MP4 Design

Angquan Yu (angquan2), Nachuan Wang (nachuan3), Zhengyang Zhang (zz44)

**Block Diagram**

图示

描述已自动生成

**Roadmaps**

|  |  |  |  |
| --- | --- | --- | --- |
| Check Point | Angquan Yu | Nachuan Wang | Zhengyang Zhang |
| CP1 (Oct. 26) | Set up test environment (RVFI monitor, random test for CPU), Arbiter | Reorder Buffer, Common Data Bus, Reservation Station, ALU | Fetcher, Instruction Queue, Issuer |
| Support imm-reg/reg-reg instructions by CP1 | | | |
| CP2 (Nov. 9) | Instruction Cache, Data Cache, More testbench (shadow memory, random test for memory) | Branch related modules, Load/Store related modules | Branch Predictor |
| Support all instructions by CP2 | | | |
| CP3 (Nov. 30) | Work on extra feature / Debug / Optimization / Reserved for expected challenges | | |
| CP4 (Dec. 7) |

**Common Data Bus (CDB)**

**Description**:

Bus that update all architectural state registes.

**Ports**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Size | From/To | Description |
| **cdb.in** | | | | |
| cdb.in.valid | in | 1 | … | Write request issued by alu/cmp/memory (require a “OR” ahead of it) |
| cdb.in.data | in | 32 | … | Data to write to speculated registers (require a mux ahead of it) |
| cdb.in.id | in | log2(N) | … | Identifier of instruction (require a mux ahead of it) |
| **cdb.out** | | | | |
| cdb.out.valid | out | 1 | … | Same meaning as above |
| cdb.out.data | out | 32 | … | Same meaning as above |
| cdb.out.id | out | log2(N) | … | Same meaning as above |
| **cdb.flush** | | | | |
| cdb.flush.en | out | 1 | … | Indicate a flush is needed |
| cdb.flush.en\_id | out | N | … | N[i] = 1 means register with identifier I should be flushed |
| **cdb.flushin** | | | | |
| cdb.flushin.en | in | 1 | ROB | Flush signal generated when branch misprediction is detected |
| cdb.flushin.id | in | N | ROB | Indicate which identifier should be flushed |

**Behavior**:

cdb.out.valid = cdb.in.valid

cdb.out.data = cdb.in.data

cdb.out.id = cdb.in.id

cdb.flush.en = cdb.flushin.en

cdb.flush.id = cdb.flushin.id

**Module Description**

**Fetcher (PC + Branch Predictor)**

**Description**:

Continuously fetch instruction from instruction queue.

**Ports**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Size | From/To | Description |
| clk | in | 1 | top |  |
| rst | in | 1 | top |  |
| mem\_inst\_read | out | 1 | inst cache | Read enable |
| mem\_inst\_addr | out | 32 | inst cache | Predicted address |
| mem\_inst\_resp | in | 1 | inst cache |  |
| mem\_inst\_rdata | in | 32 | inst cache |  |
| dataout | out | 32 | inst queue | Send instruction data |
| is\_full | in | 1 | inst queue | 1-Instruction queue is full |
| pc\_next | out | 32 | inst queue/ROB(when issues to ROB  correct) | Send next instruction pc to cache and instruction queue (usually is pc\_current + 4), when a branch miss prediction, pc\_predicted <= pc\_rob |
| br\_addr | in | 32 | address calculator | Store The branched pc to do pc prediction |
| cdb.flush.en | in | 1 | ROB | To stop pc\_predicted and dataout output to queue, and change pc\_predicted to pc\_compared |
| wen | out | 1 | Inst cache | If valid, it’s sending correct value |

**Internal States:**

|  |  |  |
| --- | --- | --- |
| Name | Size | Description |
| pc\_pred | 32 | pc predicted by pc predictor |

**Behavior**:

If (is\_full):

Valid <= 0;

pc\_next <= pc\_next;

Else if(resp):

If (flush)

pc\_next <= pc\_compared;

Else if (instruction == branch) begin

Pc\_next <= pc\_predicted;

Else

Data\_out <= mem\_inst\_rdata;

Valid <= 1;

mem\_inst\_addr <= pc\_next;

**Instruction Queue (Size L)**

**Description**:

Buffer fetched instructions.

