Beware of some English slides!

Betriebssysteme

Vorlesung im Herbstsemester 2008
Universität Mannheim

Kapitel 3b: ULIX-Hardware

Felix C. Freiling
Lehrstuhl für Praktische Informatik 1
Universität Mannheim

(Diese Folien sind nicht Teil des Basiskurses)

Overview

- Short noweb example
- Motivation hardware emulator
- ULIX hardware
  - Overview
  - Main memory
  - CPU
  - Timer
  - MMU
  - I/O Controller
- Concurrency semantics
- CPU instruction set and encoding
- CPU instruction cycle and memory access
- Demo of emulator

Short noweb Example

```
This is my first noweb example (in Java).

<<*>>=
class Test {
  <<main>>
}

@ Here's just a simple [[main]] method.

<<main>>=
public static void main(String args[]) {
  System.out.println("Hello World");
}

@ The end.
```

Tangling and Weaving

- notangle test.nw > test.java
- javac test.java
- java test

- noweave test.nw > test.tex
- pdflatex test.tex
- acroread test.pdf

Extraction of Files

- notangle –Rmain test.nw > main.java
- Can define (for example) chunks with filenames, e.g. ulix.h

  <<ulix.h>>=
  ...
  @

- Extract header file from ulix.nw:
  notangle –Rulix.h ulix.nw > ulix.h

Why an Emulator?

The Emulator

- The ULIX hardware is hypothetical, it doesn't exist as hardware
  - But it is very similar to read hardware and could be implemented on a chip
- Instead of building a chip, we write an emulator
  - Program that simulates ULIX hardware
- Serves as experimentation platform and at the same time defines the hardware semantics

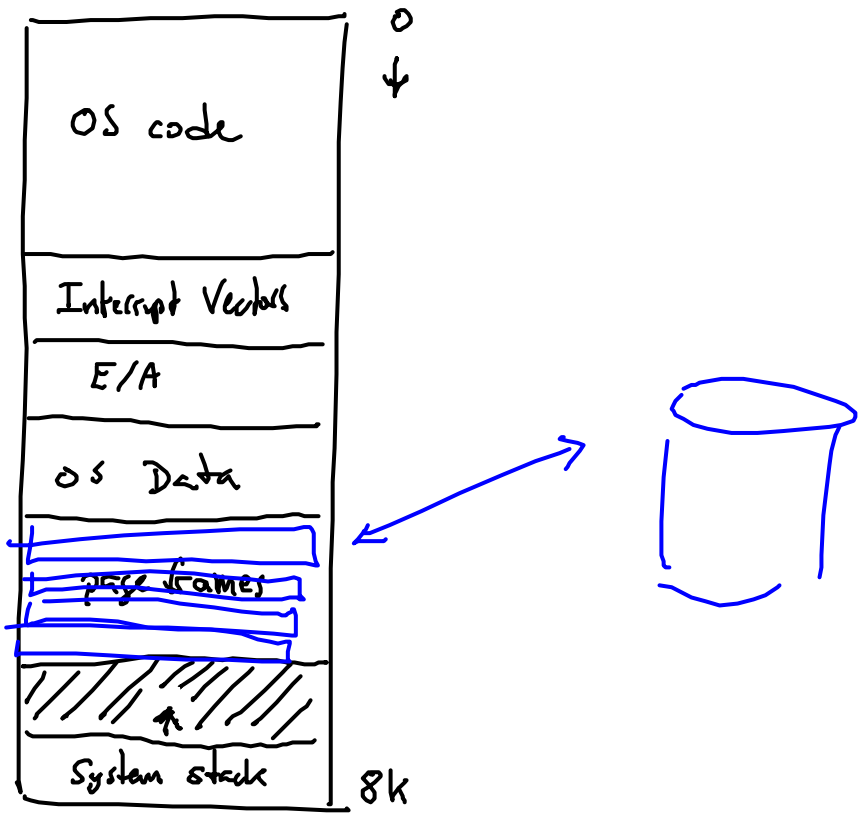
Running ULIX

- Emulator is a program
- Program reads input and starts a computation
- Input of the emulator:
  - Configuration file for the hardware
  - Memory image
  - Disk image

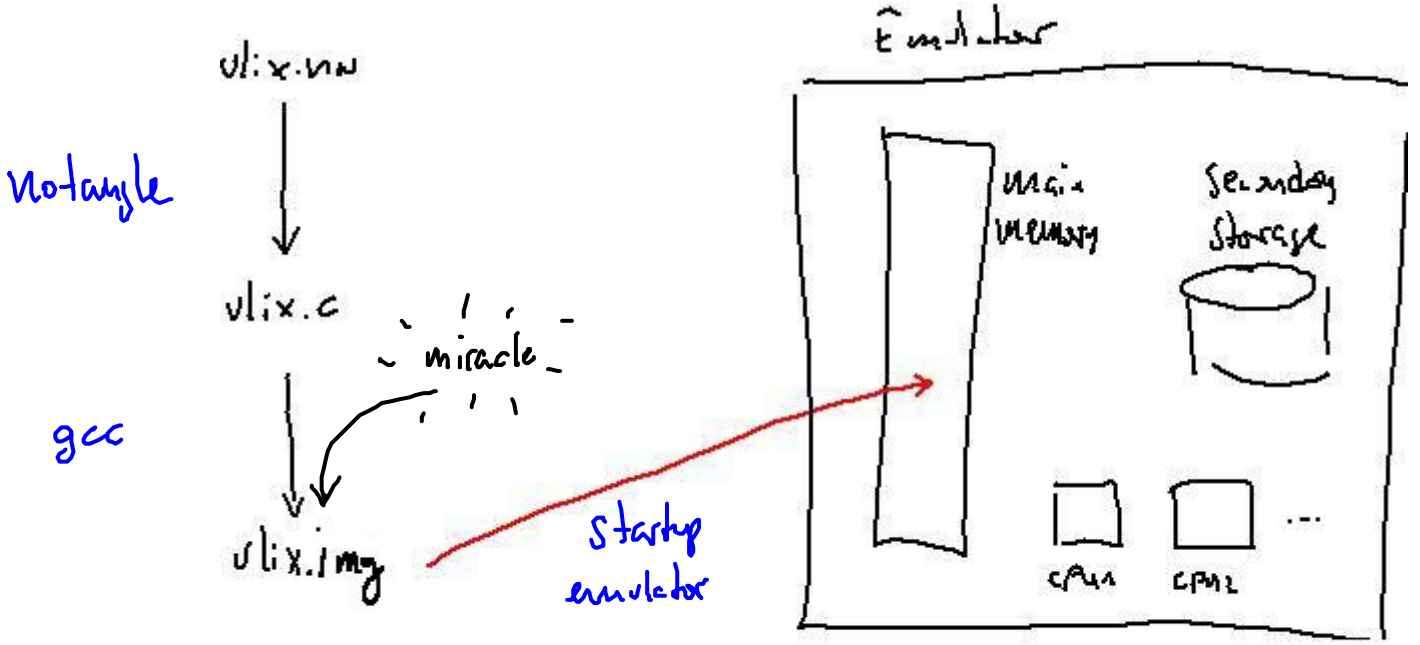
Configuration File Example

```
# traptest.conf
cpu=1
memory=8192
disk=10
sector=512
```

