Design and implementation of a RISC simulator

Wei Hong
weihong@cs.umass.edu

Tengyu Sun
tsun@cs.umass.edu

College of Computer Science, University of Massachusetts
Amherst, MA 01003

ABSTRACT

In this report, we design a MIPS-like instruction set and implement a simulator for it. The simulator supports cached memory and pipelined execution. In order to evaluate its performance, we design some benchmarks and compare the clock cycles under different simulator configurations.

1. INTRODUCTION

This report describes the design and implementation of a RISC instruction set and its simulator. The instruction set is a subset of the MIPS instruction set[1] with some modifications for simplicity. It supports basic instructions such as data transfer, flow control, integer and floating point computations. It also supports the advanced feature SIMD. The simulator is implemented in C++ with QT for the GUI. The backbone consists a hierarchical multi-way associative memory cache system and a pipelined CPU architecture with a Floating Point Unit (FPU) and a Vector Unit (VU). A few benchmark programs including exchange sort and matrix multiplication are used to test the performance of the simulator. An assembler is implemented to bridge the assembly language and the instruction set architecture.

The report is structured as follow: in section 2, we will introduce the design of our instruction set and then in section 3, we describe the implementation of the simulator. In section 4, we report the performance evaluation results. At last, we summary the report in section 5.

2. INSTRUCTION SET DESIGN

The instruction set follows the RISC design strategy. It contains 6 types of instructions, namely data transfer, arithmetic and logical, control, floating point, cache and SIMD. The data types supported are 8-bit byte, 32-bit word integer and 32-bit single precision floating point. The memory is byte addressable and the byte ordering is big-endian. Word access requires alignment.

[1] `https://imgtec.com/mips/architectures/mips64/`

Table 1: Instruction Format

Format	Field (32 bits)				
1	opcode (7)	offset (25)			
2	opcode (7)	$1 (4)	$2 (4)	immediate (17)	
3	opcode (7)	$1 (4)	$2 (4)	$3 (4)	X

There are 16 32-bit general purpose registers for integers (gpr[0] - gpr[15]), 16 32-bit registers for floating point (fpr[0] - fpr[15]), 16 64-bit vector registers for SIMD (vr[0] - v[15]). gpr[0] - gpr[15] and fpr[0] - fpr[15] can be accessed by load/store instructions. gpr[15] is usually used by flow control instructions as the default register for return address. vr[0] - vr[15] can hold 8 8-bit byte integers simultaneously. The vector element index starts from the most-significant bit, that is vr[i][0] is the higher 8 bits of vr[i]. There are also one 32-bit register for program counter ($pc) and one 32-bit status register ($status). $pc cannot be directed accessed and can only be changed by special instructions.

Instructions use fixed length encoding. Each instruction is 32-bit long. The addressing mode are immediate and displacement. The Register index requires 4 bits and the immediate operand can have up to 17 bits. The operation code is 7-bit long. The first 3 bits are used for distinguishing 6 instruction types. Each instruction can have up to three operands. Like in MIPS, depending on the number of operands, there are three kinds of instructions. Format 1 only has an offset field which is up to 25 bits. Format 2 has two register operands and one immediate. Format 3 has three register operands. The instruction format is described in Table 1.

The instruction set has 6 types. Type 1 Data transfer are in charge of loading/storing value from/to memory. Type 2 Arithmetic and logical are basic integer computation instructions. They can operate on register operand or immediate value in instructions. Type 3 Control instructions are jump and branching instructions. Also it also include break for debugging. Type 4 Floating point instructions perform floating point calculations and also integer conversion. Type 5 Cache instruction can preload a word into cache without writing to register files. Together with data transfer instructions, they are the only instruction types that can access memory. At last, Type 6 SIMD instructions can do vector operations and transfer values between integer registers and vector registers. Details about each instruction and its description are listed in Table 2.

