**Ace21064 Microprocessor**

Architecture Specification

**r0p1**

**Revision History**

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

**Content**

[1 Ace 21064 Core Basics 7](#_Toc484951901)

[1.1 Overview 7](#_Toc484951902)

[1.2 Features 7](#_Toc484951903)

[1.3 Pipeline architecture of Ace21064 9](#_Toc484951904)

[2 Level 1 Instruction Cache (iCache) 10](#_Toc484951905)

[2.1 Overview 10](#_Toc484951906)

[2.1.1 Introduction 10](#_Toc484951907)

[2.1.2 Architecture 11](#_Toc484951908)

[2.1.3 Signals 11](#_Toc484951909)

[2.1.4 Submodules 12](#_Toc484951910)

[2.1.5 Timing 12](#_Toc484951911)

[2.2 Instruction Alignment Unit 12](#_Toc484951912)

[2.2.1 Introduction 12](#_Toc484951913)

[2.2.2 Architecture 12](#_Toc484951914)

[2.2.3 Signals 12](#_Toc484951915)

[2.2.4 Submodules 12](#_Toc484951916)

[2.2.5 Timing 12](#_Toc484951917)

[3 Instruction Fetch Unit (IFU) 13](#_Toc484951918)

[3.1 Overview 13](#_Toc484951919)

[3.1.1 Introduction 13](#_Toc484951920)

[3.1.2 Signals 13](#_Toc484951921)

[3.1.3 Submodules 15](#_Toc484951922)

[3.2 Branch Prediction Unit 16](#_Toc484951923)

[3.2.1 Introduction 16](#_Toc484951924)

[3.2.2 Architecture 16](#_Toc484951925)

[3.2.3 Ports 16](#_Toc484951926)

[3.2.4 Timing 17](#_Toc484951927)

[3.2.5 Branch History Table (BHT) 17](#_Toc484951928)

[3.2.6 Pattern History Table (PHT) 19](#_Toc484951929)

[3.3 Return Address Stack (RAS) 20](#_Toc484951930)

[3.4 Branch Target Buffer (BTB) 20](#_Toc484951931)

[3.4.1 Introduction 20](#_Toc484951932)

[3.4.2 BTB Way 20](#_Toc484951933)

[3.4.3 Ports 20](#_Toc484951934)

[4 Instruction Decoder Unit(IDU) 21](#_Toc484951935)

[4.1 Overview 21](#_Toc484951936)

[4.1.1 Introduction 21](#_Toc484951937)

[4.1.2 Submodules 21](#_Toc484951938)

[4.2 Instruction Buffer 21](#_Toc484951939)

[4.2.1 Introduction 21](#_Toc484951940)

[4.2.2 Architecture 21](#_Toc484951941)

[4.3 Instruction Decoder 22](#_Toc484951942)

[4.3.1 Introduction 22](#_Toc484951943)

[4.3.2 Architecture 22](#_Toc484951944)

[5 Register Renaming Unit (RRU) 23](#_Toc484951945)

[5.1 Overview 23](#_Toc484951946)

[5.1.1 Introduction 23](#_Toc484951947)

[5.1.2 Signals 24](#_Toc484951948)

[5.1.3 Submodules 24](#_Toc484951949)

[5.2 Instructions Renaming Unit 25](#_Toc484951950)

[5.2.1 Introduction 25](#_Toc484951951)

[5.2.2 Architecture 25](#_Toc484951952)

[5.2.3 Ports 25](#_Toc484951953)

[5.3 Memory dependency predictor 26](#_Toc484951954)

[5.3.1 Introduction 26](#_Toc484951955)

[5.3.2 Architecture 26](#_Toc484951956)

[6 Instruction Schedule Unit (ISU) 27](#_Toc484951957)

[6.1 Overview 27](#_Toc484951958)

[6.1.1 Introduction 27](#_Toc484951959)

[6.1.2 Features 28](#_Toc484951960)

[7 Physical Register File (PRF) 29](#_Toc484951961)

[7.1 Overview 29](#_Toc484951962)

[7.1.1 Introduction 29](#_Toc484951963)

[8 Execution Unit 30](#_Toc484951964)

[8.1 Overview 30](#_Toc484951965)

[8.1.1 Introduction 30](#_Toc484951966)

[9 Load Store Unit (LSU) 31](#_Toc484951967)

[9.1.1 Overview 31](#_Toc484951968)

[9.1.2 Introduction 31](#_Toc484951969)

[10 Level 1 data Cache (dCache) 32](#_Toc484951970)

[10.1.1 Overview 32](#_Toc484951971)

[10.1.2 Introduction 32](#_Toc484951972)

[11 Retire Unit 33](#_Toc484951973)

[11.1 Overview 33](#_Toc484951974)

[11.1.1 Introduction 33](#_Toc484951975)

[11.2 Interrupt 34](#_Toc484951976)

[11.3 Branch Missprediction 34](#_Toc484951977)