Possible Physical Memory Image Layout



Building ULIX Memory Image



Memory Image

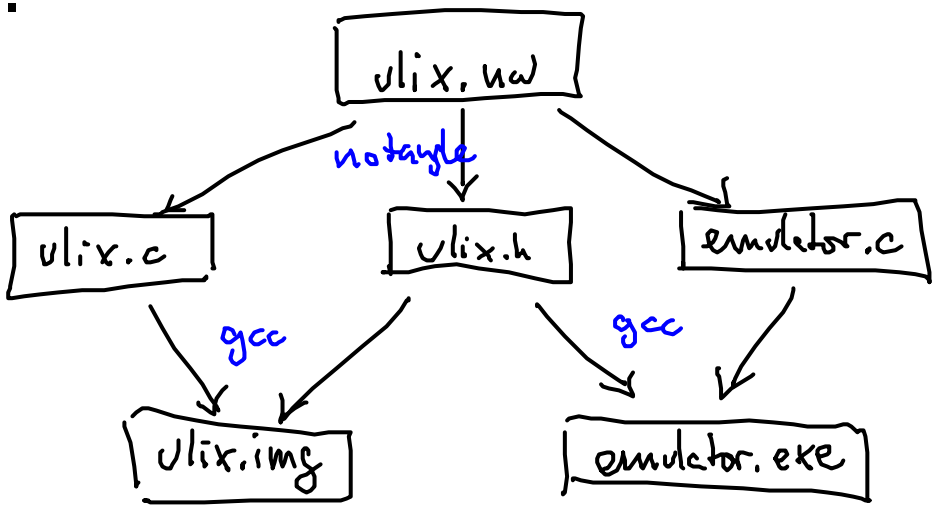
- Memory image defines the initial state of physical memory of the hardware
- Depends on the hardware configuration (size of memory)
- Memory image contains (complete) binary code of ULIX kernel

Files in ulix.nw

- ulix.c (root chunk <<*>>)
  C source code of ULIX kernel
- ulix.h
  Header file of ULIX kernel containing common declarations
- emulator.c
  C source code of ULIX emulator
- Makefile
  Makefile of ULIX project (see **`man make`**)

Header File

- Contains common declarations, e.g.
  - Width of address bus
  - Numer of multi purpose registers in CPU
  - ...



ulix.h Skeleton

File Skeletons

Here are the initial chunks of `ulix.c` and `ulix.h`.

13a ⟨*ulix.h* 13a⟩≡

```
// ulix.h

// The Ulix header file containing common declarations
// ⟨copyright notice 11a⟩

⟨common declarations 13b⟩
```

Here are the first handy declarations.

13b ⟨*common declarations* 13b⟩≡ (13a) 18a▷

```
#define boolean unsigned int
#define true 1
#define false 0
```

Defines:
`boolean`, used in chunks 70, 104, 105, 134d, 135, and 140d.
`false`, used in chunks 70, 107b, 108a, 134–36, 140e, and 141d.
`true`, used in chunks 44b, 105, 107b, 134d, 136, and 141c.

ulix.c

The ULIX kernel does not use external libraries.

14a ⟨*ulix.c* 14a⟩≡

```
// ulix.c

// The Ulix kernel
// ⟨copyright notice 11a⟩

#include "ulix.h"
// no other includes

⟨kernel declarations 34a⟩
⟨kernel global variables 103b⟩
⟨kernel functions 105a⟩
int main(int argc, argv) {
  ⟨kernel main routine 14b⟩
}
```

Defines:
`main`, never used.

