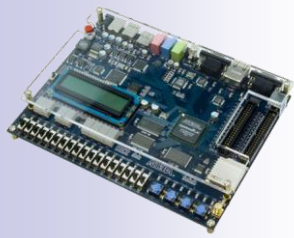
Struktury počítačových systémů

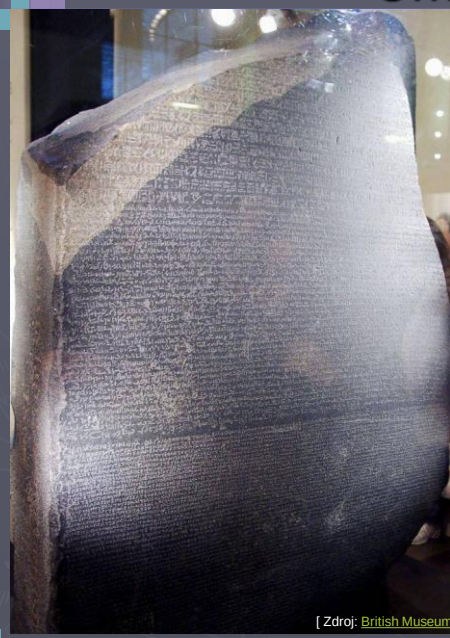


Přednáší: Richard Šusta

24. listopadu 2011 verze 1.0

ČVUT-FEL v Praze – kód předmětu A0B35SPS

Omluva



[Zdroj: British Museum]

Dnešní dnešní přednáška bude zas česko-anglická

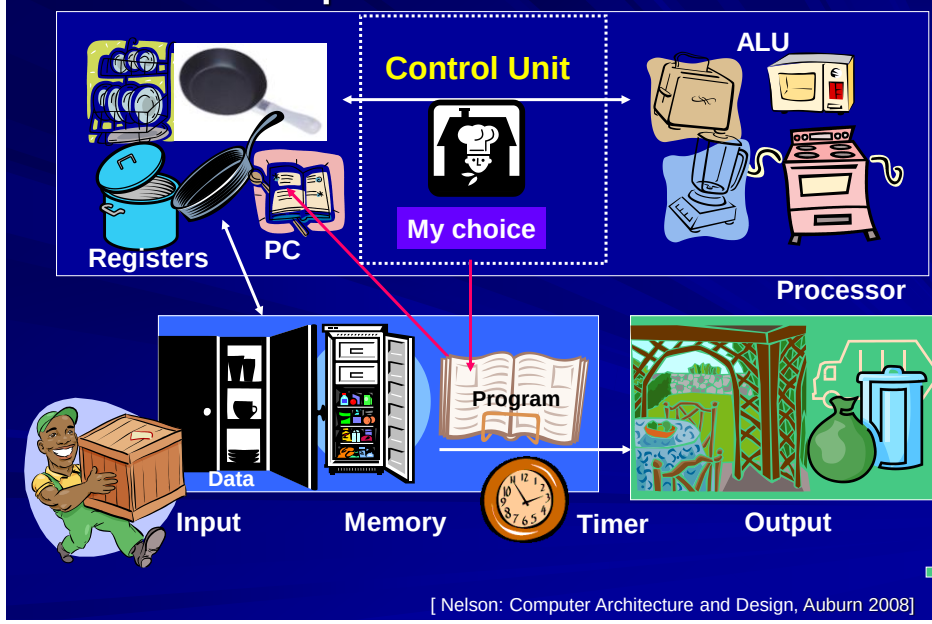
** žel nemám vše přeložené do angličtiny pro zahraniční studenty, ale ani vše v češtině*

** nicméně jsem upřednostňoval češtinu, pokud byla volba.*

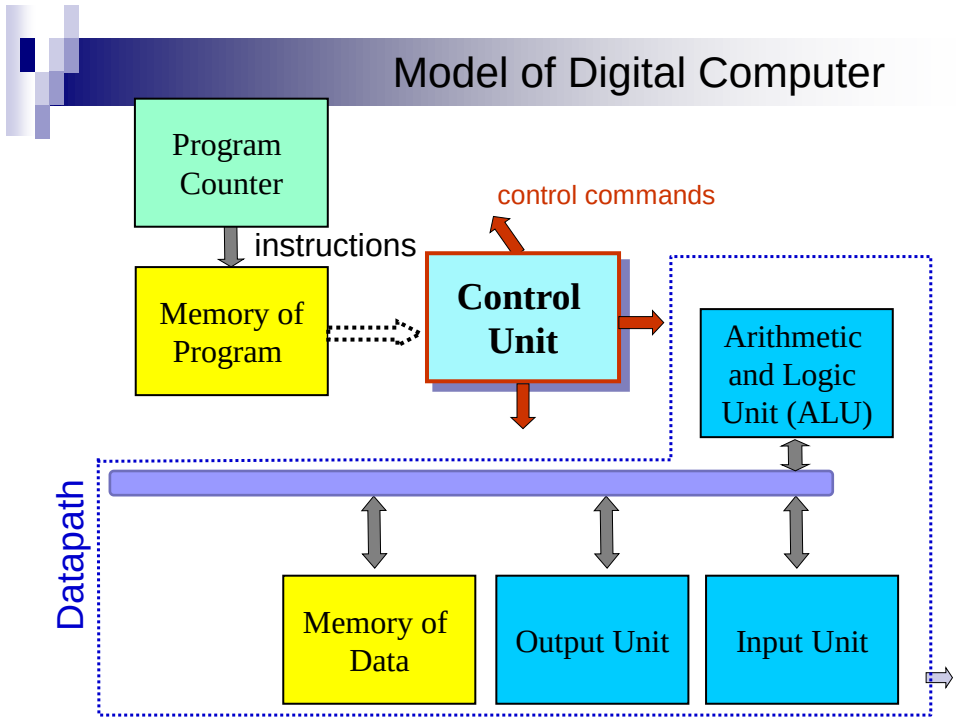
Paralelní řadiče

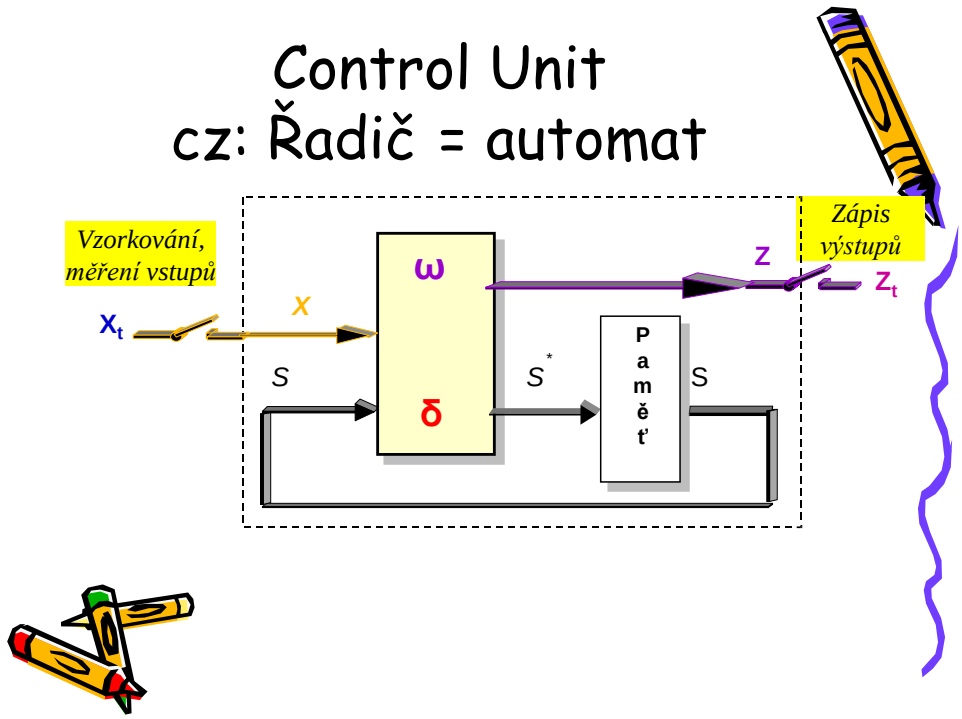
[http://i3.photobucket.com/albums/y70/nickheyward/teamachine.jpg]

Computer à la Kitchen



[Nelson: Computer Architecture and Design, Auburn 2008]


Model of Digital Computer
Program Counter
instructions
control commands
Memory of Program
Control Unit
Arithmetic and Logic Unit (ALU)
Datapath
Memory of Data
Output Unit
Input Unit

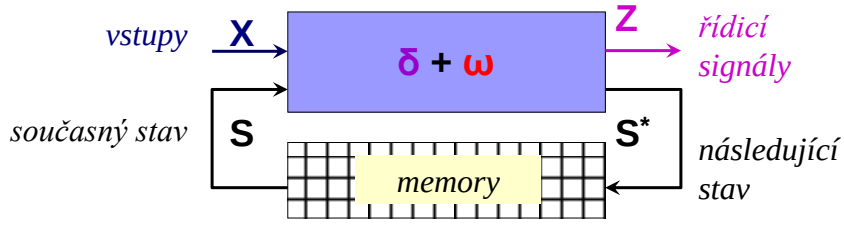

Control Unit
cz: Řadič = automat
Vzorkování, měření vstupů
Zápis výstupů
X
Z
S
S*
S
Paměť
ω
δ

Dělení automatů

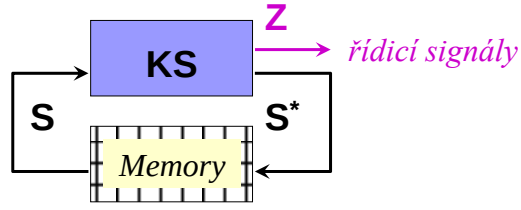
- Podle realizace **δ , ω**
  - paralelní (hardwarový)
    - *řešeno jako logický obvod či paměť*
    - **výhody**: - rychlá reakce
      - vysoká odolnost proti vnějšímu rušení
    - **nevýhody:** - složitější návrh pro nutnost řešit vnitřní hazardy
      - používá se v "Reconfigurable Computing"
    - **řešení**: - pevně naprogramovaný
      - reprogramovatelný
  - sériový (softwarový)
    - *činnost automatu se emuluje programem*
    - **výhody** - flexibilita, univerzálnost
    - **nevýhody** - pomalejší
      - citlivější na vnější rušení

⇨

Varianta kreslení řadiče



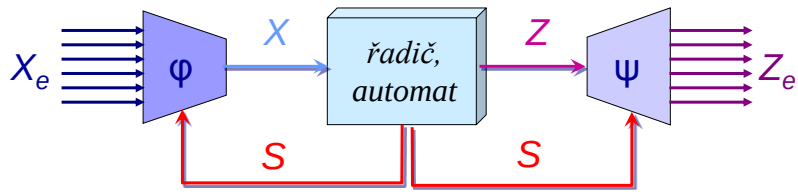
- *varianta:* **autonomní řadič** – vstupy ***X***={ ∅ }



⇨

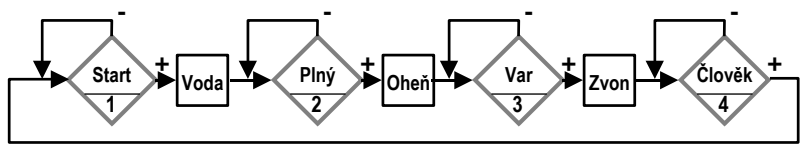
Kódování vstupů a výstupů řadiče

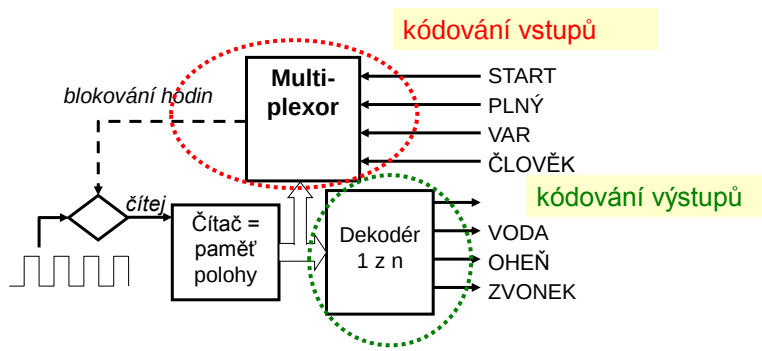
- ***Zpravidla nereagujeme na jakoukoliv kombinaci vstupů, ale na jejich výběr závislý na okamžitém stavu*** - *provádíme tak kódováním externích vstupů X_e a výstupů Z_e*
  - φ: $X_e \times S \rightarrow X$, *kde zpravidla* $|X_e| > |X|$
  - ψ: $Z \rightarrow Z_e$, *kde obvykle platí* $m < n$; $z^m \in Z$, $z_e^n \in Z_e$
- *Vhodně navržená zobrazení zjednoduší návrh řadiče/automatu*