[12 Coprocessor 36](#_Toc484951978)

**Table of Figures**

[Figure 1‑1 Block Diagram of Ace21064 7](#_Toc484951979)

[Figure 1‑2 Ace21064 Pipeline 9](#_Toc484951980)

[Figure 2‑1 Instruction Cache Hierarchy Overview 10](#_Toc484951981)

[Figure 2‑2 ace\_icache block diagram 11](#_Toc484951982)

[Figure 3‑1 Instruction Fetch Unit block diagram 13](#_Toc484951983)

[Figure 5‑1 Register Renaming Unit 24](#_Toc484951984)

**Table of Tables**

[Table 3‑1 fetch unit port signals 13](#_Toc484951985)

[Table 3‑2 ace\_fetch sub-modules 15](#_Toc484951986)

[Table 3‑3 nvmc\_biu ports 16](#_Toc484951987)

[Table 3‑4 BHT Port List 17](#_Toc484951988)

[Table 3‑5 BHT Port List 19](#_Toc484951989)

[Table 4‑1 ace\_decode sub-modules 21](#_Toc484951990)

[Table 5‑1 IRU port signals 24](#_Toc484951991)

[Table 5‑2 ace\_rename sub-modules 24](#_Toc484951992)

# 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 1‑1 Block Diagram of Ace21064

## Features

* hex-issue (machine width) out-of-order 12-stage integer pipeline
* 64bit virtual address, 40bits physical address
* 8KB Virtually indexed physically tagged 2-way set associative instruction cache
* 4KB per-page in memory manager
* 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
* Checkpoint-based mechanism for fast recovery from branch misprediction
* 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
* Dual-port 2-way associative l1 data cache is designed with 8 interleaved banks 4KB each bank
* 64-entry reorder buffer is used to support 8 wide retirement

## Pipeline architecture of Ace21064



Figure 1‑2 Ace21064 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 entry 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 8KB, 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 2‑1 Instruction Cache Hierarchy 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 8KB 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 2‑2 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

### Introduction

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 target buffer (BTB), return address stack (RAS), and branch ordering buffer (BOB) etc. The block diagram is shown below.



Figure 3‑1 Instruction Fetch Unit block diagram

### Signals

Table 3‑1 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 |
| Temporary | | | |
| inst\_q\_full\_i | input | 1 | come from instruction queue, which indicates the full status |
| Output signal for pc generator | | | |
| branch\_pc\_o | output | 64 | branch target pc from BTB or RAS |
| 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 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 |
| inst3\_vld\_r | output | 1 | instruction3 valid signal registered from fetch stage1 |
| inst4\_vld\_r | output | 1 | instruction4 valid signal registered from fetch stage1 |
| inst5\_vld\_r | output | 1 | instruction5 valid signal registered from fetch stage1 |
| inst6\_vld\_r | output | 1 | instruction6 valid signal registered from fetch stage1 |
| inst7\_vld\_r | output | 1 | instruction7 valid signal registered from fetch stage1 |
| inst0\_ali\_r | output | 32 | aligned instruction0 registered from fetch stage1 |
| inst1\_ali\_r | output | 32 | aligned instruction1 registered from fetch stage1 |
| inst2\_ali\_r | output | 32 | aligned instruction2 registered from fetch stage1 |
| inst3\_ali\_r | output | 32 | aligned instruction3 registered from fetch stage1 |
| inst4\_ali\_r | output | 32 | aligned instruction4 registered from fetch stage1 |
| inst5\_ali\_r | output | 32 | aligned instruction5 registered from fetch stage1 |
| inst6\_ali\_r | output | 32 | aligned instruction6 registered from fetch stage1 |
| inst7\_ali\_r | output | 32 | aligned instruction7 registered from fetch stage1 |
| inst0\_pc\_f1\_r | output | 64 | pc value of aligned instruction0 registered from fetch stage1 |
| inst1\_pc\_f1\_r | output | 64 | pc value of aligned instruction1 registered from fetch stage1 |
| inst2\_pc\_f1\_r | output | 64 | pc value of aligned instruction2 registered from fetch stage1 |
| inst3\_pc\_f1\_r | output | 64 | pc value of aligned instruction3 registered from fetch stage1 |
| inst4\_pc\_f1\_r | output | 64 | pc value of aligned instruction4 registered from fetch stage1 |
| inst5\_pc\_f1\_r | output | 64 | pc value of aligned instruction5 registered from fetch stage1 |
| inst6\_pc\_f1\_r | output | 64 | pc value of aligned instruction6 registered from fetch stage1 |
| inst7\_pc\_f1\_r | output | 64 | pc value of aligned instruction7 registered from fetch stage1 |

### Submodules

Table 3‑2 ace\_fetch sub-modules

|  |  |
| --- | --- |
| Module Name | Description |
| BRDEC | pre-decode logic for branch identification |
| BPD0 | branch prediction stage0 |
| BPD1 | branch prediction stage1 |
| BTB | branch target buffer |
| BOB | branch order buffer |
| RAS | return address stack |
|  |  |

