C & RV32I CPU with Peripherals

Mario Werner and Robert Schilling

November 14, 2019

Computer Organization and Networks
Graz University of Technology

C Support

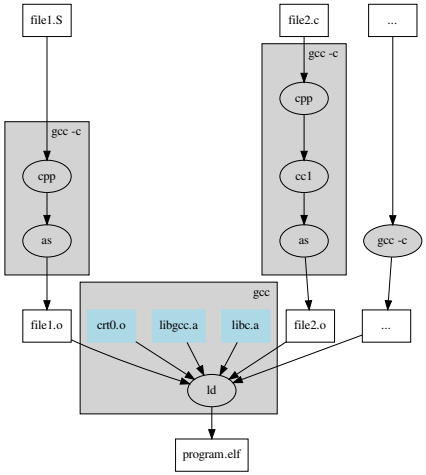
Motivation for Better Tools

- Writing large assembler programs is cumbersome.
- Keeping ABI for complex types is quite hard.
  Spec.: `https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md`
- Manual stack organization is getting complex.
- Portability of assembler code is limited. (RV32I vs RV32G)

$\rightarrow$ Use a higher level language like C.

Introduction of the New Tools

- Replacing `riscvasm.py`
  - `cpp` (preprocessor)
  - `cc1` (C compiler)
  - GNU `as` (assembler)
  - GNU `ld` (linker)
- `gcc` as unified driver for the build tools
- `qemu` as replacement for `riscvsim.py`
- Other utilities: `objcopy`, `objdump`, `ar`, ...



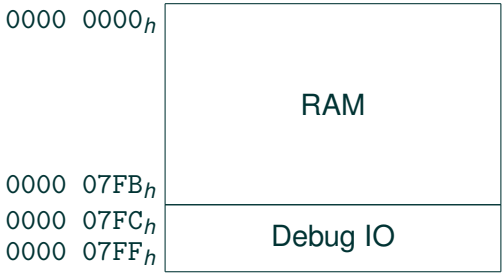
About the Tools

- `cpp`: The C Preprocessor. Generates a single flat file by processing #includes, macros, and #ifs. (`gcc -E`)
- `cc1`: Actual C frontend that translates C to assembler. (`gcc -c -S`)
- `as`: The GNU assembler. Translates assembler to object files. (`gcc -c`)
- `ld`: The GNU linker. Combines object files/libraries into a single binary. (`gcc`)
- `qemu`: Quick EMUlator that executes code via binary translation. (user-mode and full-system mode)

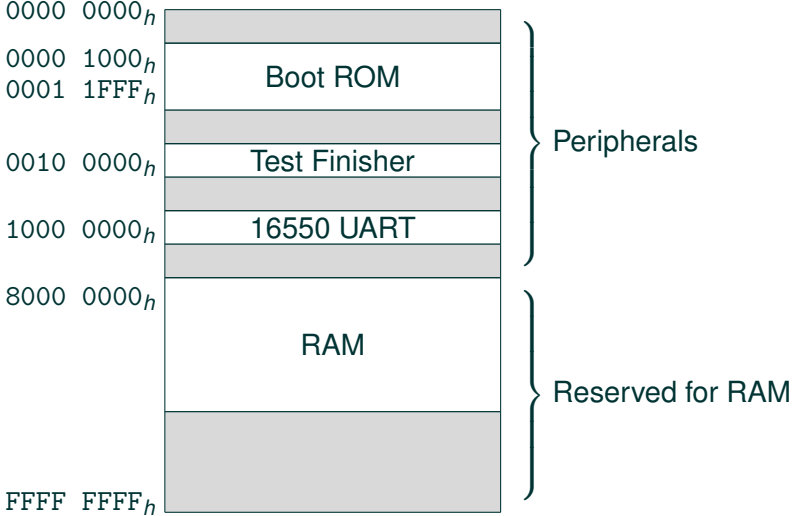
Gained Advantages

- Support for multiple files. (better code structure)
- Convenience features in ASM.
  - Pseudo instructions: `li`, `mov`, `call`, `ret`, ...
  - Local labels: `j 1f` (jump to the next label 1 in forward direction)
- Easy support for standard build tools. (e.g., `cmake`)
- Enables reuse of existing libraries. (e.g., newlib as C library)
- But requires full RV32I ISA $\rightarrow$ e.g., `qemu`