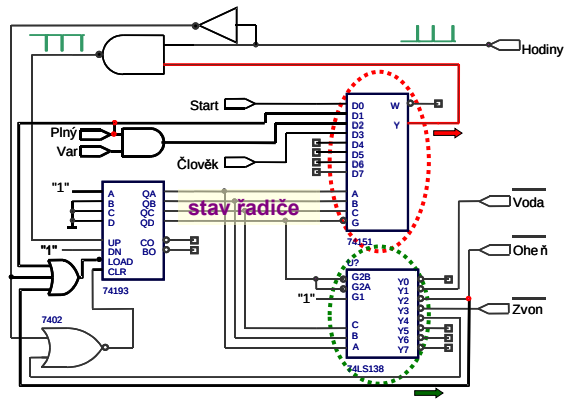
U automatu typu Moore **ω:** S -> Z

Kódování u podmínkového řadiče





LSI řadič



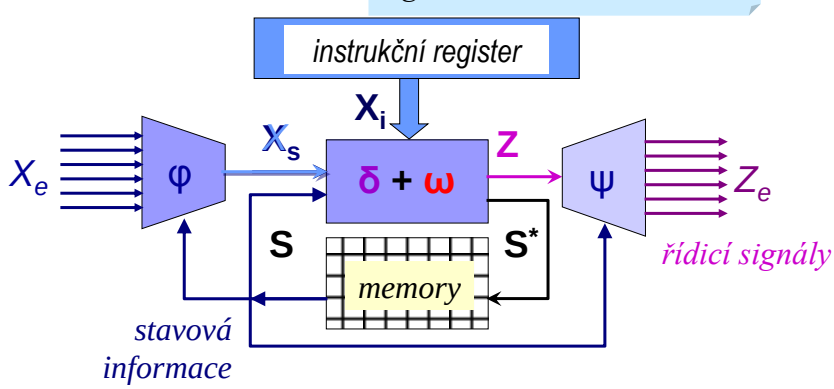
kódování vstupů

kódování výstupů

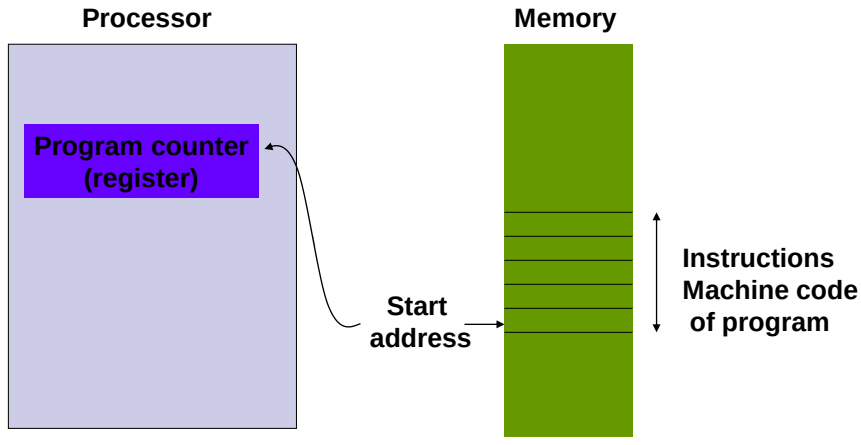
Řadič s instrukčním registrem

- Vstupy $X = X_s \cup X_i$

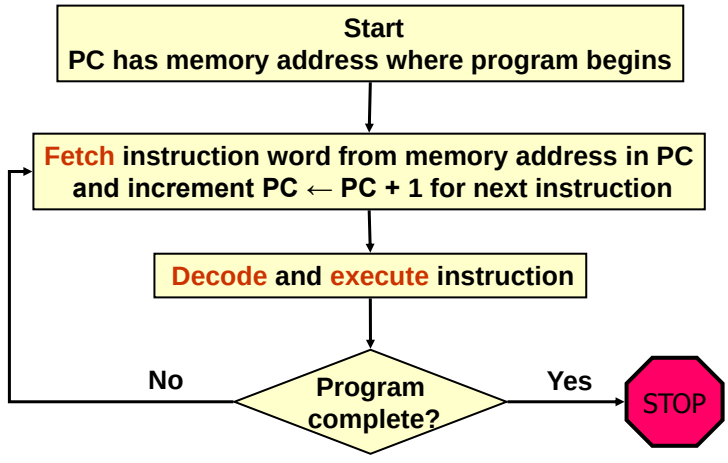
Část vstupů modifikuje algoritmus řadiče



Program



How Does It Run?

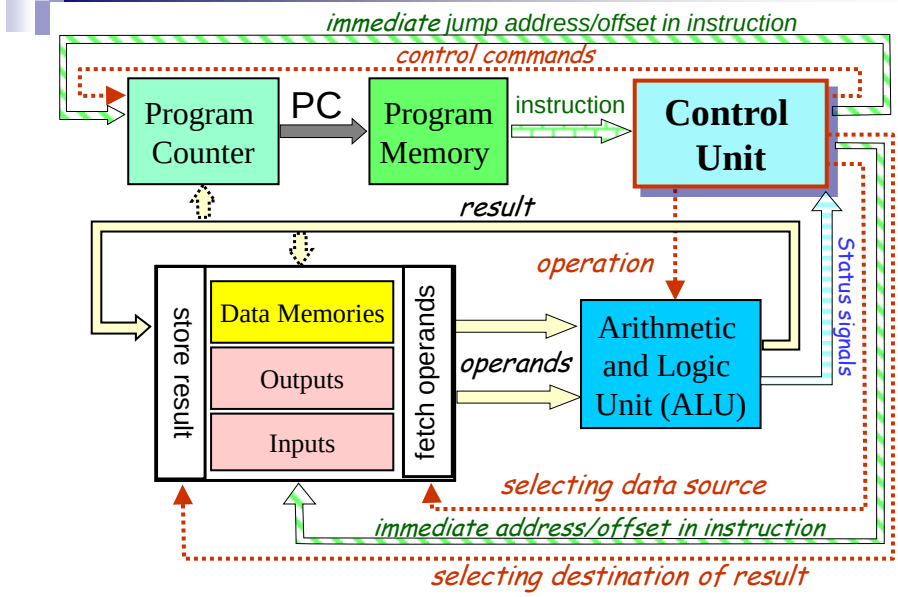


Datapath and Control

- Datapath: Memory, registers, adders, ALU, and communication buses. Each step (fetch, decode, execute) requires communication (data transfer) paths between memory, registers and ALU.
- Control: Datapath for each step is set up by control signals that set up dataflow directions on communication buses and select ALU and memory functions. Control signals are generated by a **control unit** consisting of one or more finite-state machines.

[Nelson: Computer Architecture and Design, Auburn 2008]

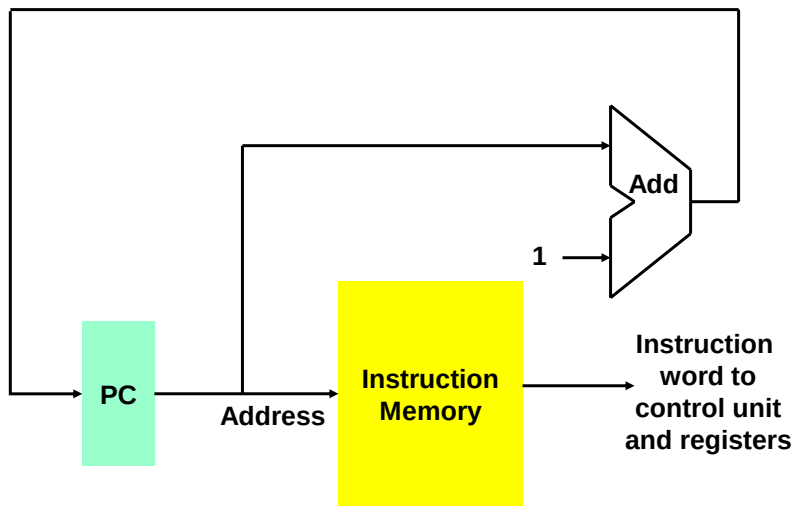
Model of Simple Digital Computer



Control Unit

- The control unit connects programs with the datapath.
  - It converts program instructions into control words for the datapath,
  - It executes program instructions in the correct sequence.
  - It generates the “constant” input for the datapath.
- The datapath also sends state information back to the control unit. For instance, the ALU status bits V, C, N, Z (overflow, carry, negative, zero) can be inspected by branch instructions to alter a program’s control flow.

PC - Program Counter

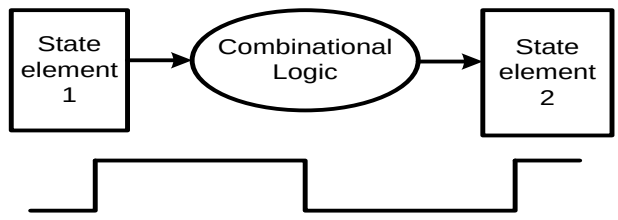


RISC Computer

RISC is an acronym for *reduced instruction set computer*, as contrasted with *complex instruction set computer*, or CISC. Modern computers are typically blends of these two design styles.

1. Load/store memory access
2. Single instruction length
3. Only elementary operations
4. Operations require single pass through pipeline

RISC Single Cycle Implementation

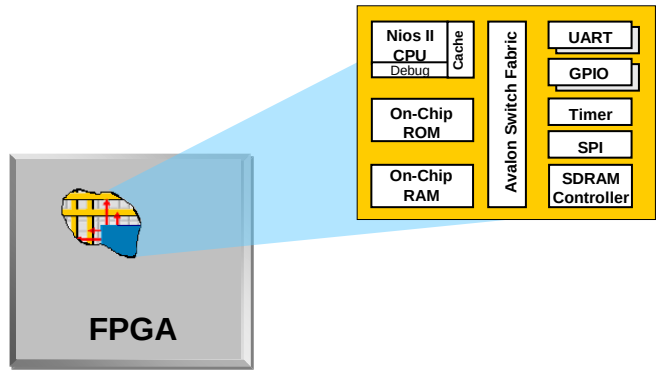


Clock Cycle

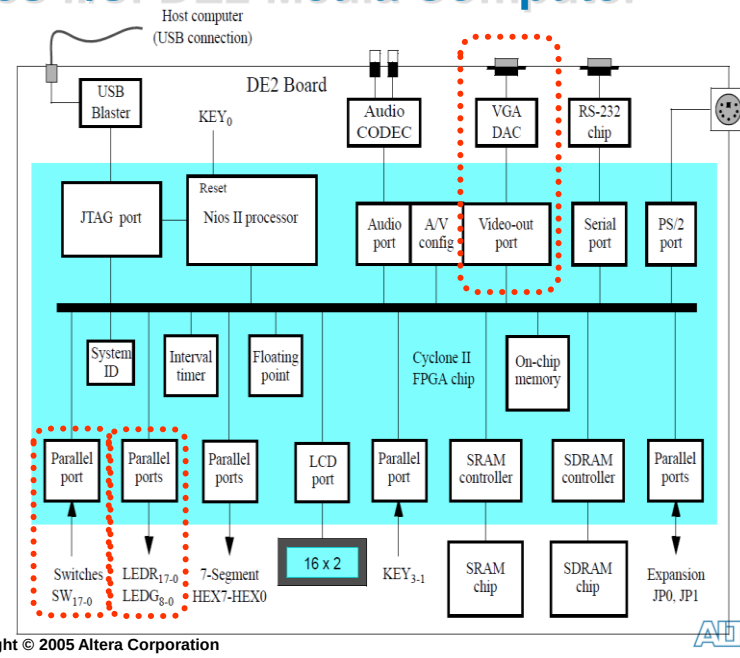
- **Typical execution:**
  - **read contents of some state elements,**
  - **send values through some combinational logic**
  - **write results to one or more state elements**
- **Using a clock signal for synchronization**
- **Edge triggered methodology**

