Beware of some German slides!

Betriebssysteme

Vorlesung im Herbstsemester 2008
Universität Mannheim

Kapitel 3c: ULIX Machine Language

Felix C. Freiling

Lehrstuhl für Praktische Informatik 1

Universität Mannheim

(Diese Folien sind nicht Teil des Basiskurses)

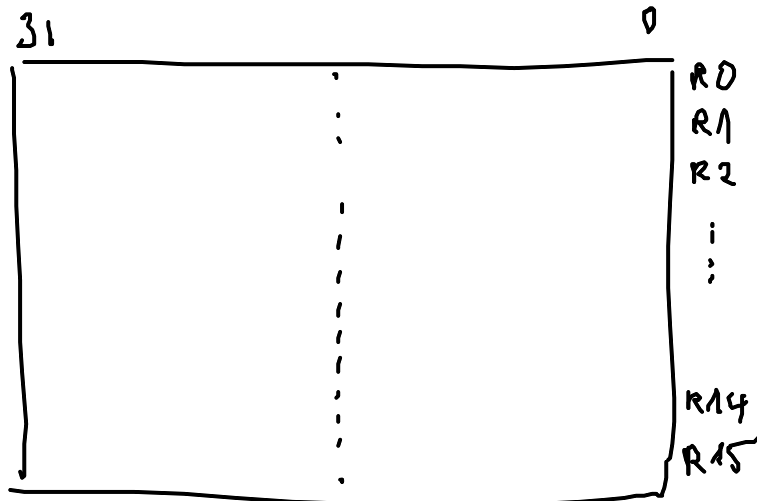
ULIX Assembler Example

```
push    int r0
add     int r0, sp, #4
sub     int sp, sp, #28
sub     int r1, r0, #4
move    int [r0 + -12], #23
move    int [r0 + -8], #42
add     int [r0 + -4], [r0 + -12], [r0 + -8]
sub     int [r0 + -4], [r0 + -12], [r0 + -8]
div     int [r0 + -20], [r0 + -12], [r0 + -8]
move    int [r0 + -4], [r0 + -20]
mul     int [r0 + -4], [r0 + -12], [r0 + -8]
move    int [r0 + -24], [r0 + -12]
mod     int [r0 + -28], [r0 + -24], [r0 + -8]
move    int [r0 + -4], [r0 + -28]
move    int sp, r0
move    int r0, [r0 + 0]
rts
```

Overview

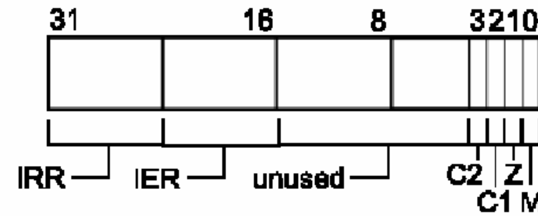
- ULIX hardware recap
- ULIX hardware instructions and encoding
- CPU instruction cycle
- Memory access
- Instruction cycle proper
- Implementation of instructions
- The ULIX compiler

