Year 2 Project

System on chip with RISC-V microprocessor and Linux operating system

By: Gan Fang (ID: 201447502)

Qiyang Ding (ID: 201447455)

Yufeng Chen (ID: 201316122)

Supervised by: Mark Bowden

Department of Electrical Engineering and Electronics

17 April 2020

Abstract

The objective of this project is to develop a SoC and an embedded operating system that can run on this SoC. This can be divided into three components, including SoC design, operating system design and operating system transplant. The SoC is implemented using Verilog hardware description language. The whole designing, simulating and testing process are on the IDE software. For operating system, it uses the xv6 as basic structure and design each component referring to the online tutorials. The whole designing and testing process are in Ubuntu environment. It is compiled after each component is finished and run on the simulating environment. The bootloader is used in transplant phase. It reads the operating system image file from USB flash disk and start running the system. Although this project was not fully completed, the SoC can run on the board and the operating system can run on the simulating environment successfully. This project improves students' understanding of computer architecture and operating system, and their programming skills.

Declaration

I confirm that I have read and understood the University's definitions of plagiarism and collusion from the Code of Practice on Assessment. I confirm that I have neither committed plagiarism in the completion of this work nor have I colluded with any other party in the preparation and production of this work. The work presented here is my own and in my own words except where I have clearly indicated and acknowledged that I have quoted or used figures from published or unpublished sources (including the web). I understand the consequences of engaging in plagiarism and

Content

List of Figures

List of Table

Chapter 1: Introduction

1.1 Background

In early 1960s, IBM developed four incompatible computers, which included different instruction set architectures, I/O systems and so on. These computers opened a new world for computer, especially computer architecture. Since 1970s, due to the rapid progress of MOS technology, there appears many instruction set architectures and minicomputers, and gradually, complicated instruction set computer (CISC) constructed by Intel such as Intel 8086 and 80386 become the microprocessor with high performance. Nowadays, according to the Jeff Dean and David Patterson, who is the chief researcher in Google and the latter one is the professor of University of California, Berkeley, a new golden age has started in computer architecture and it is relative to the development of machine learning [21]. Until now, MIPS, x86, x86-64, Arm, VLIW and many microarchitectures has exhibited their values in different areas and among these architectures, Reduced Instruction Set Computer (RISC) has gradually come to the center of the stage. It contains less instruction set so that it executes more instructions per program, but it can use less cycles to achieve the program. RISC-V is based on the reduced instruction set architecture and it is developed by David A. Patterson, from University of California, Berkeley. It is an open-source instruction set; thus, it was developed by different people and was applied in many different spheres such as deep learning accelerator and embedded system.

In addition, the other topic of the project is Linux operating system. It has become an open-source operating system and because of its characteristics of free and open source, it is widely applied in embedded system, which is also one of the most hopeful direction of RISC-V microprocessor. Therefore, RISC-V and Linux operating system for any release version will be an interesting composition in the embedded system area and also be helpful for the computer architecture or computer system. Particularly, in this project, xv6 is designed and used as the embedded system.

As for the bootloader, U-boot, which is a bootloader program for embedded CPU

developed by German DENX group, is used in this project in the transplant phase. U-Boot has strong command line functions and interactive capabilities, it can also be used as a tool for hardware testing and demonstration. After years of development, it has become the industry standard for bootloaders. Most embedded devices now use U-Boot as the bootloader by default.

1.2 Objectives

In this project, RISC-V microprocessor and Linux (xv6) will be the core parts. It aims to construct a system on chip (SoC) to run the transplanted operating system and enable it to act as a minicomputer such as getting input from keyboard. This project will improve the understanding of OS and microprocessor and programming skills.

Chapter 2: Materials and Methods

2.1 Materials

2.1.1 Software and Hardware

1) Microsoft Windows 10 Professional 64 Bit
2) Xilinx Vivado 2019.1 WebPack
3) Github
4) RARS – RISC-V Assembler and Runtime Simulator
5) Visual Studio Code
6) Virtual Box
7) Ubuntu 18.04
8) vim
9) gedit
10) riscv32-unknown-elf-gcc
11) qemu-system-riscv32
12) u-boot source
13) Digilent Nexys A7 (Xilinx Artix-7 FPGA XC7A100T-1CSG324C)

2.1.2 Reference books, articles, and web pages

2.1.2.1 Book

1) Computer Organization and Design RISC-V Edition [1]
2) Digital Design and Computer Architecture [2]
3) Introduction to computing system [3]
4) xv6: a simple, Unix-like teaching operating system [18]

2.1.2.2 Articles and Documents

1) Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions v4.2 User Guide (UG582) [4]
2) Vivado Design Suite AXI Reference Guide (UG1037) [5]

3) Combining Branch Predictors [6]
4) Slave Memories and Dynamic Storage Allocation [7]
5) RowHammer: A Retrospective [8]
6) The RISC-V Instruction Set Manual Volume I: Unprivileged ISA [9]
7) Note 5 of ECE-495/595 in ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT, OAKLAND UNIVERSITY [10]
8) The UNIX time-sharing system [19]
9) Bell labs and CSP thread [20]

2.1.2.3 Web pages

1) Longxin Cup final report (github/TrivialMIPS) [11]
2) lowrisc-chip (github/lowRISC) [12]
3) riscv_soc (github/ultraembedded) [13]
4) Nexys A7 Reference Manual [14]
5) Xilinx forum [15]
6) Digilent forum [16]
7) Design of Digital Circuit (course of ETH Zurich) [17]

2.2 Operating system part

2.2.1 Process

Week 1

Install the running environment (including *riscv32-unknown-elf-gcc* and *qemu-system-riscv32)*. *riscv32-unknown-elf-gcc* is the cross compile, which can compile the operating system source code so that it can be used in 32 bit RISC-V. *qemu-system-riscv32* can simulate the 32 bit RISC-V environment on Ubuntu. Therefore, we can test the correctness of operating system without a completed CPU. After finishing the installation, write a simple C program to test the correctness. Firstly, use the *riscv32-unknown-elf-gcc* to compile the source code and generate an executable file. Then, use *qemu-system-riscv32* to run this executable file.

Week 2

Implement several shell commands, such as *sleep, pingpong, primes, find and xargs*.

1. *sleep* can pause the operating system for a given time. (eg. *sleep 100*: the system will pause for 10 second)

2. *pingpong* can transmit a byte message between two processors via a pair of pipes, *parent_pipe* and *child_pipe*, one for each direction (in and out). The parent process will write message to *parent_pipe[1]* and the child receive this message from *parent_pipe[0]*. Then child responds with writing a message to *child_pipe[1]* and parent can read from *child_pipe[0].*

3. *primes* is based on the idea Unix Pipes. The first process feeds the numbers 2 through 35 into the pipeline. For each prime number, you will arrange to create one process that reads from its left neighbor over a pipe and writes to its right neighbor over another pipe.

4. *find* is used to find the directory or file under the given path. For example, we make a directory *aaa* and then create a file *bbb* inside *aaa*. If we want to find all files called *bbb* in current directory, we just need to write *find . bbb*. The second argument . means it starts in the current directory.

5. *xargs* reads the whole line from standard input and run a command for this line and supplying this line as arguments for the command. For example, echo can only print one argument, but if we use *xargs*. It can print all arguments in a line. It just like echo is called several times.

Week 3

Implement a simulated shell, called *nsh*. When running the operating system, write *nsh* in command line to start. In this simulated shell, we can also run some commands as in the original shell.

Implement the buddy allocation *(kernel/buddy.c)*, which allocates and frees file structs so that xv6 can have more open file descriptors.

Design how to create address space. The central data structure is *pagetable_t* (it can be used for both kernel page table and process page table), which is really a pointer to a RISC-V root page-table page. The walk function will find the PTE for a virtual address, and map pages, which installs PTEs for new mappings.

Week 4

Implement lazy allocation, which will only allocate resource until it is actually needed. That is, *sbrk* function doesn't allocate physical memory, but just remembers which addresses are allocated. When the process first tries to use any given page of memory, the CPU generates a page fault, which the kernel handles by allocating physical memory, zeroing it, and mapping it.

Week 5

Design a simple file system. All files in xv6 are limited to 268 blocks, or 268*BSIZE bytes (BSIZE is 1024 in xv6). This limit comes from the fact that an xv6 inode contains 12 "direct" block numbers and one "singly-indirect" block number, which refers to a block that holds up to 256 more block numbers, for a total of 12+256=268 blocks. This design has limited function but at least stable. Moreover, keyboard driver, console driver and time interrupt are designed in within this week.

Week 6

Compile xv6. Since the CPU is 32 bit, the source code must be modified (mainly change pointer's increasing step length). Set the compile tool and simulating tool in *Makefile* to *riscv32-unknown-elf-gcc* and *qemu-system-riscv32*.

2.2.2 Method

To make an embedded operating system that can run on CPU, there are many works

to be done and these works can be divided into several stages. The first and most difficult one is to design an operating system. At the same time, it is also important to compile and test on the RISC-V simulator. In this project, the main structure of operating system used is xv6, which is from MIT 6.828. It provides the basic interfaces introduced by Ken Thompson and Dennis Ritchie's Unix operating system [19], as well as simulating Linux's kernel design. Linux and Unix provide a similar interface, offering a surprising degree of generality. For some other operating systems, like MacOS, this interface is also implemented well. Therefore, xv6 is a good start for students or other learners to learn come basic concepts of operating systems and even design a system. Moreover, another advantage is that designer can compile and run the operating system after finishing each component. To run the operating system on simulator, we need to make fs.img file firstly (this will be introduced in the next paragraph). Then, use command *make qemu* to run the system. All compilation details are shown in *xv6-riscv/Makefile*. The Figure 2.2.2.1 below shows the part of running the image file.

Figure 2.2.2.1: make qemu (run the OS on simulator)

The second stage is to transplant the image file to chip. To finish this step, one of the preconditions is designing a USB flash disk driver. The USB is to store the operating system image file, so that it can be read by bootloader and start running on CPU. Specifically, we need to use *make fs.img* to pack all the code into a image file and the details about how to finish this step is in *xv6-riscv/Makefile.* The following figure (Figure 2.2.2.1) is the part about how to make *fs.img*.

Figure 2.2.2.2: make fs.img

make fs.img will compile all files (*.c, *.h and *.S). In order to facilitate the compilation, in line 158, the compiler will automatically generate some dependent files (.d), which is used to described the dependencies among head files (.h) and object files (.o). The reason to use dependent file is to ensure the changes in head file will not affect object files. After finishing making *fs.img*, we can copy it to USB flash disk so that it can be read by bootloader.

2.2.2.1 Design the Kernel

As mentioned in the previous section, the basic structure of xv6 is from MIT 6.828 and it makes the work convenient. However, there is still a lot of components unfinished. The general process is to write each component and refer to Linux and Unix to do some modifications.

2.2.2.2 Organization

Since the whole organization is given by xv6, this part will only briefly introduce some concepts. A reliable operating system must arrange isolation which means one process's failure does not affect the others. Each process should have its private address space which cannot be accessed by others. Isolation provides all processes a more stable environment and reduce conflicts among different processes. However, it does not mean there is totally no interaction of messages. In xv6, the most frequently used method for interaction is *pipe*.

In order to achieve isolation, xv6 is divided into two spaces, user space and kernel space. In the kernel space, CPU is allowed to execute every instruction, such as disabling interrupts, reading and writing the register that holds the address of a page table. Kernel space is intended for configuring the computer and protect the physical resources from some user's malicious instructions. In the user space, CPU can only execute some instructions, such as adding numbers. If an application in user space attempt to execute kernel space instructions, it must switch CPU to kernel space using system call (*ecall which is a series of system call instructions that CPU enters the kernel at an entry point specified by the kernel*). After the CPU switches to kernel space, the kernel can validate the arguments of the system call, decide whether the application is allowed to perform the requested operation, and then deny it or execute it. If xv6 does not have two separate spaces or does not support system call, a malicious application could, for example, change the *process identification (pid).* All the kernel source code is in *xv6-riscv/kernel* and the interface for each module is defined in *defs.h (xv6-riscv/kernel/defs.h).* The following figure shows the simplified organization.

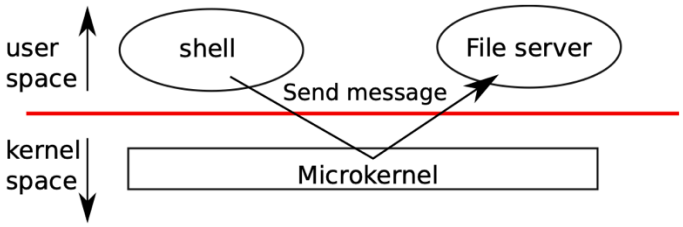

Figure 2.2.2.2.1: Simplified organization [17]

In Appendix C, Table 2 shows all system calls in xv6. For example, the *fork()* system call can be used to create child process. The *exit(status)* system call stops the current running process and release its sources, such as memory. The *wait()* system call blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent continues its execution after wait system call instruction. The *exec(filename,*argv)* system call replaces the current process image with a new process image. It loads the program into the current process space and runs it from the entry point.

In xv6, each process has its own private address to avoid other processes wrecking or spying its memory. xv6 uses page tables to give each process its own address space. Page table is a kind of data structure used by a virtual memory system in operating system to store the translation principles between virtual addresses and physical addresses. Each process has a separate page table shown in Figure 2.2.2.2.2 below. An address space starts with user's memory, including user text, data, stack and heap (used for memory allocation so that the process can expand as needed). At the top of the address space xv6 reserves a page for *trampoline* and a page mapping the process's *trapframe* to switch to the kernel space.

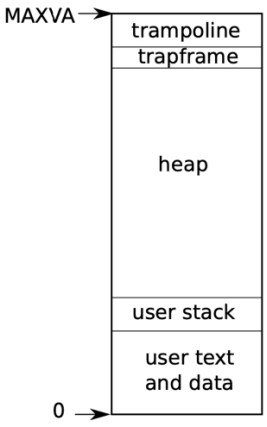

Figure 2.2.2.2.2: Virtual address space for a user process [18]

Usually, the kernel maintains many states of different processes and these states are gathered in *struct proc* (xv6-riscv/kernel/proc.h:85), such as kernel stack (*kstack*),

user stack (*ustack*) and page table (*pagetable*). Figure 2.2.2.2.3 above shows the struct proc.

```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  struct proc *parent;         // Parent process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // these are private to the process, so p->lock need not be held.
  uint64 ustack;    //Bottom of user stack for this process
  uint64 kstack;               // Bottom of kernel stack for this process
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // Page table
  struct trapframe *tf;        // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)


};
```

Figure 2.2.2.2.3: struct proc

Each process may have one or more threads and the CPU can only execute one thread at a time. In order to maximize the efficiency, the kernel can switch among different threads (suspend the currently running thread and resume another thread of the same process). As shown in Figure 2.1.3.3, each process has both user stack and kernel stack, which means all threads of this process can get access to one of these stacks. For example, if the thread is running in user space, it can only access the user stack. If the thread enters kernel space, it can only access the kernel stack.

2.2.2.3 Interfaces

xv6 maintains three main kinds of interfaces, process identification, I/O, file descriptor and pipes. Process identification (*pid*) is a unique ID number to indicate each process. Limited by the time, this operating system does not support multi-process, which means that in most of the cases, there is only one process with *pid* equals to 0. Even though xv6 cannot run several processes at the same time, it can still have more than one *pid*. For example, the parent process can create a child process using *fork()* system call. Then the data of parent process (including local variables and the content

of program counter) will be push into user stack. The *program counter* jumps to the start of child process to run child process. The *pid* of child process will be an integer (parent *pid* + 1). The Figure 2.2.2.3.1 below shows how to create child process and assign new *pid* (xv6-riscv/kernel/proc.c:246).

```
// Create a new process, copying the parent.
// Sets up child kernel stack to return as if from fork() system call.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;
  np->ustack=p->ustack;
  np->parent = p;
  *(np->tf) = *(p->tf);  // copy saved user registers.
  np->tf->a0 = 0;  // Cause fork to return 0 in the child.

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);
  safestrcpy(np->name, p->name, sizeof(p->name));
  pid = np->pid;
  np->state = RUNNABLE;
  release(&np->lock);
  return pid;
}
```

Figure 2.2.2.3.1: fork() system call

It will firstly allocate a process using *allocproc()* function and copy the user memory from parent process to child process using *uvmcopy(p->pagetable, np->pagetable, p->sz)* function. The child process has the same size, directory, *pid* and user stack as parent process. After that, the file descriptor of child process needs to be incremented by 1 based on the parent process. Finally, set the state of child process to *RUNNABLE* and release the child process's lock. The child is created and can run normally.

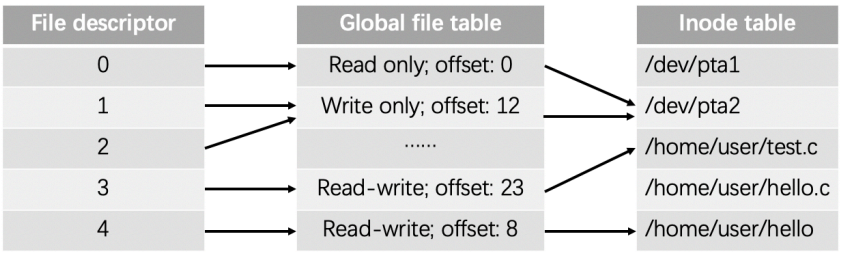

Figure 2.2.2.3.2: use of file descriptor in xv6

A file descriptor *(fd)* is a small integer representing a kernel-managed object that a process may read from or write to. Figure 2.2.2.3.2 above is the use of file descriptor. A process may obtain a file descriptor by opening a file, or by creating a pipe, or by duplicating an existing descriptor. In xv6, the number of file descriptor starts from 0 and usually choose the smallest integer first. For each process, we assign three file descriptors 0, 1 and 2. User can read from file descriptor 0, write to file descriptor 1 and write error message to file descriptor 2. The achievement of this part is in xv6-riscv/user/sh.c:150 and also shown in the following Figure 2.2.2.3.3.

```
int
main(void)
{
  static char buf[100];
  int fd;

  // Ensure that three file descriptors are open.
  while((fd = open("console", O_RDWR)) >= 0){
    if(fd >= 3){
      close(fd);
      break;
    }
  }
```

Figure 2.2.2.3.4: running the shell (including assign *fd*)

As shown in Figure 2.2.2.3.4, we use a while loop to assign file descriptor 0, 1 and 2. If fd exceeds 2, if will close the extra file descriptor and end this loop. The file descriptor is quite important in reading and writing and is a powerful abstraction, because it hides the details of what they are connected to. In Appendix D, Table 2 shows that *read* and *write* both have three parameters: it reads at most n bytes from the file descriptor, copies them into *buf* (memory address), and returns the number of bytes

read. Sometimes the returned value may be smaller than n because there is no more bytes in the file. It reads data from the start of current file and then advances that offset by the number of bytes read *(n)*. In xv6, there are several types of read function in kernel, such as *piperead(struct pipe*, uint64, int)* and *fileread(struct file*, uint64, int n)*. Figure 2.2.2.3.5 shows the *piperead* function (xv6-riscv/kernel/pipe.c:102).

```
int
piperead(struct pipe *pi, uint64 addr, int n)
{
  int i;
  struct proc *pr = myproc();
  char ch;

  acquire(&pi->lock);
  while(pi->nread == pi->nwrite && pi->writeopen){  //DOC: pipe-empty
    if(myproc()->killed){
      release(&pi->lock);
      return -1;
    }
    sleep(&pi->nread, &pi->lock); //DOC: piperead-sleep
  }
  for(i = 0; i < n; i++){  //DOC: piperead-copy
    if(pi->nread == pi->nwrite)
      break;
    ch = pi->data[pi->nread++ % PIPESIZE];
    if(copyout(pr->pagetable, addr + i, &ch, 1) == -1)
      break;
  }
  wakeup(&pi->nwrite);  //DOC: piperead-wakeup
  release(&pi->lock);
  return i;
}
```

Figure 2.2.2.3.5: piperead (read from pipe)

piperead is used to read message from pipe. Before reading message, it will firstly lock the pipe so that there is no other process writing into this pipe. As long as the pipe is not empty, it can copy out the data to destination address *(addr)*. Finally, the lock on this pipe will be released reading is finished.