3. SIMULATOR

3.1 Overview

The simulator follows the classic Model View Control (MVC) design. A simplified UML of the simulator is shown in Figure 1 which gives an overview of the architecture. A `simulator` class

Table 2: Instruction List

Type	Instruction	Encoding	Format	Description
Data Transfer	lb	0010000	lb $1,$2,im	load mem[$1+im] and sign extended into gpr[$2]
	lbu	0010001	lbu $1,$2,im	load mem[$1+im] into gpr[$2]
	sb	0011000	sb $1,$2,im	store lower 8 bits of gpr[$2] into mem[$1+im]
	lw	0010010	lw $1,$2,im	load mem[$1+im ... $1+im+3] into gpr[$2]
	sw	0011001	sw $1,$2,im	store gpr[$2] into mem[$1+im ... $1+im+3]
	lsp	0010100	lsp $1,$2,im	load mem[$1+im ... $1+im+3] into fpr[$2]
Arithmetic & Logical	add	1000000	add $1,$2,$3	gpr[$3] = gpr[$1] + gpr[$2]
	sub	1000001	sub $1,$2,$3	gpr[$3] = gpr[$1] - gpr[$2]
	addi	1010011	addi $1,$2,im	gpr[$2] = gpr[$1] + im
	subi	1010100	subi $1,$2,im	gpr[$2] = gpr[$1] - im
	mul	1000010	mul $1,$2,$3	gpr[$3] = lower 32 bits of (gpr[$1]*gpr[$2] as signed value)
	muh	1000011	muh $1,$2,$3	gpr[$3] = higher 32 bits of (gpr[$1]*gpr[$2] as signed value)
	mulu	1000100	mulu $1,$2,$3	gpr[$3] = lower 32 bits of (gpr[$1]*gpr[$2] as unsigned value)
	muhu	1000101	muhu $1,$2,$3	gpr[$3] = higher 32 bits of (gpr[$1]*gpr[$2] as unsigned value)
	div	1000110	div $1,$2,$3	gpr[$3] = gpr[$1] / gpr[$2] as signed value
	divu	1000111	divu $1,$2,$3	gpr[$3] = gpr[$1] / gpr[$2] as unsigned value
	modu	1001000	modu $1,$2,$3	gpr[$3] = gpr[$1] % gpr[$2] as unsigned value
	and	1001001	and $1,$2,$3	gpr[$3] = gpr[$1] & gpr[$2]
	or	1001010	or $1,$2,$3	gpr[$3] = gpr[$1] | gpr[$2]
	not	1001011	not $1,$0,$3	gpr[$3] = $\sim$ gpr[$1] ($0 is not relevant but required)
	xor	1001100	xor $1,$2,$3	gpr[$3] = gpr[$1] $\wedge$ gpr[$2]
	rr	1001101	rr $1,$2,$3	gpr[$1] rotate right $2 bits and store in gpr[$3]
	srl	1001110	srl $1,$2,$3	gpr[$1] logical shift right $2 bits and store in gpr[$3]
	sra	1001111	sra $1,$2,$3	gpr[$1] arithmetic shift right $2 bits and store in gpr[$3]
	sl	1010000	sl $1,$2,$3	gpr[$1] shift left $2 bits and store in gpr[$3]
	slt	1010001	slt $1,$2,$3	gpr[$3] = (gpr[$1] < gpr[$2]) as signed value
	sltu	1010010	sltu $1,$2,$3	gpr[$3] = (gpr[$1] < gpr[$2]) as unsigned value
	slti	1010101	slti $1,$2,im	gpr[$2] = (gpr[$1] < im) as signed value
	sltiu	1010110	sltiu $1,$2,im	gpr[$2] = (gpr[$1] < im) as unsigned value
Control	j	0000001	j offset	jump to offset ($pc = offset)
	jal	0000010	jal offset	jump to offset and put current $pc to gpr[15]
	beq	0000011	beq $1,$2,offset	branch an offset ($pc = $pc +offset) if gpr[$1] == gpr[$2]
	bneq	0000100	bneq $1,$2,offset	branch an offset ($pc = $pc +offset) if gpr[$1] != gpr[$2]
	bgez	0000101	bgez $1,offset	branch an offset ($pc = $pc +offset) if gpr[$1] >= 0
	bgtz	0000110	bgtz $1,offset	branch an offset ($pc = $pc +offset) if gpr[$1] > 0
	blez	0000111	blez $1,offset	branch an offset ($pc = $pc +offset) if gpr[$1] <= 0
	bltz	0001000	bltz $1,offset	branch an offset ($pc = $pc +offset) if gpr[$1] < 0
	break	0000000	break	break
Floating Point	addsp	0100000	addsp $1,$2,$3	fpr[$3] = fpr[$1] + fpr[$2]
	subsp	0100001	subsp $1,$2,$3	fpr[$3] = fpr[$1] - fpr[$2]
	mulsp	0100010	mulsp $1,$2,$3	fpr[$3] = fpr[$1] * fpr[$2]
	divsp	0100011	divsp $1,$2,$3	fpr[$3] = fpr[$1] / fpr[$2]
	sltsp	0100100	sltsp $1,$2,$3	fpr[$3] = (fpr[$1] < fpr[$2])
	witf	0100101	witf $1,$2	convert integer gpr[$1] to floating point and store in fpr[$2]
	wfti	0100110	wfti $1,$2	convert floating point fpr[$1] to integer and store in gpr[$2]
Cache	pref	0110000	pref $1,$0,im	load mem[$1+im ... $1+im+3] into cache ($0 is not relevant but required)
SIMD	move	1101010	move $1,$2	vr[$2] = vr[$1]
	copys	1101011	copys $1,$2,n	store n-th byte in vr[$1] to gpr[$2] as signed value
	copyu	1101100	copyu $1,$2,n	store n-th byte in vr[$1] to gpr[$2] as unsigned value
	insertb	1101101	insertb $1,$2,n	store lower 8 bits of gpr[$1] to the n-th byte in vr[$2]
	fillb	1101110	fiilb $1,$2	fill lower 8 bits of gpr[$1] to all bytes in vr[$2]
	vaddb	1100000	vaddb $1,$2,$3	vr[$1] = vr[$2] + vr[$3]
	vsubb	1100001	vsubb $1,$2,$3	vr[$1] = vr[$2] - vr[$3]
	vmulb	1100010	vmulb $1,$2,$3	vr[$1] = vr[$2] * vr[$3]
	vdivb	1100011	vdivb $1,$2,$3	vr[$1] = vr[$2] / vr[$3]
	vmodb	1100100	vmodbb $1,$2,$3	vr[$1] = vr[$2] % vr[$3]
	ceqb	1100101	ceqb $1,$2,$3	vr[$3] = (vr[$1] == vr[$2]) (element wise)
	cleb	1100110	cleb $1,$2,$3	vr[$3] = (vr[$1] <= vr[$2]) (element wise, signed value)
	cleub	110111	cleub $1,$2,$3	vr[$3] = (vr[$1] <= vr[$2]) (element wise, unsigned value)
	cltb	1101000	cltb $1,$2,$3	vr[$3] = (vr[$1] < vr[$2]) (element wise, signed value)
	cltub	1101001	cltub $1,$2,$3	vr[$3] = (vr[$1] < vr[$2]) (element wise, unsigned value)