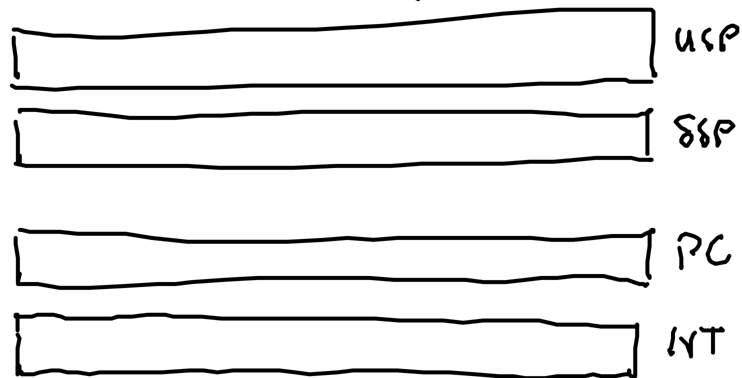
Programming Model



General Purpose Registers

contains





user mode

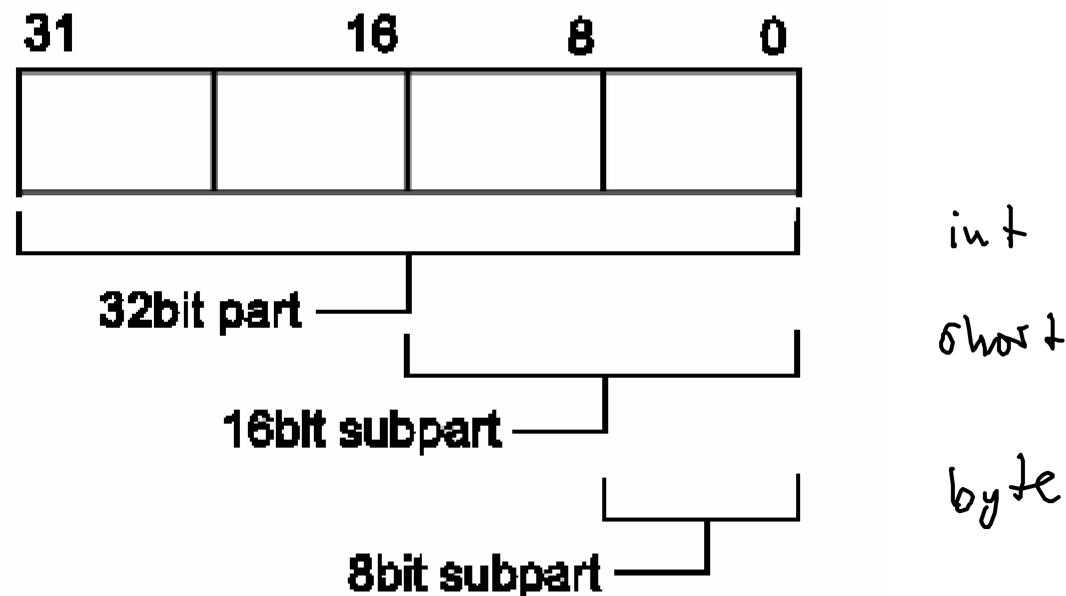
SP

system mode

(use "USP" to access usp in system mode)