RISC Nios II

- Altera's Second Generation Soft-Core 32 Bit RISC Microprocessor
  - Nios II Plus All Peripherals Written In HDL
  - Can Be Targeted For All Altera FPGAs
  - Synthesis Using Quartus II Integrated Synthesis



Nios II/s: DE2 Media Computer



Zadání NIOS úlohy

vícenásobné hodiny

Více hodin

```
--Syntakticky správný kód!
library ieee; use ieee.std_logic_1164.all;

entity multiEdge is port  (SW_0, SW_1, SW_2 : in std_logic;
                           LEDR_0 : buffer std_logic);
end entity;
architecture rtl of multiEdge is
 begin
   process (SW_0, SW_1, SW_2)
   begin
        if rising_edge(SW_0) then LEDR_0<= not LEDR_0;  end if;
        if rising_edge(SW_1) then LEDR_0<= not LEDR_0;  end if;
        if rising_edge(SW_2) then LEDR_0<= not LEDR_0;  end if;
   end process;
end rtl;
```

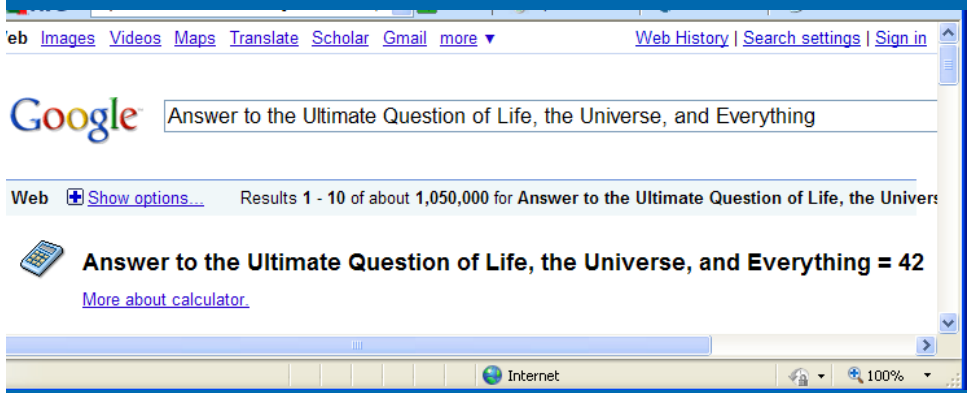
Error (10820): Netlist error at multiEdge.vhd(18):
can't infer register for LEDR_0 because its behavior
depends on the edges of multiple distinct clocks

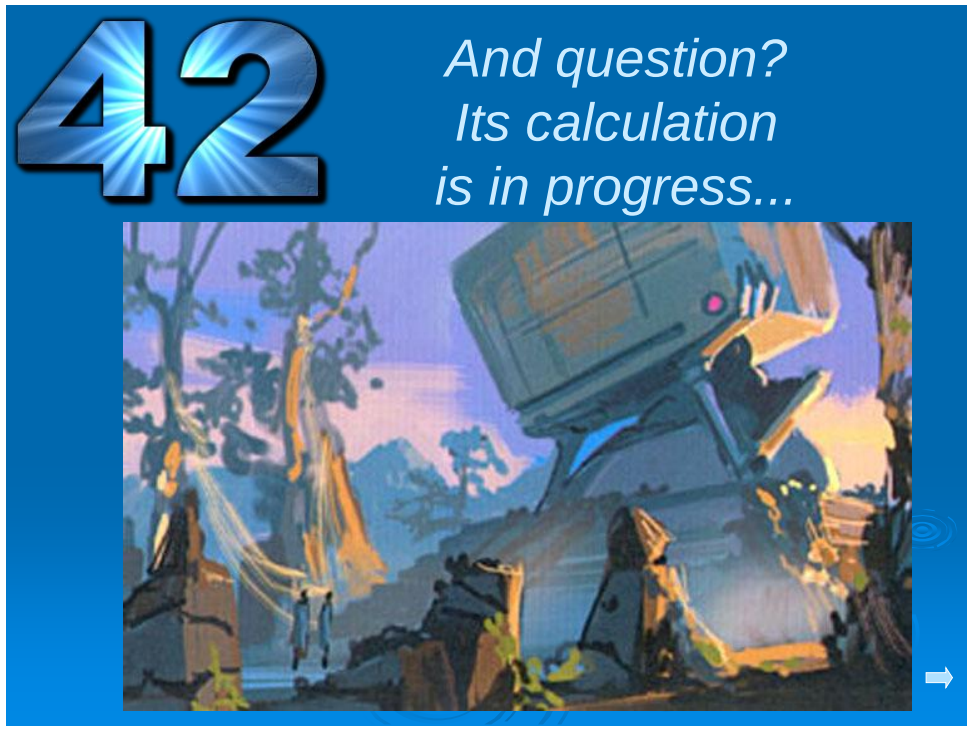
Obvod si tedy vyřešíme superprocesorem!

Základy RISC procesoru NIOS objasníme pomocí jednoduššího procesoru - navrhneme si svůj

1-bitový 1-jádrový RISC procesor s 8 instrukcemi schopný adresovat dokonce až 32 bitů paměti !
=1+1+9+32=

If You Didn't Know ***Google Knows***




42
And question?
Its calculation
is in progress...

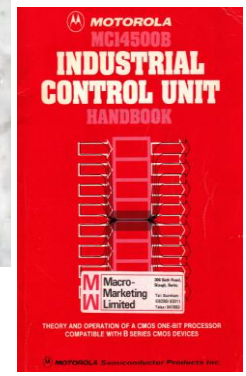

2 Rules of 42 Processor Design
DON'T PANIC
Always Know
Where Your
Towel Is
TOWEL DAY • MAY 25

Algorithm of 42

- Desing 42

1 bits processors?

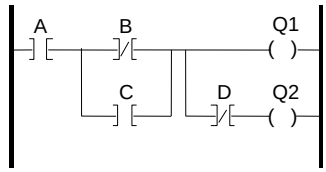
- logic control in industry



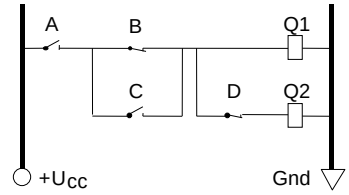
[http://www.motorola.com/]

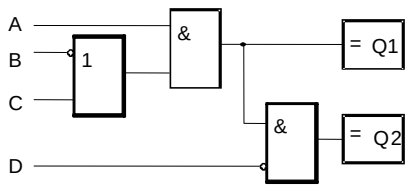
Industrial Programming Languages

Ladder

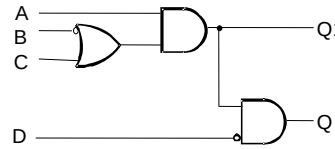


Relay





Logical Diagram



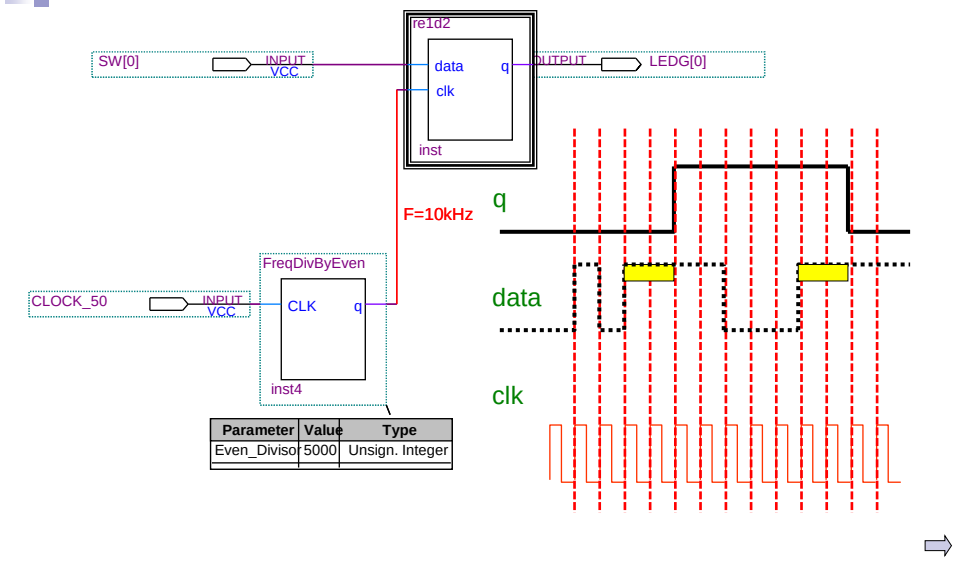
logic circuits

We begin by programming a debug version...

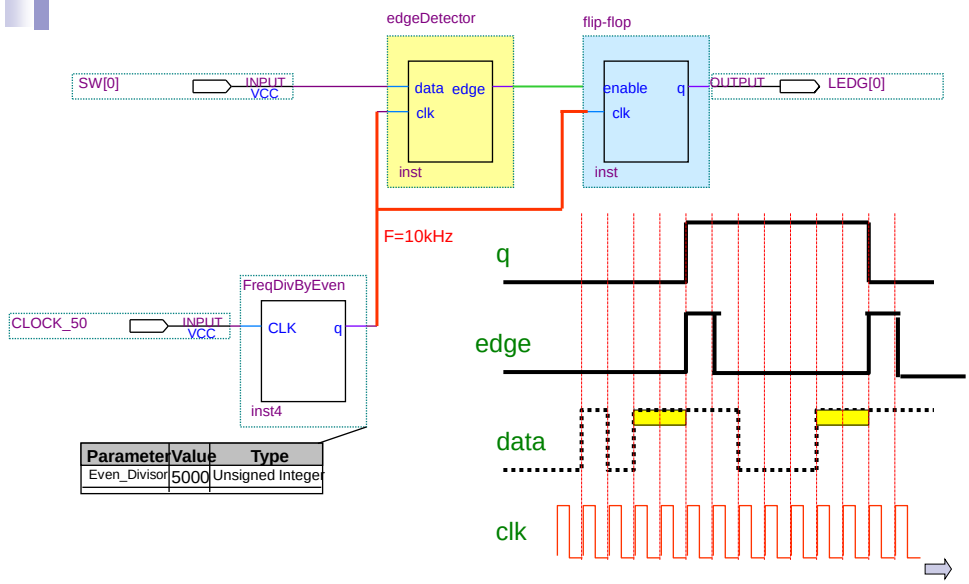
```
library ieee; use ieee.std_logic_1164.all;

entity debug1bit is port  (SW_0 : in std_logic;
                           LEDR_0 : buffer std_logic);
end entity;
architecture rtl of multiEdge is
 begin
   process (SW_0)
   begin
        if rising_edge(SW_0) then LEDR_0<= not LEDR_0;
   end if;
end process;
end rtl;
```

