock cycles are needed in current implement, and are fully pipelined.

* Control ALU

Control ALU executes control instructions, for example, conditional branches, jumps etc. these instruction can be finished in one cycle.

* Two AGENs

Address Generator (AGEN) performs address computations for memory operations, like load and store. The output of AGEN goes to the LSU.

The source operands of each function unit comes from PRF read or bypass logic.

### Introduction

# Load Store Unit (LSU)

### Overview

Load and store instructions typically use register operands to calculate their address, dependence of load and store usually unreachable until they are executed. Memory dependence prediction is implemented in current design for load and store to be execute out of order.

By the way, a load instruction compares its address with all un-committed store operation which is older in program order. In case the address matches the store address, the data in store operation will be forwarded to load instruction, moreover, all store operations should update the architectural memory data in program order.

Load queue (LQ) and store queue (SQ) are designed to maintain the uncommitted memory operations in program order. In current design, an issued load operation takes at least two cycles to execute: first cycle, the AGEN unit calculate the load address, in second cycle it goes through an address dependency check. The load operation might get its data from data cache or the store queue. And the access to data cache happens in parallel with the checking of store queue.

### Introduction

# Level 1 data Cache (dCache)

### Overview

Dual-port 2-way associative level 1 data cache, each way is designed with 8 interleaved banks 2KB each bank, data cache designed with write back strategy when write hit happens. when a store operation get a write miss, data cache controller takes write allocate strategy, which means level 1 data cache will fetch expected data from next level cache, and then performs write hit operation. When a cache refill take place, LRU strategy is used for data replacement.

### Introduction



Figure ‑ Ace21064 level 1 data cache overview

# Retire Unit

## Overview

In Current design, the retire unit contains write back stage and retire stage.

* Write back

The write back stage keeps the results from the execute stage, which may become the source of bypass network, the bypass network forwards the results from executed instructions to the dependent instructions. The instructions in register read stage and execute stage compare their register specifiers with destination register specifiers on the bypass network.

The write back stage also acts the source of branch misprediction signals.

* Retire

Ace21064 is a superscalar processor, which executes instruction out-of-order, but they update the processor’s architecture state in program order, these in order commit mechanism maintains the sequential execution model, and naturally leads to the implementation of precise exception.

### Introduction

This implementation maintains the program order among instructions with a circular FIFO with head and tail pointers, referred to reorder buffer (ROB). When an instruction is dispatched, the instruction are inserted into the ROB at tail pointer. 64-entry ROB is implemented in current design,

64-entry reorder buffer, with eight-wide retire

The ROB entry’s section are consisted of exception flag, instruction type, architectural register, current physical register, previous physical register, instruction PC, complete flag. ROB keeps probing the completed bits for the entries starting from the head pointer, and any completed instructions at the head are committed and removed from ROB. For Store instruction ROB signals the SQ to commit the store data into memory (dCache).

Architectural Register Alias Table (ArchRAT)

ArchRAT contains register mappings between architectural registers and physical registers for committed version of architectural registers. When an instruction commits, the ROB updates the ArchRAT with the instructions’ physical destination register mapping, and release the mapped physical register previously, and the released physical registers are added into the Architectural Register Free List (ArchRFL)

In these retire two cycles, first cycle the head of ROB is read and in the second cycle, the ArchRAT and ArchRFL and SQ are updated.

## Interrupt

## Branch Misprediction Recovery

In a pipeline with branch prediction, the prediction result ca be checked in several pipeline stages, they are decode stage, register read stage, and execute stage. In these stages some information can be confirmed

*Decode stage*: one instruction can be know whether it is a branch instruction, and the branch type of this instruction. If this instruction is PC-relative branch instruction, the target address can also be confirmed. So some instructions’ prediction result can be checked. In RV32I class the instruction JAL (Jump And Link) is an unconditional branch instruction which encoded as J-type format.



The target address of JAL is PC plus signed immediate offset. If this instruction is predicted not taken, then in this cycle the branch misprediction take place. And the recovery operation can be performed at once. To get the minimal penalty.

But this is the only one branch instruction can be checked the prediction result. Other unconditional branch like JALR (Jump and Link Register) can only found the misprediction take place, but it still can’t figure out where is the target address, until register read stage.

*Register Read Stage*: in this stage, processor can check in-direct branch instructions target address, if the target address prediction miss, then the pipeline can fetch start at the correct address, what’s more and important, the instructions behind this branch should be flushed, in our design non-data-capture structure is used, so when we found the branch miss prediction here, the previous instructions may be in reservation station and execution stage. So instruction order should be marked, in order to pick the correct instruction to flush.

*Execute Stage*: All branch instructions can be confirmed to be taken or not be, and this is the absolute security stage to check the branch prediction’s correctness. Of course one instruction misprediction found in this stage, costs the longest penalty. In this superscalar core, the instructions before the branch instruction includes two parts, first, all instructions before issue stage should be flushed, because all of these instructions must in the misprediction path. They can be flushed in one cycle. Second, the instructions after the issue stage, instructions executes in out-of-order mode, some of them may be in the misprediction path, so we need to record the instruction number, which realized in ROB in current design. After the instructions in miss predicted path are flushed, the CPU status should be recovered.

# Coprocessor

## Overview

Coprocessor of Ace21064 implements PC generator; power management;

## Program Counter Generator (PCGen)

### Introduction

Program counter generator generates the PC for every stage in ace pipeline, and there is several sources for next PC in a superscalar design.

### Architecture

Here is the PC generator architecture diagram, which details the next PC comes from.



Figure ‑ PC generate logic

### Ports

## Power Management (PowerM)

### Introduction

### Architecture

### Ports