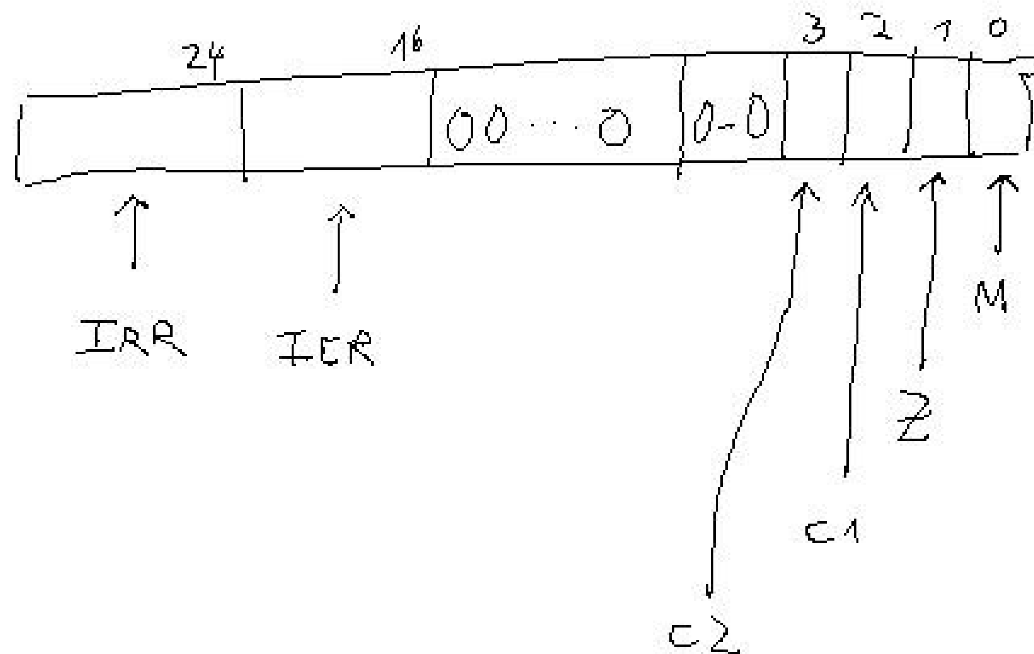
Parts of 32 Bit Registers

- 32 Bit Registers can be used as 16 and 8 Bit registers
  - Use a size parameter in the machine instruction



Program Status Word

- IRR, IER
- C1, C2 for comparison results (see later)
- Z for arithmetic (zero, see later)



UNIX is Big Endian

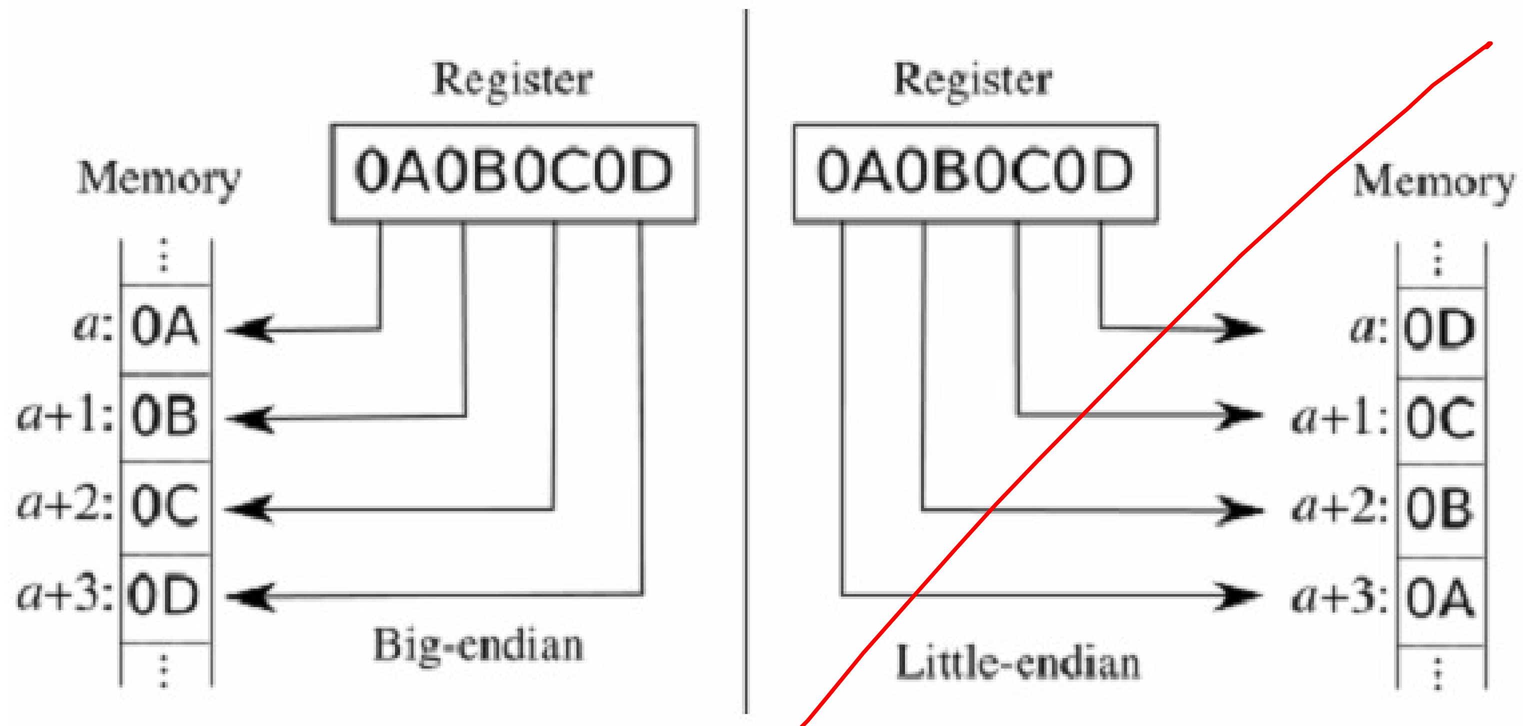
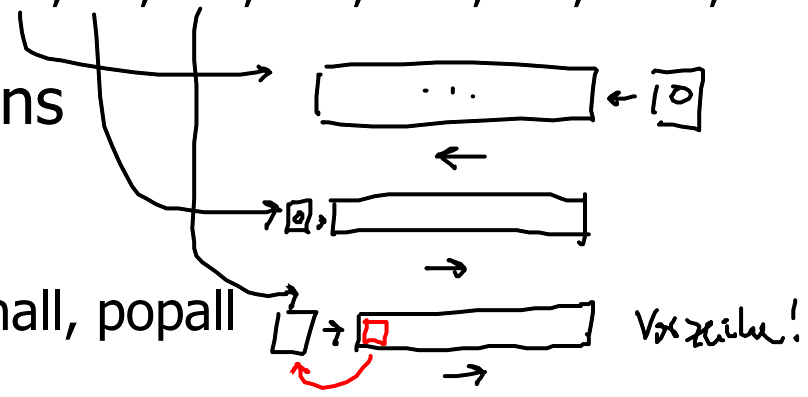