⇨

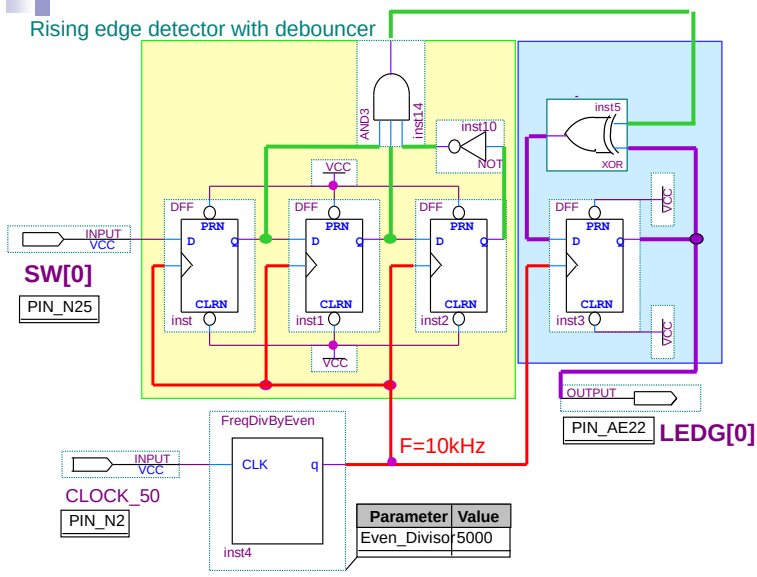
2bit Debouncer from 4th lecture



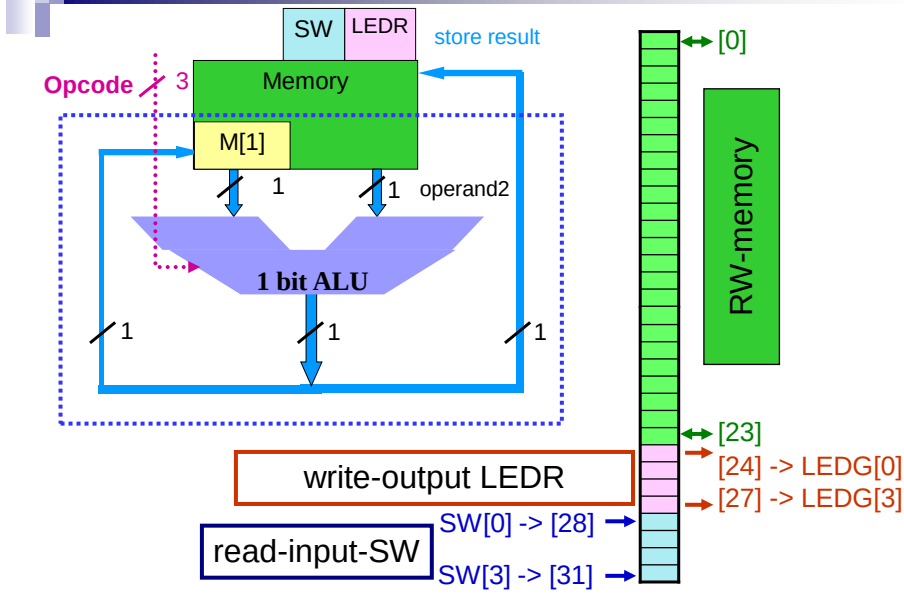
We add LED flip-flop



Hardware Analogy - Flip-Flop LED on Rising edge2



Our 1bit Processor ALU

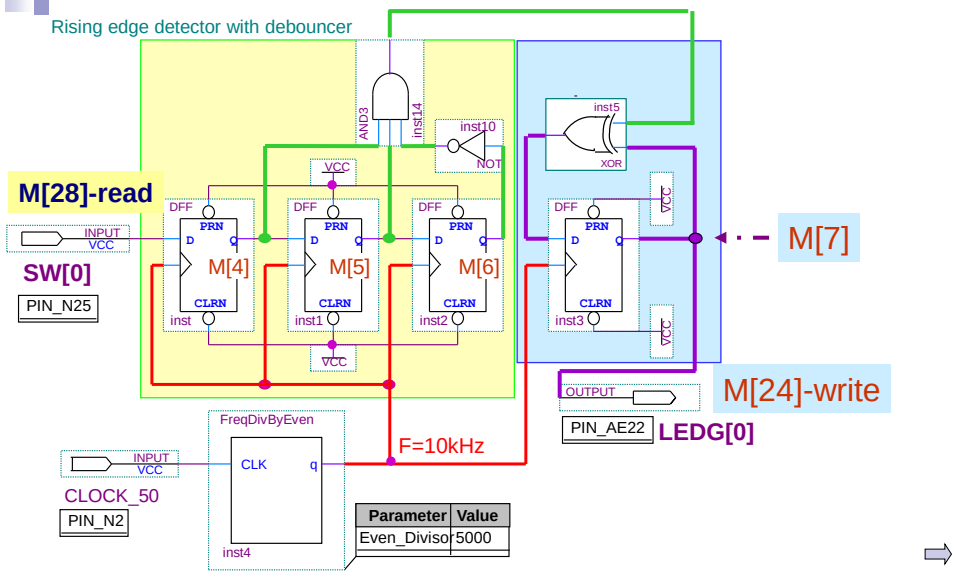


Accumulator

…A 1-operand machine, or a CPU with *accumulator-based architecture*, is a kind of CPU where, although it may have several registers, the **CPU mostly stores the results of calculations in one special register, typically called "the accumulator".** Many microcontrollers still popular as of 2010 (such as the 68HC12, the PICmicro, the 8051 and several others) are basically accumulator machines…

[http://en.wikipedia.org/wiki/Accumulator_(computing)]

Hardware Analogy - Flip-Flop LED on Rising edge2

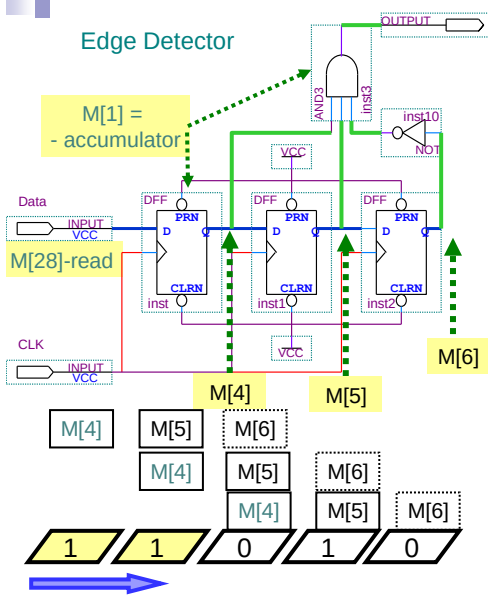


Usage of Memory

M[0]	**0**
M[1]	accumulator
M[2]	
M[3]	
M[4]	sw0_t0
M[5]	sw0_t1
M[6]	sw0_t2
M[7]	led_mem
M[8]	
M[9]	
M[10]	
M[11]	
M[12]	
M[13]	
M[14]	
M[15]	

M[16]	
M[17]	
M[18]	
M[19]	
M[20]	
M[21]	
M[22]	
M[23]	
M[24]	LEDG[0]
M[25]	
M[26]	
M[27]	
M[28]	SW[0]
M[29]	
M[30]	
M[31]	

Code for 1bit Processor

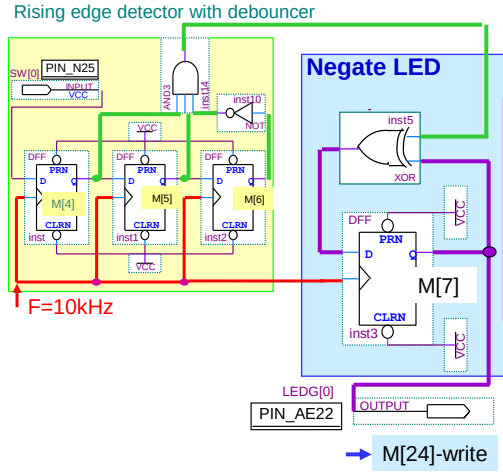



```
--IX0-REDGE...
M[1] ← not M[6]
M[1] ←M[1] and M[5]
M[1] ←M[1] and M[4]
```

```
--IX0-shift reg...
M[1] ← M[5]
M[1] → M[6]
M[1]  ← M[4]
M[1] → M[5]
M[1]  ← M[28] -- SW[0]
M[1] → M[4]
```

Code for 1bit Processor

Rising edge detector with debouncer



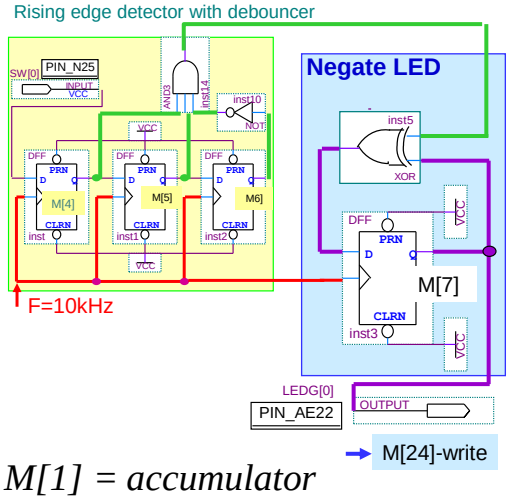
-- negate LED

M[1] ← not M[7]
M[1] → M[7]
M[1] → M[24]

⇨

Code for 1bit Processor

Rising edge detector with debouncer



LABEL0:--IX0-shift reg...
M[1] ← M[5]; M[1]→ M[6]
M[1] ← M[4]; M[1]→ M[5]
M[1] ← M[28]; M[1]→ M[4

--IX0-REDGE...
M[1] ← M[6]; M[1] ← not M[1]
M[1] ← M[1] and M[5]
M[1] ← M[1] and M[4]

if M[1]<>'0' then PC←LABEL1

-- negate LED
M[1] ← M[7]; M[1]←not M[1]
M[1]→ M[7]
M[1] → M[24]

LABEL1: PC ← LABEL0

M[1] = accumulator
PC= program counter

⇨

Usage of memory and registers

M[0]	r0	**0**
M[1]	r1	**acc – accumulator / at - assembler temporary**
M[2]	r2	
M[3]	r3	
M[4]	r4	sw0_t0
M[5]	r5	sw0_t1
M[6]	r6	sw0_t2
M[7]	r7	led_mem
M[8]	r8	
M[9]	r9	
M[10]	r10	
M[11]	r11	
M[12]	r12	
M[13]	r13	
M[14]	r14	
M[15]	r15	

M[16]	
M[17]	
M[18]	
M[19]	
M[20]	
M[21]	
M[22]	
M[23]	
M[24]	LEDG[0]
M[25]	
M[26]	
M[27]	
M[28]	SW[0]
M[29]	
M[30]	
M[31]	

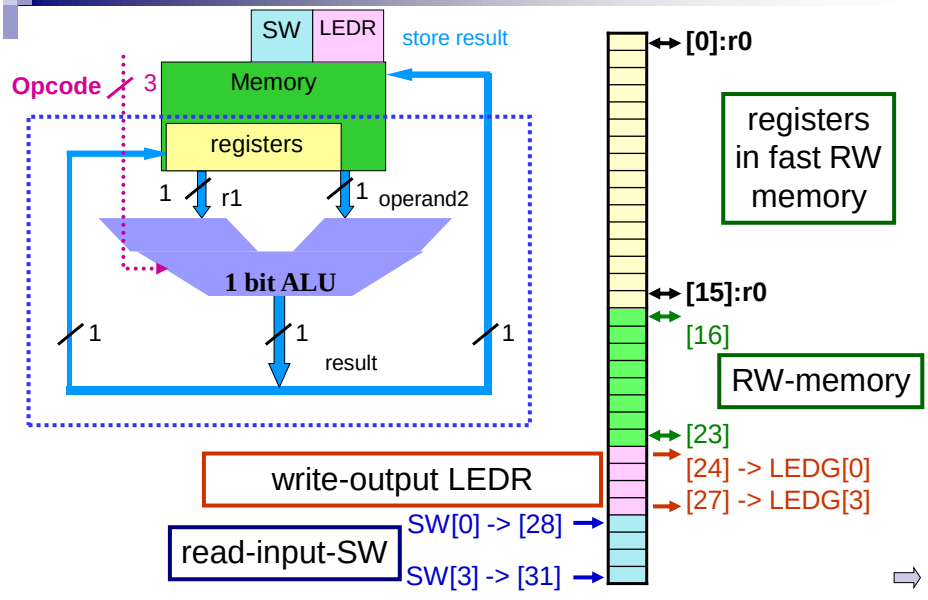
Registers

In computer architecture, a **processor register** is a small amount of storage available as part of a CPU or other digital processor. Such registers are (typically) addressed by mechanisms other than main memory and can be accessed more quickly. Almost all computers, load-store architecture or not, load data from a larger memory into registers where it is used for arithmetic, manipulated, or tested, by some machine instruction...