## Branch Prediction Unit

### Introduction

Ace 21064 use a Tournament predictor

### Architecture



### Ports

Table 3‑3 nvmc\_biu ports

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| Global interface |  |  |  |
| hclk | Input | 1 | Clock |
| hrst\_b | Input | 1 | Reset |
| Memory access bus interface | | | |
| hsel | Input | 1 | Memory access bus interface |
| htrans | Input | 2 | Memory access bus interface |
|  |  |  |  |
|  |  |  |  |
| Slave 1 access bus interface | | | |
| s1\_hsel | Output | 1 | EFC access bus interface |
| s1\_htrans | Output | 2 | EFC access bus interface |
|  |  |  |  |
| Slave 2 access bus interface | | | |
| s2\_hsel | Output | 1 | ROM1 access bus interface |
| s2\_htrans | Output | 2 | ROM1 access bus interface |

### Timing

### 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 3‑4 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 3‑5 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)

## Branch Target Buffer (BTB)

### Introduction

4-Way associative branch target buffer is realized in Ace 21064 processor

### BTB Way

### Ports

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Direction | Width | Description |
| Global interface |  |  |  |
| clk | Input | 1 | Clock |
| rst\_n | Input | 1 | Reset |
| Memory access bus interface | | | |
| hsel | Input | 1 | Memory access bus interface |
| htrans | Input | 2 | Memory access bus interface |
|  |  |  |  |
|  |  |  |  |
| Slave 1 access bus interface | | | |
| s1\_hsel | Output | 1 | EFC access bus interface |
| s1\_htrans | Output | 2 | EFC access bus interface |
|  |  |  |  |
| Slave 2 access bus interface | | | |
| s2\_hsel | Output | 1 | ROM1 access bus interface |
| s2\_htrans | Output | 2 | ROM1 access bus interface |

# 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.

.

### Submodules

Table 4‑1 ace\_decode sub-modules

|  |  |
| --- | --- |
| Module Name | Description |
| inst\_buf | instruction buffer, the fetch stage will fetch 8 instructions per cycle, 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 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, the real decode stage.

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

## 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-v2.1.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

### 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, branch mask logic are implemented for branch misprediction quick recovery. The checkpoints mechanism makes copy of SpecRAT and SpecRFL’s head pointer, when a branch instruction is encountered. And each branch instruction carries the associated SpecRAT.

Branch mask logic used to indicate the pending branches an instruction depends on.

When a branch misprediction detected, the frontend pipeline stages are completely flushed, and the branch mask is used for selective removal of instructions in backend pipeline stages.(remove only those instructions that after the branch program order); and PC is set to the correct target address, and SpecRAT is quickly restored from the checkpoint of that branch.

If a branch resolves correctly the checkpoint is cleaned of the branch.

8 branch instructions are allowed to be issued in current design, So 8 checkpoint are supported in hardware. If there is checkpoint (SpecRAT) free, the processor keeps renaming, until one branch instruction encounters.

Memory dependence prediction using store sets.



Figure 5‑1 Register Renaming Unit

### Signals

Table 5‑1 IRU port signals

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

### Submodules

Table 5‑2 ace\_rename sub-modules

|  |  |
| --- | --- |
| Module Name | Description |
| spec\_rat | speculative RAT with 8 read ports(four-wide renaming each instruction has two source register in worst case), 4 write ports(four-wide renaming, each instruction has one destination register in worst case). |
| spec\_freelist | 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 |
|  |  |
|  |  |
|  |  |

## Instructions Renaming Unit

### Introduction

For a 4-way renaming stage, there are 4 instructions should be renamed in one cycle, so there are eight source register and four destination register in worst case. So RFL and RAT (pop four destination register, read eight source register) are accessed in the same cycle,

NOTE: if there are multiple producers of the same architectural registers in the rename group, only the youngest producer updates the 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 width of current design, sRAT should designed with 8 read ports, and 4 write ports. 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, 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.

### Register Alias Table (RAT)

As mentioned in section 5.2.1, a SRAM based Register Alias Table is used in current design. The sRAT holds following features:

* 8 read ports
* 4 write ports
* 32 entries (logical register count)
* 80 Checkpoints (allows 80 branch instructions in flight)

#### 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 |

## 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. And the operation which write the renamed instruction into ROB, Reservation Station and Store Queue is called dispatch, before dispatch, it is the responsibility of dispatch logic to check for available space. If the back-end pipeline stages don’t have enough spaces in these resources, the dispatch logic generates stall signal for the 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

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 6, up to 6 instructions can be selected for execution every cycle.

### 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.

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

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

# 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.

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

### Introduction

# 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.

Missprediction

Exception

Interrupt

Store operation

storebuffer

## Interrupt

## Branch Missprediction

# Coprocessor

Power management

Performance Monitors