Similarly, *write(fd, buf, n)* writes n bytes from *buf* to the file descriptor and returns the number of bytes written. *write(fd, buf, n)* writes data at the current file offset and then advances that offset by the number of bytes written: each write picks up where the previous one left off. user can also use *pipewrite(struct pipe*, uint64, int)* and *filewrite(struct file*, uint64, int n)* to write to pipe and file. These two functions are in *xv6-riscv/kernel/pipe.c:76* and *xv6-riscv/kernel/file.c:143* respectively. Figure 2.2.2.3.6 below shows *pipewrite* function (xv6-riscv/kernel/pipe.c:76). Figure 2.2.2.3.7 shows

the *filewrite* function (xv6-riscv/kernel/file.c:143).

```
int
pipewrite(struct pipe *pi, uint64 addr, int n)
{
  int i;
  char ch;
  struct proc *pr = myproc();

  acquire(&pi->lock);
  for(i = 0; i < n; i++){
    while(pi->nwrite == pi->nread + PIPESIZE){  //DOC: pipewrite-full
      if(pi->readopen == 0 || myproc()->killed){
        release(&pi->lock);
        return -1;
      }
      wakeup(&pi->nread);
      sleep(&pi->nwrite, &pi->lock);
    }
    if(copyin(pr->pagetable, &ch, addr + i, 1) == -1)
      break;
    pi->data[pi->nwrite++ % PIPESIZE] = ch;
  }
  wakeup(&pi->nread);
  release(&pi->lock);
  return n;
}
```

Figure 2.2.2.3.6: write to pipe

```
  int max = ((MAXOPBLOCKS-1-1-2) / 2) * BSIZE;
  int i = 0;
  while(i < n){
    int n1 = n - i;
    if(n1 > max)
      n1 = max;

    begin_op(f->ip->dev);
    ilock(f->ip);
    if ((r = writei(f->ip, 1, addr + i, f->off, n1)) > 0)
      f->off += r;
    iunlock(f->ip);
    end_op(f->ip->dev);

    if(r < 0)
      break;
    if(r != n1)
      panic("short filewrite");
    i += r;
```

Figure 2.2.2.3.7: write to file

pipewrite is used to write message to pipe. Before writing message, it will firstly lock the pipe so that there is no other process reading from this pipe. As long as the pipe is not empty, it can copy the data from destination address *(addr)* to *fd*. Finally, the lock on this pipe will be released and finish writing. As for *filewrite* function, it writes a few

blocks at a time to avoid exceeding the maximum log transaction size.

File descriptors and *fork()* interact to make I/O redirection easy to implement. The *fork()* function will assign the child process with the same memory address as parent process, which means that child process shares the same file descriptors with parents. This behavior was achieved by *exec()* system call that allows the shell to implement I/O redirection, and then executing the new program (xv6-riscv/user/sh.c:82). Figure 2.2.2.3.8 shows I/O redirection. It will close and reopen the specified file with a new access mode. Then run this command (the start of new process).

```
case REDIR:
  rcmd = (struct redircmd*)cmd;
  close(rcmd->fd);
  if(open(rcmd->file, rcmd->mode) < 0){
    fprintf(2, "open %s failed\n", rcmd->file);
    exit(-1);
  }
  runcmd(rcmd->cmd);
  break;
```

Figure 2.2.2.3.8: I/O redirection

In xv6, pipe is a kernel buffer used by two processes as a pair of file descriptors, one for reading and one for writing. One process can write to the pipe so that another can read from this pipe after finishing writing. Likewise, one process can read from the pipe after another process writes to this pipe. If there are two processes parent and child, two pipes are used for them to communicate with each other. Parent process sends by writing a byte to *prent_fd[1]* and the child process receives it by reading from *parent_fd[0]*. After receiving a byte from parent process, child process responds with its own byte by writing to c*hild_fd[1]*, which parent process then reads. Figure 2.2.2.3.9 is an example of pipe message transfer (xv6-riscv/user/pingpong.c).

```c
int main(int argc, char *argv[]) {
  int parent_fd[2], child_fd[2];
  pipe(parent_fd);
  pipe(child_fd);
  char buf[64];

  if (fork()) {
    // Parent
    write(parent_fd[1], "ping", strlen("ping"));
    read(child_fd[0], buf, 4);
    printf("%d: received %s\n", getpid(), buf);
  } else {
    // Child
    read(parent_fd[0], buf, 4);
    printf("%d: received %s\n", getpid(), buf);
    write(child_fd[1], "pong", strlen("pong"));
  }

  exit(0);
}
```

Figure 2.2.2.3.9: example of pipe (*pingpong*)

In the shell, the pipe can be expressed as |. For example,

echo hello world | wc

The *echo* process creates a pipe to connect the left end of the pipeline with the right end, so *"hello world"* will be sent to the memory address indicated by *wc*. Without pipe, the CPU need to access the memory twice:

echo hello world >/tmp/xyz

wc </tmp/xyz

It firstly writes *"hello world"* to */tmp/xyz*, then reads from */tmp/xyz*. Reading from/Writing to memory will take a lot of time. The time can be saved by using pipe to transfer messages. Figure 2.2.2.3.10 shows the achievement of pipe (xv6-riscv/user/sh.c:100). If the command does not contain any pipes, it will execute the left end of pipe *(runcmd(pcmd->left))* and write result to file descriptor 1. Likewise, CPU will then execute right pipe *(runcmd(pcmd->right))* and read the result from file descriptor 0. In line 105, *close(1)* is used to ensure that file descriptor 1 is free. And after duplicate the file descriptor 1, the original file descriptor 1 and 0 should be set free.

```
100    case PIPE:
101      pcmd = (struct pipecmd*)cmd;
102      if(pipe(p) < 0)
103        panic("pipe");
104      if(fork1() == 0){
105        close(1);
106        dup(p[1]);
107        close(p[0]);
108        close(p[1]);
109        runcmd(pcmd->left);
110      }
111      if(fork1() == 0){
112        close(0);
113        dup(p[0]);
114        close(p[0]);
115        close(p[1]);
116        runcmd(pcmd->right);
117      }
118      close(p[0]);
119      close(p[1]);
120      wait(0);
121      wait(0);
122      break;
```

Figure 2.2.2.3.10: pipe

Pipe has three advantages compared to memory or disk. First, pipes automatically clean themselves up using system call. Second, the capacity of pipe is not limited, while the memory or disk can only support limited space. Third, pipe is more time-efficient compared with memory *(eg. echo hello world | wc)*.

2.2.2.4 Page Table

xv6 runs on Sv39 RISC-V, which has 39-bit virtual addresses (Figure 2.2.2.4.1). The top 25 bits are not used because *Digilent 410-292 Nexys 4 DDR Artix-7 Development Board* only have 128MB memory. A page table is stored in physical memory as a three-level tree and each layer contains 512 page table entries. Each PTE contains a 44-bit physical page number (PPN) and some flags. The page table entry in first layer points to physical addresses for page-table pages in the second level of the tree. Likewise, the page table entry in second layer points to physical addresses for page-table pages in the third level of the tree. As shown in Figure 2.2.2.4.1, each physical address is consisted of physical page number and offset. Physical page number is given by the three layer of page table entry and the offset is directly given by the bottom 12 bits of virtual address.

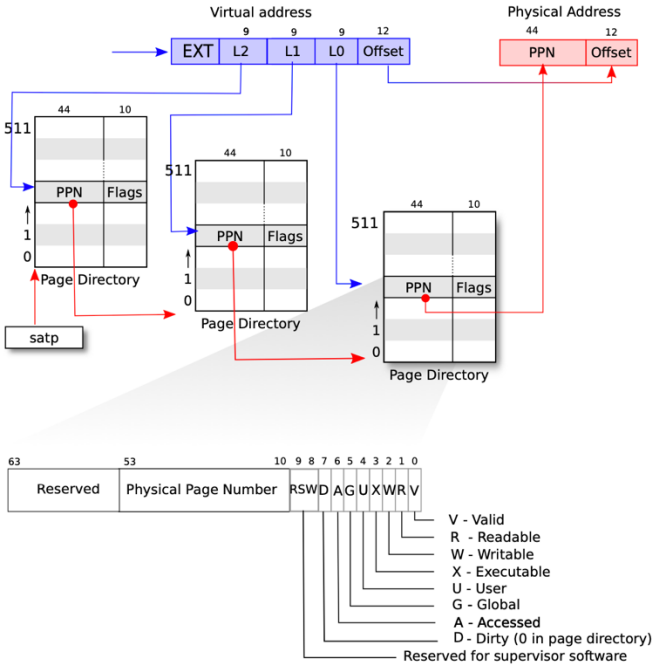

Figure 2.2.2.4.1: RISC-V page table hardware [18]

Each PTE contains several flag bits that inform the paging hardware how the state of each virtual address. In Figure 2.2.2.4.1, the bottom 8 bits contain 8 flags, including valid, readable, writable, executable, user, global, accessed and dirty. For example, if the user bit is 1, it means the user application can access. To use the page table, the kernel needs to write the root page-table page into *satp* register. The CPU can convert virtual address to physical address using the page table pointed by the *satp* register.

The kernel has its own page table. When a process enters the kernel, it will start to use kernel page table. When this process goes back to user space, it then switches back to user page table. Compared with address conversion in user space, kernel uses an identity mapping for most virtual addresses using directed mapping (Figure 2.2.2.4.2). However, not all virtual address is direct-mapped. For example, the trampoline page and kernel stack page are mapped at the top of virtual address, while it is mapped in physical memory (RAM) in physical address. In virtual address space, there is a guard page on the top of each kernel stack page. The guard page is used to avoid the overflow in kernel stack to affect other stacks or trampoline. The content of kernel memory layout is in *xv6-riscv/kernel/memlayout.h*, which is given by course 6.828.

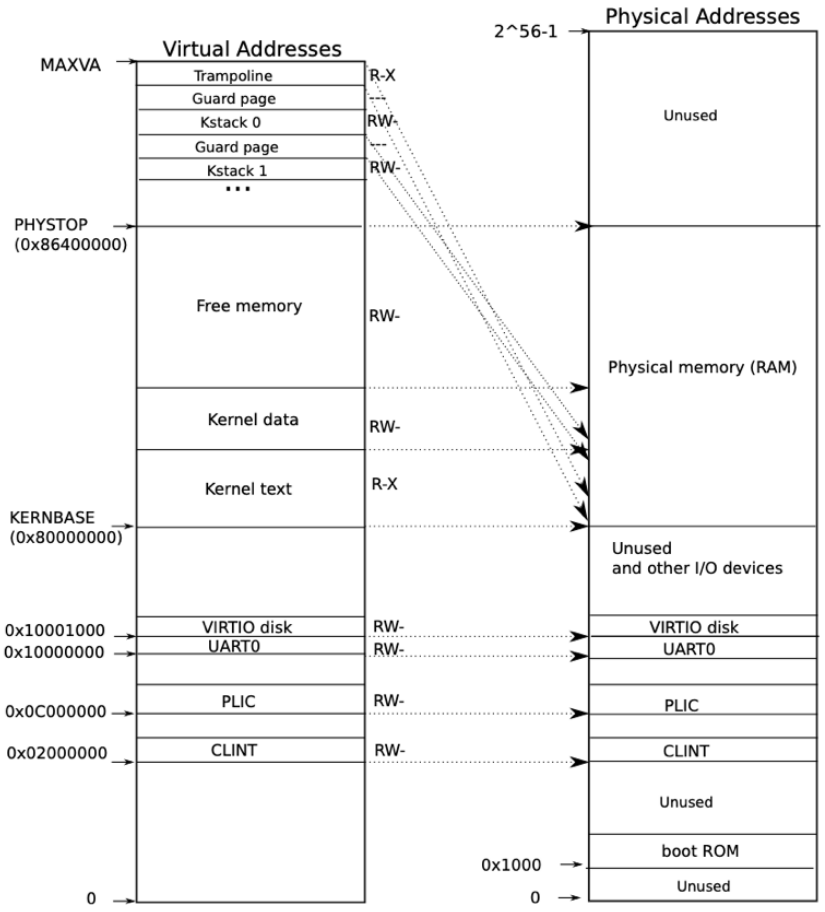

Figure 2.2.2.4.2: kernel address space (left); physical address space (right) [18]

The code about virtual address and physical address is in *xv6-riscv/kernel/vm.c*. The *kvminit* function creates a direct-map page table for the kernel and turn on paging. The *walk* function finds the address of the PTE in page table page table that corresponds to virtual address. This function will firstly check whether the virtual address exceeds the maximum address. If not, it goes through the three-layer physical page number to find the PTE. Similarly, the *walkaddr* function can convert virtual address to physical address. In line 111, *PTE2PA* function means page table entry to physical address. As for the conversion from kernel address to physical, it is supported by *kvmpa* function. The details of these and other functions are shown in xv6-riscv/kernel/vm.c.

All of above need allocator to allocate or free physical memory. In xv6, the kernel allocates a page-unit size (4096 bytes) at a time. In xv6, there are two allocation mechanisms designed for physical memory allocation: lazy allocation and buddy

allocation.

Lazy allocation simply means not allocating a resource until it is actually needed. This is common with singleton objects, but strictly speaking, any time a resource is allocated as late as possible. By delaying allocation of a resource until you actually need it, you can decrease startup time, and even eliminate the allocation entirely if you never actually use the object. In contrast, you could pre-allocate a resource you expect to need later, which can make later execution more efficient at the expense of startup time, and also avoids the possibility of the allocation failing later in program execution. The code in *xv6-riscv/kernel/trap.c* to respond to a page fault from user space by mapping a newly allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing. Figure 2.2.2.4.3 shows how to check page fault *(xv6-riscv/kernel/trap.c)*.

```
70  }else if(r_scause()==13||r_scause()==15){
71    //int ret=0;
72        /*if((ret=handle_page_fault(p->pagetable,r_stval()))!=0){
73            p->killed=1;
74            printf("fail!!\n");*/
75
76        struct proc *p=myproc();
77        pagetable_t t=p->pagetable;
78        uint64 faddr=(uint64)r_stval();
79        uint64 base=PGROUNDDOWN(faddr);
80
81        if(faddr>=p->sz){
82            printf("out of page boundary\n");
83            p->killed=1;
84            goto end;
85        }
86
87        if(faddr<p->ustack){
88            p->killed=1;
89            goto end;
90        }
91
92    char *mem=kalloc();
93        if(mem==0){
94            p->killed=1;
95            goto end;
96        }
97        memset(mem,0,PGSIZE);
98        if(mappages(t, base, PGSIZE, (uint64)mem, PTE_W|PTE_X|PTE_R|PTE_U) != 0){
99      kfree(mem);
100           p->killed=1;
101           goto end;
102     //uvmdealloc(pagetable, a, oldsz);
103     //return 0;
104   }
105 }
```

Figure 2.2.2.4.3: check page fault

If the fault address does not exceed page boundary *(line 81)*, or access the user stack *(line 87)*, the kernel will assign a page-size space to the address (*kalloc* and

memset).

The buddy memory allocation is a memory allocation algorithm that divides memory into partitions to try to satisfy a memory request as suitably as possible. This system makes use of splitting memory into halves to try to give a best fit. In xv6, we set the minimum block size to 16 bytes and maximum size cannot exceed 1 MB. Large block is split to small one which is half of large block's size. Figure 2.2.2.4.4 shows the setting of buddy allocator. The buddy list *(bd_list)* is used to store different sizes of blocks. The allocator has *sz_info* (xv6-riscv/kernel/buddy.c:28) for each size k. Each *sz_info* has a free list, an array *alloc* to keep track which blocks have been allocated, and a split array to keep track which blocks have been split. The arrays are of type char (which is 1 byte), but the allocator uses 1 bit per block (thus, one char records the info of 8 blocks).

```
static int nsizes;//=5;  // for debugging

#define LEAF_SIZE     16 // The smallest allocation size (in bytes)
#define MAXSIZE       (nsizes-1) // Largest index in bd_sizes array
#define BLK_SIZE(k)   ((1L << (k)) * LEAF_SIZE) // Size in bytes for size k
#define HEAP_SIZE     BLK_SIZE(MAXSIZE)
#define NBLK(k)       (1 << (MAXSIZE-k))  // Number of block at size k
#define ROUNDUP(n,sz) (((((n)-1)/(sz))+1)*(sz))  // Round up to the next multiple
    of sz
#define in_range(a,b,x) (((x)>=(a))&&((x)<(b)))

typedef struct list Bd_list;
```

Figure 2.2.2.4.4: setting of buddy allocator

Line 126 is *bd_malloc* function (buddy memory allocation). It will firstly lock the free block list and find a free block larger or equal to *nbytes*, starting with smallest k possible. After that, we set this chosen block indivisible and allocated. Finally, it releases the free block list and return the block base address.