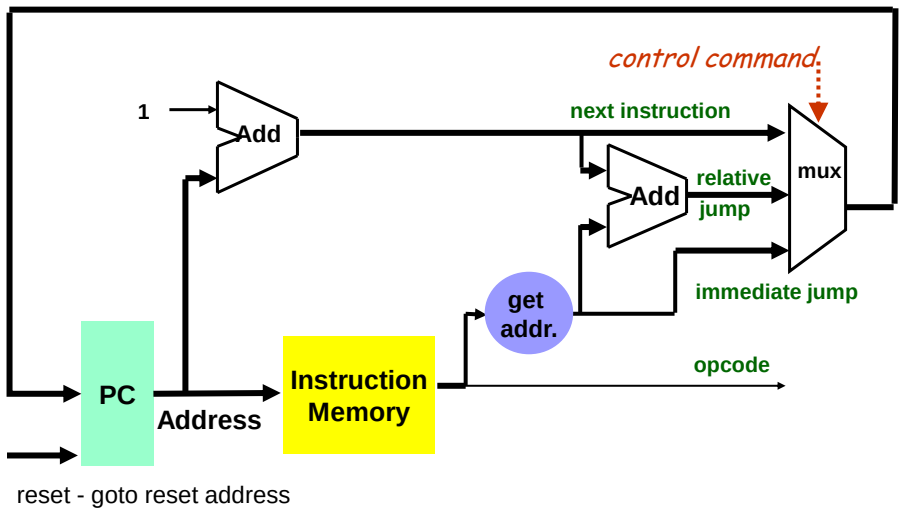
[See: http://en.wikipedia.org/wiki/Processor_register]

[Cz: viz http://cs.wikipedia.org/wiki/Registr_procesoru]

Our 1bit Processor ALU



PC - Program Counter with Jumps



Jumps addressing modes

- |jump| address | (Effective PC address = address) The effective address for an absolute instruction address is the address parameter itself with no modifications.
- |jump| offset | jump relative (Effective PC address = next instruction address + offset, offset may be negative) The effective address for a PC-relative instruction address is the offset parameter added to the address of the next instruction. This offset is usually signed to allow reference to code both before and after the instruction.
- This is particularly useful in connection with jumps, because typical jumps are to nearby instructions (in a high-level language most **if** or **while** statements are reasonably short). Measurements of actual programs suggest that an 8 or 10 bit offset is large enough for some 90% of conditional jumps.3
- Another advantage of program-relative addressing is that the code may be position-independent, i.e. it can be loaded anywhere in memory without the need to adjust any addresses.

See http://en.wikipedia.org/wiki/Addressing_mode

Instruction of 1bit Processor ALU

LD	r1 ← M[address];
ST	r1 → M[address];
AND	r1 ← r1 and M[address]
OR	r1 ← r1 or M[address]
BNE	PC ← PC+1+address if r1 ='1'
JMPI	PC ← address
NOT	r1 ← not r1
NOP	nic

Similarly with NIOS II assembler, we will write :

- M[address] as **address(r0)**, which determines M[address+ r0], r0 always =0
- In NIOS, missing 'NOT' instruction can be calculated as NOR instruction
  r1 ← (r1 nor r0) ; r0 always =0

Instruction 1bit Processor ALU

LD	r1, address(r0)	**ST**	r1, address(r0)
AND	r1, address(r0)	**OR**	r1, address(r0)
BNE	address	**JMPI**	address
NOT		**NOP**	

⇨

Pseudoinstruction MOV

In the next slides, **we use more transparent MOV shortcut that is similar to NIOS II pseudo-instruction MOV**

our MOV

MOV rN, rM ≡ LD r1, rM; ST r1, rN

NIOS II MOV

MOV rN, rM ≡ ADD rM,rN,r0 ; ≡ rN←rM+0

Note: In NIOS, pseudo-instructions are used in assembly source code like regular assembly instructions. Each pseudo-instruction is implemented at the machine level using an equivalent instruction, or a sequence of several instructions

[NIOS II Reference Handbook (file: n2cpu_nii5v1.pdf) page 8-4]

⇨

Program for 42

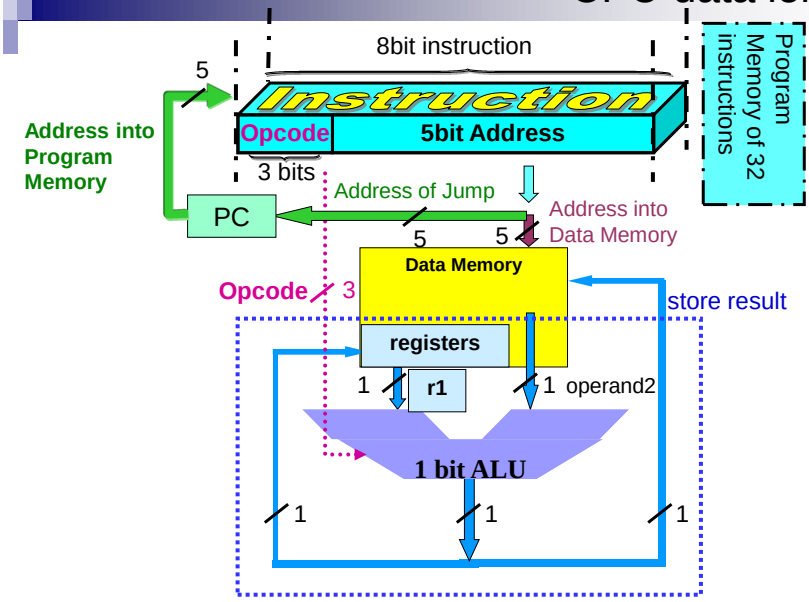
LABEL0:--IX0-shift reg...
MOV r5, r6;
MOV r4, r5;
LD r1, 28(r0); ST r1, r4;
	LABEL0:--IX0-shift reg...
	M[1] ← M[5]; M[1]→ M[6]
	M[1] ← M[4]; M[1]→ M[5]
	M[1] ← M[28]; M[1]→ M[4]

--IX0-REDGE... LD r1, r6; NOT; AND r1, r5 AND r1, r4	--IX0-REDGE... M[1] ← M[6]; M[1] ← not M[1] M[1] ← M[1] and M[5] M[1] ← M[1] and M[4]
BNE LABEL1	**if M[1]<>'0' then PC⟵LABEL1**
-- negate LED **LD r1, r7; NOT;** **ST r1, r7;** **ST r1, 24(r0);**	-- negate LED **M[1] ← M[7]; M[1]⟵not M[1]** **M[1]→ M[7]** **M[1] → M[24]**
LABEL1: JMPI LABEL0	**LABEL1: PC ⟵ LABEL0**

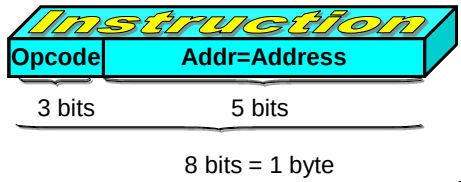
Algorithm of 42

- Designing Instruction Set

CPU data length



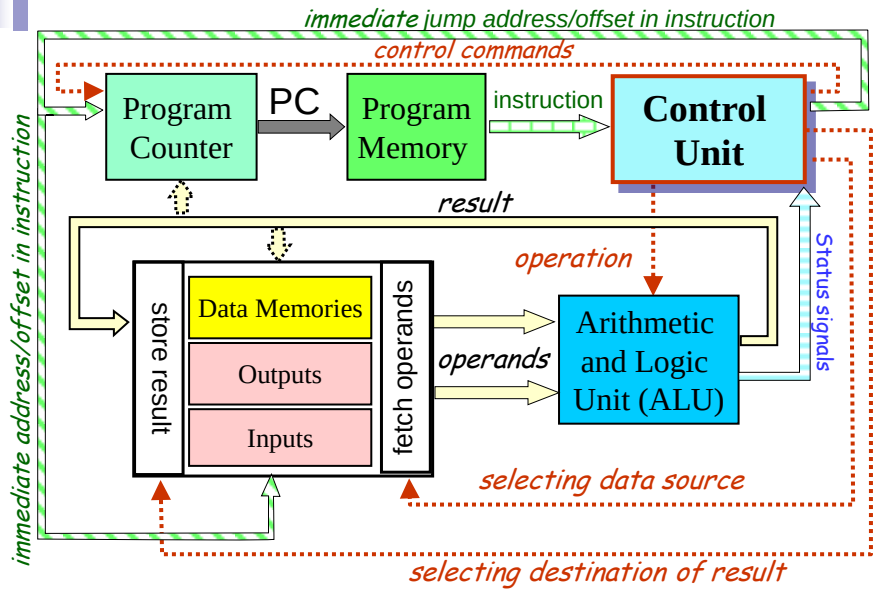
Designing Instruction - Opcodes



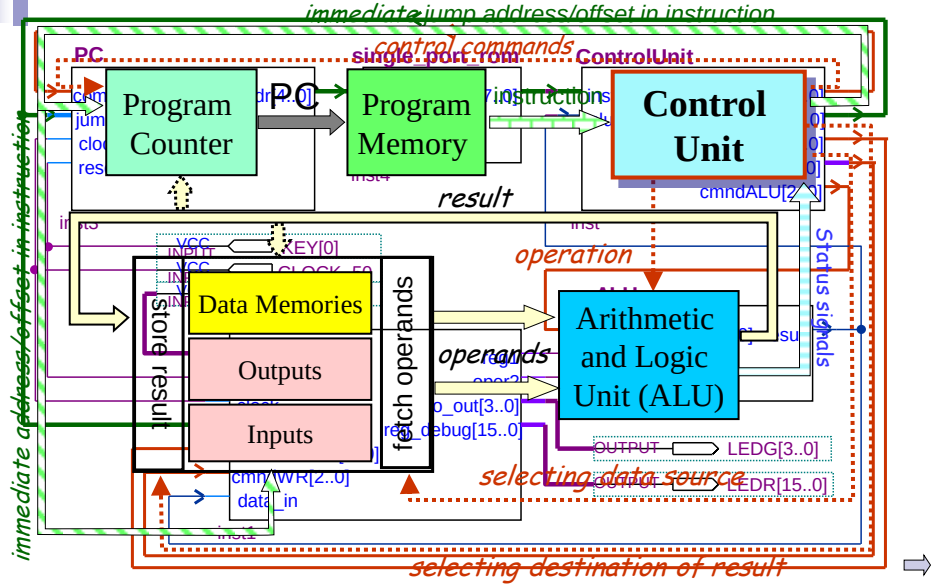
LD	r1 ← M[addr]
ST	r1 → M[addr]
JMPI	PC ← addr
BNE	PC ← PC+1+addrif r1='1'
AND, OR	r1 ← r1 and/or M[addr]
NOT	r1 ← not r1
NOP	no operation

