A simulator, why?

Modern hardware architectures:
- may be too complex to understand
- may be not useful for teaching and demonstration purposes
- may require additional costs for effective development (software development kit, test boards, etc.)
- may add unnecessary complexities to the development cycle

A simulated hardware architecture:
- may be tailored to provide exactly the “right” features for teaching and demonstration purposes
- may be provided with an integrated development kit, graphical user interface and debug tools
- may be deployed on available CS lab equipment
- will probably be a lot slower than the real one (not always a bad feature)

MIPS, MPS and uMPS

MIPS: Microprocessor (without) Interlocking Pipe Stages
- one of the original RISC processor architectures from the ’80s
- with a lot of interesting features
- still widely used (on embedded systems, but also …)

MPS:
- a complete (simulated) computer system integrating an (emulated) MIPS R3000 CPU

uMPS:
- a complete (simulated) computer system integrating an (emulated) MIPS R3000 CPU with physical and virtual memory addressing
A MIPS processor, why?

**MIPS R3000 processor with MIPS I instruction set:**
- is reasonably easy to understand
- provides useful features and insights for instructional purposes
- documentation is widely available
- is supported by the GNU gcc compiler and development kit
- does not provide a pre-defined devices interface

More info (and manuals too):
http://en.wikipedia.org/wiki/MIPS_architecture

MPS and uMPS

**MPS simulator provides:**
- a complete emulation of MIPS R3000 main processor and CP0 (MIPS I instruction set)
- RAM
- ROM (for bootstrap and basic functions)
- a basic set of devices:
  - TOD clock
  - disks
  - tapes
  - printers
  - tty-like terminals
- an integrated development kit, with graphical user interface, a cross-compiler (gcc) and debug tools

MPS and uMPS

**uMPS:**
- (almost) “all of the above”
- ethernet-like network interfaces
- physical and virtual memory addressing
- a streamlined user interface

**Why uMPS and not MPS?**
- Because having virtual memory “right from the beginning” adds unnecessary complexities when writing an OS from scratch…

**MPS and uMPS may be compiled on:**
- FreeBSD, GNU/Linux distributions (x86 and PPC)
- Sun Solaris

uMPS processor architecture

The uMPS architecture
The MIPS processor architecture

**MIPS R2/3000 Architecture**

- System Bus
- Master Pipeline/Bus Control
- Cache Control
  - MCU
  - TLS
- Local Control Logic
- 32 General Purpose Registers
- ALU
- CPU

**MIPS delayed load:**

**MIPS R2/3000 Delayed Load**

1. **Load**
   - IF
   - RD
   - ALU
   - MEM
   - WB

2. **Load delay slot**
   - IF
   - RD
   - ALU
   - MEM
   - WS

3. **CPU Cycle**

**MIPS delayed branch:**

**MIPS R2/3000 Delayed Branch**

1. **Branch Instruction**
   - IF
   - RD
   - ALU
   - MEM
   - WB

2. **Branch Delay slot**
   - IF
   - RD
   - ALU
   - MEM
   - WS

3. **Branch Target Instruction**
   - IF
   - RD
   - ALU
   - MEM
   - WS

4. **CPU Cycle**
uMPS processor architecture

uMPS processor features:

- RISC-type integer instruction set on a load-store architecture
- 32-bit word for registers/instructions/addressing (4 GB physical address space)
- Pipelined execution, delayed loads and branches
- 32 general purpose registers (GPR) denoted $0$ . . . $31$
  - Register $0$ is hardwired to zero (0)
  - Registers $1$ . . . $31$ (also with mnemonic designation)

uMPS processor features (cont’d):

- All of $1$ . . . $31$ registers may be used, but some conventions exist, for example:
  - $26$ and $27$ ($k0$ and $k1$) are reserved to kernel use
  - HI and LO, special registers for holding the results from multiplication and division operations
  - PC, the program counter

uMPS processor architecture

uMPS processor features (cont’d):

CP0 (CoProcessor 0) is incorporated into the main CPU and provides:

- Two processor operation modes:
  - Kernel-mode
  - User-mode
- Exception handling
- Virtual memory addressing

uMPS processor architecture

CP0 has 8 registers:

- Status register
  - Used for exception handling:
    - Cause
    - EPC
  - Used for virtual memory addressing:
    - Index
    - Random
    - EntryHi
    - EntryLo
    - BadVAddr
**Miscellaneous uMPS processor features:**

- **Endianness:**
  - the uMPS processor may operate in big-endian and little-endian mode (the emulator uses the endianness of the host architecture)
  - a different cross-compiler set is required
- **CP1:** optional coprocessor for floating point support
  - unimplemented
  - processor traps if floating point instructions are executed or CP1 access is attempted

**Big endianness:**

- **BIG ENDIAN**
  - High Addresses
    - Word address: 8 9 10 11
    - Low Addresses: 31 ... 0