Micro RISC-V Memory Map (2 KiB Address Space)



qemu's Memory Map (Simplified virt Machine)



Minimal Baremetal C/Assembler Example

crt0.S and serial_putchar.S:

```
.equ __RAM, 0x80000000
.equ __RAM_LENGTH, 0x8000000
    .section .text.startup
    .global _start
    .type   _start, @function
_start:
    li sp, __RAM + __RAM_LENGTH
    li a0, 0
    li a1, 0
    call main   # <retval> = main(0, NULL);
    tail exit   # exit(<retval>);
    .size _start, .-_start

    .section .text
    .global serial_putchar
    .type     serial_putchar, @function
serial_putchar:             # void serial_putchar(char c) {
    li t0, 0x10000000       # volatile char* uart_base
1:  lbu t1, 5(t0)           #       = (char*)0x10000000;
    andi t1, t1, 0x20
    beqz t1, 1b             # while (!(*(uart_base+5)&0x20));
    sb a0,0(t0)             # *uart_base = c;
    ret                     # }
    .size serial_putchar, .-serial_putchar
```

serial_hello.c

```
void serial_putchar(char c); // implemented in assembler

void exit(int code) { // Exit qemu by writing to magic memory
    volatile int* const exit_device = (int* const)0x100000;
    if(code == 0)
        *exit_device = 0x5555;
    else
        *exit_device = (code << 16) | 0x3333;
    while (1);
}

int main(int argc, char** argv) {
    const char* msg = "Hello baremetal QEMU!\n";
    while(*msg) serial_putchar(*msg++);
    return 0;
}
```

Minimal Baremetal C/Assembler Example

- Ensure that the helper scripts are in the PATH:

```
$ export PATH=.../examples/chapter_06/bin:$PATH
```

- Compile, assemble, and link the source files:

```
$ qemu-newlib-nano-gcc -nostdlib -nostartfiles -Os -o hello crt0.S serial_hello.c serial_putchar.S
# Full CMD: riscv32-elf-gcc -specs /opt/riscv/riscv32-elf/lib/rv32i/ilp32/qemu-newlib-nano.specs \
#             -nostdlib -nostartfiles -Os -o hello crt0.S serial_hello.c serial_putchar.S
```

- Look at the size of the compiled program:

```
$ riscv32-elf-size hello
   text    data     bss     dec     hex filename
    176       0       0     176      b0 hello
```

- Disassemble the binary to inspect the code:

```
$ riscv32-elf-objdump -hSl hello > hello.lss
```

- Execute the program with qemu:

```
$ qemu-run hello
# Full CMD: qemu-system-riscv32 -machine virt -nographic -serial mon:stdio -kernel hello
Hello baremetal QEMU!
```

Observations/Remarks for Baremetal Example

- Only a single slide example to show the concept but too minimal in practice.
- Relies on implicit linker script of `ld` for code layout.
- `crt0.S` usually also performs the following tasks:
  - Initializes the global pointer `gp`.
  - Initializes the data values from ROM/FLASH if needed.
  - Clears the `.bss` section to null uninitialized global variables.
  - Calls global initialization functions (*i.e.*, `__libc_init_array`).
  - Registers global termination functions (*i.e.*, `__libc_fini_array`).

$\rightarrow$ `crt0.S` is quite generic and typically provided by the toolchain.

Observations/Remarks for Baremetal Example

- Additional support libraries are also neglected.
- On RV32I, the following program would not link in this mode:

```
int main(int argc, char** argv) { return argc*argc; }
```

```
$ qemu-newlib-nano-gcc -nostdlib -nostartfiles -Os -o test crt0.S mul_main.c
/opt/riscv/riscv32-elf/bin/ld: /tmp/ccHZmvHc.o: in function `.L6':
mul_main.c:(.text.startup+0x24): undefined reference to `__mulsi3'
```

→ Compiler support libraries (e.g., `libgcc`, `libcompiler-rt`) are needed.

- Having a C standard library, while not necessary, is a huge help in practice.

```
#include <stdio.h>
int main(int argc, char** argv) { printf("Hello %04x\n", argc); }
```

```
$ qemu-newlib-nano-gcc -Os -o hello printf_hello.c
$ qemu-run hello
Hello 0000
```

Be aware of the size overhead!

```
$ riscv32-elf-size hello
   text    data     bss     dec     hex filename
  11369     113     160   11642    2d7a hello
```

→ We provide all needed libs, tools, and configs in the VM and examples repo.

