I/O Device

Zihao Yu

Institute of Computing Technology
Chinese Academy of Sciences

Introduction

CPU is a circuit which can only execute instructions

- How to let CPU display `Hello, World!`?
- How to let CPU know which key you have pressed?

Introduction

CPU is a circuit which can only execute instructions

- How to let CPU display `Hello, World!`?
- How to let CPU know which key you have pressed?

- We need I/O device!

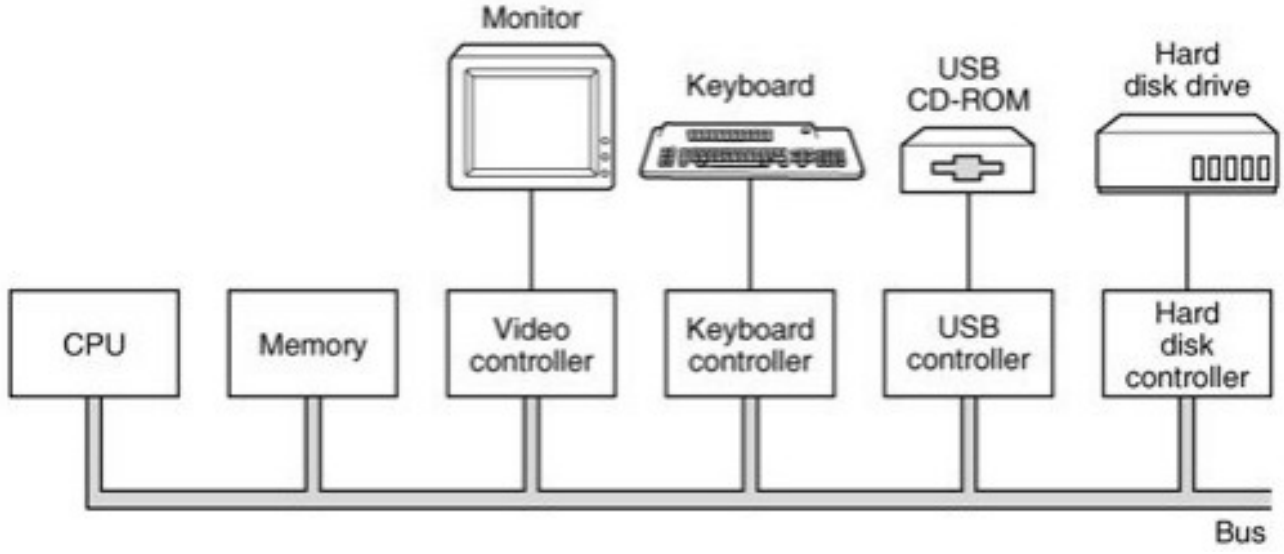
Computer users do not control the computer by sending binary instructions to the CPU