2.2.2.5 Traps

A trap may occur while executing in user space if the user program makes a system call (*ecall* instruction), or if a device interrupt. Limited by the time, this operating system does not support exception handler, which means if an exception occurs, the

system will stop directly. There are only two device drivers, keyboard driver and console driver. Besides, xv6 also provides us a timer interrupt mechanism. One example usage of timer interrupt is sleep command (xv6-riscv/user/sleep.c). The code is shown in Figure 2.2.2.5.1 below. *sleep* function will use the *sleep* system call to pause all process for a short time.

```
int main(int argc, char *argv[]) {
  if (argc != 2)
    write(2, "Error message", strlen("Error message"));

  int x = atoi(argv[1]);
  sleep(x);
  exit(0);
}
```

Figure 2.2.2.5.1: sleep function

The console driver accepts characters typed by user via the UART serial-port hardware attached to the RISC-V. The driver accumulates a line of input at a time until there is a change line character. The shell can use the read system call to fetch lines of input from the console. The *consoleinit* function (xv6-riscv/kernel/console.c:189) initialize the UART hardware and connect read and write system calls to *consoleread* and *consolewrite*.

consoleread function will firstly lock the console buffer that is used to temporarily store input. Then it waits until interrupt handler has put some input into console buffer. If it receives the EOF character (end of file) or '\n' character, it will end the stop reading. Similarly, *consolewrite* function will also lock the console buffer to ensure that there is no other process reading from or writing to it. Then it copies character from src and put on console (consputc). Figure 2.2.2.5.2 and Figure 2.2.2.5.3 are codes about *consoleread* and *consolewrite*.

```
int
consoleread(int user_dst, uint64 dst, int n)
{
  uint target;
  int c;
  char cbuf;
  target = n;

  acquire(&cons.lock);
  while(n > 0){
    while(cons.r == cons.w){
      if(myproc()->killed){
        release(&cons.lock);
        return -1;
      }
      sleep(&cons.r, &cons.lock);
    }

    c = cons.buf[cons.r++ % INPUT_BUF];
    if(c == C('D')){  // end-of-file
      if(n < target){
        cons.r--;
      }
      break;
    }
    cbuf = c;
    if(either_copyout(user_dst, dst, &cbuf, 1) == -1)
      break;
    dst++;
    --n;
    if(c == '\n'){
      break;
    }
  }
  release(&cons.lock);
  return target - n;
}
```

Figure 2.2.2.5.2: consoleread function

```
int
consolewrite(int user_src, uint64 src, int n)
{
  int i;

  acquire(&cons.lock);
  for(i = 0; i < n; i++){
    char c;
    if(either_copyin(&c, user_src, src+i, 1) == -1)
      break;
    consputc(c);
  }
  release(&cons.lock);

  return n;
}
```

Figure 2.2.2.5.3: consolewrite function

2.2.2.6 File system

The xv6 file system implementation is organized in seven layers, shown in Table 2.2.2.6.1 below.

Table 2.2.2.6.1: Layers of the xv6 file system

File descriptor
Pathname
Directory
Inode
Logging (not completed)
Buffer cache
Disk

The disk layer reads from or writes to the disk and the minimum reading/writing unit is a page (4096 bytes). The buffer cache layer is used to temporarily store blocks, which makes reading/writing faster. It also maintains a lock so that only one kernel process at a time can access. The structure of buffer cache is list (xv6-riscv/kernel/bio.c:26). Moreover, *bread* function is used to read indicated block from buffer cache and *bwrite* function is used to write a block to buffer cache. In *bread* function, it will firstly look through buffer cache looking for specified block (with *blockno*). If this block in cache, return a pointer pointing to this block (line 92). Otherwise, it reads from the disk (line 94). Similarly, the bwrite function will check whether buffer cache is locked by other process. If no other process accesses the buffer cache, it writes the buffer cache to the disk. Figure 2.2.2.6.1 shows the *bread* and *bwrite* function.

```
86  // Return a locked buf with the contents of the indicated block.
87  struct buf*
88  bread(uint dev, uint blockno)
89  {
90    struct buf *b;
91
92    b = bget(dev, blockno);
93    if(!b->valid) {
94      virtio_disk_rw(b->dev, b, 0);
95      b->valid = 1;
96    }
97    return b;
98  }
99
100 // Write b's contents to disk.  Must be locked.
101 void
102 bwrite(struct buf *b)
103 {
104   if(!holdingsleep(&b->lock))
105     panic("bwrite");
106   virtio_disk_rw(b->dev, b, 1);
107 }
```