Code Layout

- Compiler/assembler emits multiple sections:
  - `.bss`: uninitialized global variables (have to be zeroed)
  - `.data`: global variables with initialization
  - `.text`: code
  - `.rodata`: read-only constants (e.g., string literals)
  - ...

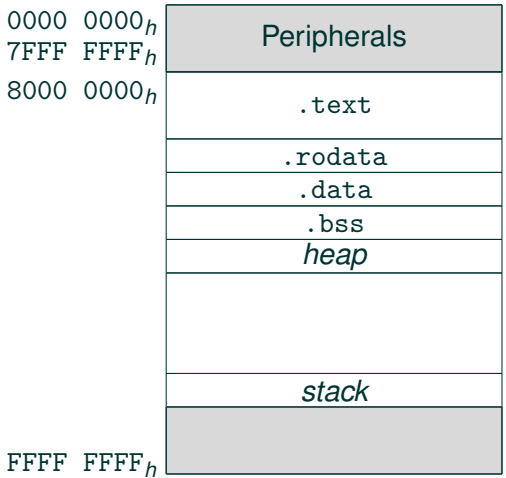
→ Global variables are never on the stack!

- Linker decides where to put the sections and patches the addresses during linking.
- Layout is customizable via linker scripts. (default: `ld --verbose`)

Common Code Layout

Properties:

- `.text`: executable, read-only
- `.rodata`: no-execute, read-only
- `.data`: no-execute, read+write
- `.bss`: no-execute, read+write, zero initialized
- *heap*: For dynamic memory. Grows towards top.
- *stack*: At top. Grows towards bottom.



RV32I vs. Micro RISC-V

- Micro RISC-V supports 19 instructions.
- The Full RV32I RISC-V instruction set (v2.2) has 47 instructions.
  Spec.: `https://riscv.org/specifications/`
- Adds support for ...
  - load/store/compare operations with `unsigned` datatype.
  - different memory access granularity (half word, bytes).
  - ALU instructions with immediate operands (e.g., `xori`).
  - Control and Status Registers (CSRs) incl. instructions.

$\rightarrow$ We need a better CPU.

RV32I vs. Micro RISC-V

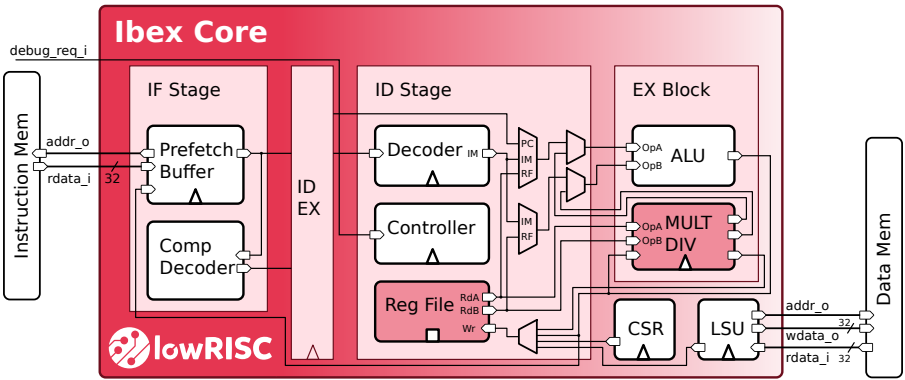
- We could add the missing instructions to Micro RISC-V.
- Instead, we use a different processor core, which “speaks” RISC-V.
- Reminder: The ISA is the contract between hardware and software.
- We use the Ibex core from lowRISC.

Ibex Core

Ibex RISC-V Core

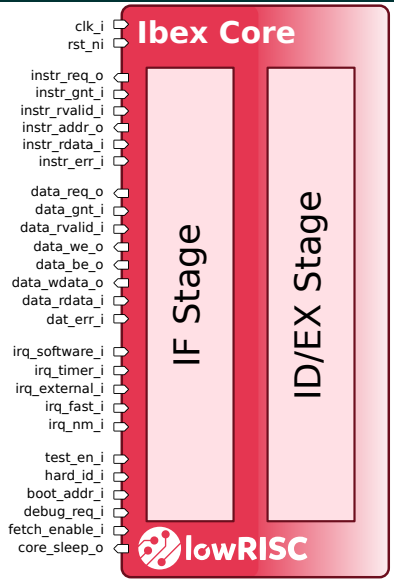
- 2-stage in-order 32-bit RISC-V processor.
- Designed to be small and efficient.
- Full support for RV32I (and more).
- Supports optional extensions (multiplication, compressed).
- Commercially used and ASIC proven.
- Extensively documented `https://ibex-core.readthedocs.io`.