Figure 4 – Big and little endianness illustrated [?]

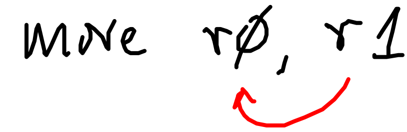
Instruction Set and Encoding

Instruction Set Overview

- Arithmetic/logic instructions
  - add, sub, and, or, xor, sl, lsr, asr, not, mul, div, mod, sdiv, smod
- Comparison instructions
  - cmp
- Memory instructions
  - move, push, pop, pushall, popall
- Jump instructions
  - jmp, jxx (jeq, jne, jl, ...)
- Subroutine instructions
  - jsr, rts, trap, rti
- Special instructions
  - swap, nop, upcast



Instruction Set Comments



- Standard notation:
  - instruction destination, source
  - Example: add r0, r1, r2

- Instruction set was designed to make building a compiler easy!
- Examples (of rather untypical instructions):
  - pushall, popall : good for subroutines
  - upcast: perform a signed upcast
    - Example: upcast int r0, short r1

short s = -3;
int i = (int) s;

Addressing Modes

There exist four different addressing modes. Whenever an instruction works on an operand, an appropriate addressing mode has to be chosen. For example, the push instruction syntax is as follows: push <operand>
where <operand> may be specified by one of the following addressing modes:

1. **Constant**: #value. The constant value which is to be pushed must be specified after a #-character. Example: push #123

   move r0, #4

2. **Absolute**: [#address]. Push the contents located at the specified memory address. Example: push [#1000]

   move r0, [#1000]   move pc, #37

3. **Register**: registername. Push the contents of a register. Example: push r12
4. **Relative**: [registername + constant offset]. In this case, the effective address is the value of the register plus the offset. The contents of the memory located at the effective address is pushed. Example: push [r0+4]