8 instructions

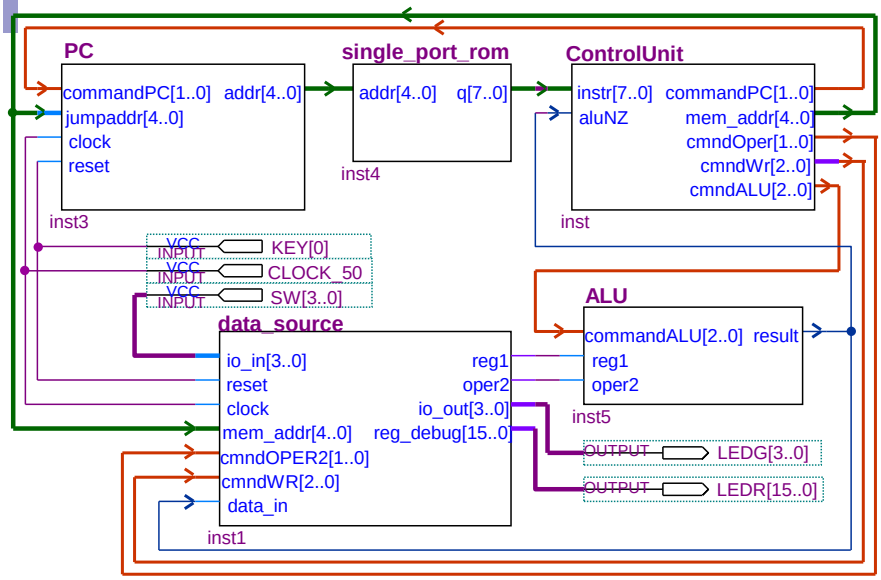
Recall: Simple Digital Computer



Fading animation: Simple Digital Computer



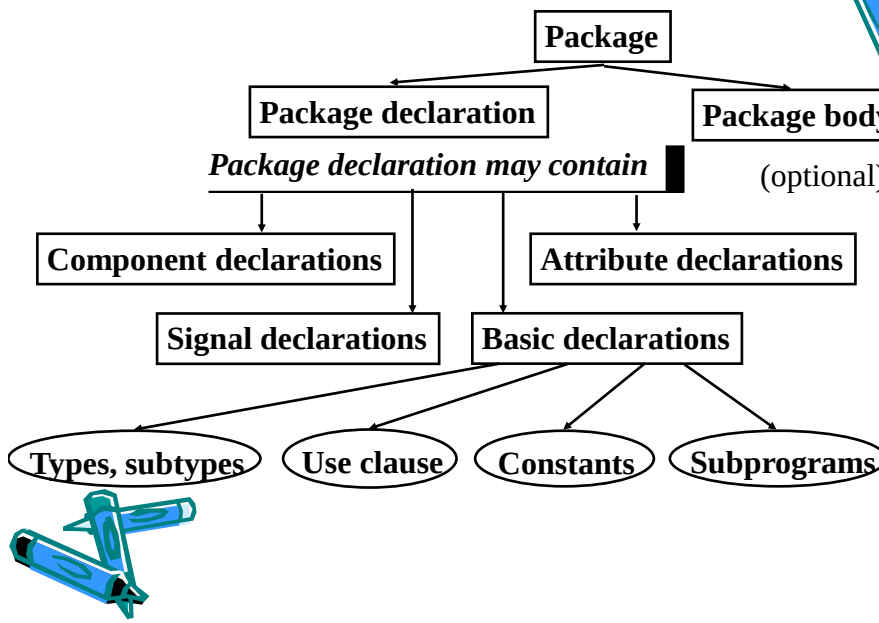
1 bit CPU



Algorithm of 42

- VHDL code

VHDL Package



Example of Package

```
package my_package is
        type binary is (on, off);
        constant pi : real : = 3.14;
        procedure add_bits3 (signal a, b, en : in bit;
        signal temp_result, temp_carry : out bit);
end my_package;
```

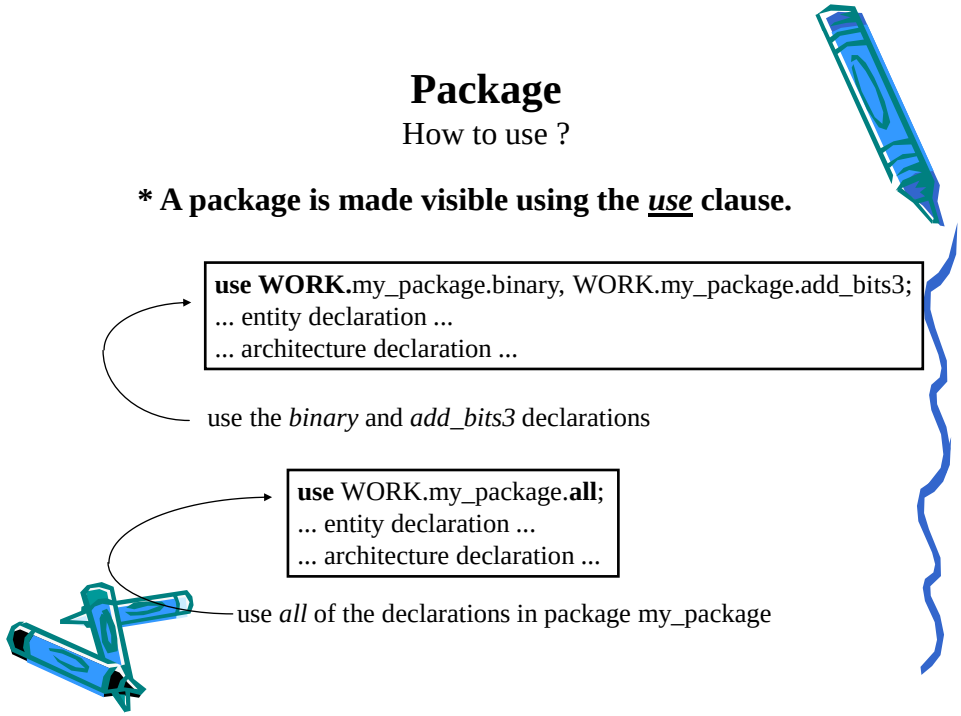
```
package body my_package is
    procedure add_bits3 (signal a, b, en : in bit;
                         signal temp_result, temp_carry : out bit) is
    begin
        temp_result <= (a xor b) and en;
        temp_carry <= a and b and en;
    end add_bits3;
end my_package;
```

The package body contains *subprogram bodies* and other declarations not intended for use by other VHDL entities

Package

How to use ?

*** A package is made visible using the *use* clause.**



Codes of Instruction of 1bit Processor 42

"001"+address	**LD**	r1 ← M[address];
"010"**+address**	ST	r1 → M[address];
"100"**+address**	AND	r1 ← r1 and M[address]
"101"**+address**	OR	r1 ← r1 or M[address]
"110"**+address**	BNE	PC ← PC+1+address if r1 ='1'
"111"**+address**	JMPI	PC ← address
"000"+"00001"	NOT	r1 ← not r1
"000"+"00000"	NOP	nic

Note:
MOV – assembler pseudo instruction
MOV rN, rM ≡ LD r1, rM; ST r1, rN

.

Our Package – Declaration Section

```
library IEEE;use IEEE.std_logic_1164.all;
 package cpu_definition is
  subtype t_word is std_logic_vector(7 downto 0);
  subtype t_opcode is std_logic_vector(2 downto 0);
  subtype t_address is std_logic_vector(4 downto 0);

  constant OPNOP: t_opcode:="000";   constant OPNOT: t_word:=OPNOP & "00001";
  constant OPLOAD: t_opcode:="010";  constant OPSTORE: t_opcode:="011";
  constant OPAND: t_opcode:="100";  constant OPOR: t_opcode:="101";
  constant OPJMP: t_opcode:="110";  constant OPBRNE: t_opcode:="111";
 -- assembler function Declaration
   type t_register is (r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15);

 function asmLOAD(operand:t_ioadress) return t_word;
 function asmLOAD(operand:t_register) return t_word;

 --followindeclarations……
 end package cpu_definition;
```

⇨

Our Package – Definitions

```
 package body cpu_definition is
   -- internal functions
function decodeOperand(addr:t_ioadress) return t_address is
    begin   return std_logic_vector(to_unsigned(addr,5)); end;
function decodeOperand(regdef:t_register) return t_address is
   begin  return '0' &std_logic_vector(to_unsigned(t_register'POS(regdef),4)); end;
 -- implementations of declared functions
function asmLOAD(operand:t_ioadress) return t_word is
   begin   return OPLOAD & decodeOperand(operand); end;
function asmLOAD(operand:t_register) return t_word is
   begin return OPLOAD & decodeOperand(operand); return result; end;
 -- following definitions.....

end package body cpu_definition;
```

PC.vhld

```
architecture rtl of PC is
signal count : t_address :=(others => '0');
begin
process (clock, reset) is  begin
   if reset = '0' then count <= (others => '0');
   elsif rising_edge(clock) then
     case commandPC is
       when pc_Jmp => count<=jumpaddr;
       when pc_Br =>
          if jumpaddr(jumpaddr'HIGH)='1' then
           -- count <= count+1-jumpaddr -- not X=-X-1
            count <= count-(not jumpaddr(jumpaddr'HIGH-1 downto 0));
            else  count<=count+1+jumpaddr; end if;
       when others => count<= count + 1;
       end case;
  end if;
end process;
addr<=count; end architecture rtl;
```

PC

commandPC[1..0] addr[4..0]
jumpaddr[4..0]
clock
reset

ROM - VHDL

```
library ieee;use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;use IEEE.std_logic_unsigned.all;
use WORK.cpu_definition.all;

entity single_port_rom is  port (addr: in t_address;q: out t_word);
end entity;

architecture rtl of single_port_rom is
type memory_t is array(0 to 31) of t_word;
constant rom : memory_t := (
    asmLOAD(r5),  --       0.
    asmSTORE(r6),   --    1.
-- .....................
    others => (others=>'0')
                        );
begin
     q <= rom(conv_integer(addr));
end rtl;
```

ROM VHDL

```
constant rom :
    memory_t := (

    asmLOAD(r5),     --0
    asmSTORE(r6),    --1
    asmLOAD(r4),     --2
    asmSTORE(r5),    --3
    asmLOAD(28),     --4
    asmSTORE(r4),    --5
    asmLOAD(r6),     --6
    asmNOT,          --7
    asmAND(r5),      --8
    asmAND(r4),      --9
    asmNOT,          --10
    asmBNE(4),       --11

    asmLOAD(r7),     --12
    asmNOT,          --13
    asmSTORE(24),    --14
    asmSTORE(r7),    --15
    asmJMP(0),       --16

    others => (others=>'0')   );
```

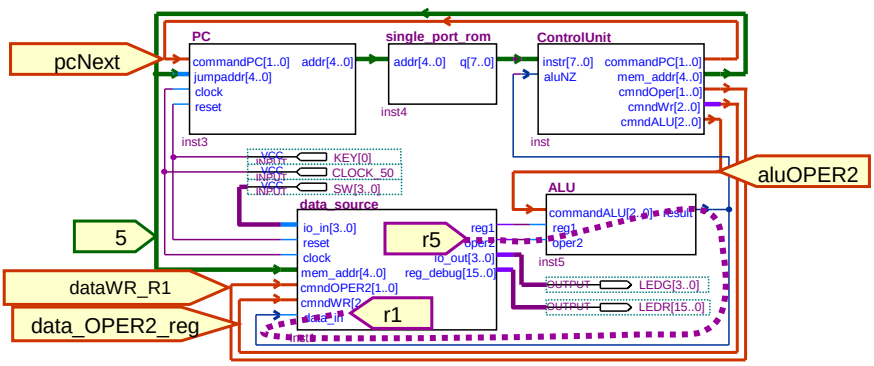
ALU

```
library IEEE; use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;use IEEE.std_logic_unsigned.all;
use WORK.cpu_definition.all;
entity ALU is port ( commandALU : t_commandALU; reg1, oper2 : in std_logic;
        result : out std_logic);
end entity ALU;
architecture rtl of ALU is
begin  with commandALU select
                result <= reg1 when aluReg1,
                (not reg1) when aluNotReg1,
                oper2 when aluOPER2,
                (reg1 and oper2) when aluAND,
                (reg1 or oper2) when aluOR,
                '0'  when others;
end architecture rtl;
```