**Ports**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Size | From/To | Description |
| clk | in | 1 | top |  |
| rst | in | 1 | top |  |
| datain | in | 32 | fetcher |  |
| pc\_next | in | 32 | fetcher |  |
| wen | in | 1 | fetcher | 1-Input data is valid (write data in) |
| ren | in | 1 | issuer | 1-Issuer want to read data from queue (rvalid will go 1 at next cycle if data is vailable) |
| is\_full\_q | out | 1 | fetcher | 1-Instruction queue is full |
| is\_empty\_q | out | 1 | issuer | 1-instruction queue is empty |
| dataout | out | 32 | issuer | instruction |
| pc\_out | out | 32 | issuer | pc |
| cdb.flush.en | in | 1 | ROB |  |
| rvalid | out | 1 | issuer | 1-dataout is ready for read (only last for 1 cycle) |

**Internal States:**

|  |  |  |
| --- | --- | --- |
| Name | Size | Description |
| top | Log2(L) | The top of the fifo |
| bottom | Log2(L) | The bottom of the fifo |
| fifo[N] | 32 | fifo |

**Behavior**:

/\* Buffer instructions and predicted addresses \*/

/\* Flush if necessary \*/

**Issuer (Decoder & Issue Logic)**

**Description**:

Decode and issue instruction.

**Ports**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Size | From/To | Description |
| inst (dataout) | in | 32 | inst queue |  |
| pc\_out | in | 32 | inst queue |  |
| ren | out | 1 | inst queue | 1-Issuer want to read data from queue (rvalid will go 1 at next cycle if data is vailable) |
| rvalid | in | 1 | inst queue | 1-dataout is ready for read (only last for 1 cycle) |
| issue\_rs1 | out | 5 | regfile | Source register index 1 |
| issue\_rs2 | out | 5 | regfile | Source register index 1 |
| issue\_rd | out | 5 | regfile | Destination register index 1 |
| issue\_imm | out | 32 | RS | Extended value |
| issue\_addr | out | 32 | ROB | Send pc\_next to ROB |
| issue\_type | out | log2(num of inst) | ROB/Regfile | send instruction |
| is\_full\_ROB | in | 1 | ROB | if full, don’t need to issue |
| is\_ empty\_q | in | 1 | inst queue | if empty, don’t need to issue |
| is\_full\_res\_alu | in | 1 | reservation station alu | if alu reservation station is full and the it’s alu instruction, stop issue |
| is\_full\_br | in | 1 | reservation station br | if br reservation station is full and the it’s br instruction, stop issue |
| is\_full\_mem | in | 1 | load/store | if load/store reservation station is full and the it’s load/store instruction, stop issue |
| sel\_reservation | out | 2 | mux | choose which reservation to issue |

**Behavior**:

/\* Decode instruction \*/

/\* Read value/id from Regfile/ROB \*/

/\* Generate MUX control signal for RS inputs \*/

**Reservation Station (Size M)**

**Description**:

Reserve issued instructions.

**Ports**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Size | From/To | Description |
| clk | in | 1 | top |  |
| rst | in | 1 | top |  |
| Issuer | | | | |
| rs\_is\_full | out | 1 | issuer | 1-The RS is full |
| issue\_en | in | 1 | issuer | 1-Write instruction to RS |
| issue\_opc | in | ? | issuer | Opcode of instruction |
| ROB/Register File | | | | |
| rs\_id\_in | in | log2(N) | ROB | Identifier of instruction |
| rs\_opr1\_rdy | in | 1 | ROB | 1-Use value instead of id/0-Otherwise |
| rs\_opr1\_id | in | log2(N) | ROB | Identifier of oprand1 |
| rs\_opr1\_val | in | 32 | ROB/regfile/decoder | Value of first oprand1 |
| rs\_opr2\_rdy | in | 1 | ROB | 1-Use value instead of id/0-Otherwise |
| rs\_opr2\_id | in | log2(N) | ROB | Identifier of oprand2 |
| rs\_opr2\_val | in | 32 | ROB/regfile/decoder | Value of first oprand2 |
| Arithmetic unit | | | | |
| isidle | in | 1 | alu/cmp | 1-Current operation has finished execution |
| exe\_resp | out | 1 | alu/cmp | 1-Execute next instruction |
| rs\_opc\_out | out | ? | alu/cmp |  |
| Written by CDB | | | | |
| cdb.in | in | cdb itf | cdb |  |
| Wrtie to CDB | | | | |
| rs\_req | out | 1 | mux | Request control of the bus |
| rs\_dataout | out | 32 | mux | Data write to bus |
| rs\_id\_out | out | log2(N) | mux | Identifier of register (aka order of instruction) |
| cdb\_resp | in | 1 | mux | Respond from CDB |
| Flush | | | | |
| cdb.flush | in | cdb itf | cdb | Flush signals |