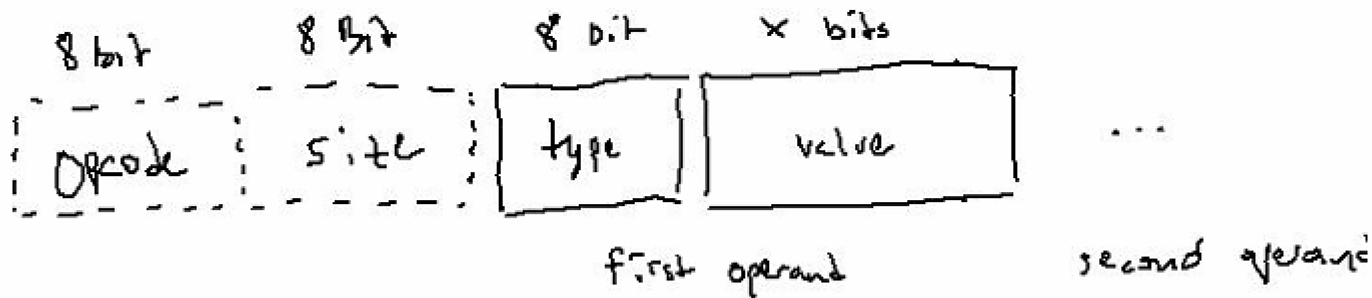
It is allowed to have multiple memory accesses within a single instruction. For example, the following instruction is valid:

```
add [#1000], r0, [r1+1024]
```

Instruction Encoding

- Instructions have
  - Opcode: defines the instruction itself
  - Size: defines the operand sizes
    - byte (8 bit)
    - short (16 bit)
    - int (32 bit)
  - Operands

move short r0, r1



Opcodes and Size Encoding

46 ⟨*opcodes of* ULIX *instructions* 46⟩≡ (47a)

```
#define OPCODE_ADD     0
#define OPCODE_SUB     1
#define OPCODE_AND     2
#define OPCODE_OR      3
#define OPCODE_XOR     4
#define OPCODE_SL      5
#define OPCODE_LSR     6
#define OPCODE_ASR     7
#define OPCODE_SR      8
#define OPCODE_NOT     9
#define OPCODE_MUL    10
#define OPCODE_DIV    11
#define OPCODE_MOD    12
#define OPCODE_SDIV   13
#define OPCODE_SMOD   14
#define OPCODE_CMP    15
#define OPCODE_MOVE   16
#define OPCODE_PUSH   17
```

...

type	encoding value
`int` (32 bit)	0
`short` (16 bit)	1
`byte` (8 bit)	2

Table 2.3: Operand Sizes of ULIX CPU.

Operand Encoding

- Operand consists of
  - operand type/addressing mode
  - value (like register number or constant value)
- Four operand types:

Addressing	Type	Length of Argument	Argument
Constant	0	32, 16 or 8bit	Constant value
Absolute	1	32bit	Memory address which points to the operand value
Register	2	8bit	Register encoding (see below)
Relative	3	40bit	Register encoding followed by offset value

Register Encoding

register function	register name	encoding
general purpose register 0	r0	0
⋮	⋮	⋮
general pupose register 15	r15	15
stack pointer	SP	16
user stack pointer	USP	17
program counter	PC	18
program status word	PSW	19
mode bit (in PSW)	M	20
zero bit (in PSW)	Z	21
compare bit 1 (in PSW)	C1	22
compare bit 2 (in PSW)	C2	23
interrupt request register (in PSW)	IRR	26
interrupt enable register (in PSW)	IER	27
begin of interrupt vector table	IVT	28

Table 2.5: Register encoding.