Figure 2.2.2.6.1: *bread* and *bwrite*

The logging layer is used to recover when crash occurs (redo and undo). However, we have no time to achieve this part. The inode layer provides individual files and each file maintains a private inode. In xv6, we have disk inode and memory inode. The disk inode refers to the file structure in disk (xv6-riscv/kernel/fs.h:32), while memory inode refers to the file structure in memory and it is also the copy from disk (xv6-riscv/kernel/file.h:17). When the process is running, it will read inode from this disk to memory. Figure 2.2.2.6.2 shows the structure of disk inode. Each inode has 12 direct address (NDIRECT=12) and 1 indirect address which points to 256 addresses, so the size of each file can be 268 blocks (each block size is 1024 bytes).

```
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

Figure 2.2.2.6.2: inode on disk

The directory layer is given by 6.828 course. The pathname layer provides hierarchical path names like */home/gan/xv6-riscv/kernel/fs.c*. The file descriptor layer abstracts many xv6 resources (e.g., pipes, devices, files) using the file system interface, simplifying the lives of application programmers.

2.3 System on chip part

2.3.1 Procedure

Week 1

1) Weekly Periodic Progress
    a. Discuss what the expected demo of the project with group members
    b. Discuss what the project needed to do (general construction of separate part)
    c. Search reference of the system on chip and CPU core
    d. Test the function of Digilent Nexys A7 board
    e. Create the github page and determine the timetable of different parts
2) Weekly Progress Explanation

Since this project included many engineering and practical work, aim and expectation should be first discussed and determined. Based on the products designed by students from Tsinghua University, an system on chip was expected to be constructed successfully if some of components could be directly utilized without designing and testing [11]. Therefore, in this project, it was proper to meet the requirements that the aim of the hardware is to basically support the running of operating system and high-level design and protocol were unecessary.

In addition, preparation of Central Processing Unit (CPU) and peripheral of System on Chip (SoC) were also involved in the first week to help the hardware design. Most importantly, this project asked students to use Github to achieve the progress tracking and code sharing. Moreover, Github repository could also back up the progress of the project so that the risk of losing work would be low.

Week 2

1) Weekly Periodic Progress
    a. Discuss which instruction set the operating system (xv6 or Linux kernel) needed (RV32IM)
    b. Construct the simple Arithmetic Logic Unit Structure on Vivado 2019.1
    c. Update ALU structure and support RV32M extension without verification (commits on Feb 8, 2020)
    d. Update ALU Control Unit without verification (commits on Feb 8, 2020)
    e. Construct register file (32 registers following RISC-V specification)
2) Weekly Progress Explanation

Before designing and constructing system on chip and Central Processing Unit, instruction set architecture would be the most significant abstraction to be fixed. RV32I and extension RV32M were necessary for xv6 or Linux Kernel. Among all the parts of the core, arithmetic logic unit (ALU) functioned as calculating the result; thus, it would be firstly designed. In addition, 32 register files were also designed because they were simple to construct when using static RAM in the FPGA board.

Week 3

1) Weekly Periodic Progress
    a. Test the ALU with new extension support and fix the bugs (commits on Feb 9, 2020)
    b. Construct Control Unit, and test ALU control and register file (commits on Feb 10, 2020)
    c. Confirm the function of SDRAM to DDR Component provided by Digilent on board (commits on Feb 12, 2020)
        a) Advantages: It could be directly used to save time
        b) Disadvantages: minimum writing time was 260ns and reading time was 210ns, which caused huge memory operation latency. In addition, it didn't have the output for a normal bus that guarantees the communication

between memory and CPU core

c) Problem existing: After reading or writing many times to memory, it would obviously lose data (maybe caused by refresh rate of DDR2 SDRAM in this FPGA)

d. Construct register between each stage used in pipeline processor

e. Test Control Unit and fix the bugs (commits on Feb 15, 2020)

2) Weekly progress explanation

Testing the components occupied half of the time in the hardware design, and in this project, all the unit tests were automati, but they didn't contain all the circumstances. Additionally, control part included *ControlUnit* and *ALUControl* because separation between control logic and ALU control logic could simplify the design complexity. Furthermore, SDRAM to DDR Component took much time to test whether it could be correctly used, and the result were negative.

Week 4

1) Weekly Periodic Progress

a. Discuss how the operating system communicate with the CPU (system call)

b. Search information to solve the problem of DDR2 SDRAM, which cause the whole project not to meet the expectation progress.

c. Replace on chip DDR2 RAM with LUT RAM and Block RAM

a) Can not support the compile file of operating system. Therefore, memory support is necessary

2) Weekly progress explanation

There were two main problems in this week. One was memory management unit while the other was the meaning of system call. The first one was due to the strange results when testing the SDRAM to DDR component and its low performance. The second one was relative to the student who designed the operating system because system call was responsible for all the I/O and other operations. It also included user mode or kernel mode in the interaction between operating system and the hardware. Hence, most of the work in this week was to search information, learn how to solve the

problems, and discuss about the problems.

Week 5

1) Weekly periodic progress
    a. Integrate all the components to be a pipeline processor with 5 stages (commits on Feb 27, 2020)
    b. Construct Hazard Unit to solve control dependencies and data dependencies by data forwarding and pipeline stall (commits on Feb 28, 2020)
    c. Construct static branch prediction (commits on Feb 28, 2020)
    d. Simple test about the Central Processing Unit (commits on Feb 29, 2020)
2) Weekly progress explanation

In this week, Central Processing Unit was integrated by different components designed before and there were some pipeline issues in 5 stage pipeline processor. Data dependencies and control dependencies would affect the correctness of the processor and Hazard Unit was responsible for dealing with these problems. More details about the issues would be introduced in methods section. In addition, in this project, due to the limited time, static branch prediction “not taken” were used in the branch instruction.

Week 6

1) Weekly periodic progress
    a. Construct cache system (commits on Mar 2, 2020)
    b. Construct communication between cache system and memory by AXI4 full bus (commits on Mar 3, 2020)
    c. Construct communication between cache and CPU core (commits on Mar 3, 2020)
    d. Try to integrate operating system into the SoC
        a) Problem: No GPIO, No USB, Unknown bugs in cache system
2) Weekly progress explanation

In the final week, memory management unit and cache system were constructed to enable the system on chip work at the lowest level. These two parts were difficult, and

it asked students to know much about the memory system and bus. Moreover, integrating boot, operating system and hardware were unsuccessful because of the loss of components and unknown bugs in hardware design especially in cache part.

2.3.2 Method

2.3.2.1 Hardware Description Language and FPGA board

Hardware Description Language was the high-level abstraction of the digital circuit and it was largely applied in Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC). This project used FPGA as the hardware platform and Verilog as the hardware description language. In addition, some IP core provided by Xilinx was written by VHDL, which is stricter than Verilog but more complex than Verilog.

FPGA board in this project was Digilent Nexys A7 and the reason why this board would be used in this project was its 128MB DDR2 SDRAM. Because the expectation of the project was to load the operating system into the system on chip, only static RAM on FPGA board could not support large program.

2.3.2.2 Central Processing Unit

2.3.2.2.1 RISC-V Instruction Set Architecture (ISA)

RISC-V instruction was specified by the RISC-V Foundation and there was an official document to talk about all the instructions [9]. The document contained different instruction set architecture with different bit length per unit. For example, 64-bit and 32-bit instruction set would not be the same. In this project, due to the design complexity of 64-bit processor and the limitation of programmable logic slices, processor was chosen to be 32-bit. In addition, some instruction sets were discarded in this project to save time. RV32I Base Integer Instruction Set and ‘M’ Standard Extension for Integer were determined to be achieved in this project. RV32I was one of the basic instruction sets in the specification of RISC-V Foundation, which included the basic instructions like *JAL, ADD* and so on. RV32M Extension was one of the

extension instruction sets and it included all the multiplication, division, remainder instructions.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Figure 2.3.2.2.1.1: RV32I Instruction Set List [9]

RV32M Standard Extension						
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Figure 2.3.2.2.1.2: RV32M Instruction Set List [9]

Figure 2.3.2.2.1.1 and 2.3.2.2.1.2 showed the type of all the instruction in RV32I and RV32M. The detail of instructions could be read in the specification provided by RISC-V foundation and in this report, only the types of instructions would be introduced. They were R-type instruction, I-type instruction, S-type instruction, B-type instruction, U-type instruction, and J-type instruction.

R-type instruction meant that it needed two source registers and one destination register. Based on different function number, it would perform as addition, subtraction, and so on. All the R-type instruction would calculate the data from source registers and put them into the destination register.

I-type instruction meant that it needed one source registers and one destination register. The difference between I-type instruction and R-type instruction was that the second source register would be replaced by a 32 bits immediate value coming from sign-extending the immediate value in the assemble code.

S-type instruction meant that it needed two source registers and no destination register. This type of instruction would enable the program to store the data into the memory. Based on the figure 2, the immediate value consisted of bit 31 to 25 and bit 11 to 7, and then this value would be sign-extended to be a 32 bits value. Next, this immediate value would be added into the data in the first source register *rs1* so that the result would be the destination memory space. The data that would be stored was in the second source register *rs2* and function number *funct3* determined which length of the data should be stored in the memory.

B-type instruction was branch instruction. It had two source registers and one immediate value. It would compare the data in both two source registers and generate

the result based on the result. If the branch hit, the program counter would get the right instruction address. The branch target address was added by sign-extended immediate value and the branch instruction address. Moreover, function code *funt3* would determine whether the comparison was signed or unsigned.

U-type instruction was special in RV32I. It only had two instructions that are LUI and AUIPC. The former was called load upper immediate and it would put 20 bits immediate value on the first 20 bits of destination registers with zeros on the last 12 bits. The latter was called add upper immediate to pc, which meant that the immediate value would be first extended with zeros on the last 12 bits and then be added to the pc, and finally store in the destination register.

J-type instruction only represented one instruction called JAL. It would add jump instruction address with sign-extended immediate value to generate the address of next instruction, and it would store the current next instruction address into the destination register.

Categories of instructions help reduce the complexity when designing the arithmetic logic unit because when designing and constructing the core, instructions themselves could be separated from outer layer. That is, designers could first regard the same type instructions as entirety and then determine each instruction when designing the arithmetic logic unit and control unit.

2.3.2.2.2 Microarchitecture

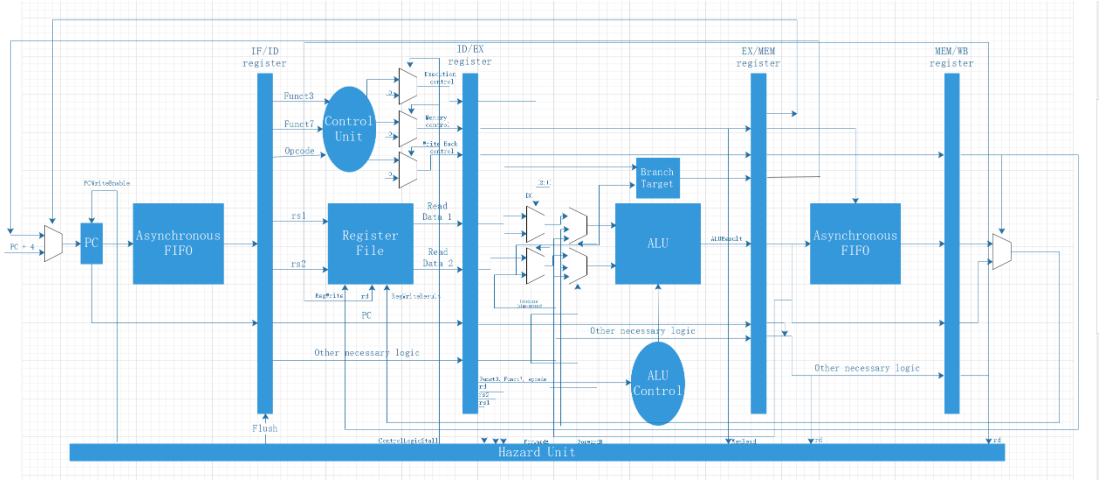

Figure 2.3.2.2.2.1: CPU structure diagram in the project

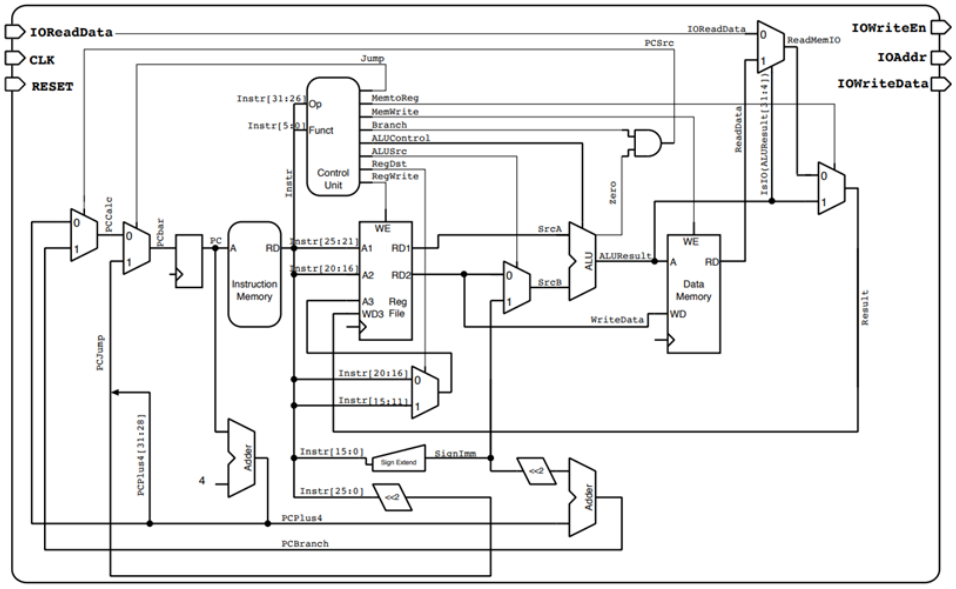

Fig 2. Block diagram of the MIPS processor that we will implement in this exercise. This diagram is almost identical to Fig 7.14 on page 387 of your textbook

Figure 2.3.2.2.2.2: Multi-cycle MIPS processor [17]

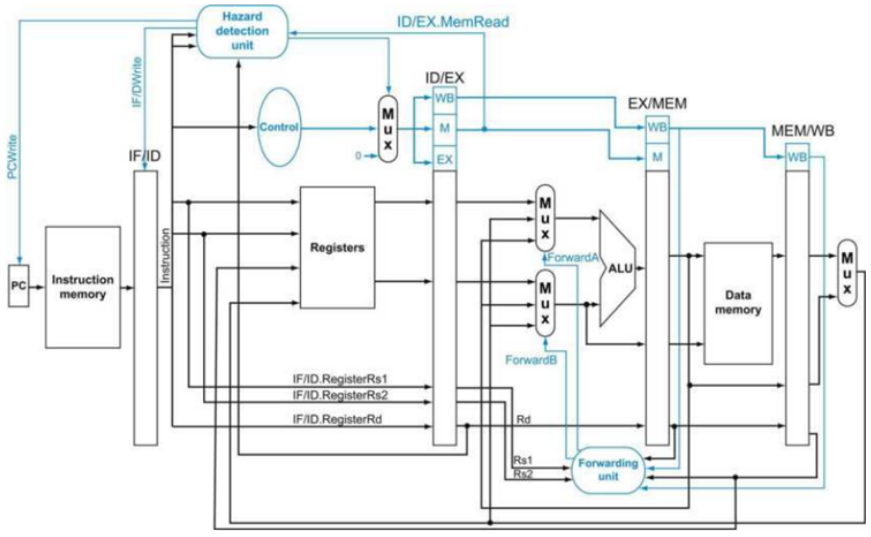

FIGURE 4.58 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit.
Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

Figure 2.3.2.2.2.3: Simple RV64 RISC-V processor [1]

Figure 2.3.2.2.2.1, 2.3.2.2.2.2, and 2.3.2.2.2.3 were three structures of Central Processing Unit and the first one was what designed in this project while the other

two were referred when designing the first one.

2.3.2.2.2.1 Pipeline

In this project, the RISC-V microprocessor utilized five-stage pipeline structure. Compared with the multi-cycle and single-cycle microprocessor, pipeline microprocessor had higher Instruction Per Cycle (IPC) so that it could have higher performance the other two structure. Furthermore, five stages were Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Operation (MEM) and Write Back (WB). IF stage meant that during the current cycle, CPU would get the instruction data from instruction cache. ID stage meant that during the current cycle, CPU would decode the instruction to get the control logic and data in the source registers from register files. It also sign-extended the immediate value to meet the requirements of the instruction. Execution stage was that CPU calculated the data and returned the results. Memory operation stage included two parts that were branch/jump logic and memory. In this stage, if the instruction was branch instruction or jump instruction, and the branch hit, program counter (PC) would get the branch target address. At the same time, if the instruction was store or load instruction, it would get the information from data cache. Write Back stage was that if the instruction needed to write data into the register, it would happen in this stage.

Five stages worked at the same cycle for different instructions and each instruction would be divided into five parts to be run one by one. In order to enable pipeline to flow quickly, there were four pipeline registers between each stage to store the data and results from previous stage, which called IF/ID Register, ID/EX Register, EX/MEM Register, and MEM/WB Register. The performance of pipeline processor could be showed by IPC. When the processor entered the full pipeline state, IPC could be 1 without any pipeline issues mentioned below. Although IPC in pipeline structure was nearly the same as the single-cycle processor, pipeline structure had much higher cycle frequency because the cycle frequency of single-cycle processor was determined by the longest instruction.

2.3.2.2.2.2 Pipeline issues

Even if the pipeline structure had such perfect performance, it had some issues when being constructed. Data dependencies and control dependencies are two issues in this project.

Data dependencies contains three dependencies that were Flow dependence (Read-after-Write), Anti dependence (Write-after-Read), and output dependence (Write-after-Write) [2]. The first data dependencies would cause incorrect results in the pipeline while the other two data dependencies would not cause problems in in-order processor, which would be discussed in in-order section and improvement section in discussion chapter. The reason was that WAR and WAW data dependencies was normal when the processor executed the instruction one by one in von Neumann model, but RAW dependency might induce next instruction to get the wrong data because the former one hadn't written its result into the destination register. In this project, data forwarding was used to deal with these issues. Data forwarding meant to transfer the data in MEM stage or WB stage to ID stage if the instruction in ID stage needed to read the register that would be also be written back by the instruction in MEM stage or WB stage. However, there was still one condition where memory operation spent more than one cycle. It needed to stall the pipeline because the instruction was incomplete in the current stage.

Control dependencies was about the address in program counter when going to the next cycle. If instruction in IF stage was branch instruction or jump instruction, the next instruction fetched would depend on the former one. If the instruction was branch instruction, branch prediction would help solve this dependency. In this project, in order to reduce the complexity, 'not taken' branch prediction was applied, which meant that the processor would predict that the branch would miss and firstly used PC + 4 as the address of next instruction to be fetched. Nevertheless, if the result was that branch hit, processor will flush the IF/ID register and ID/EX register, and keep ID/EX register to be zero for extra one cycle because these two registers are running wrong instruction so that pipeline in these two stages should be flushed and wait the correct instruction.

2.3.2.2.2.3 In-order and scalar structure

In-order structure meant that the processor would run the instruction following the order of the instruction. The advantages of this structure were that it was easy to construct; on the contrary, it has low performance if some instructions such as load instruction spent two or more cycles being complete. Originally, this project was expected to apply out-of-order execution, which meant that instruction would be run according to its data dependencies instead of order. However, due to the time limit, in-order processor replaced the out-of-order processor and the detail would be discussed in the future. Moreover, scalar processor was that the processor could only run one instruction at a time while superscalar processor could run multiple instructions at a time. In this project, due to the complexity and issues in superscalar processor, scalar processor was chosen to replace the latter.