**Internal States:**

|  |  |  |
| --- | --- | --- |
| Name | Size | Description |
| valid | Mx1 |  |
| exe | Mx1 | 1-The instruction is being excuted |
| id | Mxlog2(N) | Identification of this instruction |
| opc | Mx3 | Opcode for operation type |
| opr1\_rdy | Mx1 | 1-Value for oprand1 is ready |
| opr1\_id | Mxlog2(N) | Identifier of instruction that will produce this value |
| opr1\_val | Mx32 | Value of source register/immediate value |
| opr2\_rdy | Mx1 | Same as before |
| opr2\_id | Mxlog2(N) | Same as before |
| opr2\_val | Mx32 | Same as before |

**Behavior**:

is\_full = & valid

// Issue instruction

if (wen)

/\* Load opcode and oprands \*/

/\* Set valid bits \*/

// Execute instruction

if (~(& busy))

(for each instruction)

if (valid & opr1\_rdy & opr2\_rdy) // By default oprand of singe-oprand instruction is stored in opr1, opr2\_rdy is set to high when issued

/\* Compete for execution \*/

/\* Execute the instruction if arithmetic unit is idle \*/

busy <= 1

// Written by bus (also need to consider the case that issue and write happens simultaneously – there is a mux that select between regfile output and cdb.in)

if (cdb.in.valid)

(for each opr1/2 in RS)

if (opr1/2\_id == cdb.in.id)

opr1/2\_val <= cdb.in.data

opr1/2\_rdy <= 1

// Write to bus

/\* Compete for the control of bus \*/

/\* Write to the bus, meanwhile reset valid bit of the line \*/

// Flush

(for each instruction)

if (cdb.flush.en && cdb.flush.id[id])

valid <= 0

if (busy)

/\* Reset ALU/CMP/Multiplier \*/

**Reorder Buffer (ROB, Size N)**

**Description**:

Buffer and commit instruction/PC.

**Ports**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Size | From/To | Description |
| clk | in | 1 | top |  |
| rst | in | 1 | top |  |
| Write value | | | | |
| cdb.in | in | cdb itf | CDB |  |
| Commit | | | | |
| commit\_reg\_en | out | 1 | regfile | 1-Commit a register |
| commit\_rd | out | 5 | regfile | Destination register (x0-x31, pc) |
| commit\_val | out | 32 | regfile | Value of register |
| commit\_ls\_en | out | 1 | memory unit | 1-Commit an oldest load/store instruction |
| commit\_ls\_id | out | log2(N) | memory unit | Identifier of load/store instruction |
| Issue instruction | | | | |
| issue\_req | in | 1 | issuer | Issue request from issuer |
| issue\_type | in | 2 | issuer | Type of instruction (alu/lui/auipc, br, load/store) |
| issue\_isrd | in | 1 | issuer | 1-The instruction has a destination register |
| issue rd | in | 5 | issuer | Destination register (x0-x31) |
| issue\_addr | in | 32 | issuer | Next address of instruction (used for checking branch prediction and committing PC) |
| is\_full | out | 1 |  |  |
| id | out | log2(N) | RS | Identifier that is assign to the issued instruction |
| Flush | | | | |
| br\_addr | in | 32 | address calculator | Correct address of branching (become available when a branch instruction is written into bus)  NOTE: also wire it to the fetcher! |
| br\_valid | in | 1 | address calculator | 1-The br\_addr and br\_id become valid |
| br\_id | in | log2(N) | address calculator | Identifier of instruction |
| cdb.flushin | out | cdb itf | CDB | Generate flush signal if needed |
| flush\_dep\_rf | out | 32xlog2(N) | regfile | Change the dependence id of register if necessary |