Examples

One Operand, Constant Addressing Mode

```
Instruction: push int #9
Encoding (hexadecimal): 0D 00 00 00 00 00 09
                        |  |  |  |---------|
                        |  |  |       ^--- constant value
                        |  |  ^--- addressing mode
                        |  ^--- operand size
                        ^--- opcode
```

Two Operands, Relative Addressing Mode

```
Instruction: move byte r1, [r1-4h]
Encoding (hexadecimal): 0C 02 02 01 03 04 FF FF FF FC
                        |  |  |  |  |  |  |---------|
                        |  |  |  |  |  |       |
                        |  |  |  |  |  |       ^---- offset
                        |  |  |  |  |  ^---- register encoding of second operand
                        |  |  |  |  ^----- second operand type
                        |  |  |  ^----- register encoding of first operand
                        |  |  ^----- first operand type
                        |  ^----- operand size type
                        ^----- opcode
```

Example Binary Code

```
// nop
0x18


// jmp #0
0x11,  0x00,  0x00, 0x00, 0x00, 0x00


// move int M, #0x00
0x0C, 0x00, 0x02, 0x14, 0x00, 0x00, 0x00, 0x00, 0x00
```

More Complex Example

```
push    int r0
add     int r0, sp, #4
sub     int sp, sp, #28
sub     int r1, r0, #4
move    int [r0 + -12], #23
move    int [r0 + -8], #42
add     int [r0 + -4], [r0 + -12], [r0 + -8]
sub     int [r0 + -4], [r0 + -12], [r0 + -8]
div     int [r0 + -20], [r0 + -12], [r0 + -8]
move    int [r0 + -4], [r0 + -20]
mul     int [r0 + -4], [r0 + -12], [r0 + -8]
move    int [r0 + -24], [r0 + -12]
mod     int [r0 + -28], [r0 + -24], [r0 + -8]
move    int [r0 + -4], [r0 + -28]
move    int sp, r0
move    int r0, [r0 + 0]
rts
```

CPU Instruction Cycle

Instruction Cycle

```
52      ⟨execute local step of cpu t 52⟩≡                                        (45b)
          if (cpu[t].IRR > cpu[t].IER) { // initial interrupt check
            ⟨service interrupt at CPU t within local step 62a⟩
            continue;
          }
          ⟨load MM[PC] into CPU t 61a⟩
          if (cpu[t].IRR > cpu[t].IER) { // check for interrupt during load
            ⟨service interrupt at CPU t within local step 62a⟩
            continue;
          }
          ⟨decode loaded instruction, fetch operands and execute in CPU t 61b⟩
          if ((cpu[t].IRR > cpu[t].IER) && (cpu[t].opcode != OPCODE_TRAP)) {
            //check for interrupt during decode and execute
            // service interrupt only because of page fault (not because of trap)
            ⟨service interrupt at CPU t within local step 62a⟩
            continue;
          }
          ⟨point cpu[t].PC to next instruction 61c⟩
```

Memory Access

Low Level Memory Read/Write

53 ⟨*functions of the emulator* 16b⟩+≡

```
byte read_byte(unsigned int address) {
  // no consistency checks necessary
  return MM[address];
}
```

54a ⟨*functions of the emulator* 16b⟩+≡

```
void write_byte(unsigned int address, byte b) {
  // no consistency check at the moment
  m_memory[address] = b;
  ⟨update memory mapped I/O register (if necessary) 74c⟩
}
```

MM

MMU Function `mm_map`

- To be done ...

74b ⟨*functions of the emulator* 16b⟩+≡

```
int mm_map(int va) {
  // translate virtual address into physical address
  // using page table pointed to by PTR
  return 0;
}
```

(handwritten correction: `0` crossed out, replaced with `va`)

Defines:
  `mm_map`, used in chunks 54c and 74a.

High-Level Memory Read/Write

54c ⟨*functions of the emulator* 16b⟩+≡

```
byte read_memory_byte(unsigned int address) {
  return read_byte(mm_map(address));
}
```

```
void write_memory_byte(unsigned int address, byte value) {
  write_byte(mm_map(address), value);
}
```

Instruction Cycle Proper

Filling Parts of Instruction Cycle

- Some additional helpful variables used ...

```
61a   ⟨load MM[PC] into CPU t 61a⟩≡
        cpu[t].opcode = load_memory_byte(cpu[t].PC);
        cpu[t].decode_pointer = cpu[t].PC + 1;
```

```
61b   ⟨decode loaded instruction, fetch operands and execute in CPU t 61b⟩≡
        switch (cpu[t].opcode) {
          case OPCODE_NOP : // do nothing
            break;
          ⟨cases of opcode switch in CPU t 62b⟩
          default: emulator_panic("illegal opcode");
        }
```