**Little endianness:**

- **LITTLE ENDIAN**
  - Physical Frame Number and Offset
    - PFN 1211 Offset 0
uMPS processor architecture

uMPS virtual memory address format:

- **Segment Number**, **Virtual Page Number** and **Offset**
- **ASID** (Address Space IDentifier): 0..63 (0 for Kernel)

<table>
<thead>
<tr>
<th>SEGNO</th>
<th>VPN</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 29</td>
<td>...... 12 11</td>
<td>...... 0</td>
</tr>
</tbody>
</table>

uMPS processor architecture

Status register structure:

- **IE**: Interrupt Enable
- **KU**: Kernel/User mode (kernel = 0)
- **IM**: Interrupt Mask
- **VM**: Virtual Memory
- **BEV**: Bootstrap Exception Vector
- **CU**: Coprocessor Usable

uMPS processor architecture

Status register structure:

![Status register structure diagram]

uMPS processor status at bootstrap:

- **CP0** is enabled
- Virtual Memory is off
- Bootstrap Exception Vector bit is on
- Processor is in Kernel mode
- **PC** = 0x1FC0.0000 (in boot ROM)
Exception handling:

**EPC** (Exception PC): is automatically corrected by the CPU if **BD** bit is set, to allow re-execution of the branch.

Cause:

<table>
<thead>
<tr>
<th>BD</th>
<th>CE</th>
<th>IP</th>
<th>ExcCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>2928</td>
<td>15</td>
<td>8</td>
</tr>
</tbody>
</table>

Exception handling (cont’d):

**Cause** explained:

- **IP**: Interrupt Pending
- **BD**: Branch Delay
- **CE**: Coprocessor Error
- **ExcCode**

Exception types:

- Program Traps (PgmTrap)
- SYSCALL/Breakpoint (SYS/Bp)
- TLB Management (TLB)
- Interrupts (Ints)
Exception handling (cont’d):

Program Traps (PgmTrap)
  Address Error (AdEL & AdES)
  Bus Error (IBE & DBE)
  Reserved Instruction (RI)
  Coprocessor Unusable (CpU)
  Arithmetic Overflow (Ov)

SYSCALL/Breakpoint (SYS/Bp)
  SYSCALL instruction
  BREAK instruction

TLB Management (TLB)
  TLB-Modification (Mod)
  TLB-Invalid (TLBL & TLBS)
  Bad-PgTbl (BdPT)
  PTE-MISS (PTMs)

Interrupts (Ints)
  remember Status.IM mask and Status.IEc bit
  hardware and software interrupts

uMPS processor actions on exception:

Basic operations:
  EPC stores the current PC
  BD bit is set if required
  Cause.ExcCode is set
  Status.VM, KU and IE stacks are pushed:

Before the exception:
  \[ \text{VM, KU, IE} \]

Processor exception response:
  \[ \text{VM, KU, IE} \]

Exception-specific operations:
  Address Error (AdEL & AdES): set BadVAddr

Coprocessor Unusable (CpU): set Cause.CE

Interrupts (Ints): set Cause.IP

TLB Management (TLB):
  set BadVAddr
  load EntryHi.SEGNO and EntryHi.VPN
uMPS processor architecture

uMPS processor actions on exception (cont’d):

At the end:
- load PC with a fixed address in ROM:
  - 0x1FC0.0180 if Status.BEV is set
  - 0x0000.0080 if Status.BEV is not set

All this in one atomic operation

ROM exception handlers will perform specific actions and set some exception types:
- Bad-PgTbl (BdPT)
- PTE-MISS (PTMs)

ROM exception handler first task:

to save the current processor state (the “old” one) and to load a new state (the “new” one)

A processor state contains:
- 1 word for the EntryHi CP0 register (contains the current ASID, EntryHi.ASID)
- 1 word for the Cause CP0 register
- 1 word for the Status CP0 register
- 1 word for the PC (New) or EPC (Old)
- 29 words for the GPR registers (GPR registers $0, $k0, and $k1 are excluded)

But where is the ROM?

But where is the ROM?

uMPS physical memory map (Kernel and User modes)

How it is mapped?

ROM and device registers area:
But where is the processor state stored?

in the ROM reserved frame:

But where?

in the bottom part of the ROM reserved frame:

• Ending the exception handling:

ROM handler (hopefully) will load a processor state and:

  jump to some address

  RFE (Return From Exception): pop the KU, IE and VM stacks

Beware…

look at Cause in Old area for knowing exactly what happened

remember that KU, IE and VM stacks in Status were pushed before being stored, and will be popped when returning from the exception

remember that EPC will point to the correct address to jump to after having serviced the exception (the BD bit tells if it was the instruction at EPC or the instruction in a branch delay slot to cause the exception)