sits in the middle acting as a coordinator for the rest classes. The model part consists of a `CPU` class and a `MemSys` class. The `CPU` class handles most of the instruction executions. It also contains a `FPU` and a `VU` for dealing with floating point and vector operations. The `MemSys` controls all the memory accesses. It has a `Cache` and a `Memory` as components. `Cache` and `Memory` are both inherited from the `Storage` class which makes memory hierarchies easy to implement. Three view classes `CPUView`, `CacheView` and `MemoryView` display the CPU, cache and memory status. The model-view communications use QT's signal and slot model instead of callback handles. This makes adding new interactions easier. Besides, `Simulator` also includes a `ConfigDialog` class which can change the system setting during runtime. We will introduce each part in detail in the next few sections.

3.2 Memory and cache system

We use a hierarchical structure to build the memory-cache system. First, we define an abstract class called `Storage`. Both `Cache` and `Memory` inherits from this base class. The `Storage` class contains an integer field `cycle` defining the number of cycles it needs to execute an instruction and a pointer `nextLevel` that references to the next level of storage object, if any. It also supports 3 methods `load`, `store` and `dump`. The `load` method takes a 32-bit integer address, a pointer to the buffer for storing loaded values and the length of the memory blocks to be read as the arguments. It copies the content in the memory starting at the address into the buffer of the same size. The `store` method takes the same three arguments as the `load` method. Instead of loading the blocks into the buffer, it takes the content from the buffer and writes them into the blocks from the starting address. The `dump` method converts and concatenates the content in the storage into a string for debugging.