2.3.2.2.3 Design and Construction

2.3.2.2.3.1 Arithmetic Logic Unit (ALU) and ALU Control

In this project, ALU component would get three inputs, which were two sources, and one ALU control logic. According to the RISC-V instruction set architecture, there were six types of instructions and they would be dealt with by different logics and finally outputted the result by multiplexer. The most difficult parts in ALU were unsigned operation and RV32M instruction set. The former was due to the boundary problem of the unsigned number and the latter was because the RV32M instruction included multiplication, division, and remainder operations that toke much resource and complexity on the FPGA board. Multiplication, Division, and remainder toke the Register transfer level (RTL) synthesis circuit as the solution because this could largely lower the complexity students met when designing their own digital logic circuit for these three operations. However, RTL synthesis logic were not optimized, and it would spend 200ns or more to calculate a 32 bits multiplication; thus, this problem would affect the cycle frequency of the processor.

In this project, the control logic of ALU was separated from the Control Unit because integration might increase the complication of the control Unit and make the

designer difficult to debug. In addition, it can lower the codes and registers because ID/EX register didn't need to store the ALU control logic, but it would increase the latency in the execution stage. Furthermore, to simplify the code, ALU control logic is divided into different parts to represent which instruction it was. The ALU Control component would generate a 11 bits result, which consists of *addOp, branch, shiftOp, logicalOp, mulOp, sltOp, jalr, logicalOrArith, Funct3*. These bits were determined by *opocode, funct3, funct7* that decoded in the ID stage. For example, *ADD* instruction and *SUB* instruction were only different on *funct3*, which meant that one is 000 while the other is 010. Thus, *addOp* would be 1 if the instruction was *ADD* instruction and be 0 if the instruction was *SUB* instruction. Only the bit addOp had two meanings about the value because addition and subtraction are opposite and other bits only represent whether it was or not. Figure 2.3.2.2.3.1 and Figure 2.3.2.2.3.2 were the elaborated design schematic of ALU and ALU Control component, and the logic circuits in the diagram were generated from Verilog code by Vivado.

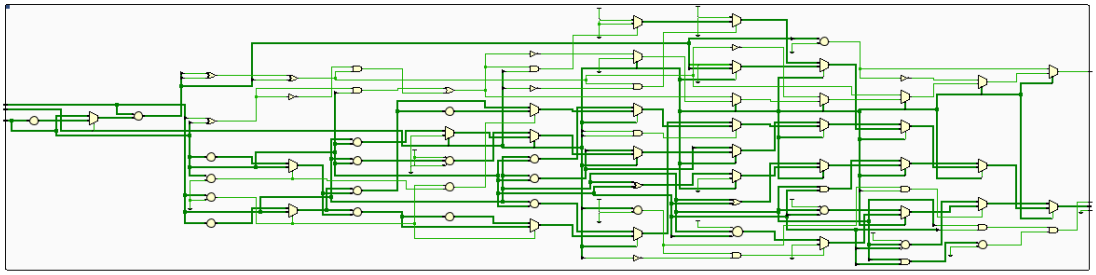

Figure 2.3.2.2.3.1: ALU Elaborated Design Schematic

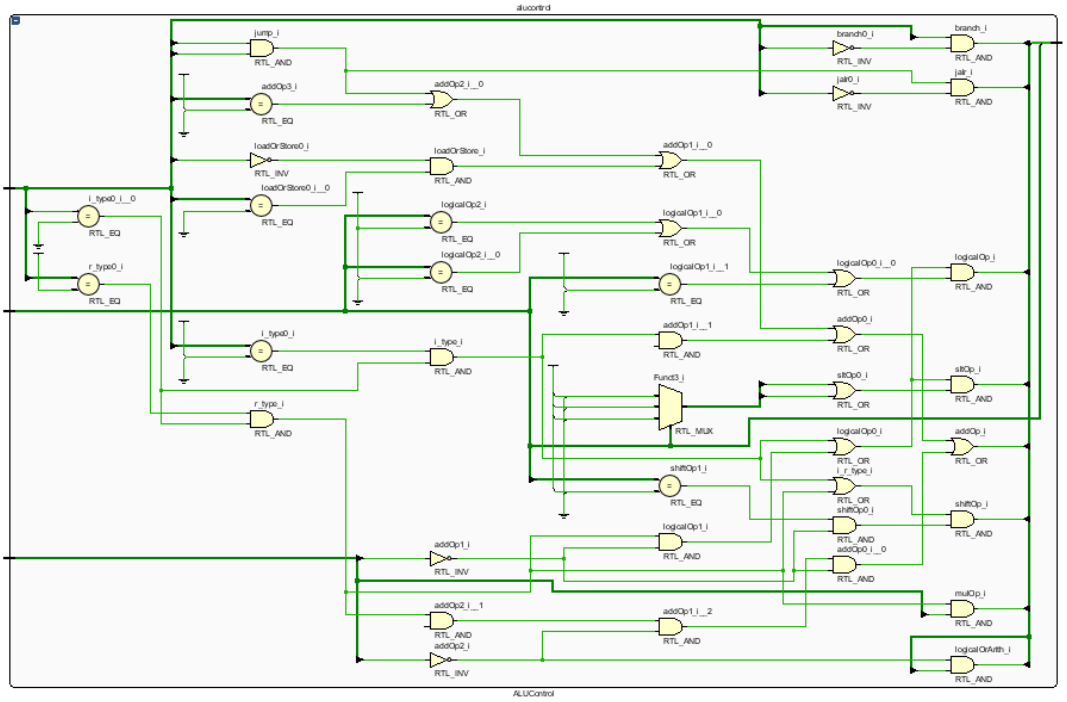

Figure 2.3.2.2.3.2: ALUControl Elaborated Design Schematic

2.3.2.2.3.2 Register file

Register File was the place to contain 32 registers. To keep the register read and write in one cycle, static RAM in the FPGA board was used. Therefore, the processor could make sure that reading registers and writing to registers would be quick enough. Figure 2.3.2.2.3.3 was the elaborated design schematic of Register File component, and the logic circuits in the diagram were generated from Verilog code by Vivado.

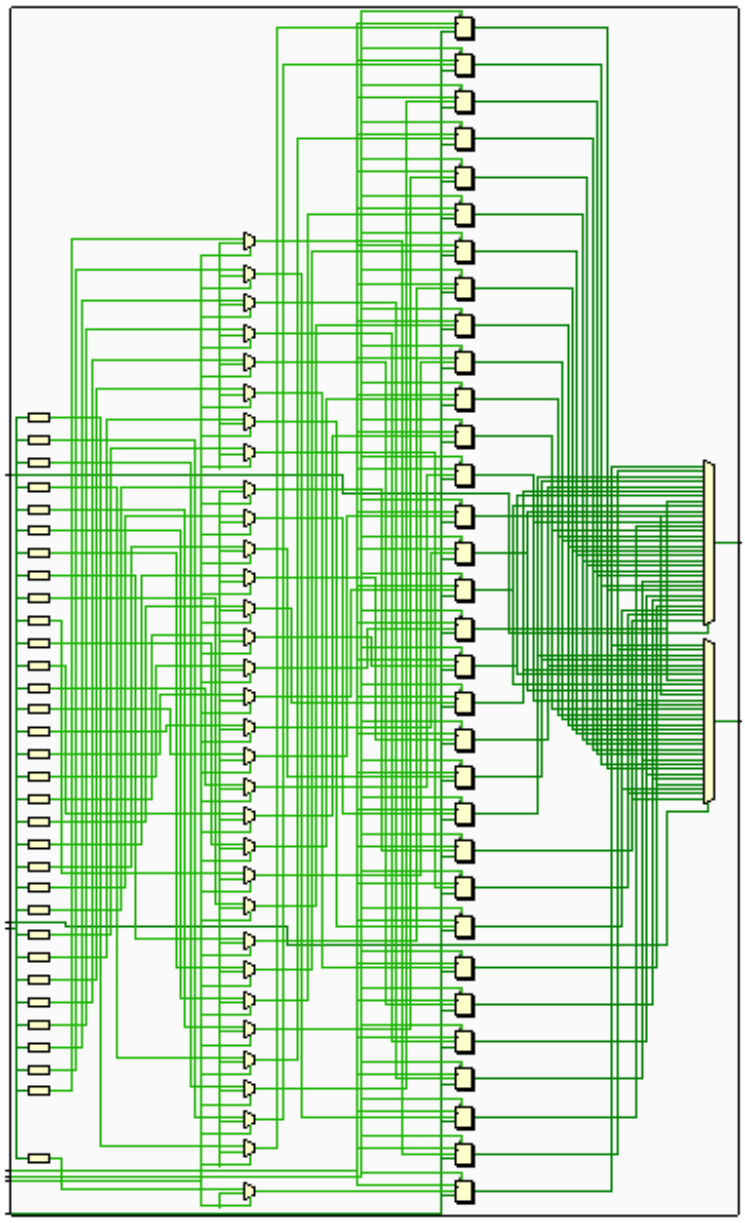

Figure 2.3.2.2.3.3: RegisterFile Elaborated Design Schematic

2.3.2.2.3.3 Stage Registers

There were four stage registers called IF/ID register, ID/EX Register, EX/MEM register, and MEM/WB register. They were to store the data of each stage to keep the pipeline flow. They were constructed by static RAM in FPGA because they wouldn't occupy many resources so that the usage of static RAM could keep the registers to finish store during a short time (less than one cycle).

2.3.2.2.3.4 Control Unit

In this project, ControlUnit was responsible for generating the control logic except for the ALU control logic. It would compose all the logic into three different logic clusters that are *EX_control*, *MEM_control*, *WB_control*. The first one was all the control logic useful in EX stage, the second one was that useful in MEM stage, and the final one was that useful in WB stage. Figure 2.3.2.2.3.4 was the elaborated design schematic of Control Unit component, and the logic circuits in the diagram were generated from Verilog code by Vivado.

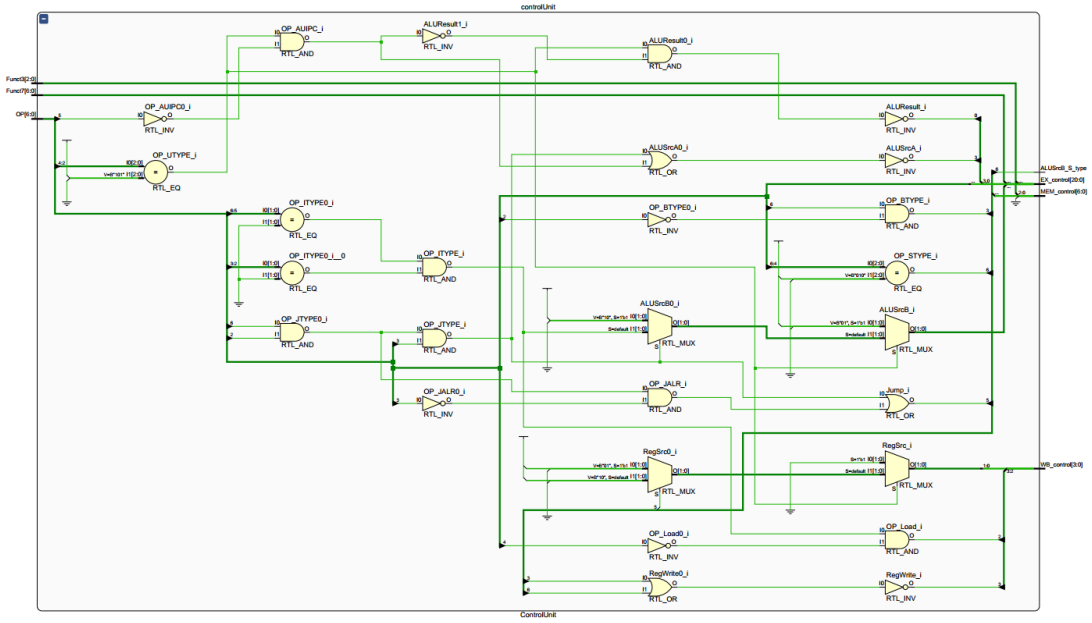

Figure 2.3.2.2.3.4: ControlUnit Elaborated Design Schematic

2.3.2.2.3.5 Hazard Unit

Hazard Unit was responsible for dealing with the pipeline issues such as data dependencies. It would get the destination register of instruction in MEM stage and WB stage, and then compared them with two sources register in the ID stage. If source registers were the same as the destination register, which was flow dependence (RAW), the hazard Unit would enable the result in MEM stage or WB stage to be the result of the following source registers. Moreover, if instruction in Execution stage was load instruction, it was necessary to stall the pipeline if the instruction in ID stage needed to read the same register that is the destination register of the instruction in Execution

stage. To stall the pipeline, hazard unit will flush the ID/EX register and fill it with zero. It would also keep all the logic in Fetch stage and Decode stage same through an Enable Control logic circuit. Figure 2.3.2.2.3.5 was the elaborated design schematic of Hazard Unit component, and the logic circuits in the diagram were generated from Verilog code by Vivado.

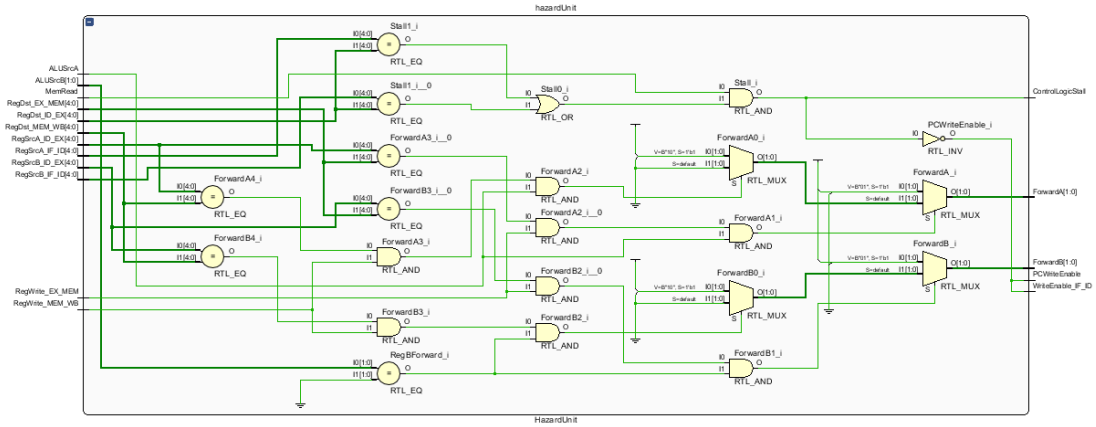

Figure 2.3.2.2.3.5: HazardUnit Elaborated Design Schematic

2.3.2.2.3.6 CoreTop

CoreTop was the top of the core, which connected all the components to enable them work together. It included a register called Program Counter (PC), the choice of source data for the ALU, bits dispatching of sign-extended immediate value, and the wire connection of each component. Figure 2.3.2.2.3.6 was the elaborated design schematic of top structure of CPU component, and the logic circuits in the diagram were generated from Verilog code by Vivado.

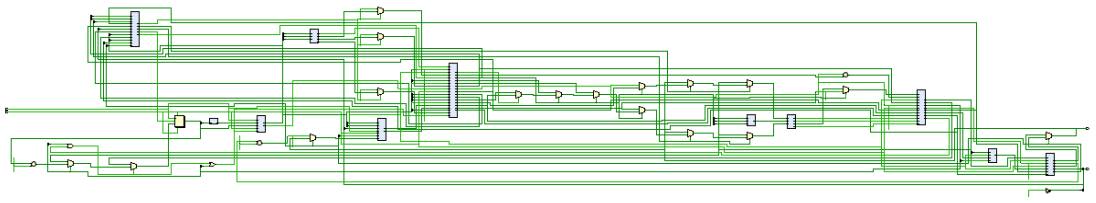

Figure 2.3.2.2.3.6: CoreTop Elaborated Design Schematic

2.3.2.3 System on Chip

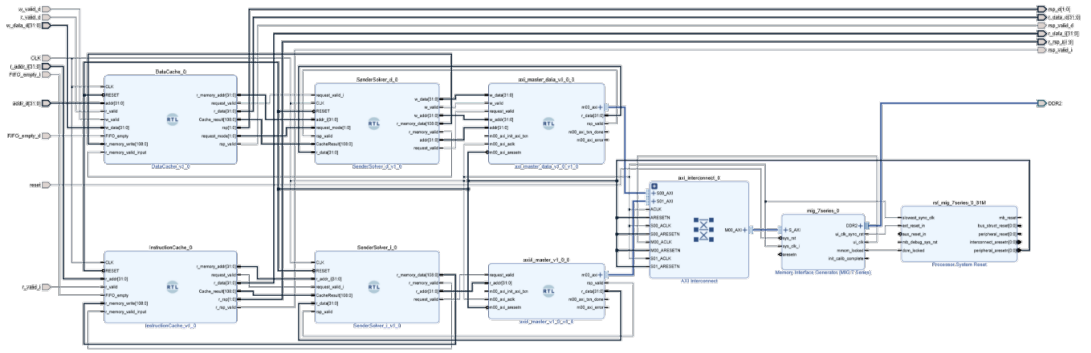

Figure 2.3.2.3.1: System on chip Block Design Schematic (Only Cache and MIG)

As the figure 2.3.2.3.1 showed, the system on chip part was designed by Vivado block design so that designer could not spend much time on port naming and connecting.

2.3.2.3.1 Cache

Due to the time limitation, cache part including structure design and code would have many problems and unknown bugs. Therefore, some design described in the method part might be incorrect because it was about the current design. In discussion section, some of the problems and their solution would be contained. Moreover, there were no automatic tests in cache part.

2.3.2.3.1.1 Storage Structure

Data/Instruction Cache was constructed by Xilinx Parameterized Macros (XPM) Single Port RAM (xpm_memory_spram). This single port RAM used LUT to construct a distributed RAM so that the RAM could provide result within one cycle. The size of cache was 111616 bits, which was 1 KB because the byte size of cache is 109. Since the size of cache was small, cache could use LUT RAM; otherwise, Block RAM would be better with higher latency. However, instruction cache should keep low latency because instruction cache was not necessary to be large enough for the processor. Furthermore, to increase the hit rate of the cache, cache used two associativity. Two

associativity meant that one word in cache could store two data, which helped the cache hit because there was enough space to avoid the situation that instruction read A, read B, and read A again.

One byte of the cache consisted of Valid bit (V), Used bit (U), Dirty bit (D), Tag bits, Data bits. The detail of the arrangement in the cache was as the Figure 2.3.2.3.2 showed. Set bits in address was to choose which block in the cache and due to 1KB size of cache, 10 bits of Set would be necessary. In addition, cache used least recently used policy to evict the block in cache. Used bit was to show which one in the cache is recently used and then evict the other one for the new data. Moreover, cache would only write the dirty block into the memory if it had to be evicted so as to reduce the memory operation.

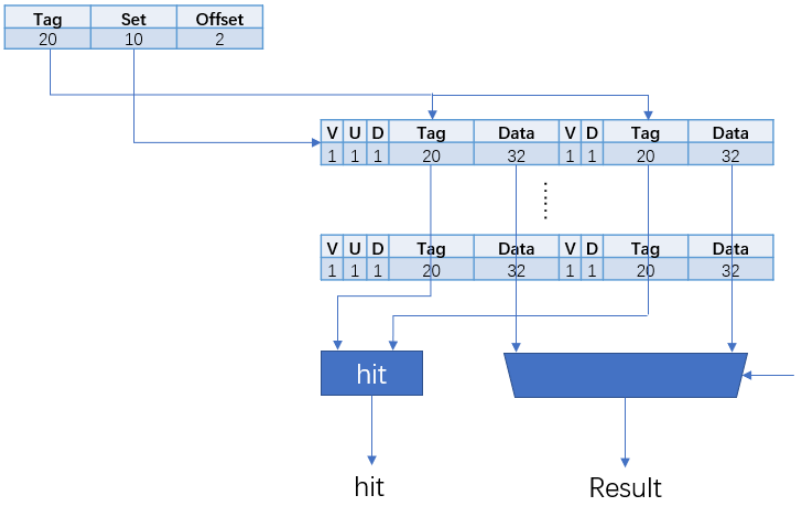

Figure 2.3.2.3.2: Storage Diagram and Basic cache structure

2.3.2.3.1.2 Cache Structure

In this project, cache was constructed by three stage pipeline structure. The first stage was Read stage, the second one was Comparison stage, and the final one was Response Stage. The first stage was to read the data from cache according to the tag that was bits 11 to 2. The second stage was to compare the offset with the address that instruction required to read or write. There were several results of comparison stage, which were cache hit (read instruction), cache hit (write instruction), cache miss (read instruction), and cache miss (write instruction). Cache hit (write instruction) would

replace the data reading from cache during comparison stage. In Response stage, if the cache miss, it would response to the CPU that it needed preparation and sent the request to a component specially solving it. If the cache hit, it would response to CPU that all data was ready, and CPU would get the result when they required the data again. The difference between instruction cache and data cache was that instruction cache didn't take writing into account and it would only deal with the reading request. Figure 2.3.2.3.3 and Figure 2.3.2.3.4 were the elaborated design schematic of instruction cache and data cache component, and the logic circuits in the diagram were generated from Verilog code by Vivado.

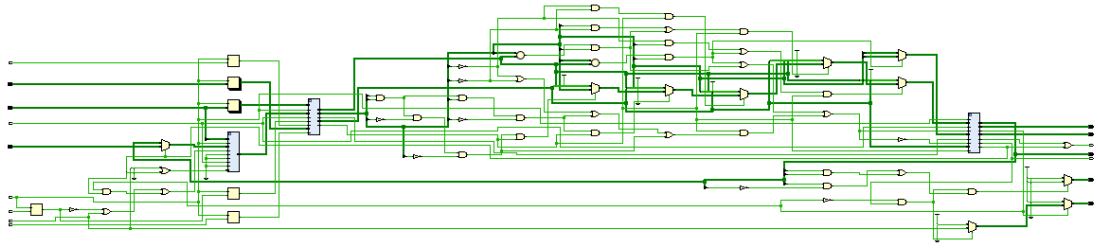

Figure 2.3.2.3.3: Data Cache Elaborated Design Schematic

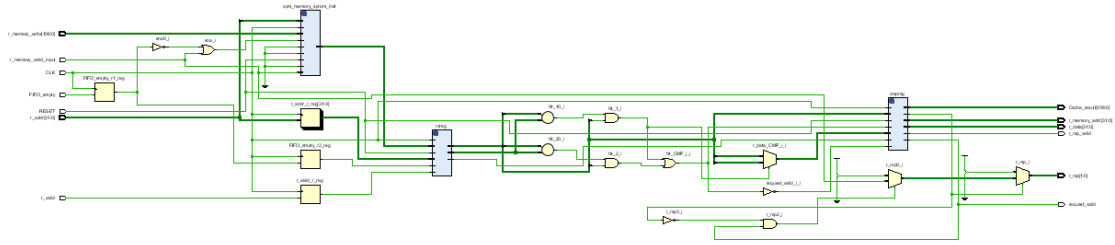

Figure 2.3.2.3.4: Instruction Cache Elaborated Design Schematic

2.3.2.3.1.3 Communication between CPU and cache

Communication between cache and CPU was designed to use asynchronous FIFO. This asynchronous FIFO was constructed by Asynchronous FIFO (xpm_fifo_async) in Xilinx Parameterized Macro (XPM). There were two FIFOs between Cache and CPU, and the data pushed would have valid bit and data bits to inform Cache or CPU whether the operation was ready or not.

2.3.2.3.2 Memory Interface and Request Solver

Request Solver part would have some problems and unknown bugs because there were no tests due to time limitation. However, memory interface would work correctly because it had been tested when dealing with SDRAM to DDR component. This test would be provided with more detail in the later section.

2.3.2.3.2.1 Request Solver

Request solver included two parts that are sender solver part and Advanced eXtensible Interface (AXI) part. The former would deal with the request from the cache and translate them into proper code type and then sent them to the AXI master. The latter included two AXI4 master and an AXI interconnect to translate the information from sender solver into what the AXI4 bus required.

The reason why cache would not directly translate the request into AXI4 mode was that it was complicated if the cache included all the communication between memory and cache. Therefore, sender solver would have duty on this. It would not only deal with the request reading or writing data into the memory but also got the result from AXI4 master, and then returned them to the cache. The difference between sender solver for data cache and sender solver for instruction cache was that instruction cache only needed to read data from memory while the other one needed to read or write the data from or into the memory.

AXI part included two AXI4 masters and one AXI interconnect. The version of AXI4 master is 1.0 and the version of AXI interconnect is 2.1 (Rev. 21). Both of two components used Xilinx IP core to simplify the design complexity. AXI4 protocol was developed by ARM and it could solve the communication problems between different components. Since memory interface only supports AXI4 FULL instead of AXI4 LITE, a simple form of AXI4 protocol, all the AXI part should be designed for AXI4 FULL protocol. AXI protocol was burst-based and it had five independent transaction channels that were Write Address Channel, Write Data Channel, Write Response Channel, Read Address Channel, Read Data Channel. More details about AXI4 FULL bus would refer to UG1037 and the figure below [5]. In Vivado 2019.1 block diagram

design, it would automatically connect the AXI4 master and slave. Only data sending and receiving in both two masters would be dealt by designer. AXI4 master was responsible for sending request in the protocol; thus, the sender solver would connect to an AXI4 master and sent the information to that component. In this project, Memory interface would become AXI slave, which would receive requests and respond. AXI interconnect was to help multiple AXI4 master to connect to one slave in this project. It would be synthesized by Vivado automatically and didn't need to be changed.

2.3.3 Memory Interface and SRAM to DDR component

2.3.3.1 SRAM to DDR component

Before talking about the memory interface, SRAM to DDR component was given by Digilent that could enable the designer to regard DDR2 SDRAM as static RAM. The FPGA board had 128MB DDR2 SDRAM, and the word size was 16 bits. Digilent provided all the options and constraints of this component. Designer only needed to use eight ports to communicate with memory. Since it didn't have valid bit transfer, before reading and writing, there should be a period to keep the input stable. On the official document, the minimum cycle time for reading was 210ns and writing was 260ns. However, there were some problems in this component, and they would be mentioned in the later chapter.

2.3.3.2 Memory Interface

Due to the problems in SRAM to DDR component, Memory Interface Generator (MIG) provided by Xilinx could generate a memory interface to replace this component. The version of MIG was 4.2 (Rev. 1). It could automatically generate an AXI4 interface to connect to the AXI4 interconnect component. The option of Digilent Nexys A7 DDR2 SDRAM was shown by Figure 2.3.3.1.

Parameter Name	Option
Memory Selection	DDR2 SDRAM
Option for Controller 0 - DDR2 SDRAM	
Clock Period	3,077 ps
PHY to Controller Clock Ratio	4:01
Memory Type	Components
Memory Part	MT47H64M16HR-25E
Data Width	16
ECC	Disabled
Data Mask	Yes
Number of Bank Machines	4
ORDERING	Strict
AXI Parameter Options C0 - DDR2 SDRAM	
Data Width	32
Arbitration Scheme	RD_PRI_REG
Narrow Burst Support	1
Address Width	27
ID Width	2
Memory Options C0 - DDR2 SDRAM	
Input Clock Period	3077 ps (324.992MHz)
Burst type	Sequential
Output Drive Strength	Fullstrength
RTT (nominal) - ODT	50ohms
Controller Chip Selection Pin	Enable
Memory Address Mapping Selection	Second
FPGA options	
System Clock	Single-Ended
Reference Clock	Differential
System Reset Polarity	ACTIVE LOW
Internal Vref	Yes
IO Power Reduction	ON
XADC instantiation	Enable
Extended FPGA Options	
Internal Termination Impedence	50 Ohms
System Signals Selection	
sys_clk_i	Bank 35, E3(MRCC_P)

Figure 2.3.3.1: Memory Interface Generator Option for Digilent Nexys A7

2.3.3.3 System clock generation

On FPGA board, there was only one clock generator and its frequency was 100MHz. However, to fulfill the clock requirements of cache and MIG, Clock Wizard, an IP core provided by Xilinx would be utilized. It could use system clock to generate the clock whose frequency was expected. For example, since MIG needs 3077ps and 200MHz clock, these two clocks frequency both needed to be generated by Clock Wizard.

2.4 Bootloader

2.4.1 Architecture

The source code of U-Boot can be downloaded from U-Boot official website. From the downloaded source code, the source code related to RISC-V was extracted and used,

and then delete other unnecessary parts to prevent confusion. The architecture of U-Boot consists of many directory trees.

(1) | --board:

This direction includes files related to some existing development boards. Each development board appears in the current directory as a subdirectory.

(2) | --common:

This direction implements the commands supported by the U-Boot command line. And each command corresponds to a file. For example, the *bootm* command corresponds to *cmd_bootm.c.*

(3) | --cpu:

This direction concludes a directory related to a specific CPU architecture. Each CPU supported under U-Boot corresponds to a subdirectory under this directory.

(4) | --disk:

This section contains the support for disks.

(5) | --doc:

This direction concludes documentation directory.

(6) | --drivers:

Device drivers supported by U-Boot are placed in this directory, such as various network cards, CFI-supported Flash, serial ports, and USB.

(7) | --fs:

This direction contains supported file systems. U-Boot now supports *cramfs, fat, fdos, jffs2,* and *registerfs.*

(8) | --include:

This direction includes header files used by U-Boot, as well as assembly files that support various hardware platforms, system configuration files, and files that support the file system.

(9) | --lib_xxxx:

This direction contains architecture-dependent library files. For example, ARM-related libraries are placed in *lib_arm.*

(10)| --net:

This direction contains the code related to the network protocol stack, the implementation of the BOOTP protocol, the TFTP protocol, the RARP protocol, and the NFS file system.

(11)| --tools:

This direction includes U-Boot tools, such as: *mkimage, crc.*

2.4.2 Boot mode

Most Boot Loaders include two different operation modes. The first one is Boot loading mode, which is also called Autonomous mode. In this mode, Boot Loader loads the operating system into RAM from a solid-state storage device on the target machine, and the entire process does not involve user intervention. This mode is the normal working mode of Bootloader. The second mode is the Downloading mode. In this mode, the Boot Loader on the target machine will download files from the Host through a serial connection or network connection, such as downloading the kernel image and root file system image. The file downloaded from the host is usually first saved by the Bootloader to the RAM of the target machine, and then written by the Bootloader to the FLASH solid-state storage device on the target machine. This mode of Bootloader is usually used when the kernel and root file system are installed for the first time; in addition, this system mode of Bootloader will also be used for future system updates. Boot Loader working in this mode usually provides a simple command line interface to its end users.