Ibex RISC-V Core



Ibex RISC-V Core - What is new?

- Instruction interface.
- Data interface.
- Interrupt interface.
- Status and control ports.



Ibex RISC-V Core: Status and Control Ports

- Configurable core identifier.
- Configurable boot address.
  - Core starts +0x80 from this address.
- External debug interface (read registers, single-step mode, . . .).
- Sleep mode output $\rightarrow$ Ibex core waits to be waken up.
- Fetch enable input $\rightarrow$ core can be halted.

Ibex RISC-V Core: Interrupt Interface

- Support for different interrupt sources.
  - Non-maskable interrupts.
  - Timer interrupts.
  - External interrupts.
- All interrupt inputs are level-sensitive.
- Configuration via `mstatus`, `mie`, `mtvec`.
- Status in `mip` and/or within the interrupt source peripheral.
- More on the topic later.

Ibex RISC-V Core: Instruction and Data Interface

- Different instruction and data memory interface.
  - $\rightarrow$ Supports parallel fetch of instructions and data.
- What do we need?
  1. Address
  2. Data
  3. Meta data like the access type (read/write)
- Data is not always available $\rightarrow$ synchronization is needed.
- Handshake protocol between Ibex and memory.

Handshakes and Synchronization

Ibex Bus Protocol - Request Channel

- `req_o`: Request valid if high.
- `addr_o[31:0]`: Address, word aligned.
- `we_o`: Write Enable, high for writes, low for reads.
- `be_o[3:0]`: Byte Enable. Is set for the bytes to write/read.
- `wdata_o[31:0]`: Data to be written to memory.

- `gnt_i`: The other side accepted the request. Outputs may change in the next cycle.

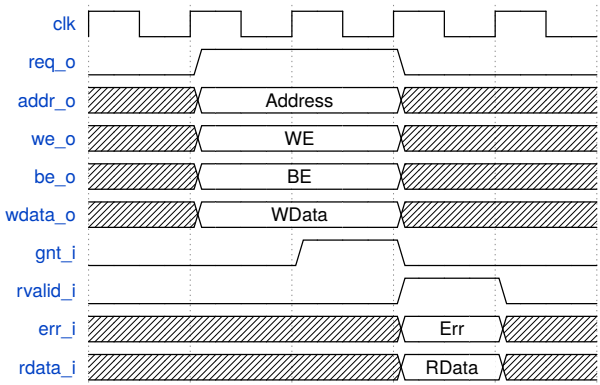
$\rightarrow$ Request transferred if `req_o` and `gnt_i` are high for one cycle.

Ibex Bus Protocol - Response Channel

- `rdata_i[31:0]`: Data read from memory for read requests.
- `err_i`: Error response from the bus or the memory: request cannot be handled. High in case of an error.
- `rvalid_i`: When `rvalid_i` is high, `err_i` and `rdata_i` hold valid data. This signal will be high for exactly one cycle per request.

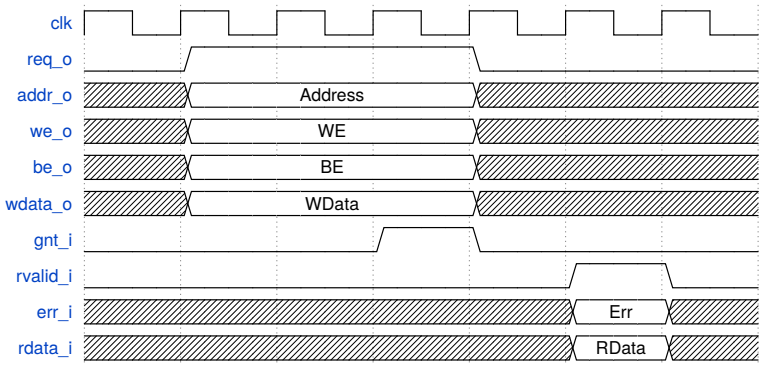
A simple Memory Transaction

1. Start the request by applying `req_o`, `addr_o`, `we_o`, `be_o`, `wdata_o`.
2. Receiver grants request when `gnt_i` goes high.
3. Response received later when `rvalid_i` is high.