- Instead, control the computer by I/O device
  - Mouse, keyboard, screen, …

I/O Device

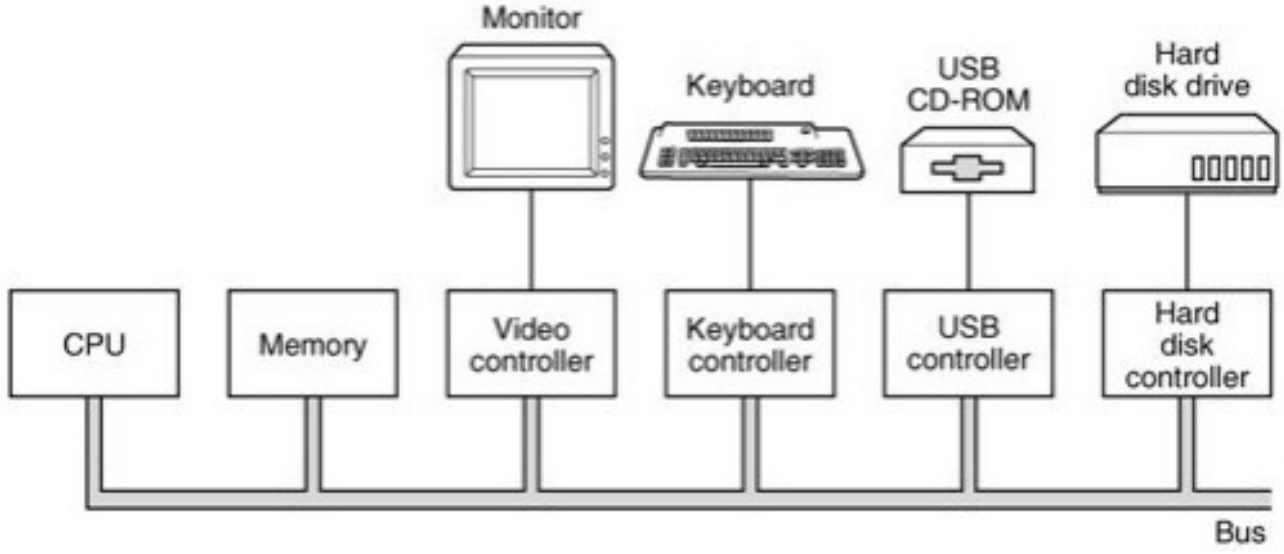
Device Controller

A digital module which controls the actual device to work



Device Controller

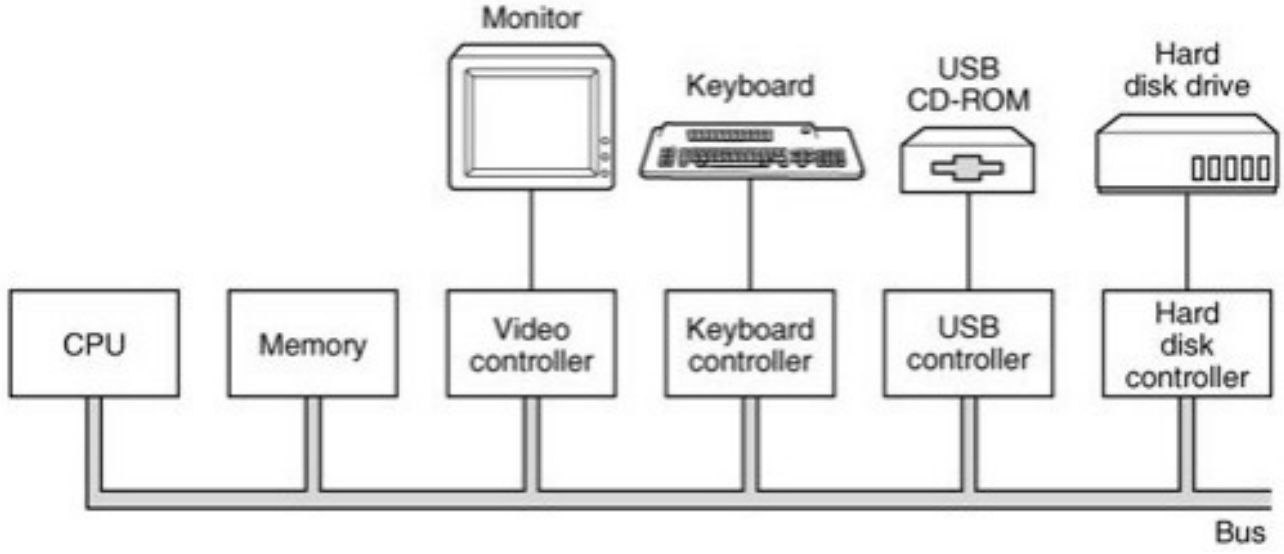
A digital module which controls the actual device to work