14b ⟨*kernel main routine* 14b⟩≡ (14a)

```
⟨initialize kernel global variables 103d⟩
⟨set up virtual memory (never defined)⟩
⟨set up thread management (never defined)⟩
⟨start the operating system (never defined)⟩
```

emulator.c

15 ⟨*emulator.c* 15⟩≡

```
// emulator.c

// Code fragments for the implementation of the
// Ulix hardware emulator.
// ⟨copyright notice 11a⟩

#include "ulix.h"
⟨includes of the emulator 16a⟩

⟨declarations of the emulator 20b⟩
⟨global variables of the emulator 18d⟩
⟨functions of the emulator 16b⟩
int main(int argc, argv) {
  ⟨initialize global variables of the emulator 42c⟩
  ⟨main routine of the emulator 76c⟩
}
```

Makefile

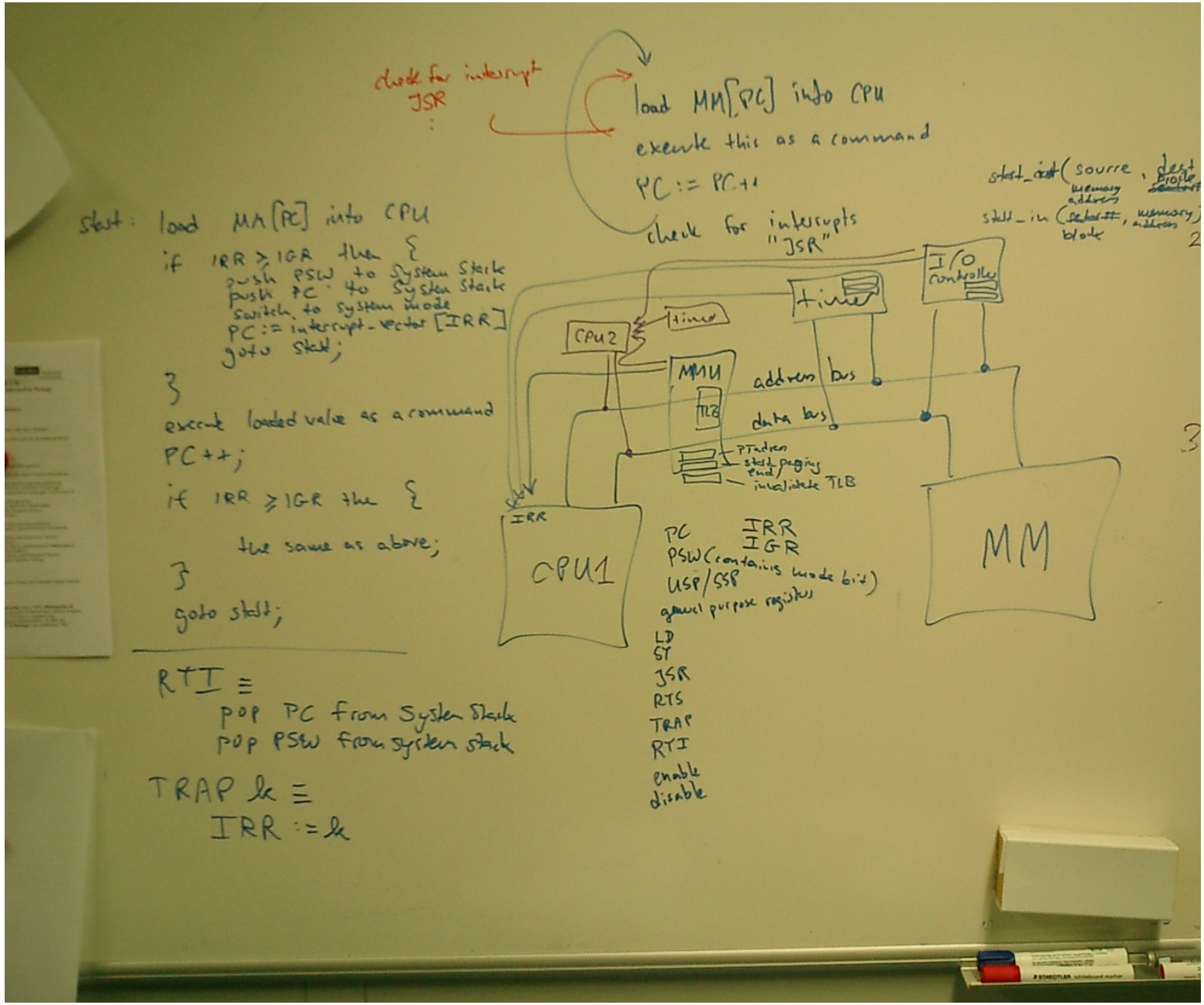
```
# Makefile for Ulix
#
# tested under Win32 using Cygwin

all: ulix.pdf ulix.h emulator.c

ulix.pdf: ulix.tex
   pdflatex ulix.tex
   bibtex ulix
   pdflatex ulix.tex
   pdflatex ulix.tex
   makeindex ulix
   pdflatex ulix.tex
   pdflatex ulix.tex

ulix.tex: ulix.nw
   noweave -autodefs c -index -delay ulix.nw > ulix.tex

ulix.c: ulix.nw
   notangle -L -Rulix.c ulix.nw > ulix.c

ulix.h: ulix.nw
   notangle -L -Rulix.h ulix.nw > ulix.h

emulator.c: ulix.nw
   notangle -L -Remulator.c ulix.nw > emulator.c
```