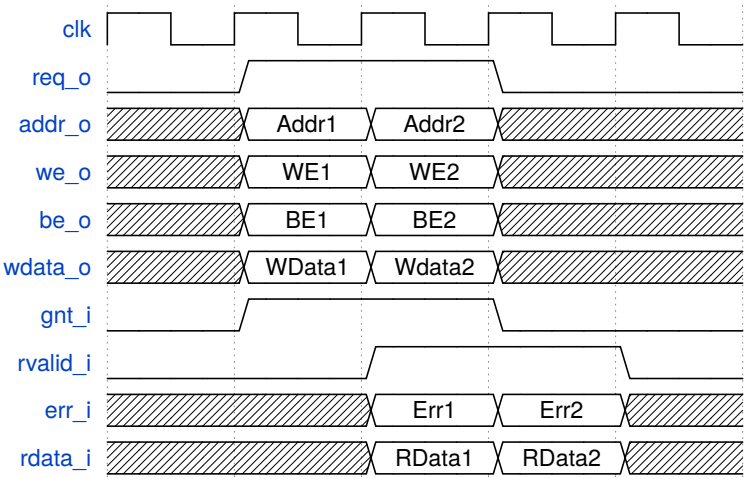
A *slow* Memory Transaction

- What if we have a slow memory?
- `gnt_i` is delayed.
- `rvalid_o` with data comes in a later cycle.



Back-to-Back Memory Transaction

- What if we want to transfer more data?
- Assert multiple requests in a burst.
- Responses are in request order.



Memory Transfer Example

- Memory accesses with different granularity
- Ibex only performs word-aligned accesses
- `be` denotes which bytes are written in the access

```
  .global main
  .type   main, @function
main:               # Define 3 constants we want to wrote
  LI t0, 0xa        # 8-bit value
  LI t1, 0xa8bc     # 16-bit value
  LI t2, 0xa3a8cb65 # 32-bit value
                    # Define 3 addresses
  LI a0, 0x101000   # 32-bit aligned address
  LI a1, 0x101001   # Unaligned byte-address
  LI a2, 0x101003   # Unaligned byte-address

  SB t0, 0(a0)      # Write byte to first address
  SH t1, 0(a1)      # Write half-word to second address
  SW t2, 0(a2)      # Write word to third address
                    # Read data back
  LB  t3, 0(a0)
  LHU t4, 0(a1)
  LW  t5, 0(a2)
  # Check if we read the same data
  SUB t0, t0, t3
  SUB t1, t1, t4
  SUB t2, t2, t5
  # Write result to DebugIO
  SW t0, 0x7fc(zero)
  SW t1, 0x7fc(zero)
  SW t2, 0x7fc(zero)

  RET
```

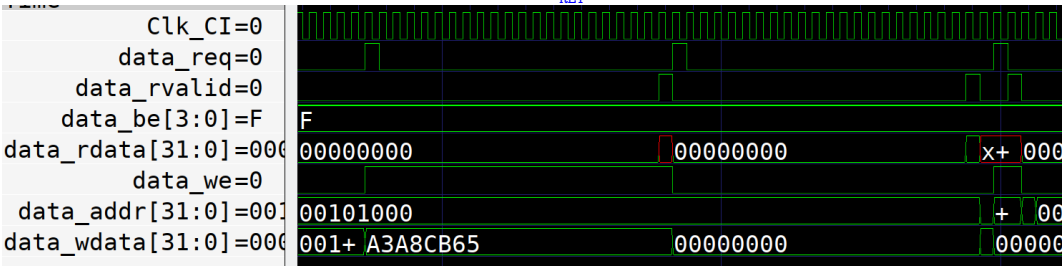
Slow Memory Transfer Example

- We have slow memory.
- `rvalid` is delayed by 20 clock cycles.

```
  .global main
  .type   main, @function
main:
  LI t0, 0xa3a8cb65  # 32-bit value
  LI a0, 0x101000    # 32-bit aligned address

  SW t0, 0(a0)       # Write byte to first address
  LW  t2, 0(a0)      # Read data back
  SUB t0, t0, t2     # Check if we read the same data
  SW t0, 0x7fc(zero) # Write result to DebugIO

  RET
```



Alternative Bus Protocols

- AXI / AMBA (ARM)
- CoreConnect (IBM)
- TileLink (Berkley/SiFive)
- Wishbone (OpenCores)
- Avalon (Intel/Altera)
- And a lot of custom protocols ...