Uses cpu 42b, emulator_panic 16b, and OPCODE_NOP 46.

```
61c   ⟨point cpu[t].PC to next instruction 61c⟩≡
        cpu[t].PC = cpu[t].decode_pointer;
```

Servicing an Interrupt

62a ⟨*service interrupt at CPU* t *within local step* 62a⟩≡

```
int request = IRR;
IRR = 0; // clear interrupt signal
write_psw_byte(t, PSW_MASK_IRR, PSW_POS_IRR, cpu[t].IRR);
// step 1: push PSW to system stack
write_memory_int(cpu[t].SSP, cpu[t].PSW);
cpu[t].SSP -= 4;
// step 2: push return address to system stack
write_memory_int(cpu[t].SSP, cpu[t].PC);
cpu[t].SSP -= 4;
// step 3: switch to system mode
cpu[t].M = SYSTEM_MODE;
write_psw_flag(t, PSW_MASK_M, PSW_POS_M, SYSTEM_MODE);
// raise interrupt level unless already non-maskable
if (request >= MIN_NMI_LEVEL) {
  cpu[t].IER = MIN_NMI_LEVEL;
} else {
  cpu[t].IER = request;
}
write_psw_byte(t, PSW_MASK_IER, PSW_POS_IER, cpu[t].IER);
// step 4: jump to interrupt handler
PC = read_memory_int(cpu[t].IVT + 4*(request-1));
```

Implementation of Instructions

- Basically done using C operations
- Example: SL (shift left)

SL

63d ⟨*cases of opcode switch in CPU* t 62b⟩+≡

```
case OPCODE_SL:
  read_operand_size();
  skip_operand();
  int source = read_operand();
  int amount = read_operand(OPERANDSIZE_BYTE);
  int result = source << amount;
  write_destination_operand(PC + 2, result);
  break;
```

Example: AND

⟨*cases of opcode switch in CPU* t 62a⟩+≡

```
case OPCODE_AND:
  read_operand_size();
  skip_operand();
  int result = read_operand() & read_operand();
  write_destination_operand(PC + 2, result);
  break;
```

Example: PUSHALL

PUSHALL

69a ⟨*cases of opcode switch in CPU* t 62b⟩+≡

```
case OPCODE_PUSHALL:
  if ((PSW & PSW_MASK_M) == MODE_SYSTEM) {
    for(int i = 0; (i < R.length); i++) {
      write_memory_int(SSP, R[i]);
      SSP += 4 * STACK_GROW_DIRECTION;
    }
  } else {
    for(int i = 0; (i < R.length); i++) {
      write_memory_int(USP, R[i]);
      USP += 4 * STACK_GROW_DIRECTION;
    }
  }
  break;
```

Example: JSR

⟨*cases of opcode switch in CPU* t 62a⟩+≡ (61b) ◁74

```
case OPCODE_JSR:
  cpu[t].operand_size = OPERANDSIZE_INT;
  int destination = read_operand(t);
  if ((cpu[t].PSW & PSW_MASK_M) == MODE_SYSTEM) {
    write_memory_int(cpu[t].SSP, (unsigned int) decode_pointer);
    cpu[t].SSP = cpu[t].SSP - 4;
  } else {
    write_memory_int(cpu[t].USP, (unsigned int) decode_pointer);
    cpu[t].USP = cpu[t].USP - 4;
  }
  cpu[t].decode_pointer = destination;
  break;
```

Example: TRAP

75c ⟨*cases of opcode switch in CPU* t 62a⟩+≡

```
cpu[t].operand_size = OPERANDSIZE_BYTE;
int num = read_operand(t);
cpu[t].IRR = num;
write_psw_byte(t, PSW_MASK_IRR, PSW_POS_IRR, num);
```