3.2.1 Memory

The `Memory` class inherits from the `Storage` class, it implements the methods inherited from the `Storage` class straight forwardly. The data is stored in an byte array so it is byte addressable. The pointer to the next level is set to be a null pointer.

3.2.2 Cache

An overview of the cache memory hierarchy is shown in Figure 2. The `Cache` class we implemented supports multiple-way associativity with write back/through and LRU/Random policies. The `Cache` class also inherits from the `Storage` class. It consists of multiple cache lines as the basic building block.

Cache line.

The cache line is defined using the struct `Cacheline`. It has a boolean field called `valid` that indicates if this cache line is empty or not, a boolean field `dirty` that indicates if the content in the cache line is newer than that in the lower level storage. The `lru` field indicates its priority in the LRU replacing mechanism. It also contains a pointer `data` referencing to an array of bytes that served as the data blocks.

The cache consists of a two-dimensional array of `Cacheline`. The rows correspond to indices of memory blocks and the columns correspond to the ways associated to the same index. The total size of cache is the number of rows times the number of columns. The size of the cache, number of ways and number of index can be set at the initialization stage.

Cache functions.

The `inCache` method takes an address and return the cache line position if the address exists in the cache or null if not. Since the memory address starts from 0, it calculates the block index as (address / cache line size) and tag value as (address $\%$ cache line size). The block index corresponds to the index in the cache array. Then it iterates all the ways of cache lines attached to that index to see if any tag matches the tag of this address. If there is any match, it returns the position of the cache line, otherwise null.

The `evict` method takes a requested address and return an clean cache line position corresponding to that address. It looks up in the cache line array and search for any empty lines of that address, which means that the valid flag is false. If such line exists, it returns its position. If all lines are occupied, which means all valid flags is true, it evicts a line according to the replace policies. We have implemented two replace policies: (1) random eviction (2) LRU eviction. Under the former scenario, a random number between $[0, \text{number of ways} - 1]$ is generated. If the dirty bit of this cache line is true, which means it contains newer value than the lower level of storage, it will be evicted to the lower level of storage. Then the cache line is emptied and the position of it is returned. The LRU is implemented as follows. Each cache line at a specific index is assigned a LRU number. When a line at this index is updated, its LRU number is set to 0, all others are incremented by 1. Then higher number means the less recently used.

The `load` method allows us to read a block starting at a specific address in memory. It first checks if the address exists in the current cache level by calling the `inCache` method. If the returned position is null, there is a cache miss. It will call the the next level of storage's `load` method, which could be another cache or the memory. The data will be passed along the hierarchical call chain. The newly loaded data have to be written into the cache. An empty cache line is created by calling the `evict` method. If the returned position is not null, there is a cache hit. The data in the current cache line is loaded and passed up.

The `store` method allows us to write an array of data blocks into a specific address in memory. First, it also checks if the cache line corresponding to the requested address exists in the cache by calling the `inCache` method. If yes, it reports a cache hit and copies the data blocks into the current level of cache. Two write policies are implemented: (1) write back (2) write through. In the former scenario, we just set the dirty flag of the cache line to be true, update the LRU numbers and done. In the latter scenario, we still need to call the `store` method in the next level of storage to pass the data down. If the address doesn't have a corresponding cache line in the cache, we have a cache miss. The `load` method is called to bring the data blocks from next level into the cache. Then it's treated as in the cache hit case.

3.2.3 Memory system

The `MemSys` acts as a mediator between the cache-memory hierarchy and the CPU. Both `Memory` and `Cache` are encapsulated in it. The `MemSys` class exposes four method `loadWord`, `loadByte`, `storeWord`, `storeByte` for CPU fetching instructions or issuing load/store instruciton. It also keeps a `countdown` integer for CPU clocking. Each time CPU issues an request, it calculates the required cycles to complete that request and only return the result after that cycle has passed. Internally, `MemSys` uses a string `request` keeping the current serving request, if a different request comes in (possible under the pipeline mode), it will not response until it finishes its current serving request. That means this memory system has only one port and only allows a single access at a time.