Demo

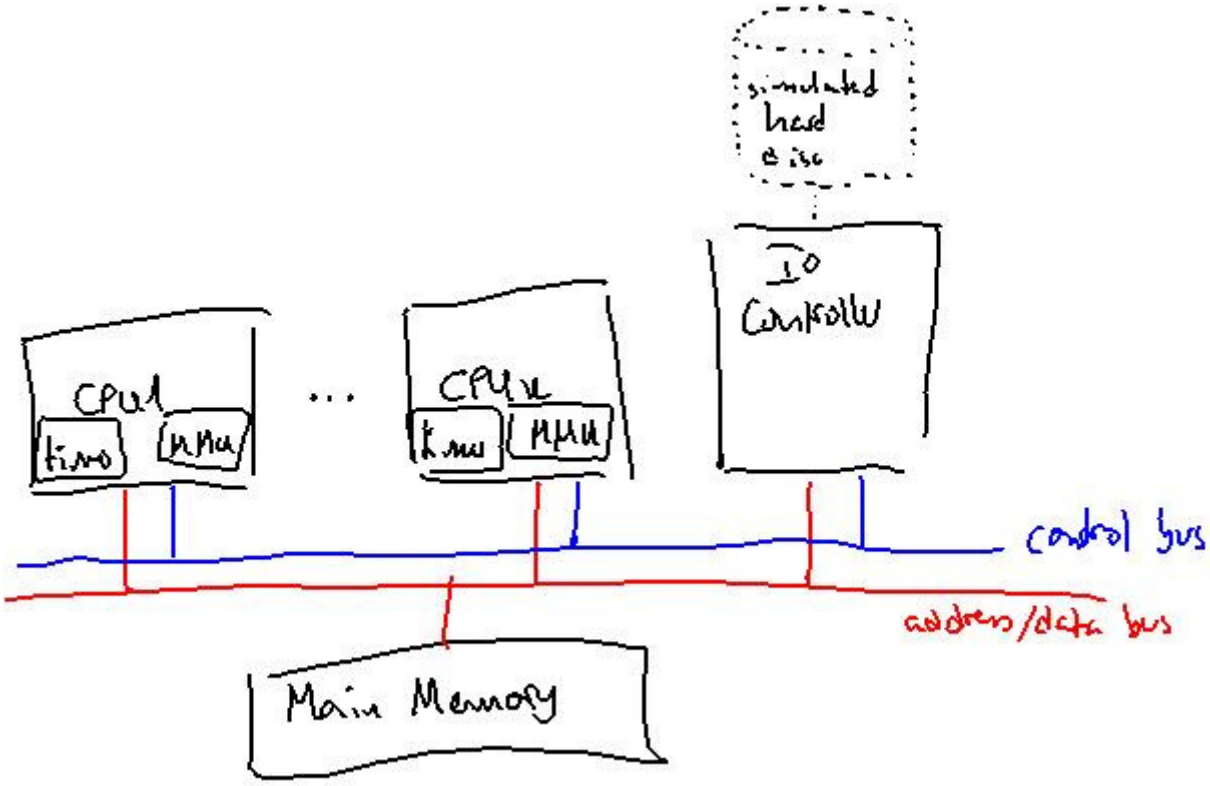
- Extraction of parts of ULIX
  - ulix.h
  - ulix.c
  - emulator.c
- make -n

ULIX Hardware Overview

Historic Overview



Modern Overview



Main Memory

Main Memory Array MM

We will simplify our life here and assume that physical memory starts from address 0 and goes up to MAX_ADDRESS. Starting from MAX_ADDRESS+1 there is no physical memory anymore and this address cannot be encoded on the address bus. Of course, MAX_ADDRESS depends on ADDRESS_BUS_BITS: if there are a address bus bits, then MAX_ADDRESS is equal to $2^a - 1$.

18c ⟨*common declarations* 13b⟩+≡ (13a) ◁18b 20a▷

```
#define MAX_ADDRESS 65535
```

Defines:
MAX_ADDRESS, used in chunks 18d and 103a.

Here we define the main memory as it is used in the emulator.

18d ⟨*global variables of the emulator* 18d⟩≡ (15) 22a▷

```
byte MM[MAX_ADDRESS];
```

Uses byte 41b and MAX_ADDRESS 18c.

The ULIX CPU

General Purpose Registers

We abstract from the number of general purpose registers here. We assume that they are each as large as an `int`, i.e., usually 32 bit.

20a ⟨*common declarations* 13b⟩+≡ (13a) ◁18c 20d▷

```
#define GPR_REGISTERS 16
```

Defines:
`GPR_REGISTERS`, used in chunk 20.

In the emulator we use an array `GPR` to store the contents of the general purpose registers of the CPU. Since we might have multiple CPUs, we hold all data in a C structure similar to a Java class.

20b ⟨*declarations of the emulator* 20b⟩≡ (15) 35a▷

```
struct CPU {
  int GPR[GPR_REGISTERS]; // general purpose registers
  ⟨other declarations in CPU structure 21a⟩
};
```

Uses `GPR_REGISTERS` 20a.

To initialize the CPU we use default values in the registers.

20c ⟨*functions of the emulator* 16b⟩+≡ (15) ◁16b 35b▷

```
void init_cpu(CPU* c) {
  for (int i=0; i < GPR_REGISTERS; i++) {
    c->GPR[i] = 0;
  }
  ⟨initialize other registers declared in CPU structure, accessed via c-> 21b⟩
}
```

Defines:
`init_cpu`, used in chunk 42c.
Uses `GPR_REGISTERS` 20a.

Processor Modes

20d ⟨*common declarations* 13b⟩+≡ (13a) ◁20a 21d▷

```
#define USER_MODE 1
#define SYSTEM_MODE 0
```

Defines:
SYSTEM_MODE, used in chunks 22b, 29b, and 62a.
USER_MODE, used in chunk 56b.

Special Registers: PC

21a ⟨*other declarations in* CPU *structure* 21a⟩≡ (20b) 21c▷

```
unsigned int PC; // program counter
```

Defines:
PC, used in chunks 21b, 29, 56b, 61–65, 67–69, 73, and 74a.

The initial value of the program counter depends on the boot sequence of the hardware. For simplicity we assume that the PC is initialized with 0, i.e., the CPU starts executing at the first instruction in main memory.