- Connect CPU by bus from one side
  - Receive commands from CPU

Device Controller

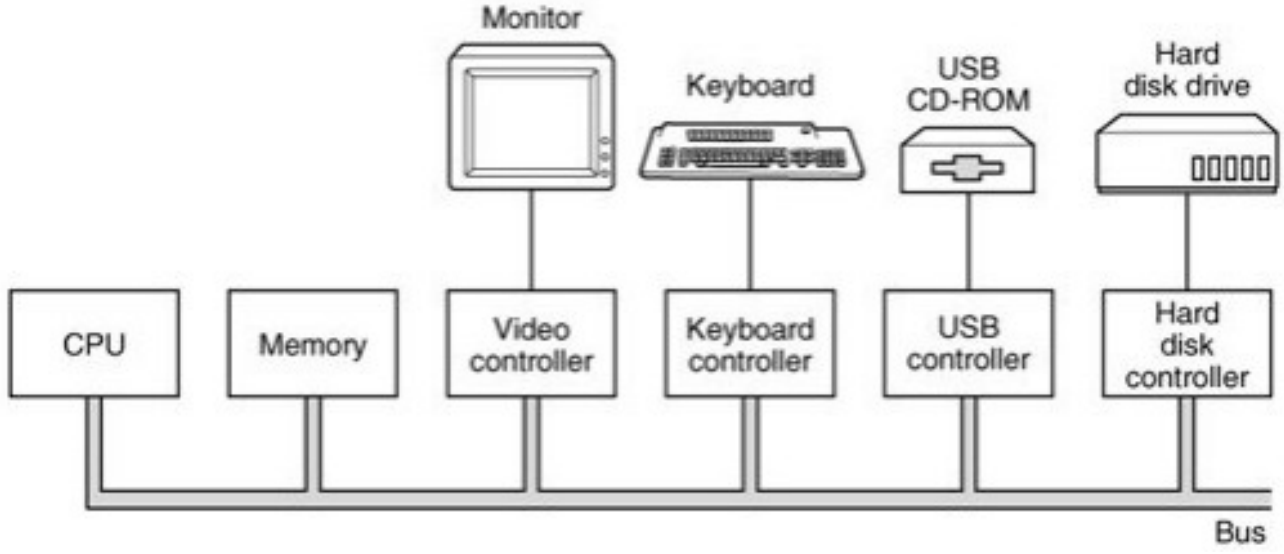
A digital module which controls the actual device to work



- Connect CPU by bus from one side
  - Receive commands from CPU
- Connect the actual device from the other side
  - Send signals to the actual device

