TSTE12 Design of Digital Systems Lecture 12

Kent Palmkvist

LINKÖPING UNIVERSITY

Agenda

- Microprocessor structures and programming
- Assembly language
- C-language low level programming

Practical issues

- Project presentation no later than 1/11
- Two or three sessions
  - 2-3 groups/session
- 1 group presents while others are acting as audience, then swap
- 20 minutes for each group, including demo
- Projector, DE2-board, screen, keyboard, speakers available in presentation room.

Microprocessor usage

- Suitable for complex programming
  - User interfaces
  - Complex state machine behavior
- Standard components
- Longer response time
  - Responses in range of us, ms, or more
- High resource utilization
  - ALU, registers etc.
- Sequential processing
  - one instruction at a time

Why leave microprogrammed structures

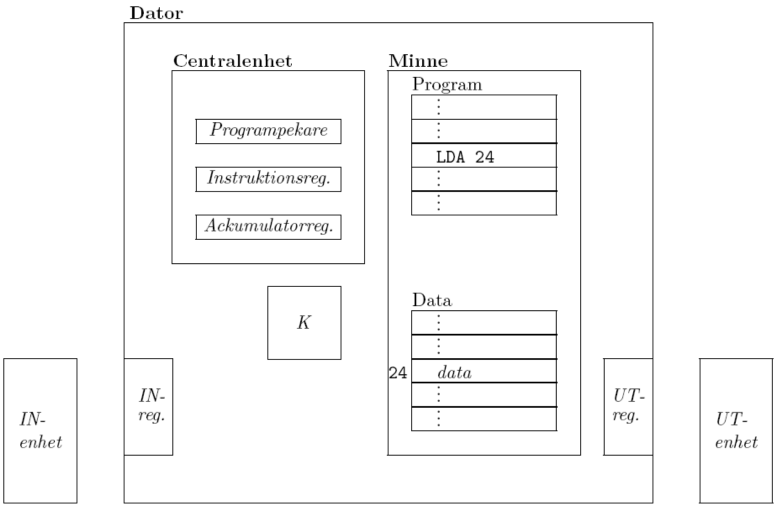
- Assembly language simplifies programming
  - No need to understand all small details
  - Lot of timing issues hidden
- Smaller memory footprint than microprogrammed
  - Previous microprogrammed example: long sequence of event for loading register value
  - Many control bits never used at the same time

Why leave microprogrammed structures, cont.

- Increase reuse
  - Architecture of processor may change while keeping the assembly language format
    - Example: 8086->80386->pentium->core2->i7
    - Sometimes binary compatible
  - Compilers of high-level languages
    - C/C++, JAVA, Python, Perl,....

Model computer example

- Computer
  - Central processing
    - Program pointer
    - Instruction regist
    - Ackumulator
  - Memory
    - Program
    - Data
  - Peripherals
    - Inputs
    - outputs



General Microprocessor Structure

- Similar to microprogrammed structure
- Program information stored in memory
  - Shared with data contents
- Program counter
  - Point to next instruction to execute
- Instruction Register
  - Current executed instruction (not visible to programmer)

Programmer model

- General Purpose Registers
  - Single or multiple registers
- Special purpose registers
  - Program counter (PC, point to next instruction to execute)
  - Stack pointer (SP, temporary space + return adresses)
  - Index registers (addressing modes, pointers)
  - Flag register (indicate result properties from operations, e.g. plus, zero)
- Memory space
  - Read or write to memory cells
  - Some addresses does not have memory cells

Microprocessor behavior

- Fetch
  - Read Program instruction from memory (pointed to by program counter (PC) register)
- Decode
  - Determine what to happen, create control signals, fetch register values
- Execute
  - Update register values, move data to/from memory, arithmetic/logic operations, jumps,M

Assembly level programming

- Describe each instruction used to implement behavior
  - Work on internal registers and/or memory cells
- Platform dependent
  - Each processor family have their own instruction set
  - Many models of the same CPU family share instruction set (e.g., 8086 – core i7)
- Maximum detail (compared to C etc.)

Instruction types

- Memory access
  - Includes I/O input and output
  - Support various addressing modes
- Arithmetic and logic
  - Modify/calculate register values
  - Include shift and rotate
- Register transfer
  - Move values between registers

Instruction types, cont.

- Branch and Jump
  - Includes conditional branch/jump
- Stack, Subroutines
  - pop/push, call, return from subroutine
- Control
  - Enable/disable interrupts, hardware breakpoints etc

Adressing modes

- Immediate
  - Data in instruction itself, e.g. movia r1,0x12
- Direct
  - Address defined in instruction, e.g ldw r1,0x1234
- Indirect
  - Register contains address to use, e.g. ldw r1,0(r2)
- Indexed
  - Address plus offset, e.g. ldw r1,0x1324(r2)