Example: UPCAST

76b ⟨*cases of opcode switch in CPU* `t` 62a⟩+≡

```
case OPCODE_UPCAST:
  read_operand_size(t);
  int destination_operand_size = operand_size;
  skip_operand(t);
  read_operand_size(t);
  int source_operand_size = operand_size;
  int source = read_operand(t);
  int result = 0;

  switch(destination_operand_size) {
    case OPERANDSIZE_INT:
      switch(sourceOperandSize) {
        case OPERANDSIZE_SHORT: // short -> int
          result = (short) source;
          break;
        ⋮
```

(handwritten correction: "short" in `(short) source` crossed out and replaced with "int")

Uff!

ULIX Compiler

ULIX Compiler

- Back end of GCC
- Written by Balthasar Biedermann
- Ready for testing ...

Input

```
int main(void){
    int a = 23;
    int b = 42;
    int c = a + b;
    c = a - b;
    c = a / b;
    c = a * b;
    c = a % b;
}
```

Output

```
   .text
.globl main
main:
   push    int   r0
   add     int   r0, sp, #4
   sub     int   sp, sp, #28
   sub     int   r1, r0, #4
   move    int   [r0 + -12], #23
   move    int   [r0 + -8], #42
   add     int   [r0 + -4], [r0 + -12], [r0 + -8]
   sub     int   [r0 + -4], [r0 + -12], [r0 + -8]
   div     int   [r0 + -20], [r0 + -12], [r0 + -8]
   move    int   [r0 + -4], [r0 + -20]Zuerst werde
   mul     int   [r0 + -4], [r0 + -12], [r0 + -8]
   move    int   [r0 + -24], [r0 + -12]
   mod     int   [r0 + -28], [r0 + -24], [r0 + -8]
   move    int   [r0 + -4], [r0 + -28]
   move    int   sp, r0
   move    int   r0, [r0 + 0]
   rts
```

Input

```
int main(void){
  int a = 23;
  int b = 42;
  int c = a & b;
  c = a | b;
  c = a ^ b;
  c = ~a;
}
```

Output

```
   .text
.globl main
main:
   push    int r0
   add     int r0, sp, #4
   sub     int sp, sp, #16
   sub     int r1, r0, #4
   move    int [r0 + -12], #23
   move    int [r0 + -8], #42
   and     int [r0 + -4], [r0 + -12], [r0 + -8]
   or      int [r0 + -4], [r0 + -12], [r0 + -8]
   xor     int [r0 + -4], [r0 + -12], [r0 + -8]
   not     int [r0 + -4], [r0 + -12]
   move    int sp, r0
   move    int r0, [r0 + 0]
   rts
```

Input

```
int main(void){
  signed char c = 4;
  signed short s = c;
  signed int i = s;
  i = c;
  s = (signed short) i;
  c = (signed char) s;
  c = (signed char) i;
}
```

Output

```
   .text
 .globl main
main:
   push   int r0
   add    int r0, sp, #4
   sub    int sp, sp, #21
   sub    int r1, r0, #4
   move   byte       [r0 + -7], #4
   upcast short      [r0 + -6], byte [r0 + -7]
   upcast int [r0 + -4], short [r0 + -6]
   upcast int [r0 + -4], byte [r0 + -7]
   move   int [r0 + -15], [r0 + -4]
   move   short      [r0 + -6], [r0 + -13]
   move   short      [r0 + -17], [r0 + -6]
   move   byte       [r0 + -7], [r0 + -16]
   move   int [r0 + -21], [r0 + -4]
   move   byte       [r0 + -7], [r0 + -18]
   move   int sp, r0
   move   int r0, [r0 + 0]
   rts
```

Ausblick

- ULIX Assemblierer gerade in Arbeit!
- Bald möglich: beliebige C-Programme auf die ULIX-Hardware bringen