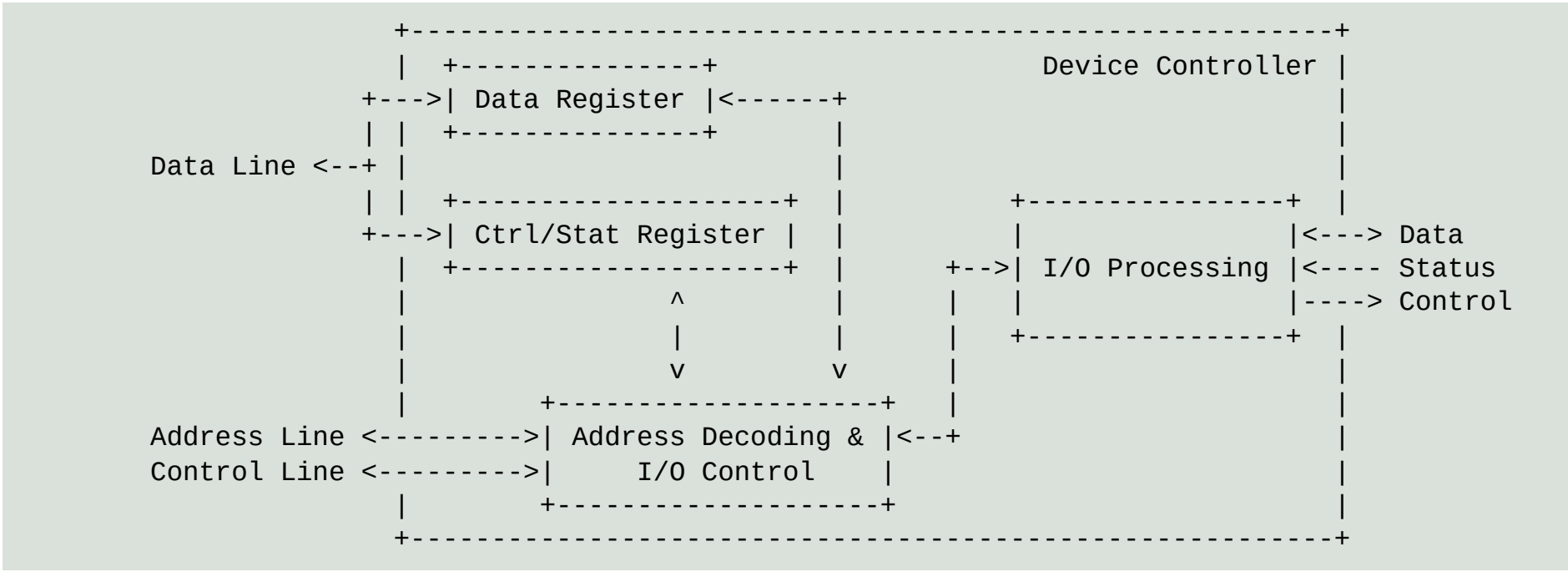
Device Controller

A digital module which controls the actual device to work



- Connect CPU by bus from one side
  - Receive commands from CPU
- Connect the actual device from the other side
  - Send signals to the actual device
- Translate the commands from CPU to signals

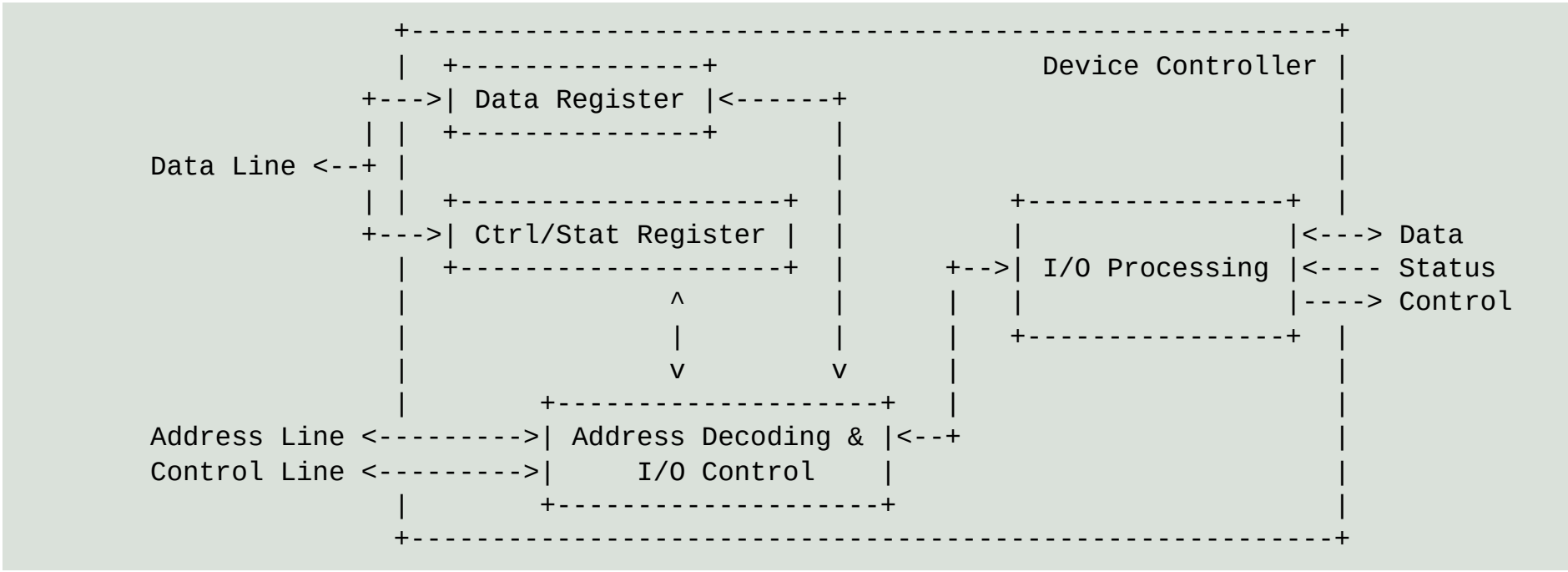
Inside Device Controller



3 important kinds of device registers visible to CPU:

- data exchange - data register
- command & control - control register
- status detecting - status register

Inside Device Controller



3 important kinds of device registers visible to CPU:

- data exchange - data register
- command & control - control register
- status detecting - status register

Other parts are invisible to CPU

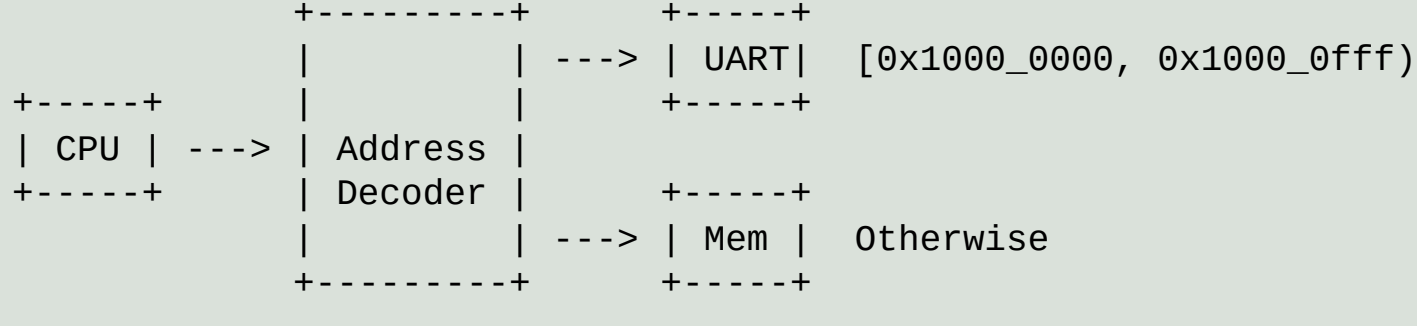
- Abstraction! CPU controls device by accessing device registers

Device Programming

Memory-Mapped I/O (MMIO)

To make device registers accessible by CPU, there should be referred in some way

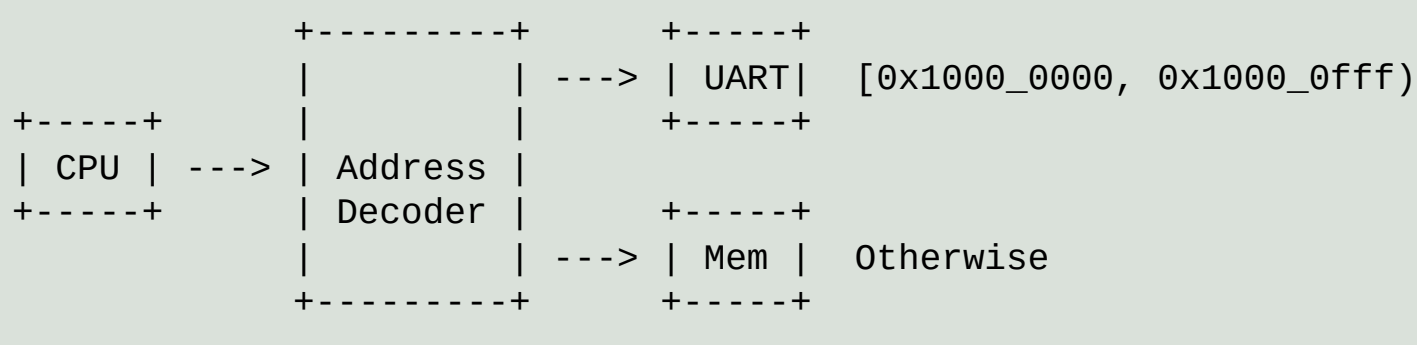
- Memory-Mapped I/O
  - map device registers to some memory addresses



Memory-Mapped I/O (MMIO)

To make device registers accessible by CPU, there should be referred in some way

- Memory-Mapped I/O
  - map device registers to some memory addresses



When CPU issues a memory request, the address decoder will map the request to the corresponding device according to the memory address