Peripherals

Memory-Mapped Peripherals

- We want to extend the CPU with different functionality.
- Add peripherals on the memory bus of the CPU.
- Use memory instructions (load/store) to access peripherals.
- Peripherals can be mapped to any addresses.

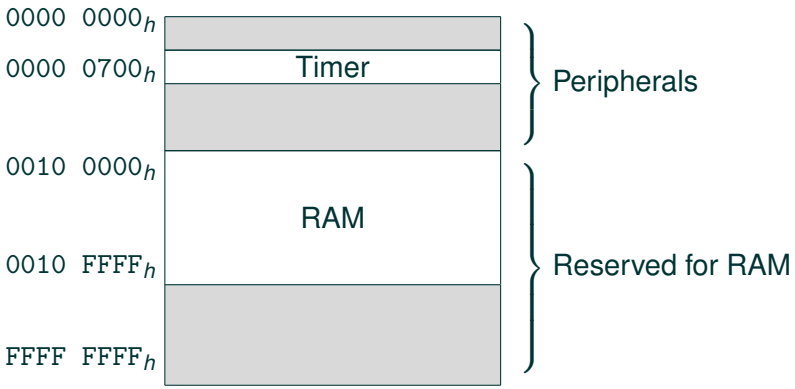
Timer

- Helps in solving the problem of
  - measuring timing differences.
  - performing periodic tasks.
- A timer is basically an up-counter with a known clock.
- Various extensions possible:
  - configurable clock input (counting events).
  - clock pre-scalers.
  - comparison register (PWM).
  - interrupts.

Our Timer for Ibex

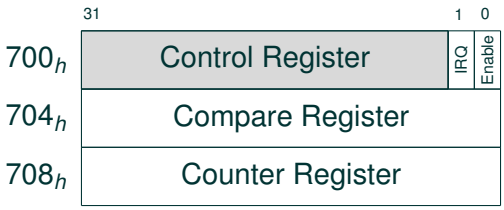
- Uses system clock for counting.
- 2-bit control register (enable/disable, IRQ pending).
- 32-bit compare register.
- 32-bit counter register.
- Triggers interrupt when counter reaches comparison value.
- Mapped to base address `0x700`.

Ibex Memory Map including Timer Peripheral



Our Timer for Ibex

Timer Memory Map:



`Enable`: [`RW`] Timer enable.

`IRQ`: [`R`] IRQ pending, [`W`] '1' clear IRQ.

Polling Example Program:

```
  .global main
  .type   main, @function
main:
  LI t1, 1
  SW t1, 0x700(x0)      # Enable timer
  LI t2, 0x100          # Timer compare value
loop:
  LW t1, 0x708(x0)      # Poll timer
  BLE t1, t2, loop      # Check if compare value reached

  SW t1, 0x7FC(x0)      # Write time to DebugIO
  RET                   # End main
```

Debug Input/Output

- Purely a debug peripheral.
- Identical to the one used in Micro RISC-V.
- Only supports reading and writing 32-bit integers.
- Immediate access to data without synchronization.
- Directly accesses files.

$\rightarrow$ Not possible inside a real chip.

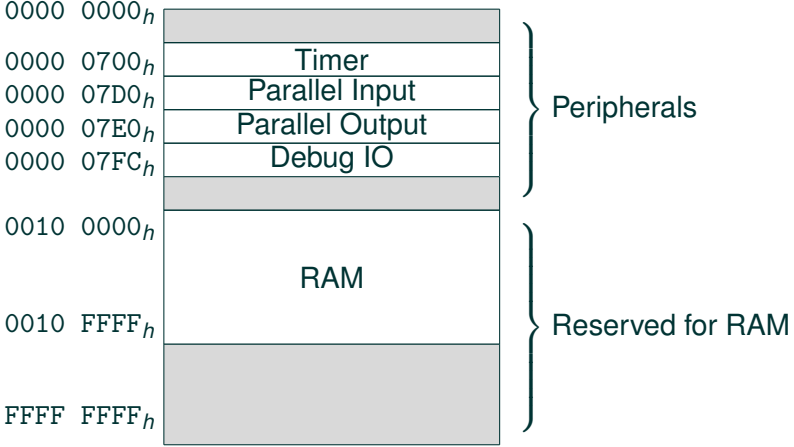
Parallel Input/Output

- On real communication channels, data is not always ready.

