Contents Lecture 4

- An introduction to computer organization.
- The machine.c program for simulating a microprocessor.

An Introduction to Computer Organization

computer organization: generic view of a computer
computer architecture: more details to make it efficient

- Processor or CPU (central processing unit)
- Memory
- Hard disk
- Network interface
- Input device, e.g. keyboard
- Output device, e.g. screen
- Software

The hardware to study in this course

- We will study the parts which affect the performance of your program:
  - You will learn how slow memories are.
  - You will learn why and how cache memories can speedup programs.
  - You will not learn how hard disks work (take the Operating systems course)
  - You will learn a little about a "trick" the machine to "think" it has gigabytes of RAM for each running program (also OS course)
- The essential parts in this course are:
  - Processor
  - Memory hierarchy (registers, caches, main memory, swap)
  - Software

Components in a Processor

A processor has the following hardware components (among others):

- An arithmetic and logic unit (called ALU) which e.g. can add two numbers and jump to a function
- Three sets of registers:
  - Integer registers `r0..r31` for integers and addresses (i.e. for pointers but as you know, an address is just a number so it makes sense...)
  - Floating point registers `f0..f31` for real numbers
  - SIMD vector registers `v0..v31` which can contain e.g. four array elements at a time — either floating point or integers.
- A special register called the *program counter* which holds the address of the memory location which contains the next *instruction* to fetch and execute.

What a Processor can Do

A processor can do the following things (among others):

- Transfer data from a memory cell A to a register R (called a read)
- Transfer data from a register R to a memory cell A (called a write)
- ALU operations on the contents of registers and store result in another register
- Branch — to be explained later

Instructions

- An instruction is a number stored in a memory cell which tells the CPU what to do.
- For instance, the instruction to add registers R12 and R29 and save the result in register R3 on a Power is specified by the number: 2087512596.
- A processor does the following:
  - Fetches the instruction in the memory cell pointed out by the program counter.
  - Reads the contents of some registers specified in the instruction. Recall the registers are located in the CPU.
  - Performs the ALU operation specified in the instruction.
  - Writes back the result to another register also specified in the instruction.
  - Adds one to the program counter and repeats these steps.

2087512596 means add r3,r12,r29 for a Power CPU

- An instruction has fields with names and widths in bits eg:

PO	RT	RA	RB	OE	EO	Rc
6	5	5	5	1	9	1

- The opcode (operation code) tells the processor what it should do and the other fields tells it e.g. with which registers.
- PO is primary opcode, RT is target register, RA and RB are source registers, OE indicates whether overflow should be reported, EO is extended opcode, and RC indicates whether a condition register should be set. Ignore OE and RC for now.
- Our Power add instruction has the following values for these fields:

31	3	12	29	0	266	0
011111	00011	01100	11101	0	100001010	0

Why this is encoded as 2087512596

- We can interpret the sequence of bits as a polynomial.
- Denote each bit b_i where we count i from the right.
- The instruction then becomes: $\sum_{i=0}^{i=31} b_i \times 2^i$

31	3	12	29	0	266	0
011111	00011	01100	11101	0	100001010	0

- If we make groups of four bits we can easily convert the sequence to a number in base 16, called a hexadecimal number.
- Although we don't usually write an instruction in base 10, here it is:

0111	1100	0110	1100	1110	1010	0001	0100	
7	c	6	c	e	a	1	4	= 2087512596

Some rules

- The processor architect decides which fields to use and their positions.
- The processor can understand what to do by extracting the opcode fields and fetching the operands.
- The primary opcode is *always* in position 0 (IBM counts from the left in a word).
- To instead subtract two registers, the extended opcode 40 is used.
- The extended opcode is not always used, e.g. when adding a constant half of the instruction (16 bits) specifies the constant and the fields RB, OE, EO, and RC are not present.
- All instructions have the same width — not on X86 though.
- Power is an example of a so called RISC processor, which are superior to CISC processors. The ARM processor in your phone is also a RISC processor.
- We will explain the difference between RISC and CISC later, but the R stands for Reduced and the C for Complex.

An example processor

- Look at the source code of the file `machine.c`.
- That program simulates a simple processor with registers, and a memory.
- First an assembler file is read and translated to machine code which is stored in the memory starting at address zero.

Assembler program: $10 + 13$

```
0: sub     0, 0, 0    ; set register 0 to 0.
1: addi    1, 0, 10   ; R1 = R0 + 10 = 0 + 10
2: addi    2, 0, 13   ; R2 = R0 + 13 = 0 + 13
3: add     3, 1, 2    ; R3 = R1 + R2 = 10 + 13
4: halt    0, 0, 0    ; stop the computer!
```

Assembler program: $\sum_{i=1}^{100} i$

```
0: sub     1, 1, 1      ; sum = R1 = 0
1: sub     2, 2, 2      ; i = R2 = 0
2: addi    3, 2, 100    ; R3 = 100
3: sgt     4, 2, 3      ; R4 = R2 > R3
4: bt      0, 4, 8      ; Branch to 8 if R4 nonzero
5: add     1, 1, 2      ; sum += i
6: addi    2, 2, 1      ; i++
7: ba      0, 0, 3      ; Goto 3
8: halt    0, 0, 0      ; Halt the machine
```

The result will be in R1 when the machine halts.