The `MemSys` is also responsible for communication between the `Simulator` class. Whenever the `Memory` or `Cache` changes its

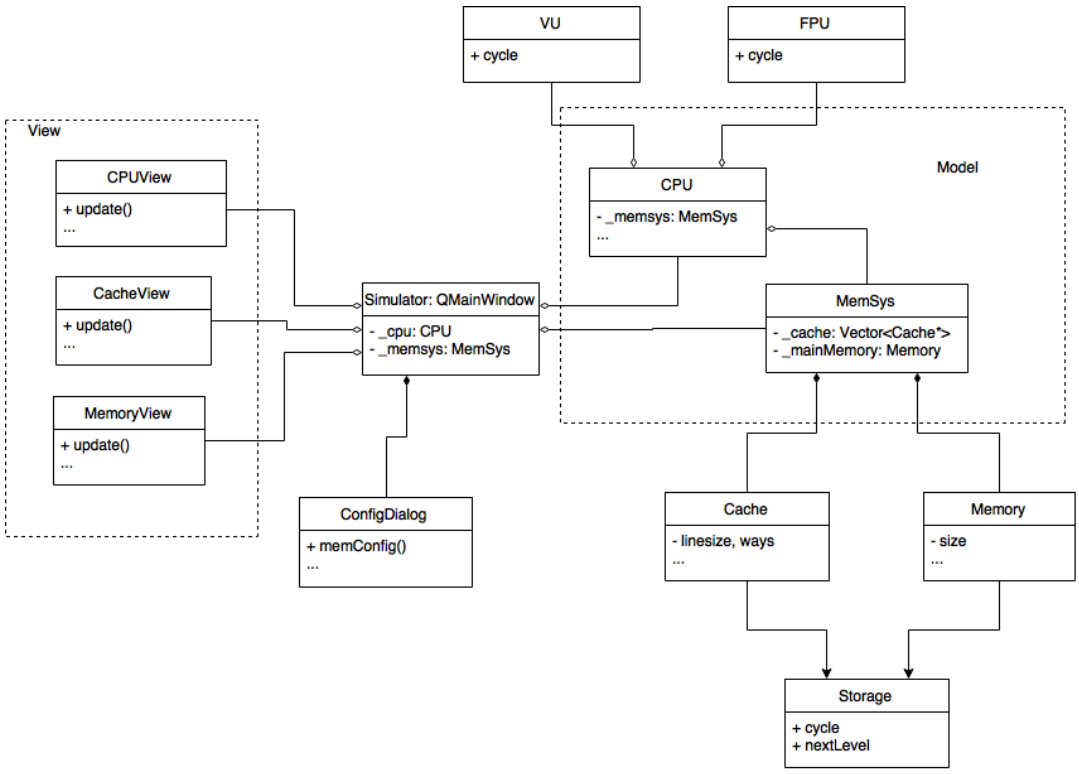

Figure 1: Overview of simulator architecture

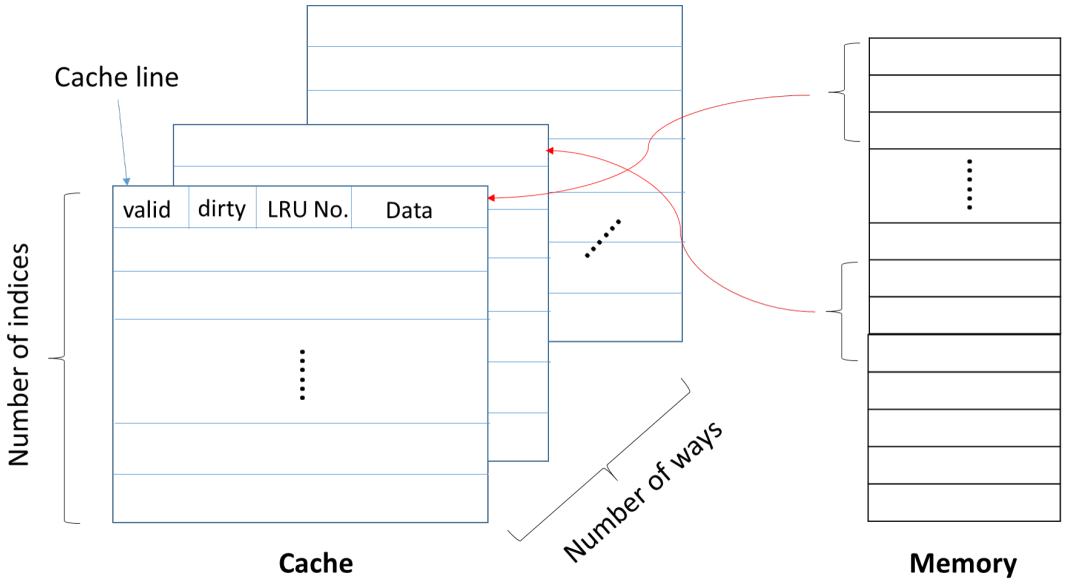

Figure 2: Schematic of memory and cache implementation

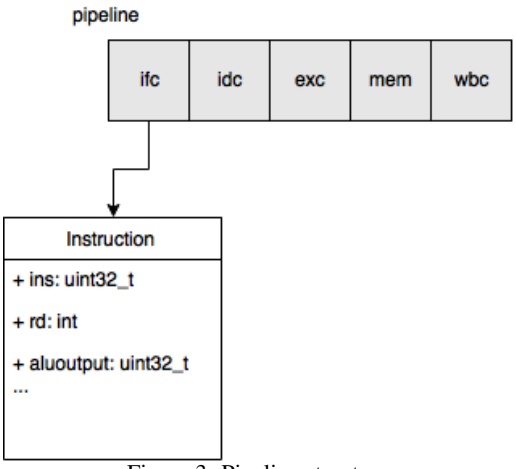

Figure 3: Pipeline structure

internal status, it will emit a notifying signal to the `Simulator`, and `Simulator` will in turn call the update methods in the corresponding view. `Simulator` can talk to the `MemSys` to change its configuration. When a user modifies the memory system configuration in the `ConfigDialog` class, the `Simulator` will notify `MemSys` through signaling and `MemSys` will initialize a new setting of cache-memory hierarchy by calling the `init` method.

`MemSys` provides one layer of indirection. It hides the implementation of the cache and memory and makes the interfaces between other classes easier to maintain.

3.3 CPU

The `CPU` class is where most of the simulations happens. This class keeps a `clk` field for clock cycle, a `pc` field for the program counter, and also the register files and pipeline states. It maintains a pointer to the `MemSys` for memory access and pointers to a `FPU` and a `VU` for special instructions. Besides, it can communicate with `Simulator` to notify the update of CPU status and get the commands from user interface.

3.3.1 Pipeline

The `CPU` class uses a standard 5-stage MIPS pipeline. Every instruction can be completed in 5 cycles: instruction fetch cycle (ifc), instruction decode cycle (idc), execution cycle (exc), memory access cycle (mem) and write-back cycle (wbc). There are five methods `ifc`, `idc`, `exc`, `mem` and `wbc` implementing these cycles. The pipeline `pipe` is implemented as a array of length 5. Each position in the array corresponds to a