2.4.3 Boot process

The ultimate purpose of u-boot is to read the kernel from flash, put it into memory, and then start the kernel. U-boot starts in two phases. The first phase is the assembly phase, which runs mainly in the *start.S* file. The main work at this stage is to initialize the hardware. When the RISC-V computer is powered on, it initializes itself and runs the boot loader, which is stored in read-only memory. U-Boot loads the xv6 kernel into memory. Then, in machine mode, the CPU starts executing xv6 from the *_entry* function. When the program jumps to the start_boot function, the second phase using

C language begins. This phase is like a loop, starting the console shell first, then processing the isolated zombies until the shell exits and repeats. After completing these operations, the system starts. U-Boot is essentially a bare-metal program rather than an operating system. Once U-Boot starts running, other programs cannot run at the same time. Once U-Boot finishes running, you cannot go back to U-Boot. Therefore, U-Boot died after it started the kernel. If you want to see the U-Boot interface again, you can only restart the system. Restart is not a resurrection of U-Boot just now, restart is just another life of U-Boot.

Chapter 3: Results

Project Github repository webpage: https://github.com/dingqy/ELEC222-Project
xv6-riscv Github repository webpage: https://github.com/sggfang/xv6-riscv.git
RISC-V Core Github repository webpage: https://github.com/dingqy/riscv-core
All work had been uploaded into the repository and it also included the progress track of each student.

3.1 Operating system part

This operating system has not been fully completed, and some components have limited functions. For example, it does not have CPU scheduling algorithm and even some components achieve the lock mechanism, the deadlock still occurs. Therefore, the multi-thread mechanism is temporarily unavailable. Moreover, the maximum size of file is 268 blocks (268*2014 bytes). The reason has been discussed at the end of previous chapter. In modern operating system, such as MacOS, user can create any size files as long as the disk is larger enough. Therefore, the upper bound of file size is a nonnegligible issue.

However, some components have been done well. For example, the pipes used to transfer messages can greatly reduce the use of data buses so that the CPU can execute more instructions per unit time. The buddy allocation and lazy allocation help to firstly allocate memory in block unit and delay the use of memory so that CPU can make more space for other processes and make full use of memory. Furthermore, there are some small programs, such as *sleep* command, *find* command, pipe application and shell simulation.

The *sleep* command is used to pause the whole system a few time (ie. *sleep 100* will pause the system for ten second). The find command is used to kook for all files under specified directory (Figure 3.1.1). There is an interesting program called *prime*. It is based on the idea Unix Pipes. The first process feeds the numbers 2 through 35 into the pipeline. For each prime number, you will arrange to create one process that reads

from its left neighbor over a pipe and writes to its right neighbor over another pipe. All the numbers can be divided by the prime number will be discarded as shown in the following figure. Figure 3.1.2 shows the function of *prime*.

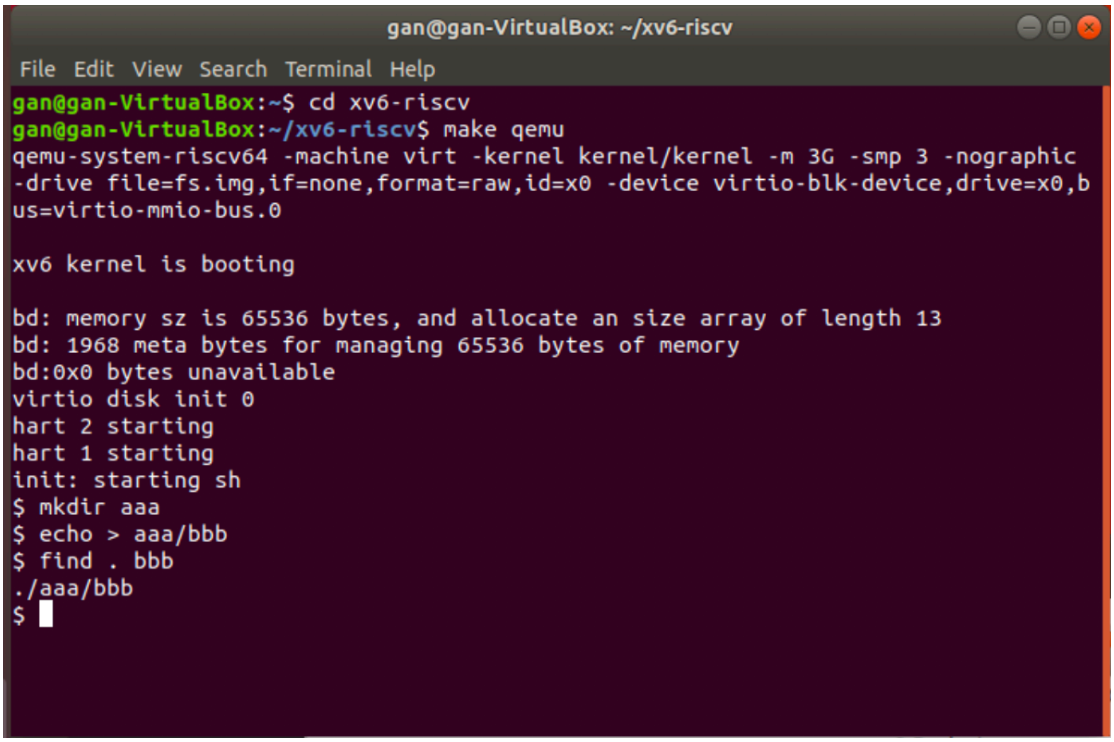

Figure 3.1.1: find command

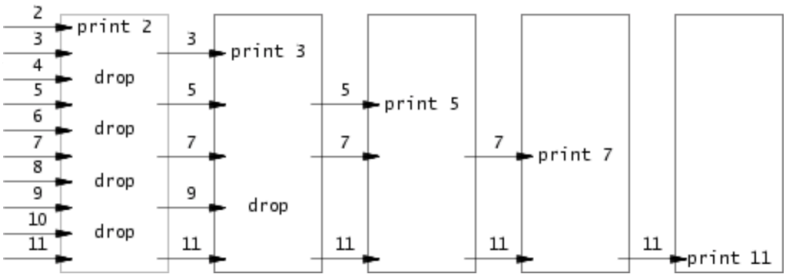

Figure 3.1.2: pipe transfer (prime)

The *find* is used to find the directory or file under the given path. For example, we make a directory *aaa* and then create a file *bbb* inside *aaa*. If we want to find all files called *bbb* in current directory, we just need to write *find . bbb*. The second argument . means it starts in the current directory. When the operating system is running, the user can type *nsh* to run a simulated shell. This shell has the same function with original shell and the only different thing is that each command starts with @ instead of $.

3.2 System on chip part

3.2.1 Testing result

3.2.1.1 Automatic tests

All the tests in current project progress were unit tests about each component. It would generate corresponding inputs and compare output with the expected signal. The test of core was to use simple RISC-V program to observe the signal in the circuit. Due to the uncomplete cache, during the test, instruction memory and data memory were all replaced by simple static RAM on FPGA board.

Due to the time limitation, some tests were not finished. They are complete verification of core, cache test, request solver test, and SoC test. In addition, since due to temporary change of Control Unit, current Control Unit test could be utilized. However, during simple core test and previous Control Unit test, it could work successfully.

3.2.1.2 Testing Diagram

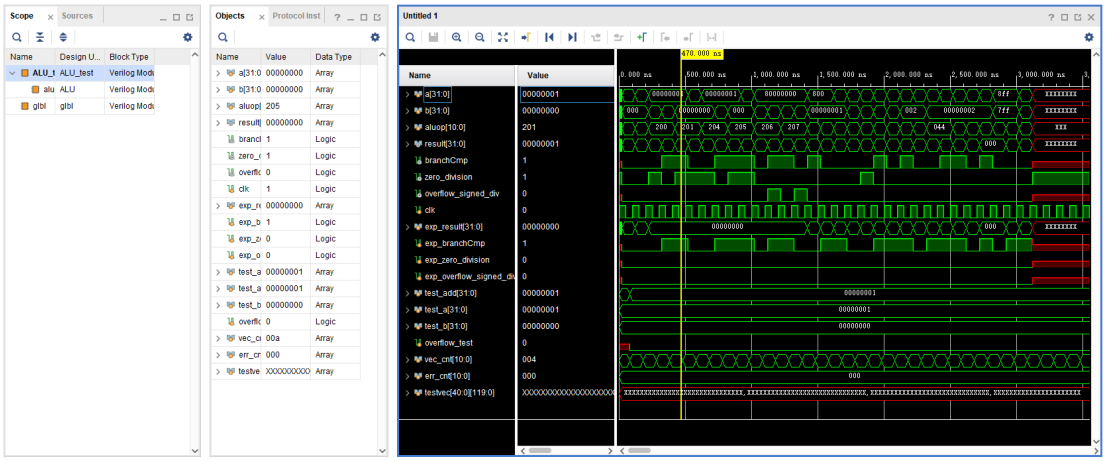

Figure 3.2.1: ALU Simulation Diagram

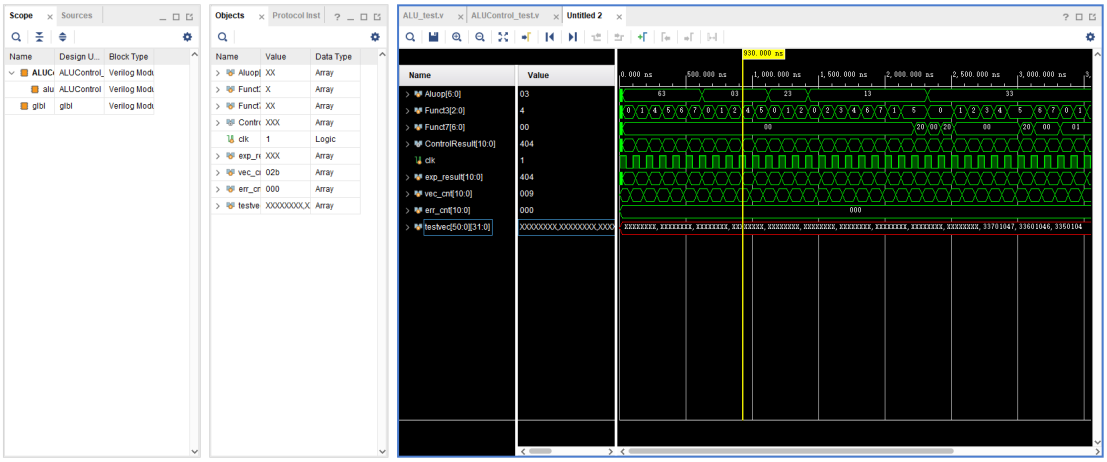

Figure 3.2.2: ALU Control Simulation Diagram

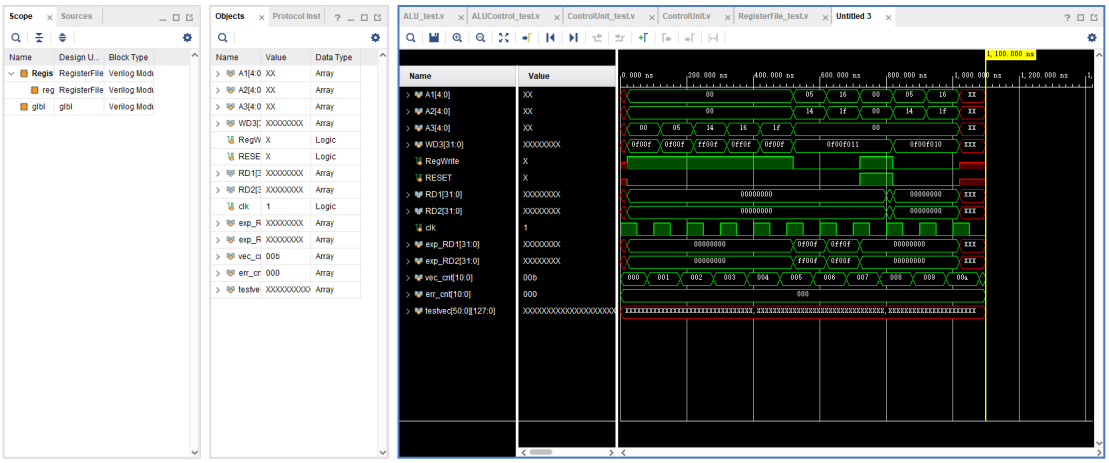

Figure 3.2.3: RegisterFile Control Simulation Diagram

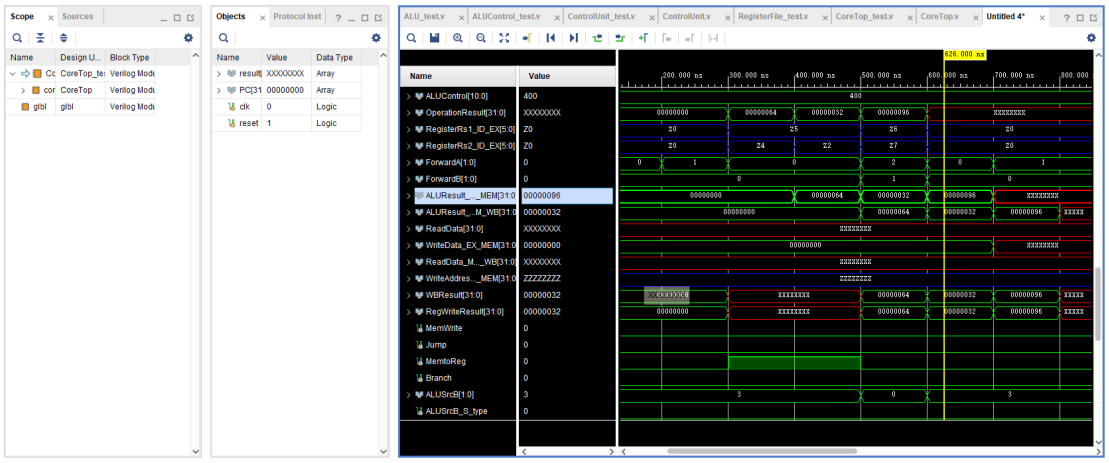

Figure 3.2.4: CoreTop Control Simulation Diagram (part of the simulation diagram)

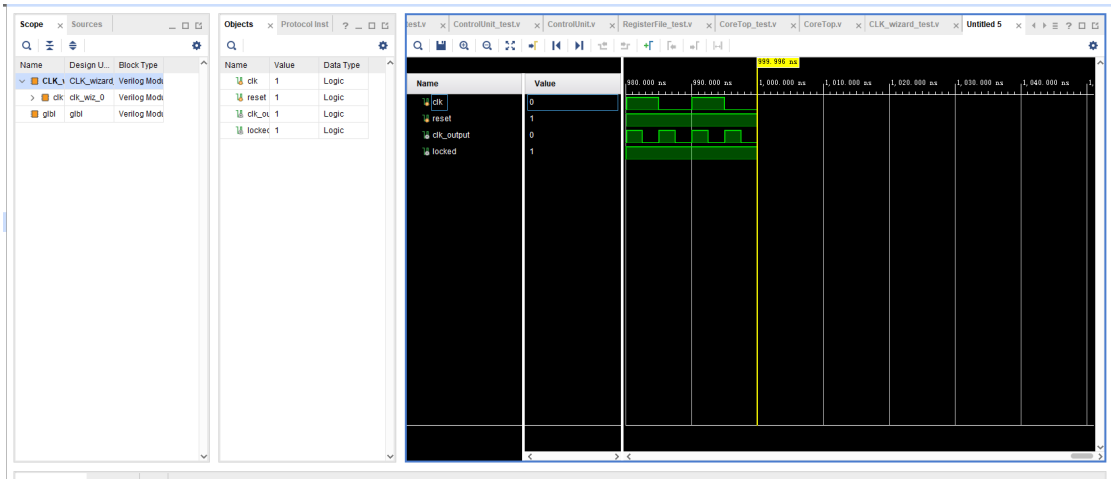

Figure 3.2.5: Clock Wizard Control Simulation Diagram

3.2.1.3 SRAM to DDR component test

This test was implemented to execute on the FPGA board; thus, there was no simulation diagram. In 'tb' file folder, there was *MMU_test.xdc* and *MMU_test*. The former was the constraints of the test and the latter was the circuit built in the FPGA board. The final result was that when writing and reading repeatedly from one memory location, the LED on FPGA board to show the data in the memory would be firstly correctly and then become the value that was unknown, and finally, all the LED would not light. One of the possible reasons was the memory fresh rate and detail about this result would be discussed below.

Chapter 4: Discussion and Conclusions

In this project, there were no data collection and analysis; thus, all the results were the simulation result. In addition, since not all the components were completed or worked correctly, the whole System on Chip could not have direct exhibition on the board. However, all the simulation showed that the 5-stage pipeline processor core could work, although they were not tested entirely. Moreover, SRAM to DDR component testing were shown on the board and there existed some problems when actually testing on the board. Additionally, CPU benchmark was not applied due to the time limitation and the default main frequency was 100 MHz.

As mentioned in previous chapter, some components of this operating system have been done well. For example, the pipes used to transfer messages can greatly reduce the use of data buses so that the CPU can execute more instructions per unit time. The buddy allocation and lazy allocation help to firstly allocate memory in block unit and delay the use of memory so that CPU can make more space for other processes and make full use of memory. Furthermore, there are some small programs, such as *sleep* command, *find* command, pipe application and shell simulation.

4.1 Limitation of SoC

4.1.1 Test limitation

Test limitation is one of the most significant problems during this project because test may take huge amount of time to include all the circumstances. Moreover, from unit tests to system tests, they are too complicated to test them simply. For example, only a CPU core instruction set test needs to test more than fifty instructions and their dependency issues. Therefore, in this project, there are only unit tests of CPU core component and simple system test of CPU core due to the limited time of the project.

4.1.2 Performance limitation

The core is a 5-stage pipeline scalar in-order processor. The Instruction Per Cycle

(IPC) was lower than 1. In addition, due to the large memory operation time, it may be difficult to keep the frequency up to 100 MHz. For example, if the project applied SRAM to DDR component, which spends 210ns for reading and 260ns for writing, memory operation may waste more than 20 cycles. Therefore, the design of cache would determine the performance of the CPU. Moreover, performance limitation of the SoC may affect the final exhibition when executing operating system.

4.1.3 Application limitation

Due to limited tests and uncomplete components, before generating bitstream into the FPGA board, all the results will be the simulation of Vivado. It doesn't mean that all the logic circuit would work correctly on FPGA board. This would cause many unknown problems and bugs when actually using the board. For example, SRAM to DDR component behave normally when simulating in the Vivado, but it caused some strange problems when testing on the board.

FPGA board was also one of application limitation because when the system on chip become more and more complex, the logic circuit needs more LUTs, RAMs, and I/Os, which caused FPGA not to support the logic. However, in the current progress of the project, it is not a problem, but it may be if the system becomes complete.

4.2 Limitation of xv6

Compared with the initial goal, there is still some works unfinished. For example, it does not have CPU scheduling algorithm and even some components achieve the lock mechanism, the deadlock still occurs. Therefore, the multi-thread mechanism is temporarily unavailable. Moreover, the maximum size of file is 268 blocks (268*2014 bytes). The reason has been discussed at the end of previous chapter. In modern operating system, such as MacOS, user can create any size files as long as the disk is larger enough. Therefore, the upper bound of file size is a nonnegligible issue. In the initial goal, we used to plan to design a network driver, but the time is not enough.

Most of modern operating systems, such as Unix, Linux, MacOS and Windows, have graphics interface, which mean they can provide better visual experience for user.

For example, the user can go through the files by looking the map. However, in xv6, user can only look at the command line and interaction only using the command.

Another limitation is that xv6 cannot deal with exceptions. This problem is caused by the time limitation. If given another two or three weeks, the exception handler may be able to be finished.

4.3 Problems and obstacles

4.3.1 RISC-V instruction set determination

Choosing proper instruction set is a large problem in this project. Due to little knowledge about the hardware/software interface, determination of instruction set architecture affects the progress of making plan of the project. Originally, RV32I is enough for operating system. However, when designing operating system, RV32IMA is necessary. Nevertheless, RV32A may be unnecessary because it is used to support atomicity of multi-thread. Finally, RV32IM are determined and becomes two instruction sets designed and constructed in the core.

4.3.2 Scalability and readability of component design

The style of naming components and wires, and comments are essential when writing Verilog code. Compared with VHDL, Verilog is not as strict as VHDL so that when designing complicated system, Verilog may cause less understandability. Therefore, comments and proper names can help a lot when writing Verilog code. However, it still lowers the efficiency because there were too much wires and components, which induces designer to spend a lot of time reading code and understanding what code means.

Scalability are also crucial when designing. Arithmetic Logic Unit is designed twice because when designing at first time, the scalability and readability of the Verilog code are low so that RV32M extension set could not be correctly added into that ALU design. In addition, more than fifty instructions also increase the difficult of whole structure due to the different function of instruction. For example, there were signed instructions and unsigned instructions that make logic more difficult to distinguish.

4.3.3 The complexity of Cache design

There are different types of cache including L1 cache, L2 cache, and L3 cache. In this project, to simplify the design, there is only L2 cache; however, it doesn't mean that it is a proper design choice because it needs a lot of experiments to find an efficient way to decide the proper memory hierarchy.

Moreover, the associativity of cache is also a problem when designing the structure of the cache. One associativity is the simplest design, but it may cause a large number of cache miss. Therefore, to avoid cache miss that induces the lower performance, two associativity were used in this project. However, it makes the cache design be more complicated and causes a lot of bugs and uncorrected connection.

4.3.4 The complexity of copy-on-write fork

In xv6, the *fork()* system call copies all of the parent process's user-space memory into the child. If the parent is large, this step will take a long time and take up a lot of free space. The inefficiency is particularly clear if the child calls *exec()*, since *exec()* will throw away the copied pages, probably without using most of them. A better solution is to make parent process and child process to share the same physical memory. The goal of *copy-on-write fork* is to defer allocating and copying physical memory pages for the child until the copies are actually needed. For example, if both parent and child use a page, and one or both writes it, a copy is truly needed.

The difficulty of this mechanism is that it marks all the user PTEs in both parent and child as not writable. When either process tries to write one of these pages, the CPU will force a page fault, which requires kernel page-fault handler to detect this case, allocate a page of physical memory for the faulting process. It sounds similar to lazy allocation, but with a little difference. Although it took half of a week in this mechanism, there is still some unknown problems and it has to be replaced by lazy allocation.

4.3.5 Poor file system

Currently xv6 files are limited to 268 blocks, or 268*BSIZE bytes. This limit

comes from the fact that an xv6 inode contains 12 "direct" block numbers and one "singly-indirect" block number, which refers to a block that holds up to 256 more block numbers, for a total of 12+256=268 blocks.

Some modern operating system, such as Unix, support “doubly-indirect” block, in addition to direct blocks and a “singly-indirect” block. It contains only 11 direct blocks The first 11 elements of *ip->addrs[]* should be direct blocks; the 12th should be a “singly-indirect” block; the 13th should be “doubly-indirect” block. With this mechanism, it allows user to allocate 65803 blocks for each file (compared with 268 blocks in xv6). However, this is too difficult to implement, so a compromised method is to implement a simple one instead.

4.4 Memory Management Unit Design

4.4.1 SRAM to DDR component

After testing the SRAM to DDR component provided by Digilent. There were two main problems. The one was that after read and write to a location many times, the data in such memory location would lose. The other one was that reading and writing memory were too slow to enable the processor to have expected performance.

For the first problem, the ILA IP core showed that SRAM to DDR component outputted the right waveform. Therefore, one of the possible reasons was the charge loss in DDR2 SDRAM. This component doesn’t set the DDR2 memory fresh rate and according to the RowHammer problem, after several reading and writing, the data in memory may be lost [8].

The reason for the other problem was that this component didn’t take performance into consideration as the Digilent forum mentioned [16]. Hence, 210ns reading period and 260ns writing period could not be solved if using such component. However, it was not acceptable for an expected 100MHz CPU.

4.4.2 Memory Interface Generator

Memory Interface Generator (MIG) was an IP core provided by Xilinx, which can automatically generate a memory interface for the FPGA board. However, the option

of this FPGA board was difficult to find. Digilent official website only provided parts of the option for this DDR2 SDRAM component and other option should be tested or be necessary to read many documents.

Furthermore, the interface part of MIG IP core could use regular RAM interface or AXI4 interface, but it is difficult to understand what actually these two interfaces work.

4.4.3 Advanced eXtensible Interface 4 (AXI4)

The first problem is how AXI4 worked. It is necessary to read several hundred pages in UG1037 to understand how AXI4 protocol transfer data between two components [5]. In addition, there exists a lot of ports in AXI4 protocol on Memory Management Unit AXI4 slave interface, which should be distinguished clearly when designing AXI4 connection.

4.5 Test

One of the most important problem is test problem. It is because tests may take much time to validate the correction of a component that the time spending on tests was almost the same as design. However, there were still many components that were not tested entirely or even not tested. Furthermore, there is a complicated verification process about RISC-V System on Chip and it is too difficult to follow in this project.

4.6 Debug problem

Debug in Vivado is not as easy as that in software development. The first debug way is to simulate the Verilog code and observe all the signal in the simulation. However, for a relatively large system, it is inefficient to find debug by eyes. Therefore, debug in this project causes huge time waste. The other debug way is to use ILA core when testing the Verilog code on board. Nevertheless, using ILA core will be more difficult than testing by simulation because it is not simple to let the core show what the designer wants to know.

4.7 Project management and time management