21b ⟨*initialize other registers declared in* CPU *structure, accessed via* c-> 21b⟩≡ (20c) 22b▷

```
c->PC = 0;
```

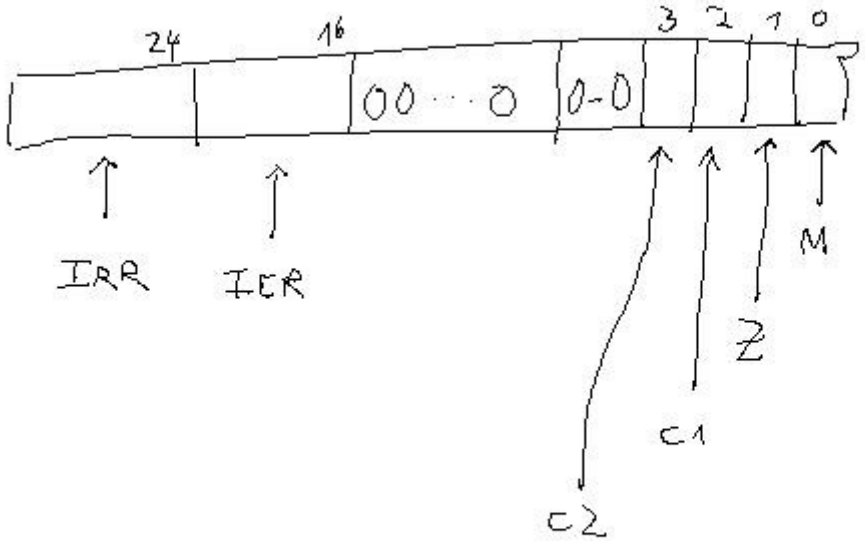
Uses PC 21a.

Special Registers: PSW

21c ⟨*other declarations in* CPU *structure* 21a⟩+≡ (20b) ◁21a 23a▷

```
unsigned int PSW; // program status word
```

Defines:
PSW, used in chunks 22b, 29, 51c, 56b, 62a, 67–69, 71, and 73a.

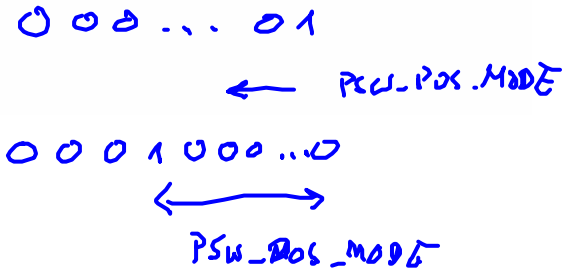


PSW Mode Position and Mask

21d ⟨*common declarations* 13b⟩+≡ (13a) ◁20d 31a▷

```
#define PSW_POS_MODE 0
```

Defines:
PSW_POS_MODE, used in chunk 22.



22a ⟨*global variables of the emulator* 18d⟩+≡

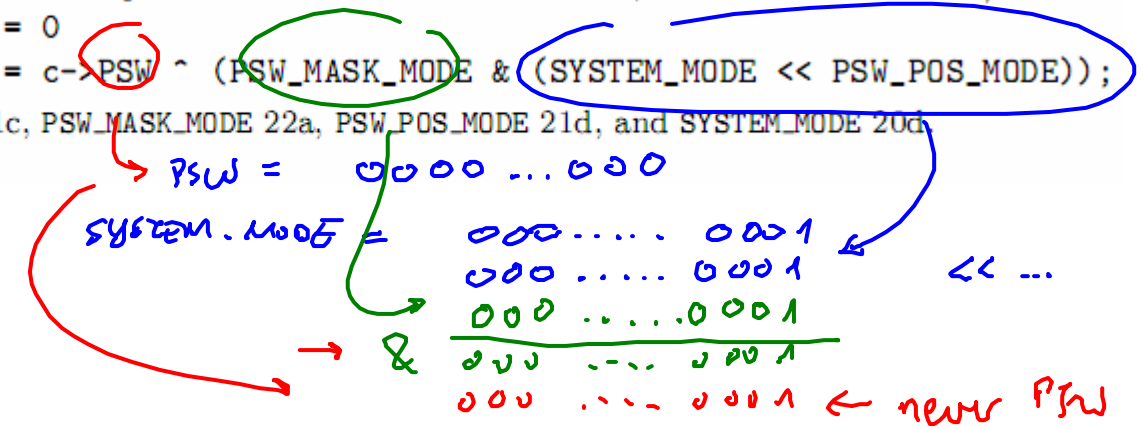
```
int PSW_MASK_MODE = 1 << PSW_POS_MODE;
```

Defines:
PSW_MASK_MODE, used in chunks 22b and 56b.
Uses PSW_POS_MODE 21d.

22b ⟨*initialize other registers declared in* CPU *structure, accessed via* c-> 21b⟩+≡

```
c->PSW = 0
c->PSW = c->PSW ^ (PSW_MASK_MODE & (SYSTEM_MODE << PSW_POS_MODE));
```

Uses PSW 21c, PSW_MASK_MODE 22a, PSW_POS_MODE 21d, and SYSTEM_MODE 20d.



Stack Pointers

23a ⟨*other declarations in* CPU *structure* 21a⟩+≡ (20b) ◁21c 28a▷

```
unsigned int USP; // user stack pointer
unsigned int SSP; // system stack pointer
```

Defines:
SSP, used in chunks 23b, 29, 56b, 62a, 67–69, 71, 73a, 127d, and 128b.
USP, used in chunks 23b, 56b, 67–69, 71, 127b, and 128b.

In the ULIX hardware, we do not assume any special initial value of these registers. The program that runs at boot time should (and must) initialize these registers appropriately. We silently initialize the stack pointers in the emulator anyway.

23b ⟨*initialize other registers declared in* CPU *structure, accessed via* c-> 21b⟩+≡ (20c) ◁22b 28b▷

```
c->USP = 0;
c->SSP = 0;
```

Uses SSP 23a and USP 23a.

Interrupts

28a ⟨*other declarations in* CPU *structure* 21a⟩+≡ (20b) ◁23a 55a▷

```
unsiged int IVT; // start of interrupt vector table
```

We initialize it to a standard value. Note that multiple CPUs should be able to define independent interrupt vector tables in one single main memory. This is why it is important that the starting address of the table should be configurable at runtime.