execution stage (see Figure 3). Elements in the array are the structure `Instruction`. `Instruction` keeps information about the instruction being executed. It resembles the registers between pipeline stages in hardware. Some key fields are `opcode` for keeping instruction code, `rd1`, `rd2`, `rd3`, `imm` for storing operands, `stage` for indicating execution stage and `aluoutput`, `fpuoutput`, `vuoutput`, `lmd` for keeping execution cycle result. The execution of the instruction is simulated by passing the structure through the `pipe` array and filling its fields in different stages.

ifc.

Fetch instructions from `MemSys` by calling `loadWord` method and create a `Instruction` instance. If in a pipelined execution mode, examine the instruction at `mem` stage to see if it is a branch. If it is a branch and it is taken, change the `pc` accordingly. Otherwise increase the `pc` by 4. If the next stage in `pipe` is empty, change its `stage` to 1 and pass the structure to the next stage.

idc.

Decode the instruction. First extract the 7 bits to get the instruction opcode, and then decode the rest operands according to the format (Table 1). After decoding, load operands from register files. If under the pipelining mode, we need to resolve potential data hazards. In the 5-stage pipeline setting, we only need consider read after write (RAW) hazard since the write back cycle is at the last stage and only one instruction can write at a time. Examine every instruction in the pipeline after idc stage and see if its destination register is one of the operands we need to read. If the instruction has finished its execution, we can forward the result in `aluoutput` (or `fpuoutput`, `vuoutput`, `lmd`) to the operand we need. Otherwise, we need to stall the pipeline, and wait for the operand to be ready. When the operands are ready, we can pass the instruction to the next stage.

exc.