- The address decoder inside the device will further map the request to the device registers

The Function of Device Registers

Read the friendly manual

- Example: Xilinx UART 16550 v2.0 LogiCORE IP Product Guide

Table 2-4: **Register Address Map**

LCR(7)	Address Offset	Register Name	Access Type	Description
0	0x1000	RBR	RO	Receiver Buffer Register
0	0x1000	THR	WO	Transmitter Holding Register
0	0x1004	IER	R/W	Interrupt Enable Register
x	0x1008	IIR	RO	Interrupt Identification Register
x	0x1008	FCR	WO	FIFO Control Register
1	0x1008	FCR	RO	FIFO Control Register
x	0x100C	LCR	R/W	Line Control Register
x	0x1010	MCR	R/W	Modem Control Register
x	0x1014	LSR	R/W	Line Status Register
x	0x1018	MSR	R/W	Modem Status Register
x	0x101C	SCR	R/W	Scratch Register
1	0x1000	DLL	R/W	Divisor Latch (Least Significant Byte) Register
1	0x1004	DLM	R/W	Divisor Latch (Most Significant Byte) Register

Device Programming

Program with assembly instructions

```
lui t0, 0x10000    # t0 = 0x10000000
addi t1, zero, 65  # 65 = the code of 'A'
sb t1, 0(t0)
```

Program with C code

```
void putch(char c) {
  char *p = (char *)0x10000000ul;
  *p = c;
}
```

Device Programming(2)

TASK 1 - implement a simple behavior model for UART

- Insert a condition statement when accessing memory

```
extern "C" void mem_write(unsigned addr, unsigned wdata, unsigned char wmask)
{
  if (addr == 0x10000000) {  // write to UART
    fputc(wdata & 0xff, stderr);   // in stdio.h
    return;
  }
  // write to memory array
}
```

- Implement `putch()` in `abstract-machine/am/src/riscv/npc/trm.c`
- Run hello program
  - For co-simulation, you should recognize the store instruction writing to UART, and skip it for the reference model

```
cd am-kernels/kernels/hello
make ARCH=minirv-npc run
sudo apt install libsdl2-dev
make ARCH=native run   # run on native for the reference result
```

Run More Programs

With UART, we can let program output something

TASK 2 - run more programs

- Download `klib-minirv-npc.a` and put it under `abstract-machine/klib/build/`
  - Overwrite the original file if exists
- Try to run `dhrystone`, `coremark` and `microbench`
  - Under `am-kernels/benchmarks`
  - You may run on `ARCH=native` first for the reference result
  - Since now there is no way to measure time on the CPU, just ignore the score displayed
  - Run `microbench` with `mainargs=train` to test more instructions: `make ARCH=minirv-npc run mainargs=train`

Run More Programs(2)

TASK 3 (optional) - run LLaMa

- Clone the repo

```
git clone -b dev https://github.com/OSCPU/archbench
```

- Compile and run LLaMa

```
cd archbench/bench/202.llama
make ARCH=minirv-npc run mainargs=test
make ARCH=minirv-npc run mainargs=train
```

- For `mainargs=test`, it will just output `Once Once` due to low accuracy of the small model
- For `mainargs=train`, it will output `Once upon a time, there was a little girl named Lily.`
  - It may cost about 5 minutes to finish

Be Careful with C Code

Poll the Status of UART

Sometimes device is not ready to accept commands

- We should first check the status to wait before it is ready

```
#define BUSY 0
void putch(char c) {
  char *data = (char *)0x10000000ul;
  char *status = (char *)0x10000004ul;
  while (*status == BUSY); // wait until ready
  *data = c;
}
```