28b ⟨*initialize other registers declared in* CPU *structure, accessed via* c-> 21b⟩+≡ (20c) ◁23b 32▷

```
c->IVT =  100;
```

ULIX Interrupt Levels

number	class/example
0	none
1	TRAP
2	timer interrupt
3	I/O interrupt
4	MMU interrupt (page fault)
5	division by zero (non-maskable)
6	basic protection violation (non-maskable)
7	invalid machine instruction encoding (non-maskable)

31a ⟨*common declarations* 13b⟩+≡

```
#define MAX_INTERRUPT_LEVEL          7
#define MIN_INTERRUPT_LEVEL          0

#define INTERRUPT_LEVEL_NONE         0
#define INTERRUPT_LEVEL_TRAP         1
#define INTERRUPT_LEVEL_TIMER        2
#define INTERRUPT_LEVEL_IO           3
#define INTERRUPT_LEVEL_MMU          4
#define INTERRUPT_LEVEL_DIV_BY_ZERO  5
#define INTERRUPT_LEVEL_BASIC_PROTECTION  6
#define INTERRUPT_LEVEL_INVALID_ENCODING  7

#define MIN_NMI_LEVEL    5
```

Interrupt Requests

- CPU has
  - an interrupt request register IRR
  - an interrupt enable register IER
- Devices (including the processor) wishing to interrupt set IRR to a specified level
- Service interrupt only if IRR is larger than IER
  - Can use this to mask interrupts

Interrupt Registers

31b ⟨*other register declarations in* CPU *structure* 31b⟩≡

```
int IRR;
int IER;
```

byte

byte

Both registers are initialized to 0 to startup in an "interrupt-free" environment.

32 ⟨*initialize other registers declared in* CPU *structure, accessed via* c-> 21b⟩+≡ (20c) ◁28b 55b▷

```
c->IRR = 0;
c->IER = 0;
```

Uses IER 31b and IRR 31b.

Timer Unit

Timer Unit

- Can be used to send regular interrupts to CPU
- Programmed through registers:
  - Duration register
  - Start/Stop register

Timer State Registers

35a ⟨*declarations of the emulator* 20b⟩+≡ (15) ◁20b 42a▷

```
struct timer_unit {
  ⟨declarations in the timer_unit structure 35c⟩
};
```

To initialize a new timer unit, we can use the following function.

35b ⟨*functions of the emulator* 16b⟩+≡ (15) ◁20c 44c▷

```
void init_timer_unit(timer_unit* t) {
  ⟨initialize entries of the timer unit structure pointer t 36a⟩
}
```

- The register PTRU (*periodic timer remaining units*) stores the remaining time units of the periodic timer, i.e., the number of "ticks" remaining until a timer interrupt is issued. When PTRU counts to 0, a timer interrupt is issued.
- The register PTSU (*periodic timer start units*) stores the initial value of the periodic timer, i.e., the value to which it is set when it runs out. Basically, this is the *period* of the regular timer interrupt. A value of 0 in this register means that the periodic timer is turned off.

We encode the state of the timer unit in the emulator as a pair of integers.

35c ⟨*declarations in the* `timer_unit` *structure* 35c⟩≡ (35a) 36b▷

```
int PTRU;
int PTSU;
```

Timer Command Registers

The timer unit can be manipulated using the two registers PTCR and PTPCR. The name PTCR stands for *periodic timer command register* and PTPCR stands for *periodic timer parameter command register*. These two registers are *write-only* registers, i.e., you can write to them but if you read them, the result may be undefined.

If you want to program the timer unit to issue a timer interrupt every x ticks, you have to do the following:

1. Write the value x to PTPCR.
2. Write a 1 to PTCR.

In case you want to turn the periodic timer off, you perform the same steps with a value of $x = 0$.

The two command registers are reflected in correspondingt entries within the emulator.

36b ⟨*declarations in the* `timer_unit` *structure* 35c⟩+≡ (35a) ◁35c

```
int PTPCR;
int PTCR;
```

Defines:
PTCR, used in chunks 36c and 75c.
PTPCR, used in chunks 36c and 75c.