Execute the instruction according to its opcode. If it only needs integer computation, we can finished the execution within the `CPU` class, otherwise, we need to forward the instruction to `FPU` or `VU` for special handling. In `CPU`, we can do data transfer, arithmetic and logical, control and cache instruction execution. If it is data transfer, we compute the memory address and store it in `aluoutput`. If it is arithmetic and logical, we save the computation result in `aluoutput`. And if it is control, we calculate the new `pc` address and put it in `aluoutput`. If it is floating point, we forward the instruction to `FPU` and wait for the result to store in `fpuoutput`. Similarly, SIMD instruction will be forwarded to `VU` and result saved in `vuoutput`. After computation, the instruction is passed to the mem stage.

mem.

This stage is only for data transfer instructions. If it is a load instruction, we load memory use the address stored in `aluoutput` and put it into `lmd`. If it is a store instruction, we store the value in operand to the address in `aluoutput`. Other type of instructions can directly pass this stage if the next stage is empty.

wbc.

Write `aluoutput` (or `fpuoutput`, `vuoutput`, `lmd`) to destination registers and destroy the `Instruction` structure.

The `step` method calls each stage function in reverse order (from `wbc` to `ifc`) and increase the `clk` by 1 afterwards. This completes one step of simulation. To simulating the whole program, we can call `step` until there are no more instructions.

3.3.2 FPU

`FPU` handles floating point operations. It can simulate multi-cycle floating point calculations. Internally, it has a `cycle` and `countdown` field. Like `MemSys`, it will only response request from `CPU` after certain number of cycles. Since it is separated from `CPU`, for future extension, the pipeline can support multi-issue easily.

3.3.3 VU

`VU` deals with vector operations. It also support multi-cycle calculation. Internally, vectors are saved as an array of bytes and computations are element wise. The separation of `VU` is also for future implementation of multi-issue.

3.4 Assembler

The assembler takes the instructions in assembly language and encode them into 32-bit integers. The assembler executes in two passes. In the first pass, it strips all the comments that starts with # symbol and stores all labels into a label map, which use the line number as the value. In the second pass, it replaces the labels in other clauses with the relative line numbers and encodes the instructions according to a preloaded map of opcodes. The result is output in binary format to a file.

3.5 User interface and use guide

The graphical interface is implemented in QT. A screenshot of the GUI is shown in Figure 4. The main window is the `Simulator` class. It can be roughly divided into three parts.

The left panel is for displaying CPU status which is implemented by `CPUView` class. Clock cycle, program counter, register files and pipeline can be seen here. There are also several buttons controlling the simulation. Pipeline `on/off` button can turn on/off the pipeline. `step` can simulate one clock cycle. `run` button executes the program at the speed of 100 millisecond one cycle. In this mode, you can see the changes in CPU or memory system easily. You can also stop the simulation whenever you want. `exe` button runs the program at full speed. Usually you can only see the final status of the program. `reset` button can clear the pipeline and register files and set the clock and pc to 0.

The middle panel is for visualizing cache. It has tabs for switching between different level of caches and on the top, there is an `on/off` button for turning on/off cache. When the cache is turned off, the tab panel will become invisible. This part is implemented by `CacheView`.

Finally, the right part is `MemoryView` where you can observe memory data and also change the display format. At the scroll area, you can press the address button and add or remove a breakpoint to the program. If the button is pressed down (turn blue), it means it is a breakpoint.

You can change the memory system configuration using the `configs` menu. After click the menu, a dialog (Figure 5) will show up. New cache and memory configurations can be set here. This dialog is a class of `ConfigDialog` and is a component of `Simulator`. The `Memory System` also provides options to dump the memory data into a file or clear the memory. The `Program` provides a way to load program direct into memory.

3.6 Testing

random, sequence access for memory small programs for CPU

4. PERFORMANCE EVALUATION

4.1 Benchmark

We designed the following benchmark programs: (1) Exchange sort (2) Matrix multiplication (3) Matrix multiplication with SIMD for performance evaluation.

4.2 Experiment result

5. SUMMARY

pipeline itself does not add much. memory is the bottle neck. a good cache will boost performance.

good software engineer design is crucial. signal slots add a mediator too many callback links.