Assembler program for $n!$

```
int fac(int n)
{
        if (n == 1)
                return 1;
        else
                return n * fac(n-1);
}

int main()
{
        printf("fac(%d) = %d\n", 5, fac(5));
        return 0;
}
```

Assembler program for $n!$

- Issues:
  - Since the `fac` function makes a function call, it must remember where it was once the called function is completed.
  - In this case, the called function happens to be `fac` but recursion is irrelevant, i.e. recursion does not make this more complicated — recursion makes life easier.
  - The value of the parameter `n` must also be remembered somehow.
  - Where is the parameter `n` in the first place? In a register or in memory?
- Solutions:
  - We use a stack and allocate a so called *stack frame* for each function call.
  - The stack frame has space for the return address and the local variables (i.e. `n`).
  - We put the parameters in registers R1, R2, ..., R10.
  - With more than ten parameters, the additional are pushed on the stack.

Factorial program

```
; R1 is the stack pointer
; R31 holds the return address after a CALL
; R3, R4 etc hold parameters.
;
; execution starts here.
sub     0,0,0           ; initialize R0 to zero
addi    1,0,1024        ; initialize stack pointer
call    0,0,19          ; call main
call    0,0,26          ; call exit
```

Factorial program

```
;
; function FAC: parameter N comes in R3.
;
st      31,1,-1          ; save return address
st      3,1,-2           ; save parameter N
subi    1,1,2            ; decrement stack pointer
seqi    4,3,1            ; R4 = N == 1
bf      0,4,12           ; branch if N != 1
; return 1
addi    3,0,1            ; R3 = 1
addi    1,1,2            ; increment stack pointer
jmp     0,31,0           ; jump to return address
```

Factorial program

```
;
; return n * fac(n-1)
subi    3,3,1           ; N-1
call    0,0,4           ; recursive call. result in R3
ld      4,1,0           ; reload parameter
mul     3,3,4           ; R3 = N * fac(N-1)
ld      5,1,1           ; reload return address
addi    1,1,2           ; increment stack pointer
jmp     0,5,0           ; jump to return address
```

Factorial program

```
;
; function MAIN
;
st      31,1,-1         ; save return address
subi    1,1,1           ; decrement stack pointer
addi    3,0,5           ; N = 5
call    0,0,4           ; call fac
ld      5,1,0           ; reload return address
addi    1,1,1           ; increment stack pointer
jmp     0,5,0           ; jump to return address
;
; function EXIT
;
halt    0,0,0           ; halt the machine
```

The stack: contents when recursion stops

1023	3	return address from init
1022	23	return address from main
1021	5	parameter
1020	14	return address from fac
1019	4	parameter
1018	14	return address from fac
1017	3	parameter
1016	14	return address from fac
1015	2	parameter
1014	14	return address from fac
1013	1	parameter

Some questions

- What will happen if our main tries to compute 600!?
  The stack pointer we be decremented and point into the instructions and overwrite some and then the machine will either discover a correct but meaningless instruction or an incorrect instruction.
- When is this a problem on real machines?
  When highly recursive functions declare large local data stored in stack frames (not references pointing to large objects), especially on multiprocessors since then the stack for each thread is smaller.
- Do you understand how instruction execution works now? Hope so.

Summary so far

- What would the performance of the machine we just saw be? Disastrous, because:
  - Both instructions and data always access main memory which is slow.
  - Only one instruction is executed at a time.
- The advanced microprocessors you will learn about is the result of 50 years of computer architecture research.
- How can we make clever use of the transistors on a chip to execute *existing* programs faster?
- The other side of the question is, how can we write *new* programs which exploit the existing machine? This is what our course will be about.

Computer architecture

- We can add some more instructions to the description in `machine.c` and translate it to VHDL (a hardware description language) and then send in a file and get back a chip (and an invoice, unfortunately). So, are we done now?
- For correctness, yes.
- For performance, no.
- What can we do?
  - Add small fast cache memories.
  - Execute multiple instructions at the same time.
  - Add many processors on one chip — a chip multiprocessor.
  - Connect multiple chips through some kind of network — a parallel computer.

Computer architecture research

- Actually, before we do anything, we need to understand where the performance bottlenecks are.
- A computer architect is free to do anything with billions transistors but is forced to obey three rules:
  - Performance: a new idea must lead to substantial performance improvements
  - Programmability: the machine should run C (and FORTRAN if it's a supercomputer) effectively.
  - Cost-effectiveness.
- Usually also binary compatibility is required for market acceptance (see what happened to Intel's IA-64 Itanium — called Titanic)
- Numerous businesses have invented faster computers without considering these rules, wasted big money and/or gone bankrupt.