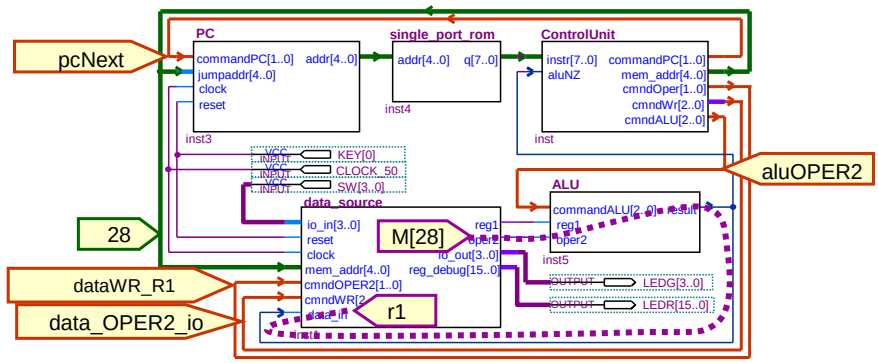
Algorithm of 42

- Function

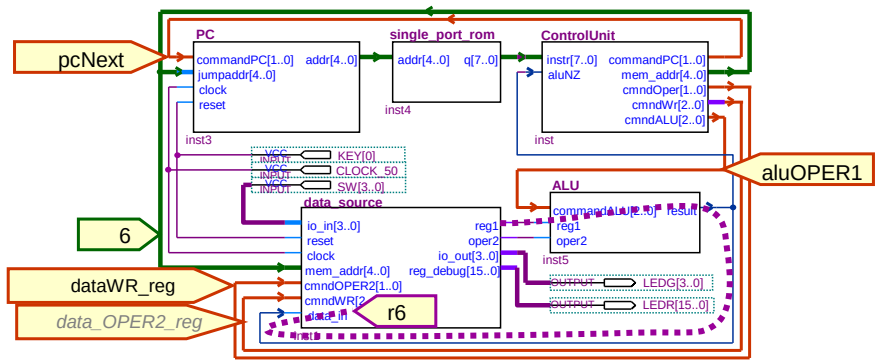
asmLOAD(r5) = LD r1, r5



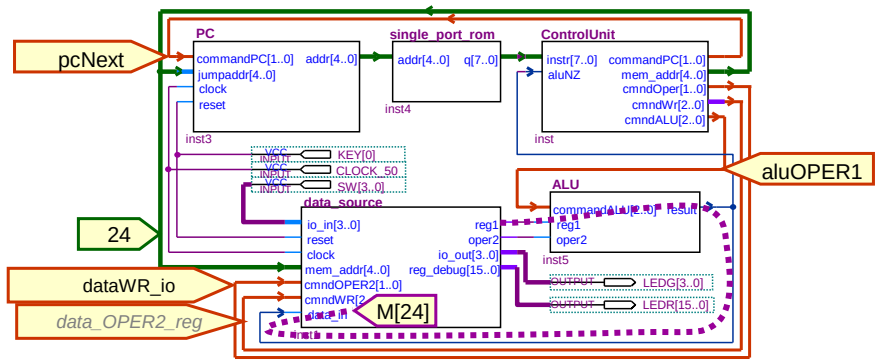
asmLOAD(28) = LD r1,28(r0)



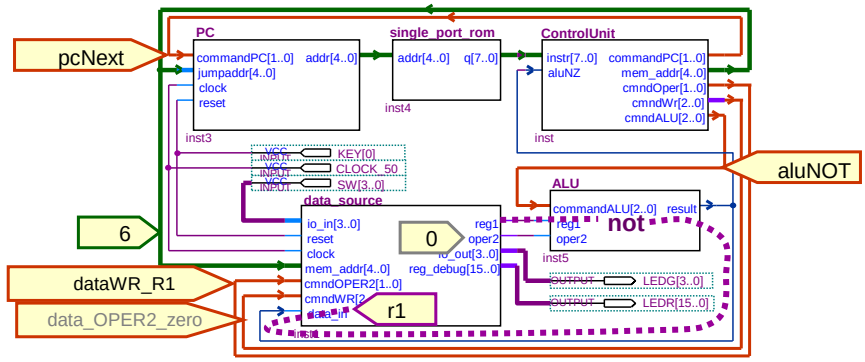
asmSTORE(r6) = ST r1, r6



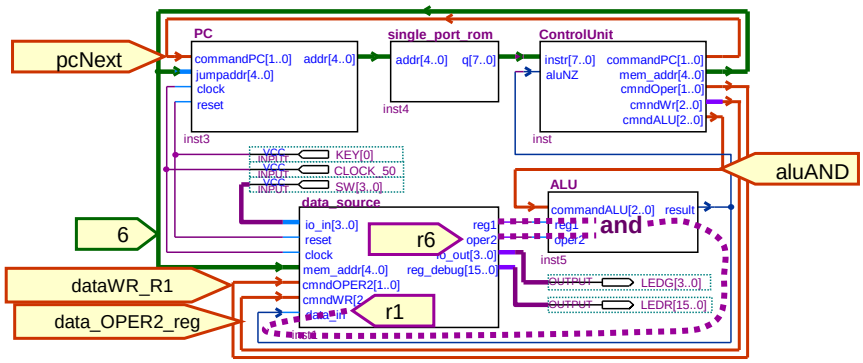
asmSTORE(24) = ST r1, 24(r0)



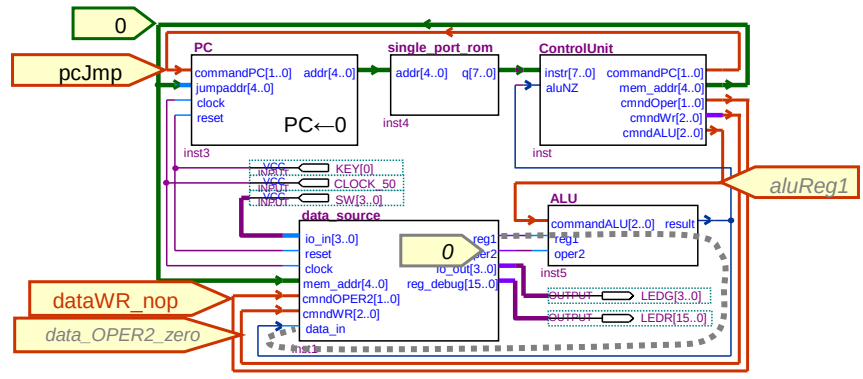
asmNOT = NOT r1



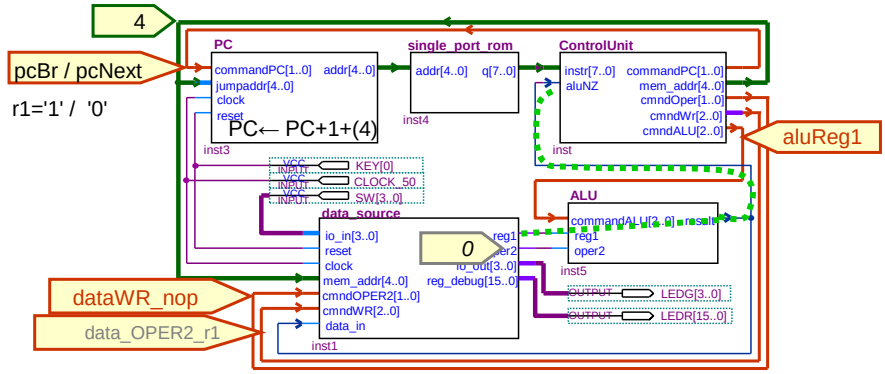
asmAND(r6) = AND r1, r6



asmJMP(r6) = JMPI 0



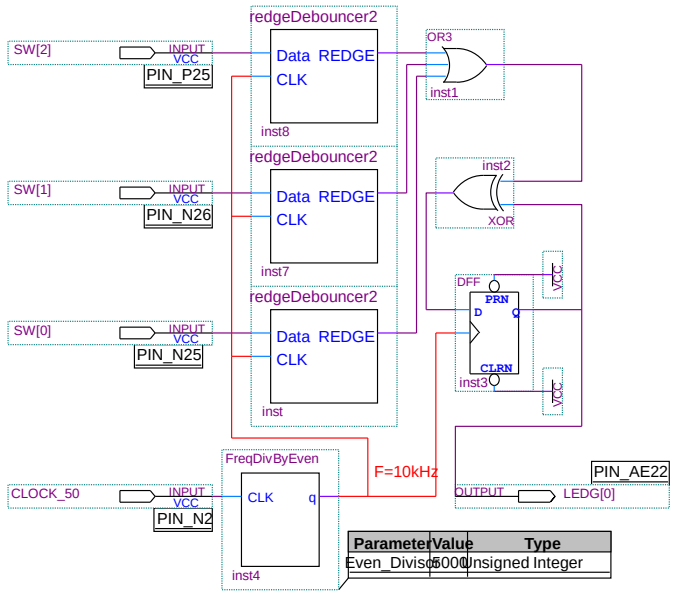
asmBRNZ(4) = BRNZ 4



Algorithm of 42

- Extending memory

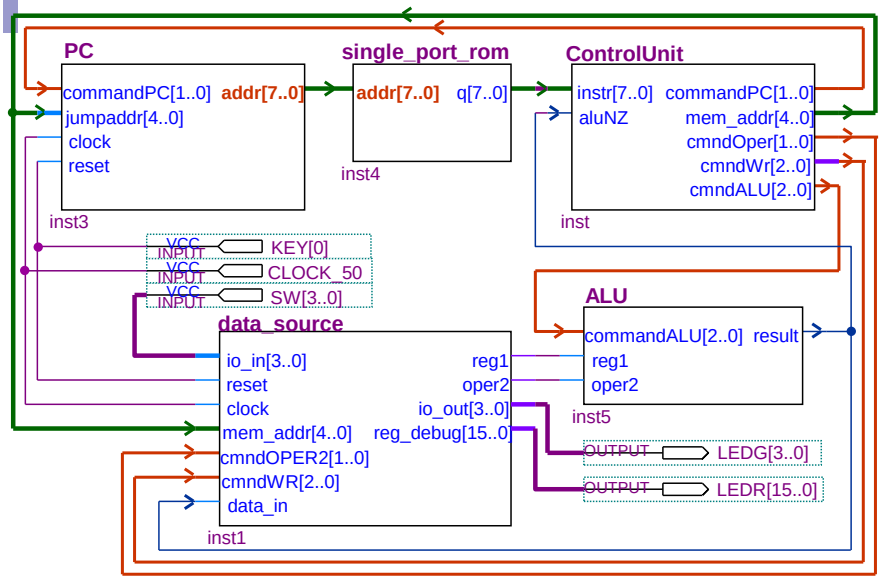
More SW bits with 1 bit Processor



ROM

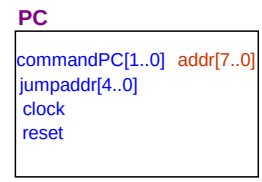
```
constant rom : memory_t := (
    asmLOAD(r5),            -- 0.
    asmSTORE(r6),           --1.
    asmLOAD(r4),            -- 2.
    asmSTORE(r5),           --3.
    asmLOAD(28),            -- 4.
    asmSTORE(r4),           -- 5.

    asmLOAD(r9),            -- 6.
    asmSTORE(r10),          -- 7.
    asmLOAD(r8),            -- 8.
    asmSTORE(r9),           -- 9.
    asmLOAD(29),            -- 10
    asmSTORE(r8),           -- 11.

    asmLOAD(r12),           -- 12.
    asmSTORE(r13),          -- 13.
    asmLOAD(r11),           -- 14.
    asmSTORE(r12),          -- 15.
    asmLOAD(30),            -- 16
    asmSTORE(r11),          -- 17.

    asmLOAD(r6),            -- 18.
    asmNOT,                 -- 19.
    asmAND(r5),             -- 20.
    asmAND(r4),             -- 21.
    asmBNE(11),             -- 22.

    asmLOAD(r10),           -- 23.
    asmNOT,                 -- 24.
    asmAND(r9),             -- 25.
    asmAND(r8),             -- 26.
    asmBNE(6),              -- 27.

    asmLOAD(r13),           --28.
    asmNOT,                 -- 29.
    asmAND(r12),            -- 30.
    asmAND(r11),            -- 31.
    asmNOT,                 -- 32.
    asmBNE(4),              -- 33.

    asmLOAD(r7),            -- 34.
    asmNOT,                 -- 35.
    asmSTORE(24),           -- 36.
    asmSTORE(r7),           -- 37.
    asmJMP(0),              -- 38.

    others => (others=>'0')
                );
```