Algorithm 1: Exchange sort(integer array A)

```
l = length of A;
for i from 0 to l - 2 do
    for j from i + 1 to l - 1 do
        if A[i] > A[j] then
            swap A[i] and A[j];
        else
            continue;
        end
    end
end
```

Algorithm 2: Matrix multiplication(left matrix L, right matrix R)

```
a = number of rows in L;
b = number of columns in L;
c = number of columns in R;
Create result matrix res of size a*c, initialized with 0s;
for i from 0 to a - 1 do
    for j from 0 to c - 1 do
        for k from 0 to b - 1 do
            res[i][j] += L[i][k]*R[k][j];
        end
    end
end
return res;
```

Algorithm 3: 2 by 2 Matrix multiplication with SIMD(left matrix L, right matrix R)

```
Create result matrix res of size 4, initialized with 0s;
Flatten L into a vector v1 of size 8 by rows;
Flatten R into a vector v2 of size 8 by columns;
Create vector v3 with size 8;
for i from 0 to 7 do
    v3[i] = v1[i] * v2[i];
end
for i from 0 to 3 do
    for j from 0 to 3 do
        res[i][j] = v3[(i*2+j)*2] + v3[(i*2+j) + 1];
    end
end
return res;
```

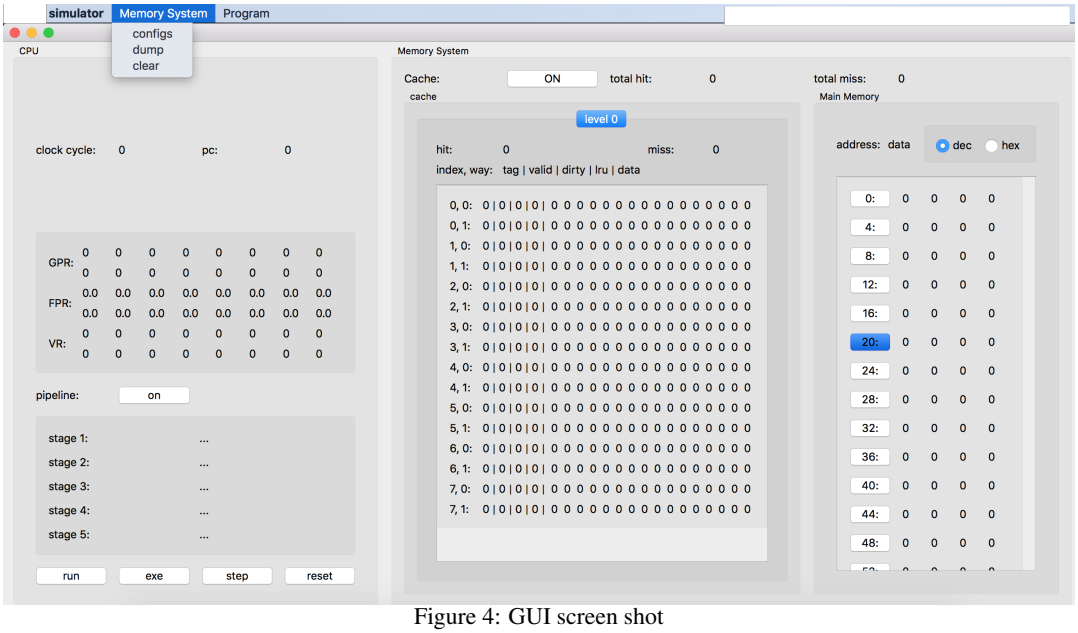

Figure 4: GUI screen shot

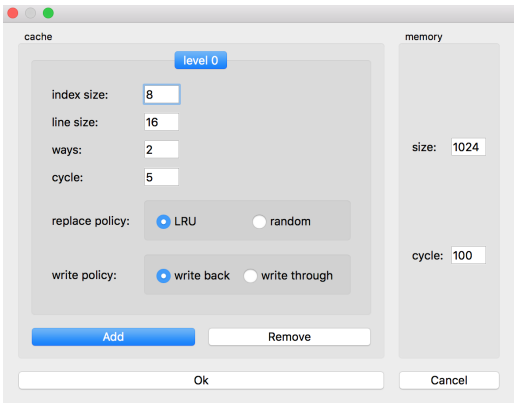

Figure 5: Configuration Dialog