**Internal States:**

|  |  |  |
| --- | --- | --- |
| Name | Size | Description |
| p\_inst\_old | log2(N) | Pointer to the oldest instruction |
| p\_inst\_new | log2(N) | Pointer to the newest instruction |
| For N buffered instructions | | |
| valid | Nx1 | 1-The slot is occupied by an instruction |
| itype | Nx2 | Type of instruction (alu/lui/auipc, br, load/store) |
| isfinish | Nx1 | 0-The instruction is still executing |
| isrd | Nx1 | 1-The instruction will write to a general-purpose register |
| rd | Nx6 | Destination register of instruction |
| val | Nx32 | The value produced by that instruction |
| addr | Nx32 | Address of the next instruction |

**Behavior**:

// Issue

if (issue\_req)

id = newest\_inst + 1

newest\_inst <= newest\_inst + 1

/\* Store parameters/data/address of instruction… \*/

// Write

if (cdb.in.valid)

val[cdb.in.id] <= cdb.in.data

isdep <= 0

// Commit

if (isfinish[oldest\_inst])

/\* Commit instruction based on its type \*/

/\* Commit PC \*/

isvalid[oldest\_inst] <= 0

oldest\_inst <= oldest\_inst + 1

// Flush (when a branch instruction passes the address to ROB)

if (cdb.in.valid)

if (type[cdb.in.id] == br && addr[cdb.in.id] != br\_addr)

/\* Generate flush signal (0 or 1) for all N instruction \*/

/\* Set all valid bit of wrong instruction to 0 \*/

(Take care of relative indexes of oldest instruction and branch instruction)

/\* Deliver right address \*/

for (PC and x1-x32)

generate new

**Register File**

**Description**:

Storing 32 architecture state register and PC.

**Ports**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Size | From/To | Description |
| clk | in | 1 | top |  |
| rst | in | 1 | top |  |
| Read value (Issue) | | | | |
| issue\_sr1 | in | 5 | issuer | Source register 1 (x0-x31, pc) |
| issue\_sr2 | in | 5 | issuer | Source register 2 (x0-x31, pc) |
| regfile\_sr1\_rdy | out | 1 | RS | 1-Use value/0-Use id |
| regfile\_sr1\_id | out | log2(N) | RS | Identifier of dependent register |
| regfile\_sr1\_value | out | 32 | RS |  |
| regfile\_sr2\_rdy | out | 1 | RS | 1-Use value/0-Use id |
| regfile\_sr1\_id | out | log2(N) | RS | Identifier of dependent register |
| regfile \_sr2\_value | out | 32 | RS |  |
| Commit | | | | |
| commit\_reg | in | 1 | ROB | 1-Commit a register |
| commit\_rd | in | 5 | ROB | Destination register (x0-x31, pc) |
| commit\_val | in | 32 | ROB | Value of register |
| commit\_pc | in | 1 | ROB | 1-Commit PC |
| commit\_addr | in | 32 | ROB |  |
| Flush | | | | |
| cdb.flushin.en | in | 1 | ROB |  |
| flush\_depid | in | 32x(log2(N)+1) | regfile | Change the dependence id of register if necessary (1 valid bit in MSB) |

**Internal States:**

|  |  |  |
| --- | --- | --- |
| Name | Size | Description |
| pc | 32 |  |
| reg x0-x31 | 32x31 |  |

**Behavior**:

// Read

if (regfile \_sr1[5])

regfile\_sr1\_value = pc

else

regfile\_sr1\_value = reg[regfile \_sr1]

/\* Same for sr2 \*/

// Commit

if (commit\_reg)

reg[commit\_rd] <= commit\_val

if (commit\_pc)

pc <= commit\_addr

// Flush

…

**ALU/CMP**

**Description**:

Similar to ALU/CMP in mp2, but need additional port to communicate with reservation station. CMP also need to calculate the address and pass it to reorder buffer after getting control of CDB.