Memory Management Unit

MMU Page Descriptor Trees

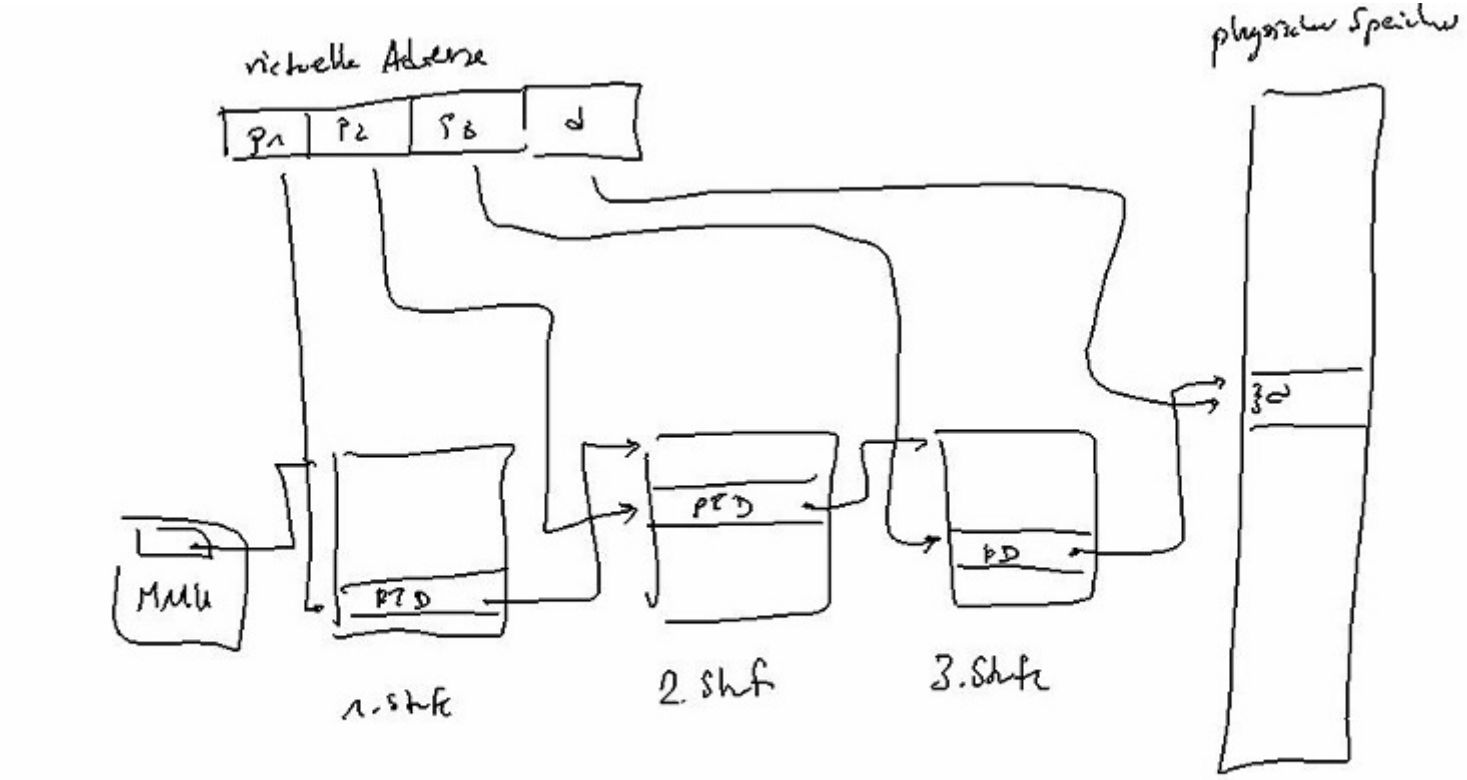
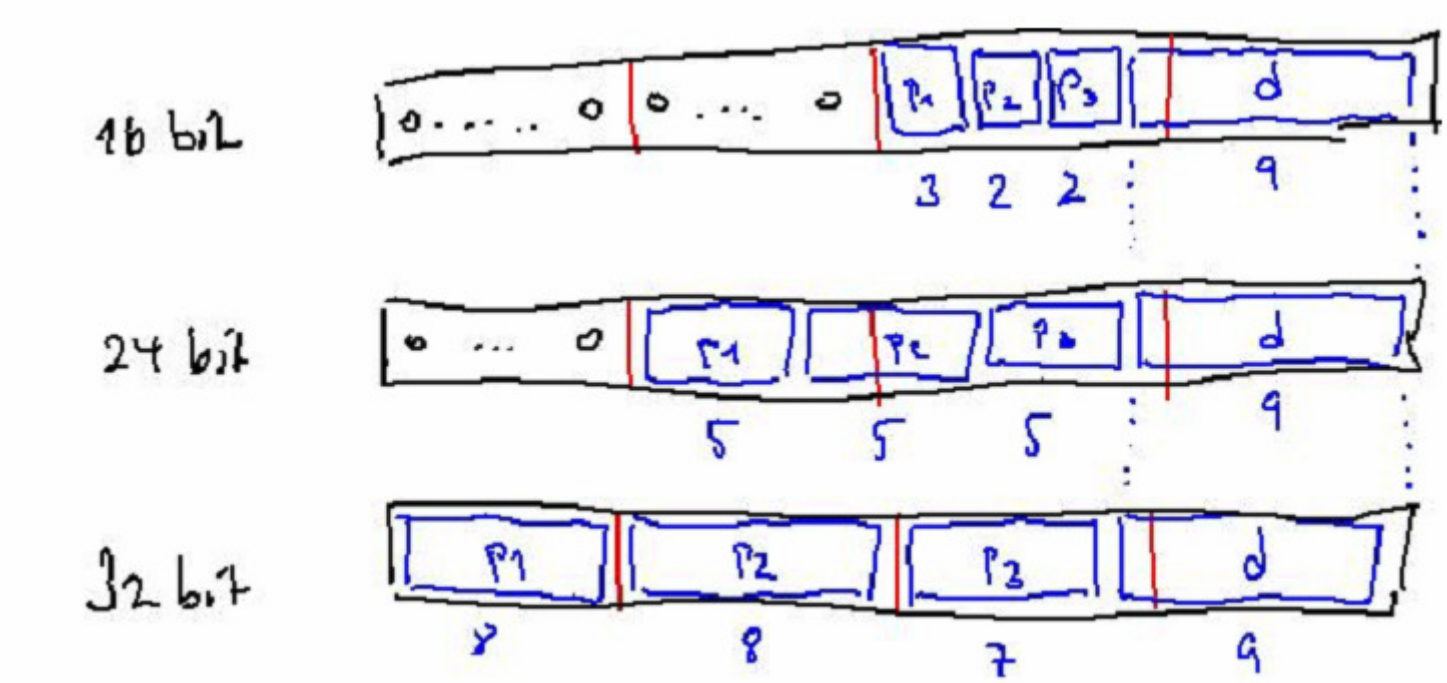


Figure 2.8: Structure of a page-descriptor tree and relation to virtual address.