The project management was difficult and complex because students didn't know how long it would take for students to finish their parts. Additionally, there were too many unknown problems for this project. The most serious problem that affected the progress in System on Chip part was SRAM to DDR component. To replace this component, students had to use two weeks, one-third of the whole project, to read relative reference books and documents and then started to design. It also caused the project to stop at constructing instead of integrating all the parts including operating system, hardware and boot program. Due to no General Purpose Input/Output (GPIO) and not enough static RAM on FPGA board, there was no way to load the xv6 operating system into the memory so that the group could not start the final test. Six weeks also asked the students to strictly follow the plan of the project with no extra time for the risk of difficult problems, but the reality was that the expectation would not happen at most of the time.

4.8 Improvement

4.8.1 Current bugs and their improvement

Table 4.8.1.1 Bugs and solutions

Bugs	Solution
Data cache uses single port LUT memory (cannot read and write at the same time)	Replace single port memory with dual port distributed memory in XPM
When cache hit and the block is dirty, current logic circuits could not correctly send a request to store the data in block.	Add an extra port for the previous data in the block so that the request can include correct data.
No AXI4 design in AXI4 master component	Construct new logic circuit in this component
Cache cannot response when it fetches the block that CPU wants.	Add more logic circuit for updating the condition of cache to CPU
CPU should request again if the cache	When writing back to cache, the data

doesn’t have the data in the memory that CPU request to.	could also be pushed to the CPU through bus.

4.8.2 CPU core Improvement

Since the current CPU core is in-order and scalar, the improvement can change the structure of the core. The expected change is out-of-order execution and superscalar processor. Out-of-order execution needs a scheduler and a reorder buffer to keep the instruction order. It can lower the penalty when memory operation needs to stall the pipeline because it can enable another instruction that doesn’t depend on the current instruction to be executed. Two scalar processor can be better in this project because superscalar processor can execute multiple instructions at the same time and two scalar processor will not increase too much complexity.

Branch prediction is also an important part about the performance of CPU core. In this project, to simplify the design, static branch prediction is used; however, dynamic branch prediction will be better to apply. Combining local branch prediction and global branch prediction could predict the loop and if condition better [6]. In addition, Branch Target Buffer (BTB) can also be used to increase the performance when executing branch instruction.

Interrupt controller is also the one that operating system requires. All the I/O and error needs interrupt controller to stop the processor and work all them. System call in Operating System is to use *ECALL* instruction to inform the CPU to enter kernel mode such as reading from I/O device.

4.8.3 Cache Improvement

Due to time limitation, there was less tests on cache part so there were many bugs as mentioned above. Thus, the fixation of bugs was the most essential improvement. In addition, the hierarchy of cache could be more separated, which means that there could be L1 cache and L3 cache instead of only L2 cache. On the other hand, cache latency should be taken into consideration. Most importantly, tests should be increased for

cache part because it would be more difficult than simulating on the Vivado because designer could only know what happens when they generate bitstream into the FPGA board and use ILA core to observe the signal in the digital circuit.

4.8.4 System on chip

In this project, many parts of system on chip are not completed. General Purpose Input/Output (GPIO), PS/2 Driver, VGA Driver, Ethernet Driver, USB Driver, UART/JTAG controller. To enable the project to be the same as expectation, GPIO, VGA, PS/2, and UART are necessary because GPIO can help designer load the program into the memory and UART driver can keep the boot program and boot the operating system. VGA and PS/2 Driver are for exhibition and these two parts can allow the keyboard and screen to work successfully. All the components will be connected by AXI4 FULL protocol.

4.8.5 xv6 design improvement

All the limitations and problems mentioned in previous chapter can be solved and further explored. For the copy-on-write fork mechanism, it can be finished by deeper reading the source code and referring to some tutorials on Stackoverflow. For the file system, it can be divided into several parts and finished step by step. For example, firstly change the *ip->addrs[] content,* secondly write the “singly-indirect” block and “doubly-indirect” block respectively. Or the second step can be further divided if necessary. For the exception handler, it is easier compared with previous two. The simplest method is to abort the current process and print which process was aborted and the exception type on the console.

In modern operating system, different user can have their private configuration and resources. It is decided when user starts the system and enter their own name and password. This is important to protect user’s privacy, so it worth spending time. As for the graphics interface, it requires more hardware components, such as Graphics Processing Unit, and more complex software design. It is quite interesting to explore this part and may be time-consuming as well.

4.8.6 Project management Improvement

Since the project is a system construction project, the work division in the project can be better. Currently, one designs all the system part, one designs operating system, the other transplants u-boot onto the system on chip. However, this work division makes the student who designs hardware affords huge workload, which causes the System on Chip part to be delayed as long as some unexpected problem happens. Therefore, an improvement is that two students can design the hardware, which can be separated that one student focus on CPU core and the other one can construct the peripheral on the System on Chip. All the software part can be constructed by one student because it can use existing operating system and boot program, and just needs to transplant them into the proper form.

Time management can also be improved for such a complicated system. Due to only six weeks normal period, more preparation should be made before the starting time. For example, the instruction set determination cab be ensured before starting the project. Additionally, the expectation of the project should also be lowered to fulfill the time limitation such as transplanting ucore instead of xv6 operating system.

4.8.7 Ethical, Security, Intellectual, Commercial aspects

Due to the open-source character of RISC-V, only IP core provided by Xilinx may have some limitation for using. For example, if using IP core for commercial usage, designer should first apply a license from Xilinx. Otherwise, the designer may violate the regulations of Xilinx.

Security for this project was a large problem but not necessary for current progress. For example, for DDR2 SDRAM, RowHammer problem can be one of the computer hardware security problems [8].

RISC-V performs well in embedded system industry; thus, if the power efficiency of the SoC in this project could be optimized, commercial usage may be possible, but it would also be a difficult problem to weigh between the performance and power consumption.

For xv6, one essential point is to ensure the isolation between user space and kernel space, and among different processes. The user instructions can only use the system call to access kernel resources. Each process should have its private memory space to avoid data clash. Moreover, the operating system also need to ensure each user's information and configuration secure. Currently, there is no individual user in xv6, which means all user share the same resources. However, after further improvement, this will become an important point.

4.8.8 Sustainability

This project uses FPGA as the development board. However, FPGA can be regarded as a perfect development tool for design instead of manufacturing because the resource usability is low for FPGA.

4.8.9 Future of the technology

RISC-V was created by University of California, Berkeley, and now it has been invested by many IC companies. Since RISC-V should not take compatibility of old software and hardware into account, the instruction set can be as few as possible and the structure of RISC-V was also simple than MIPS and ARM processor. Therefore, RISC-V can have better resource usability and as it is open-source, developers can design what they need so that the development community of RISC-V is welcomed. Most importantly, Internet of things may be one of the future technologies and RISC-V, which is processor who is an open-source, lower power assumption, and with highly developed community, were likely to be welcomed in such embedded system industry.

4.9 Conclusion

In conclusion, although this project was not integrated together, there still several works has been achieved. In operating system part, we develop many components, including interfaces, page table, traps and file system. The interface is consisted of process identification, I/O interface, file descriptor and pipe. The page table part has the algorithm to convert from virtual address or kernel address to physical address.

Besides, in this part, there are two address allocation algorithms, lazy allocation and buddy allocation. For traps in xv6, it can accept and deal with the interrupts from keyboard and system time. It can also deal with console interrupt to print messages. As for the file system, it can only support limited sized file (258 blocks).

System on chip part was all the hardware in this project. It should include all the components that supported the operating system to execute successfully. For Central Processing Unit part, it was designed to be a 5-stage pipeline microprocessor. The microarchitecture of the processors was in-order and scalar, which was simpler. For System on Chip peripheral part, only cache system has been almost completed without tests.

The advantage of this project is that is an exhaustive work involving operating system implementation, microprocess and computer component. From the point of complexity, it fully arose students' motivation to learn deeper and spend more energy in this work. However, there is also a disadvantage that there were too many works to finfish so that this project cannot be integrated successfully. Based on this experience, we will plan the whole work before working and manage time better.

Reference

[1] D. A. Patterson, Computer Organization and Design RISC-V Edition: The Hardware Software Interface, 1 edition ed. Morgan Kaufmann, 2017.

[2] D. Harris and S. Harris, Digital design and computer architecture. Morgan Kaufmann, 2010.

[3] S. Patel and Y. Patt, Introduction to Computing Systems: from bits & gates to C & beyond. McGraw-Hill Professional, 2019.

[4] X. Inc., "Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions v4.2 User Guide (UG586)." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v4_2/ug586_7Series_MIS.pdf

[5] X. Inc., "Vivado Design Suite AXI Reference Guide (UG1037)." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[6] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, 1993.

[7] M. V. Wilkes, "Slave memories and dynamic storage allocation," IEEE Transactions on Electronic Computers, no. 2, pp. 270-271, 1965.

[8] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019.

[9] A. Waterman, K. Asanovic, and S. Inc., "The RISC-V instruction set manual, volume I: Unprivileged ISA," in Document Version 20191213, 2019.

[10]O. U. ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT, "Notes - Unit 5." [Online]. Available: http://www.secs.oakland.edu/~llamocca/Courses/F16_ECE495/Notes%20-%20Unit%205.pdf

[11]"NonTrivial-MIPS." https://github.com/trivialmips/nontrivial-mips

[12]"lowrisc-chip." https://github.com/lowRISC/lowrisc-chip

[13]"riscv_soc." https://github.com/ultraembedded/riscv_soc

[14]D. Inc. "Nexys A7 Reference Manual." https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual

[15]X. Inc. "Community Forums." https://forums.xilinx.com/

[16]D. Inc. "Forum." https://forum.digilentinc.com/?_ga=2.166305335.861222769.1586779602-1437218926.1586248521

[17]O. Mutlu. "Design of Digital Circuits - Spring 2019." https://safari.ethz.ch/digitaltechnik/spring2019/doku.php?id=schedule

[18]Pdos.csail.mit.edu, 2020. [Online]. Available: https://pdos.csail.mit.edu/6.828/2019/xv6/book-riscv-rev0.pdf. [Accessed: 17- Apr- 2020].

[19] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. Commun. ACM, 17(7):365–375, July 1974.

[20]"Bell Labs and CSP Threads", Swtch.com, 2020. [Online]. Available: https://swtch.com/~rsc/thread/ [Accessed: 17- Apr- 2020].

[21]J. Dean, D. Patterson, and C. Young, "A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution," IEEE Micro, vol. 38, no. 2, pp. 21-29, 2018, doi: 10.1109/MM.2018.112130030.

Appendix A

Y2 project (ELEC222/ELEC273) – *Role allocation (responsibility matrix)*

Notes:

- ***Assessors will request to see this sheet in the bench inspection day.***
- ***See overleaf for details on titles and associated roles and responsibilities.***

	Member Name	**Title(s)**
1	Gan Fang	Project Manager Designer Implementer/Developer Technical Writer
2	Qiyang Ding	Project Manager Designer Implementer/Developer Technical Writer
3	Yufeng Chen	Project Manager Designer Implementer/Developer Technical Writer
4		
5		

Responsibility Matrix

KEY:

R – Responsible (accountable) for completion of task. (Task can be delegated to this person.)

S – Supports task.

C – Requires communication about the task.

Title	Project Activity				Deliverables			
	Requirements/ Scope	Design	Implementing	Testing	Poster	Blog	Bench	Report
Project Manager	R	C	S	S	S	S	S	S/R
Designer	C	R	S	S	S	S	S	S
Implementer/ Developer	C	S	R	S	C	C	R	S
Technical Writer	C	S	S	S	R	R	C	R

Typical Roles

Title	Role	Responsibilities
Project Manager	Responsible for developing, in conjunction with the supervisor, the project scope. The Project Manager ensures that the project is delivered on time and to the required standards.	• Managing and lead the project team. • Managing the coordination of the partners and the working groups.
Designer	Designing the system (the circuit, the code, etc.).	• Creating the required block diagrams, circuit diagrams, flow charts, etc.
Implementer/Developer	Implementing the suggested design	• Connecting the systems/circuit • Writing the code

Title	Role	Responsibilities
Technical Writer	Documenting the project progress and deliverables	• Recording and maintaining all meeting logs • Updating the logbook. • Creating project blog • Creating project poster • Writing project report
<Title>	<Role>	• <Responsibility>
<Title>	<Role>	• <Responsibility>
<Title>	<Role>	• <Responsibility>
<Title>	<Role>	• <Responsibility>

Y2 project (ELEC222/ELEC273) - *Contribution to project deliverables*

Notes:

- ***Assessors will request to see this sheet in the bench inspection day.***
- ***Typical deliverables include: Bench, Poster, Blog, Report, Code, Circuit, etc.***

Member name	Deliverable(s)	Comments
Gan Fang	Design operating system, including interfaces, page table, traps and file system. Transplant OS to FPGA board. Design poster; Write blog; Write report; Introduce the OS part in Bench Inspection;	
Qiyang Ding	Design 5 stages pipeline CPU core, Cache System, Memory interface generator, AXI4 bus, and other system on chip components. Design poster; Write blog; Write report; Introduce the CPU and SoC part in Bench Inspection;	
Yufeng Chen	Design bootloader (U-boot). Design poster; Write blog; Write report; Introduce the U-boot part in Bench Inspection;	

Y2 project (ELEC222/ELEC273) - *Attendance record*

Notes:

- ***This sheet should be updated by group members weekly when they meet to discuss the project.***
- ***Assessors will request to see this sheet in the bench inspection day.***

Member name	**Attended the weekly meeting? (Yes/No)**					**Comments**
	Week 1	**Week 2**	**Week 3**	**Week 4**	**Week 5**	
Gan Fang	Yes	Yes	Yes	Yes	Yes	
Qiyang Ding	Yes	Yes	Yes	Yes	Yes	
Yufeng Chen	Yes	Yes	Yes	Yes	Yes	

Year 2 Project (ELEC222/273) – *Supervisor meeting – Week 1*

Date: 30/01/2020 Supervisor: Mark Bowden

Project Title: System on chip with RISC-V microprocessor and Linux operating system

Student Names /Attendees:	1. Gan Fang	2.Qiyang Ding
3.Yufeng Chen	4.	5.

Summary of week's activities:

1. Install running environment. (including risc32-unknown-elf-gcc, spike, pk and qemu)

Problem, issues and concerns:

1. Not receive the equipment.

Tasks for next week/Actions for next meeting:

1. Configure the Linux source
2. ALU design of RV32I and simple control logic unit
3. Read information about u-boot

Supervisor use only

Progress Assessment: □ Unsatisfactory □ Satisfactory □ Good

Comments/Recommendations:

Need to consider project planning carefully by next week

Supervisor Signature: Mark Bowden

Year 2 Project (ELEC222/273) – *Supervisor meeting – Week 2*

Date: 06/02/2020 Supervisor: Mark Bowden

Project Title: System on chip with RISC-V microprocessor and Linux operating system

Student Names /Attendees:	1.Gan Fang	2.Qiyang Ding
3.Yufeng Chen	4.	5.

Summary of week's activities:

1. Read *Instructions: Language of the computer*
2. Explore the structure of U-boot
3. ALU design of RV32I and simple control logic unit
4. Design Linux configuration file and Makefile for RISC-V

Problem, issues and concerns:

How is ECALL achieved

Tasks for next week/Actions for next meeting:

1. Build U-boot
2. Cut the unnecessary part and test on QEMU simulating environment
3. Memory controller design and test

Supervisor use only

Progress Assessment: □ Unsatisfactory ☐ Satisfactory □ Good

Comments/Recommendations:

Supervisor Signature: Mark Bowden

Year 2 Project (ELEC222/273) – *Supervisor meeting – Week 3*

Date: 13/02/2020 Supervisor: Mark Bowden

Project Title: System on chip with RISC-V microprocessor and Linux operating system

Student Names /Attendees:	1. Gan Fang	2. Qiyang Ding
3. Yufeng Chen	4.	5.

Summary of week's activities:

1. Build U-boot.
2. Memory controller design and test.
3. Achieve *sleep, find, xargs and pipeline* functions, shell simulation in original shell and dynamic memory allocation

Problem, issues and concerns:

1. Long memory access latency.
2. Need faster algorithm for buddy allocation.

Tasks for next week/Actions for next meeting:

1. Test U-boot and integrate with OS
2. Drivers on soc design.
3. Copy-on-write fork, alarm call, file system and memory map

Supervisor use only

Progress Assessment: □ Unsatisfactory ☐ Satisfactory □ Good

Comments/Recommendations:

Need to make sure the group has enough time to carry out the integration stage.