**Memory Unit**

**Description**:

Manage the load/store operation

**Ports**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Size | From/To | Description |
| Interface with data cache (similar to mp3, skip) | | | | |
| Interface with reservation station (similar to ALU/CMP, skip) | | | | |
| Commit | | | | |
| commit\_ls | in | 1 | ROB | 1-Commit an oldest load/store instruction |
| commit\_ls\_id | in | log2(N) | ROB | Identifier of load/store instruction |
| lsq\_finish | out | 1 | ROB | Load/store instruction finish |
| CDB | | | | |
| cdb.in | out | cdb itf | cdb | Write to CDB when a load instruction is committed |

**Internal States:**

|  |  |  |
| --- | --- | --- |
| Name | Size | Description |
| mar | 32 | Address register |
| mdr | 32 | Data register |

**Behavior**:

Load/store data when corresponding instruction is going to commit, also write to CDB when a load instruction gets its value from memory.

Note that PC (and any possible rd) is committed by ROB after receiving lsq\_finish signal.

**Arbiter**

**Description**:

It is the interface between cache and main memory. Both instruction cache and data cache connect to the arbiter, which interfaces with memory. Since main memory only has a single port, arbiter determines the priority on which cache request will be served first in the case when both caches miss and need to access memory on the same cycle.

**Ports**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Size | From/To | Description |
| clk | Input | 1 | From top |  |
| rst | input | 1 | From top |  |
| d\_pmem\_rdata\_256 | output | 256 | To data cache | 256-bit data bus for receiving data from physical memory. |
| d\_pmem\_resp | output | 1 | To data cache | Active high signal generated by the memory interface indicating that the memory operation has completed. |
| d\_pmem\_wdata\_256 | Input | 256 | From data cache | 256-bit data bus for sending data to physical memory. |
| d\_pmem\_address | input | 32 | From data cache | Physical memory is accessed using this 32-bit signal. It specifies the physical memory address that is to be read or written. |
| d\_pmem\_read | input | 1 | From data cache | Active high signal that tells the memory interface that the address is valid and the cache is trying to perform a physical memory read. |
| d\_pmem\_write | input | 1 | From data cache | Active high signal that tells the memory interface that the address is valid and the cache is trying to perform a physical memory write. |
| i\_pmem\_rdata\_256 | output | 256 | to instruction cache | 256-bit data bus for receiving data from physical memory. |
| i\_pmem\_resp | output | 1 | to instruction cache | Active high signal generated by the memory interface indicating that the memory operation has completed. |
| i\_pmem\_wdata\_256 | input | 256 | From instruction cache | 256-bit data bus for sending data to physical memory. |
| i\_pmem\_address | input | 32 | From instruction cache | Physical memory is accessed using this 32-bit signal. It specifies the physical memory address that is to be read or written. |
| i\_pmem\_read | input | 1 | From instruction cache | Active high signal that tells the memory interface that the address is valid and the cache is trying to perform a physical memory read. |
| i\_pmem\_write | input | 1 | From instruction cache | Active high signal that tells the memory interface that the address is valid and the cache is trying to perform a physical memory write. |

**Behavior**:

We use a state machine to depict the behavior of arbiter:

地图在桌子上

低可信度描述已自动生成

**State description:**

|  |  |
| --- | --- |
| Name | Description |
| Idle | Initial state of arbiter |
| Serve\_instruction | Serving the instruction cache |
| Serve\_data | Serving the data cache |

idle : begin

i\_pmem\_resp = 0;

i\_pmem\_rdata\_256 = 0;

d\_pmem\_resp = 0;

d\_pmem\_rdata\_256 = 0;

pmem\_read = 0;

pmem\_write = 0;

pmem\_address = 0;

pmem\_wdata\_256 = 0;

end

Serve\_instruction: begin

i\_pmem\_resp = pmem\_resp;

i\_pmem\_rdata\_256 = pmem\_rdata\_256;

d\_pmem\_resp = 0;

d\_pmem\_rdata\_256 = 0;

pmem\_read = i\_pmem\_read;

pmem\_write = i\_pmem\_write;

pmem\_address = i\_pmem\_address;

pmem\_wdata\_256 = i\_pmem\_wdata\_256;

end

Serve\_data: begin

i\_pmem\_resp = 0;

i\_pmem\_rdata\_256= 0;

d\_pmem\_resp = pmem\_resp;

d\_pmem\_rdata\_256 = pmem\_rdata\_256;

pmem\_read = d\_pmem\_read;

pmem\_write = d\_pmem\_write;

pmem\_address = d\_pmem\_address;

pmem\_wdata\_256 = d\_pmem\_wdata\_256;

end

**Instruction Cache/Data Cache**

**Description**:

Similar to cache in mp3.