Poll the Status of UART

Sometimes device is not ready to accept commands

- We should first check the status to wait before it is ready

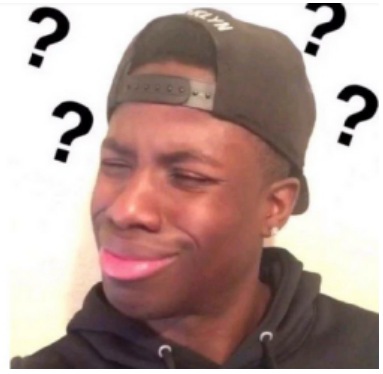
```c
#define BUSY 0
void putch(char c) {
  char *data = (char *)0x10000000ul;
  char *status = (char *)0x10000004ul;
  while (*status == BUSY); // wait until ready
  *data = c;
}
```

```
riscv64-linux-gnu-gcc -march=rv32g -mabi=ilp32 -O2 -c a.c
riscv64-linux-gnu-objdump -d a.o
```

```
00000000 <putch>:
   0:   100007b7   lui     a5,0x10000
   4:   0007c703   lbu     a4,4(a5) # 10000004
   8:   00071463   bnez    a4,10 <.L5>

0000000c <.L4>:
   c:   0000006f   j       c <.L4>

00000010 <.L5>:
  10:   00078223   sb      a0,0(a5)
  14:   00008067   ret
```



What Happened?

Compiler considers that `data` and `status` are normal pointers, which point to normal memory data

```
#define BUSY 0
void putch(char c) {
  char *data = (char *)0x10000000ul;
  char *status = (char *)0x10000004ul;
  while (*status == BUSY);
  *data = c;
}
```

- Inside `while`, the data pointed by `status` seems unchanged
  - `*status == BUSY` keeps the same every time it is evaluated

What Happened?

Compiler considers that `data` and `status` are normal pointers, which point to normal memory data

```
#define BUSY 0
void putch(char c) {
  char *data = (char *)0x10000000ul;
  char *status = (char *)0x10000004ul;
  while (*status == BUSY);
  *data = c;
}
```

- Inside `while`, the data pointed by `status` seems unchanged
  - `*status == BUSY` keeps the same every time it is evaluated

Compiler decides to move the evaluation of `*status == BUSY` to the outside of `while`, to reduce the number of evaluation

```
#define BUSY 0
void putch(char c) {
  char *data = (char *)0x10000000ul;
  char *status = (char *)0x10000004ul;
  int is_busy = (*status == BUSY);
  while (is_busy);
  *data = c;
}
```

Be Careful with Compiler Optimization

Fact - `status` is a pointer pointing to a device register

- Its value will change, even if the program does not write to it!
  - That is, `*status == BUSY` may yield different results every time it is evaluated
- But the compiler does not know about this fact

Be Careful with Compiler Optimization

Fact - `status` is a pointer pointing to a device register

- Its value will change, even if the program does not write to it!
  - That is, `*status == BUSY` may yield different results every time it is evaluated
- But the compiler does not know about this fact

Solution - use keyword `volatile` to tell compile

- `status` is pointing to a data which may change itself
- Therefore, the data accessing should be strictly performed
  - Do not optimize it!

```
        volatile char *status
// ...
        volatile char *data
// ...
```

```
        00000000 <uart_putch>:
           0:   10000737    lui     a4,0x10000
           4:   00074783    lbu     a5,4(a4) #
10000004
           8:   fe078ee3    beqz    a5,4
           c:   00070223    sb      a0,0(a4)
          10:   00008067    ret
```

Try to Poll the Status

TASK 4 - poll the status of UART before sending a character

- Add a status register for UART

```
extern "C" int mem_read(int addr) {
  if (addr == 0x10000004) {  // read UART status
    return (rand() & 0x7) == 0 ? 0 : 1; // ready with probability 12.5%
  }
  // read from memory array
}
```

- Modify `putch()` to wait for ready before sending a character
  - For co-simulation, you should recognize the load instruction to read from UART, and copy the read result from RTL to the reference model
    - This guraruntees that after the execution of the load instruction, the states of RTL and reference model are still the same

I/O Extension in AM

I/O Extension

There are different devices in different platforms

- The ways to access them are also different
  - The function of device registers
  - The address of device registers

I/O Extension

There are different devices in different platforms

- The ways to access them are also different
  - The function of device registers
  - The address of device registers

Add an abstraction layer: IOE (I/O Extension)