Supervisor Signature: Mark Bowden

Year 2 Project (ELEC222/273) – *Supervisor meeting – Week 4*

Date: 19/02/2020 Supervisor: Mark Bowden

Project Title: System on chip with RISC-V microprocessor and Linux operating system

Student Names /Attendees:	1.Gan Fang	2.Qiyang Ding
3.Yufeng Chen	4.	5.

Summary of week's activities:
1. Test U-boot and integrate with OS
2. Drivers on soc design.
3. Copy-on-write fork, alarm call, file system and memory map

Problem, issues and concerns:
1. Not finish U-boot on time

Tasks for next week/Actions for next meeting:
1. Continue and test U-boot
2. Drivers design
3. Copy-on-write fork and rewrite file system

Supervisor use only

Progress Assessment: □ Unsatisfactory ☐ Satisfactory □ Good

Comments/Recommendations:

Continue working according to the plan this week. Meet early next week to discuss about the probability of fully completing the plan.

Supervisor Signature: Mark Bowden

Year 2 Project (ELEC222/273) – *Supervisor meeting – Week 5*

Date: 27/02/2020 Supervisor: Mark Bowden

Project Title: System on chip with RISC-V microprocessor and Linux operating system

Student Names /Attendees:	1. Gan Fang	2. Qiyang Ding
3. Yufeng Chen	4.	5.

Summary of week's activities:
1. U-boot
2. User-level threads and alarm, simple file system

Problem, issues and concerns:
1. Do not have enough time to finish Copy-on-write allocation, but it can be replaced by lazy allocation and buddy allocation, which have been finished before.
2. Perhaps this OS cannot meet the initial goal, such as connecting to internet.

Tasks for next week/Actions for next meeting:
1. Integrate the project
2. Test bootloader, transplant OS on RISC-V
3. Ethernet driver and VGA driver design

Supervisor use only

Progress Assessment: □ Unsatisfactory ☐ Satisfactory □ Good

Comments/Recommendations:
Need to be very clear about which parts are self-generated and which elements are being used temporarily for the bench inspection.

Supervisor Signature: Mark Bowden 27/2

Appendix B

Member	Work
Gan Fang	**Generally:** Design operating system, including interfaces, page table, traps and file system. Transplant OS to FPGA board. Design poster; Write blog; Write report; Introduce the OS part in Bench Inspection; **Explanation:** The main work is to design each component of xv6 based on the given structure (MIT 6.828). The first thing is to read the xv6 tutorial book and the given structure code. Some part of these reading work was finished at the end of last semester, and the rest part was read as I implement each component (It works like a tutorial). The next work is deciding what interfaces we need in xv6. I design three interfaces, including process identification, I/O interface, file descriptor and pipe. The process identification is used to identify each process and abstract the interaction between two processes. File descriptor is used by each process pointing to a private memory address. It strengthens the isolation and avoids data clash. In xv6, the input interface is keyboard port and output interface is console port. The keyboard port can get command from user. The user input will be “packed” as a single unit (each unit was divided by “change line” character). The kernel will deal with one unit at a time. The console port can display the user input

	and the system reaction on the screen. It will refresh thousands of times every second. The pipe is a kind of buffer to transfer messages among processes. Usually, there are two pipes between two processes. The third work is page table mapping, including design the translation between virtual address (in user space) and physical address, and between kernel address and physical address. The details are shown in Chapter 2 operating system part. I also design two allocation mechanisms, lazy allocation and buddy allocation. The fourth work is traps design. In xv6, I design the keyboard interrupt to respond to user command, redirection of memory address of process (used in lazy allocation). However, I don't provide any exception handlers, so xv6 cannot respond to any exceptions. The fifth work is design file system. Limited by the time, this file system is too simple to satisfy the initial goal. It only allow limited size for each file and the directory is "single directed" (only know its child, doesn't know its parent). The final work should be transplanting the xv6 to the board. Since the CPU work is not finished, the transplant cannot be conducted. However, I still use the qemu to simulate the RISC-V environment to running xv6, and the result is shown on bench inspection day and Chapter 3.
Qiyang Ding	**Generally:** Determine the expectation of the project and give some advice about the project management; Design and construct System on Chip, including Central Processing Unit, Cache, and other peripheral; Write blog;

	Write report; Introduce the SoC part in Bench Inspection; **Explanation:** The origin of the project was proposed by the student and then some content in the project was changed during discussion. System on chip part was all the hardware in this project. It should include all the components that supported the operating system to execute successfully. During this project, RISC-V Instruction Set Architecture are chosen to be the ISA, which is suitable for the xv6 operating system. RV32I and RV32M were determined to be constructed because these two instruction sets were necessary for the operating system. In the System on Chip part, it was divided into another two parts that were Central Processing Unit part and SoC peripheral part. For Central Processing Unit part, it was designed to be a 5-stage pipeline microprocessor. Due to the time limitation, branch prediction is 'not taken' static branch prediction. The microarchitecture of the processors was in-order and scalar, which was simpler to design and construct. For System on Chip peripheral part, Cache system, GPIO, JTAG/UART, VGA Driver, Ethernet Driver, PS/2 Driver, USB Driver were all included. However, only cache system has been almost completed without tests because there is no enough time when SRAM to DDR component provided by Digilent could not be applied directly. Cache system were a 3-stage pipeline structure to enable the cache to have higher performance. It also included request solvers that helped the cache send requests to the Memory management Unit. The communication between Cache system is designed by AXI4 Bus, which is a protocol of

	data transfer created by ARM. There were two AXI4 Masters, one AXI interconnect, and one AXI4 Slave. Memory Management Unit was designed by Memory Interface Generator provided by Xilinx, and it would automatically generate the AXI4 slave interface for communication. Finally, to integrate the operating system into the SoC, student try to directly load the machine code od operating system into the memory that was constructed by static RAM on the board, but the size of such RAM could not afford the size of operating system. In addition, due to no GPIO, there was no way to load the operating system into memory. Except for designing hardware, there were also much discussion between students. For example, the choice of instruction set architecture or system call definition were both the topics in the discussion.
Yufeng Chen	**Generally:** Design bootloader (U-boot). Design poster; Write blog; Write report; Introduce the U-boot part in Bench Inspection; **Explanation**: 1. Read the book <Computer organization and design> and understand the basic structure of Risc-V. 2. Participate in the completion of Sustainable Development and Ethics report and the final project report. 3. Participate in the construction of the blog, the design and presentation of the poster and the progress of the project on GitHub. 4. Research on the part about U-Boot in Risc-V, learn the language compiled by U-Boot and try to compile the complete

	and usable U-Boot code. In addition, learn the basic knowledge about CPU and Linux system from it and put forward good suggestions during the development of the project.

Appendix C

Table 1: xv6 kernel source file

File	Description
bio.c	Disk block cache for the file system.
console.c	Connect to the user keyboard and screen.
entry.S	Very first boot instructions.
exec.c	exec system call.
file.c	File descriptor support.
fs.c	File system. (not finished)
kalloc.c	Physical page allocator.
kernelvec.S	Handle traps from kernel, and timer interrupts.
log.c	File system logging and crash recovery.
main.c	Control initialization of other modules during boot.
pipe.c	Pipes.
plic.c	RISC-V interrupt controller.
printf.c	Formatted output to the console.
proc.c	Processes and scheduling.
sleeplock.c	Locks that yield the CPU.
start.c	Early boot code.
string.c	C string and byte-array library.
syscall.c	Dispatch system calls to handling function.
sysfile.c	File-related system calls.
sysproc.c	Process-related system calls.
trampoline.S	Assembly code to switch between user and kernel.
trap.c	C code to handle and return from traps and interrupts.
uart.c	Serial-port console device driver.
vm.c	Manage page tables and address spaces.

Appendix D

Table 2: xv6 system calls

System call	Description
fork()	create a new process (child process)
exit(status)	Terminate the current process. status: If this is 0 or EXIT_SUCCESS, it indicates success. If it is EXIT_FAILURE, it indicates failure.
wait(*status)	Wait for a child process to exit and copy the child's exit status to xstatus
kill(pid)	Terminate process indicated by pid
getpid()	Get process identification
sleep(n)	Suspend all processes for n/10 second
exec(filename,*argv)	Load a file and execute it. filename: filename. *argv: argument list
sbrk(n)	Grow process's memory by n bytes
open(filename,flag)	Open a file flag: read or write or read/write
read(fd, buf, n)	Read n bytes from an open file into buf
write(fd,buf,n)	Write n bytes to an open file
close(fd)	Release the file descriptor of an open file
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mkdir(dirname)	Create a new directory
mknod(name,major,minor)	Create a device file
fstat(fd)	Return info about an open file
link(f1,f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

Appendix E

Description of SoC component Input/Output (Verilog) in this project

1. CoreTop.v

Input/output name	**Input/output function**
CLK (input)	The clock of the processor
RESET (input)	Reset the processor (all register goes to zaro)
DATA [31:0] (output)	Used for debug
ADDRESS [31:0] (output)	Used for debug

2. InstructionMemory.v (It is used temporarily and will be replaced by InstructionCache.v)

Input/output name	**Input/output function**
A [5:0] (input)	Address of instruction
RD [31:0] (output)	Instruction read from memory

3. IF_ID_Register.v

Input/output name	**Input/output function**
PC [31:0] (input)	Program Counter
CLK (input)	The clock of the processor
RESET (input)	Register goes to zero
Instr [31:0] (input)	The instruction read from memory/cache
flush (input)	Used when the pipeline needs to stall
PC_o [31:0] (output)	Output of program counter
Instr_o [31:0] (output)	Output of instruction

4. ControlUnit.v

Input/output name	**Input/output function**

OP [6:0] (input)	Opcode of instruction (Instruction[6:0])
Funct3 [2:0] (input)	Funct3 number of instructions
Funct7 [6:0] (input)	Fucnt7 number of instruction
EX_control [20:0] (output)	Use to control execution stage
MEM_control [6:0] (output)	Use to control Memory operand fetch
WB_control [3:0] (output)	Use to control Write Back stage

5. RegisterFile.v

Input/output name	**Input/output function**
A1 [4:0] (input)	Register number in rs1
A2 [4:0] (input)	Register number in rs2
A3 [4:0] (input)	Register number needed to write back
WD [31:0] (input)	Write Data
RegWrite (input)	Write Enable
CLK (input)	The clock of the processor
RESET (input)	Register goes to zero
RD1 [31:0] (output)	Data in rs1
RD2 [31:0] (output)	Data in rs2

5. ID_EX_Register.v

Input/output name	**Input/output function**
CLK (input)	The clock of the processor
RESET (input)	Register goes to zero
Enable (input)	Write Enable (pipeline stall)
flush (input)	Register goes to zero (branch target hit)
SrcA_i [31:0] (input)	Data in rs1 input
SrcB_i [31:0] (input)	Data in rs2 input
EX_control_i [20:0] (input)	Execution control input
MEM_control_i [6:0] (input)	Memory control input

WB_control_i [3:0] (input)	Write Back control input
U_type_immediate_i [31:0] (input)	U-type instruction immediate value input
J_type_immediate_i [31:0] (input)	J-type instruction immediate value input
I_type_immediate_i [31:0] (input)	I-type instruction immediate value input
B_type_immediate_i [31:0] (input)	B-type instruction immediate value input
S_type_immediate_i [31:0] (input)	S-type instruction immediate value input
RegDst_i [4:0] (input)	Register destination (rd) input
PC_i [31:0] (input)	Program counter input
ALUSrcB_S_type_i (input)	Control SrcB if the instruction type is S-type
RegisterRs1_i [4:0] (input)	Register rs1 (use for forwarding)
RegisterRs2_i [4:0] (input)	Register rs2 (use for forwarding)
EX_control [20:0] (output)	Execution control output
MEM_control [6:0] (output)	Memory control output
WB_control [3:0] (output)	Write Back control output
U_type_immediate [31:0] (output)	U-type instruction immediate value output
J_type_immediate [31:0] (output)	J-type instruction immediate value output
I_type_immediate [31:0] (output)	I-type instruction immediate value output
RegDst [4:0] (output)	Register destination (rd) output
PC [31:0] (output)	Program counter input
SrcA [31:0] (output)	Data in rs1 input
SrcB [31:0] (output)	Data in rs2 input
B_type_immediate [31:0] (output)	B-type instruction immediate value input
S_type_immediate [31:0] (output)	S-type instruction immediate value input
RegisterRs1 [4:0] (output)	Register rs1 (use for forwarding)
RegisterRs2 [4:0] (output)	Register rs2 (use for forwarding)
ALUSrcB_S_type (output)	Control SrcB if the instruction type is S-

	type

6. ALUControl.v

Input/output name	**Input/output function**
Aluop [6:0]	Opcode of instruction (Instruction[6:0])
Funct3 [2:0]	Funct3 number of instructions
Funct7 [6:0]	Fucnt7 number of instruction
ControlResult [10:0]	Alu control logic

7. ALU.v

Input/output name	**Input/output function**
a [31:0] (input)	Input 1
b [31:0] (input)	Input 2
aluop [10:0] (input)	ALU control logic
result [31:0] (output)	Calculation result
branchCmp (output)	Branch comparison result
zero_division (output)	Divide zero
overflow_signed_div (output)	Division overflow

8. EX_MEM_Register.v

Input/output name	**Input/output function**
CLK (input)	The clock of the processor
RESET (input)	Register goes to zero
MEM_control_i [6:0] (input)	Memory control input
WB_control_i [3:0] (input)	Write Back control input
ALUResult_i [31:0] (input)	ALU calculation result
StoreData_i [31:0] (input)	Data needed to be stored in memory (s-type instruction)
branchCmp (input)	Branch comparison result

zero_division (input)	Divide zero
overflow_signed_div (input)	Division overflow
RegDst_i [4:0] (input)	Register destination (rd) input
PC_i [31:0] (input)	Program counter input
BranchTargetAddress_i [31:0] (input)	Branch Target Address
WB_control [3:0] (output)	Write Back control output
ALUResult [31:0] (output)	ALU calculation result
StoreData [31:0] (output)	Data needed to be stored in memory (s-type instruction)
branchCmp (output)	Branch comparison result
zero_division (output)	Divide zero
overflow_signed_div (output)	Division overflow
RegDst [4:0] (output)	Register destination (rd) output
PC [31:0] (output)	Program counter output
MEM_control [6:0] (output)	Memory control output
BranchTargetAddress [31:0] (output)	Branch Target Address

9. DataMemory.v (It is used temporarily and will be replaced by DataCache.v)

Input/output name	**Input/output function**
CLK (input)	The clock of the processor
A	Address of reading memory
WE	Write Enable
WD	Write Data
RD	Read Data

10. MEM_WB_Register.v

Input/output name	Input/output function
CLK (input)	The clock of the processor
RESET (input)	Register goes to zero

WB_control_i [3:0] (input)	Write Back control input
ReadData_i [31:0] (input)	Data read from memory
ALUResult_i [31:0] (input)	ALU calculation result
PC_i [31:0] (input)	Program Counter of current instruction
WB_control [3:0] (output)	Write Back Control logic
RegDst [4:0] (output)	Register Destination
ReadData [31:0] (output)	Data read from memory output
ALUResult [31:0] (output)	ALU calculation result output
PC [31:0] (output)	Program Counter of current instruction output
RegDst_i [4:0] (input)	Register Destination output

11. HazardUnit.v

Input/output name	**Input/output function**
RegSrcA_ID_EX [4:0] (input)	Register rs1 in execution stage
RegSrcB_ID_EX [4:0] (input)	Register rs2 in execution stage
RegDst_EX_MEM [4:0] (input)	Register Destination in memory operation stage
RegDst_MEM_WB (input)	Register Destination in write back stage
RegWrite_EX_MEM (input)	Register Write control in memory operation stage
RegWrite_MEM_WB (input)	Register Write control in write back stage
ALUSrcA (input)	ALU Source A control logic
ALUSrcB [1:0] (input)	ALU Source B control logic
MemRead (input)	Determine whether the instruction in memory operation stage needs to read memory
RegDst_ID_EX [4:0] (input)	Register Destination in execution stage
RegSrcA_IF_ID [4:0] (input)	Register rs1 in decode stage

RegSrcB_IF_ID [4:0] (input)	Register rs2 in decode stage
ForwardA [1:0] (output)	Determine whether ALU Source A needs data forwarding
ForwardB [1:0] (output)	Determine whether ALU Source B needs data forwarding
WriteEnable_IF_ID (output)	Control the Write Enable of IF/ID register
ControlLogicStall (output)	Make control logic input zero
PCWriteEnable (output)	Control the PC to be next PC value or to stall

12. Cache System (Block Design)

Input/output name	**Input/output function**
w_valid_d (input)	CPU request write data into memory
r_valid_d (input)	CPU request read data from memory
w_data_d [31:0] (input)	The data that needed to be written
CLK (input)	The Cache Clock (three times as Processor clk)
r_addr_i [31:0] (input)	The address of which CPU wants to read instruction
FIFO_empty_i (input)	The empty port of FIFO between CPU and instruction cache
addr_d [31:0] (input)	The address of which CPU wants to read or write data
FIFO_empty_d (input)	The empty port of FIFO between CPU and data cache
reset (input) (input)	Reset cache
r_valid_i (input)	CPU request read instruction from memory

rsp_d [1:0] (output)	Response to CPU from data cache
r_data_d [31:0] (output)	Data read from memory
rsp_valid_d (output)	Response valid port from data cache
r_data_i [31:0] (output)	Instruction read from memory
r_rsp_i [1:0] (output)	Response to CPU from instruction cache
rsp_valid_i (output)	Response valid port from instruction cache
DDR2 (output)	DDR2 port connection to physical memory