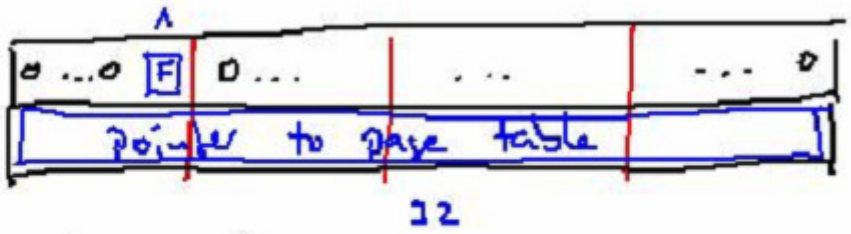
Format of Virtual Address



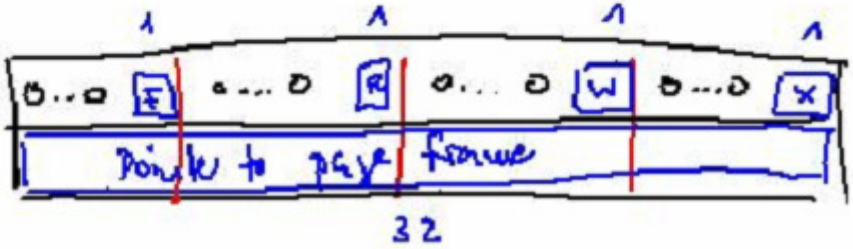
- Three different operating modes: 16, 24, 32 bit (to keep things simple)

Page (Table) Descriptor

page table descriptor



page descriptor



field	meaning
V	valid flag: 0 = null descriptor 1 = page (table) descriptor)
R	read flag: 0 = access within operand fetch phase of instruction cycle not allowed 1 = access within operand fetch phase allowed
W	write flag: 0 = access within write phase of instruction cycle not allowed 1 = access within write phase allowed
X	execute flag: 0 = access within instruction fetch phase of instruction cycle allowed 1 = access within instruction fetch phase not allowed
pointer	pointer to physical address of next page table or page frame

MMU Registers

- UPTR, SPTR: User/system page table register
- MMUCR: MMU command register
  - Turns MMU on and off
- MMUSR: MMU status register
  - Can be used to check whether MMU is on or off

- No TLB/Caches

Secondary Storage
I/O Controller

Secondary Storage

41a ⟨*common declarations* 13b⟩+≡ (13a) ◁31a 41b▷

```
#define SECTOR_SIZE 512
#define NUM_SECTORS 10000 // or any other large number
typedef sector_id unsigned int;
```

Defines:
NUM_SECTORS, used in chunk 41b.
sector_id, used in chunk 102a.
SECTOR_SIZE, used in chunk 41b.

Here's the actual definition of secondary storage.

41b ⟨*common declarations* 13b⟩+≡ (13a) ◁41a 41c▷

```
typedef byte[SECTOR_SIZE] sector;
sector[NUM_SECTORS] secondary_storage;
```

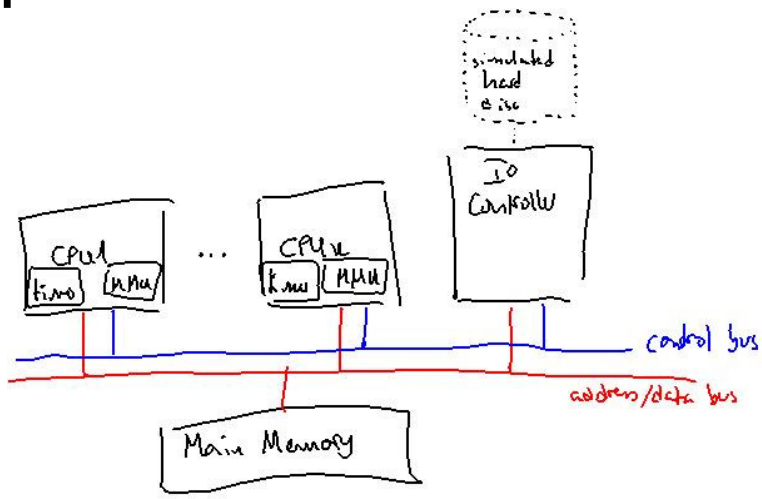
API

- To be defined ...

Concurrency Semantics

Active Units in ULIX Hardware

- Have their own instruction cycle:
  - CPU
  - Timer Unit
  - I/O Controller
- MMU is passive (can only cause synchronous interrupt)



Local vs. Global Step

- Local step = one iteration of some instruction cycle
  - CPU or Timer or IO controller
- Global step = one local step of some active unit
- Only one local step at a time: **Interleaving semantics**

44b ⟨*global instruction cycle of the emulator* 44b⟩≡

```
while (true) {
  d = next_device(d);
  switch (d) {
    ⟨case statements of switch in global instruction cycle 45a⟩
  default: emulator_panic("illegal device id in global instruction cycle");
  }
}
```

The function `next_device` implements a relatively simple round-robin multiplexing strategy.

44c ⟨*functions of the emulator* 16b⟩+≡ (15) ◁35b 45b▷

```
device_id next_device(device_id d) {
  switch (d) {
    IO_CONTROLLER : return CPU0;
                    break;
    CPU0          : return TIMER_0;
                    break;
    TIMER_0       : return CPU1;
                    break;
    CPU1          : return TIMER_1;
                    break;
    TIMER_1       : return CPU2;
                    break;
    CPU2          : return TIMER_2;
                    break;
    TIMER_2       : return CPU3;
                    break;
    CPU3          : return TIMER_3;
                    break;
    TIMER_3       : return IO_CONTROLLER;
                    break;
    default: emulator_panic("illegal device id in next_device");
  }
```

Global Instruction Cycle

Executing Local Steps of CPUs

45a ⟨*case statements of switch in global instruction cycle* 45a⟩≡ (44b) 75a▷

```
case CPU0 : local_step_cpu(0);
            break;
case CPU1 : local_step_cpu(1);
            break;
case CPU2 : local_step_cpu(2);
            break;
case CPU3 : local_step_cpu(3);
            break;
```

Uses `local_step_cpu` 45b.

Executing a local step needs an initial plausibility check.

45b ⟨*functions of the emulator* 16b⟩+≡ (15) ◁44c 51c▷

```
void local_step_cpu(int t) {
  if ((t<=0) || (t>= num_cpus)) {
    emulator_panic("illegal CPU number in local_step_cpu");
  }
  ⟨execute local step of cpu t 52⟩
}
```

Instruction Set and Encoding