- provide virtual device registers
- provide 3 APIs

```
void ioe_read(int reg, void *buf);
void ioe_write(int reg, void *buf);
bool ioe_init();
```

Virtual Device Registers in IOE

```
// abstract-machine/am/include/amdev.h
AM_DEVREG( 1, UART_CONFIG,  RD, bool present);
AM_DEVREG( 2, UART_TX,      WR, char data);
AM_DEVREG( 3, UART_RX,      RD, char data);
AM_DEVREG( 4, TIMER_CONFIG, RD, bool present, has_rtc);
AM_DEVREG( 5, TIMER_RTC,    RD, int year, month, day, hour, minute, second);
AM_DEVREG( 6, TIMER_UPTIME, RD, uint64_t us);
AM_DEVREG( 7, INPUT_CONFIG, RD, bool present);
AM_DEVREG( 8, INPUT_KEYBRD, RD, bool keydown; int keycode);
AM_DEVREG( 9, GPU_CONFIG,   RD, bool present, has_accel; int width, height, vmemsz);
AM_DEVREG(10, GPU_STATUS,   RD, bool ready);
AM_DEVREG(11, GPU_FBDRAW,   WR, int x, y; void *pixels; int w, h; bool sync);
AM_DEVREG(12, GPU_MEMCPY,   WR, uint32_t dest; void *src; int size);
AM_DEVREG(13, GPU_RENDER,   WR, uint32_t root);
AM_DEVREG(14, AUDIO_CONFIG, RD, bool present; int bufsize);
AM_DEVREG(15, AUDIO_CTRL,   WR, int freq, channels, samples);
AM_DEVREG(16, AUDIO_STATUS, RD, int count);
AM_DEVREG(17, AUDIO_PLAY,   WR, Area buf);
AM_DEVREG(18, DISK_CONFIG,  RD, bool present; int blksz, blkcnt);
AM_DEVREG(19, DISK_STATUS,  RD, bool ready);
AM_DEVREG(20, DISK_BLKIO,   WR, bool write; void *buf; int blkno, blkcnt);
AM_DEVREG(21, NET_CONFIG,   RD, bool present);
AM_DEVREG(22, NET_STATUS,   RD, int rx_len, tx_len);
AM_DEVREG(23, NET_TX,       WR, Area buf);
AM_DEVREG(24, NET_RX,       WR, Area buf);
```

Programs read the timer by `ioe_read(TIMER_UPTIME, &us);`

- They do not care about how to implement the timer

Add Timer

TASK 5 - add timer

- Implement a simple behavior model for timer

```
unsigned long long get_time() {
  // Return the number of microsecond after simulation.
  // Can be implemented by gettimeofday().
}
extern "C" int mem_read(int addr) {
  if (addr == 0x10000004) { /* ... */ } // read UART status
  // read the low 32 bits of timer
  else if (addr == 0x20000000) { return get_time() & 0xffffffff; }
  // read the high 32 bits of timer
  } else if (addr == 0x20000004) { return get_time() >> 32; }
  // read from memory array
}
```

- Implement `__am_timer_uptime()` by accessing the timer model in `abstract-machine/am/src/riscv/npc/timer.c`
- Run test for timer

```
cd am-kernels/tests/am-tests
make ARCH=minirv-npc run mainargs=t
make ARCH=native run mainargs=t  # run on native for the reference result
```

Add Timer(2)

TASK 6 - run benchmarks again

- Now they should report a meaningful score

It is time to run Mario!

TASK 7 - run Mario

- Clone the repo

```
git clone https://github.com/NJU-ProjectN/fceux-am
```

- Download the ROM file for Mario, and put it under `fceux-am/nes/rom/` with file name `mario.nes`
- Modify the configuration

```
--- fceux-am/src/config.h
+++ fceux-am/src/config.h
@@ -1,5 +1,5 @@
 #ifndef __CONFIG_H__
 #define __CONFIG_H__

-#define HAS_GUI
+//#define HAS_GUI
 #define SIZE_OPT
```

- Compile and run
  - can not play because no keyboard is connected to CPU now

```
cd fceux-am
make ARCH=minirv-npc run
```

END