10/13/2024 21:10

Assembly program example

- Calculate the sum of products

```
.include "nios_macros.s"

            .global _start
_start:
            movia r2, AVECTOR    /* Register r2 is a pointer to vector A */
            movia r3, BVECTOR    /* Register r3 is a pointer to vector B */
            movia r4, N
            ldw r4, 0(r4)               /* Register r4 is used as the counter for loop iterations */
            add r5, r0, r0              /* Register r5 is used to accumulate the product */
LOOP:       ldw r6, 0(r2)               /* Load the next element of vector A */
            ldw r7, 0(r3)               /* Load the next element of vector B */
            mul r8, r6, r7              /* Compute the product of next pair of elements */
            add r5, r5, r8              /* Add to the sum */
            addi r2, r2, 4              /* Increment the pointer to vector A */
            addi r3, r3, 4              /* Increment the pointer to vector B */
            subi r4, r4, 1              /* Decrement the counter */
            bgt r4, r0, LOOP            /* Loop again if not finished */
            stw r5, DOT_PRODUCT(r0)     /* Store the result in memory */
STOP:       br STOP

N:          .word           6 /* Specify the number of elements */
AVECTOR:    .word           5, 3, −6, 19, 8, 12 /* Specify the elements of vector A */
BVECTOR:    .word           2, 14, −3, 2, −5, 36 /* Specify the elements of vector B */
DOT_PRODUCT: .skip 4
```


Assembly results

- Translate instruction to binary form
- Indicate value in each memory adress

```
 1                           .include "nios_macros.s"
 2
 3                           .global _start
 4                           _start:
 5 0000 34008000                     movia r2, AVECTOR       /* Register r2 is a pointer to vector A */
 5      04008010
 6 0008 3400C000                     movia r3, BVECTOR       /* Register r3 is a pointer to vector B */
 6      0400C018
 7 0010 34000001                     movia r4, N
 7      04000021
 8 0018 17000021                     ldw r4, 0(r4)           /* Register r4 is used as the counter for loop iterations */
 9 001c 3A880B00                     add r5, r0, r0          /* Register r5 is used to accumulate the product */
10 0020 17008011             LOOP:   ldw r6, 0(r2)           /* Load the next element of vector A */
11 0024 1700C019                     ldw r7, 0(r3)           /* Load the next element of vector B */
12 0028 3A38D131                     mul r8, r6, r7          /* Compute the product of next pair of elements */
13 002c 3A880B2A                     add r5, r5, r8          /* Add to the sum */
14 0030 04018010                     addi r2, r2, 4          /* Increment the pointer to vector A */
15 0034 0401C018                     addi r3, r3, 4          /* Increment the pointer to vector B */
16 0038 C4FF3F21                     subi r4, r4, 1          /* Decrement the counter */
17 003c 16F83F01                     bgt r4, r0, LOOP        /* Loop again if not finished */
18 0040 15004001                     stw r5, DOT_PRODUCT(r0) /* Store the result in memory */
19 0044 06FF3F00             STOP:   br STOP
20
21                           N:
22 0048 06000000             .word   6                       /* Specify the number of elements */
23                           AVECTOR:
24 004c 05000000             .word   5, 3, -6, 19, 8, 12     /* Specify the elements of vector A */
24      03000000
24      FAFFFFFF
24      13000000
24      08000000
25                           BVECTOR:
26 0064 02000000             .word   2, 14, -3, 2, -5, 36    /* Specify the elements of vector B */
26      0E000000
26      FDFFFFFF
26      02000000
26      FBFFFFFF
27                           DOT_PRODUCT:
28 007c 00000000             .skip 4...
```

Program flow

- Very similar to microcode
  - Single sequential execution of instructions
  - Branch/jump used to implement loops, conditional statements
- Subroutines implements function calls
  - Subroutine call saves next instructions location before jump to subroutine
  - At end of subroutine restore PC to make jump back to instruction after subroutine call
  - Usually implemented using stack or special LR register

Interrupts

- Give response without polling/checking continuously
- Interrupt sequence due to external event
  - Timer, I/O, Illegal instruction, etc.
- Interrupt routine at predefined location in memory
- Sequence being interrupted must not notice interrupt
  - Save processor state, and restore after completed interrupt routine
  - Similar to a subroutine call, but without any instruction making the call

LINKÖPING UNIVERSITY

C-level programming

- Platform independent or with little platform dependence
  - Big endian vs little endian
  - Word size (8, 16, 32, 64)
- Possible to describe interrupt routines etc (same as assembly language)
- Use of hardware through memory mapped I/O
  - Store values into registers
  - Read values from registers

C-level programming, cont.

- Registers in the processor not directly accessible
  - Compiler decides where to put variables (registers, memory etc.)
- Simple constructs may be translated into long sequences of assembly code
- Less control of code
- Possible to mix with assembly language

I/O example

- Parallel input port for switches
  - Decode memory address, read value directly
- Parallel output port for LED
  - Write to a register driving the LEDs
- Pointers used to reference memory

```
#define SWITCHES_BASE_ADDRESS 0x10000010
#define LEDR_BASE_ADDRESS 0x10001000

int main(void)
{
  int * red_leds = (int *) LEDR_BASE_ADDRESS;                /* red_leds is a pointer to the LEDRs */
  volatile int * switches = (int *) SWITCHES_BASE_ADDRESS;   /* switches point to toggle switches */
  while(1)
    {
      *(red_leds) = *(switches);                             /* Red LEDR[k] is set equal to SW[k] */
    }
  return 0;
}
```


Additional subjects

- Floating point calculations and hardware
- Caches
- Virtual memory
- ….