→ We need synchronization with control signals.

- Different protocols and standards.
  - Serial protocols: RS232, SPI, USB, SATA, ...
  - Parallel protocols: PATA/IDE, IEEE 1284 (Printer), ...
- We use a simple parallel interface with few control signals.
  - 8-bit data port.
  - Simple valid/ready flow-control.
  - Interrupt for receiving data.
- Mapped to `0x7D0` (Parallel Input) and `0x7E0` (Parallel Output).

Ibex Memory Map including Parallel Input/Output



Parallel Input

Parallel Input Memory Map:

`Valid`: [`R`] data valid, [`W`] '1' clear valid.

`IRQ`: [`R`] IRQ pending, [`W`] '1' clear IRQ.

Example Program:

```
  .global main
  .type   main, @function
main:
POLL_PARIN:
  LW t1, 0x7D0(x0)     # Read PARIN CTRL REG
  ANDI t1, t1, 1
  BEQZ t1, POLL_PARIN

  LW t2, 0x7D4(x0)     # Read available data
  LI t1, 1
  SW t1, 0x7D0(x0)     # Acknowledge Data
  SW t2, 0x7FC(x0)     # Write data to DebugIO

  RET                  # End main
```

Parallel Output

Parallel Output Memory Map:

`Busy`: [R] Dev. busy. [W] '1' start sending.
`IRQ`: [R] IRQ pending, [W] '1' clear IRQ.

Example Program:

```
  .global main
  .type   main, @function
main:
  LI t1, 'a'              # Data we want to send
  SW t1, 0x7E4(x0)        # Write data to the peripheral
  LI t2, 1
  SW t2, 0x7E0(x0)        # Start transmission

POLL_PAROUT:
  LW t2, 0x7E0(x0)        # Read transmission status
  ANDI t2, t2, 1          # Check if busy bit set
  BNEZ t2, POLL_PAROUT    # and re-test again if so

  RET                     # End main
```

Example: Echo program reading from Parallel Input to Parallel Output

- Read from parallel input until newline `\n` (`0x0A` occurs)
- Write characters to parallel output

```
  .global main
  .type   main, @function
main:

  LI t1, 1             # Helper constant
  LI t4, 0x0A          # Newline charater
POLL_PARIN:
  LW t2, 0x7D0(x0)     # Read PARIN CTRL REG
  ANDI t2, t2, 1
  BEQZ t2, POLL_PARIN

  LW t3, 0x7D4(x0)     # Read available data
  SW t1, 0x7D0(x0)     # Acknowledge Data

  SW t3, 0x7E4(x0)       # Write data to the peripheral
  SW t1, 0x7E0(x0)       # Start transmission

POLL_PAROUT:
  LW t2, 0x7E0(x0)       # Read transmission status
  ANDI t2, t2, 1         # Check if busy bit set
  BNEZ t2, POLL_PAROUT   # and re-test again if so

  BNE t3, t4, POLL_PARIN   # We didn't read a newline yet
  RET                      # End main
```

Example: Measuring time using the timer

- Measure time in C

```c
#include <stdio.h>
#include <stdint.h>

typedef struct
{
    uint32_t ctrl;
    uint32_t compare;
    uint32_t counter;
} ibextimer_t;

static ibextimer_t volatile* timer =
  (ibextimer_t volatile*)0x700;

static void start_timer(void) {
    timer->counter = 0;
    timer->ctrl |= 0x1;
}

static uint32_t get_time(void) {
    uint32_t res = timer->counter;
    // prevent gcc from reordering the timer read ...
    asm volatile("" ::: "memory");
    return res;
}
```

```c
uint32_t fibonacci(uint32_t value) {
  if(value < 2)
      return value;
  return fibonacci(value-2) + fibonacci(value-1);
}

int main(int argc, char** argv)
{
  start_timer();
  for(uint32_t i=0; i<11; ++i) {
    uint32_t begin = get_time();
    uint32_t result = fibonacci(i);
    uint32_t duration = get_time() - begin;
    printf("%02ld: result = %ld, begin = %ld, duration = %ld\n", i, re
//     printf("%02ld: %ld\n", i, result);
  }
  return 0;
}
```

C & RV32I CPU with Peripherals

Mario Werner and Robert Schilling

November 14, 2019

Computer Organization and Networks
Graz University of Technology