Extended 1 bit CPU to 256 Instructions



PC of Extended42

```
architecture rtl of PC is
signal count : t_word:=(others => '0');
begin
process (clock, reset) is  begin
  if reset = '0' then count <= (others => '0');
 elsif rising_edge(clock) then
   case commandPC is
     when pc_Jmp => count<=jumpaddr & "000";
     when pc_Br =>
        if jumpaddr(jumpaddr'HIGH)='1' then
         -- count <= count+1-jumpaddr -- not X=-X-1
          count <= count-(not jumpaddr(jumpaddr'HIGH-1 downto 0));
          else  count<=count+1+jumpaddr; end if;
     when others => count<= count + 1;
     end case;
 end if;
end process;
addr<=count; end architecture rtl;
```



JMPI requires now label with 8 bit memory alignment

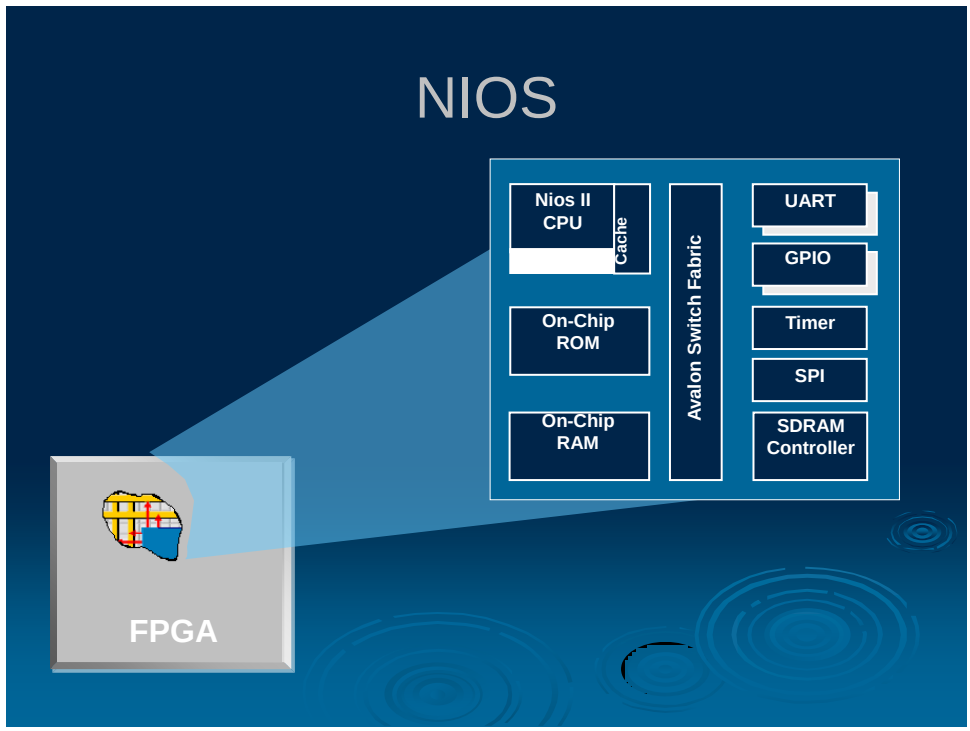
Memory Alignment

A memory address addr, is said to be n-byte aligned when n is a power of two and addr is a multiple of n bytes. In this context a byte is the smallest unit of memory access, i.e. each memory address specifies a different byte. An n-byte aligned address would have log2(n) least-significant zeros when expressed in binary.

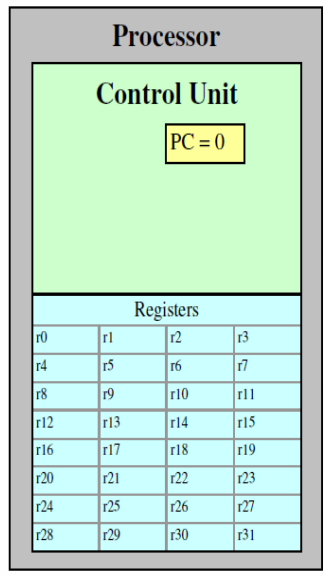
The alternate wording b-bit aligned designates a b/8 byte aligned address (ex. 64-bit aligned is 8 bytes aligned).

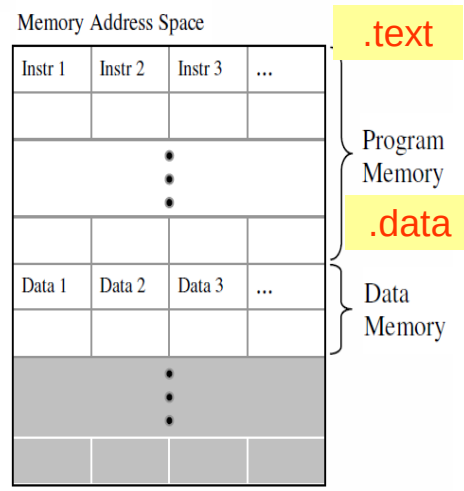
[http://en.wikipedia.org/wiki/Data_structure_alignment]

Jump requires label with 8 byte alignment



NIOS Simplified System





Programming NIOS

in GNU assembler

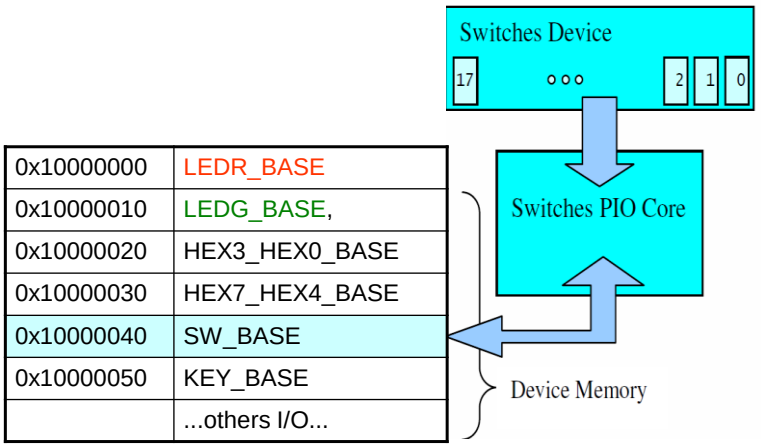
http://tigcc.ticalc.org/doc/gnuasm.html

http://sources.redhat.com/binutils/docs-2.12/as.info/

by Altera monitor – lieratura on DVD

- Doporuceno_Altera_Monitor_Program_Tutorial.pdf
- n2cpu_nii5v1.pdf – instruction set reference

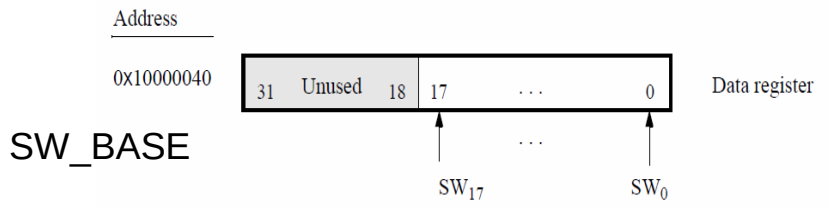
I/O DE2 Basic & Media Computer

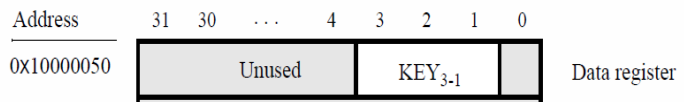


0x10000000	LEDR_BASE
0x10000010	LEDG_BASE,
0x10000020	HEX3_HEX0_BASE
0x10000030	HEX7_HEX4_BASE
0x10000040	SW_BASE
0x10000050	KEY_BASE
	...others I/O...

Literatura on DVD: DE2_Media_Computer.pdf

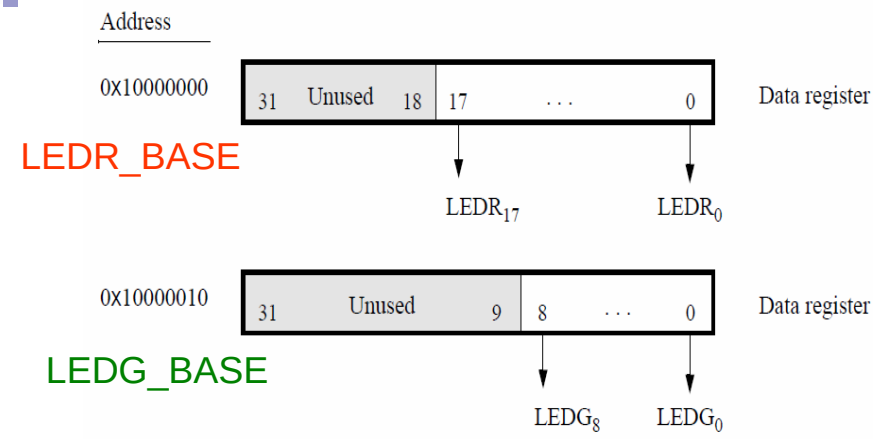
Switch and Push Button





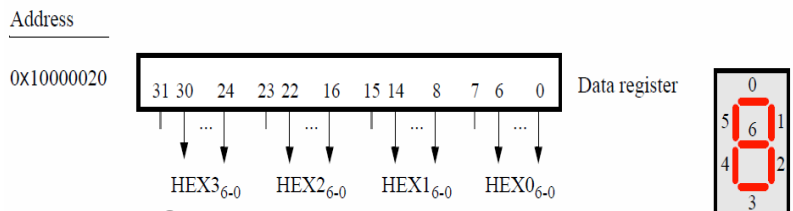
KEY_BASE

LED

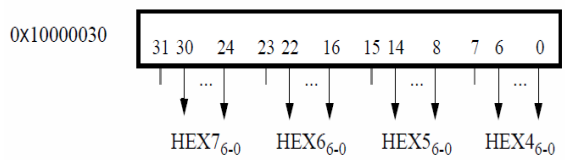


HEX

HEX3_HEX0_BASE



HEX7_HEX4_BASE



NIOS DE2 Media Computer

256 MB: 128 MB, 16 MB, 112 MB

Start-address	End-address	Element	Size
0x00000000	0x007FFFFF	SDRAM	8 MB synchron. DRAM
--			*120 MB*
0x08000000	0x0807FFFF	RAM	512 kB VGA pixel buffer
			15,5 MB
0x09000000	0x09001FFF	On-chip-RAM	8 kB VGA char. buffer
			111,92 MB
0x10000000	0x10000003	LEDR	4 bytes-write-only
0x10000010	0x10000013	LEDG	4 bytes-write-only
0x10000020	0x10000023	HEX3_HEX0	4 bytes-write-only
0x10000030	0x10000033	HEX7_HEX4	4 bytes-write-only
0x10000040	0x10000043	SW	4 bytes-read-only
0x10000050	0x1000005F	KEY_BASE	4 bytes data +12 bytes control
.. other I/O			

Wiki: Memory Mapped IO

Memory-mapped I/O (**MMIO**) and **port I/O** (also called **isolated I/O** or **port-mapped I/O** abbreviated **PMIO**) are two complementary methods of performing input/output between the CPU and peripheral devices in a computer. An alternative approach, not discussed here, is using dedicated I/O processors — commonly known as channels on mainframe computers — that execute their own instructions.

Memory-mapped I/O (not to be confused with memory-mapped file I/O) uses the same address bus to address both memory and I/O devices – the memory and registers of the I/O devices are mapped to (associated with) address values. So when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices. Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the data bus to the desired device's hardware register. To accommodate the I/O devices, areas of the addresses used by the CPU must be reserved for I/O and must not be available for normal physical memory. The reservation might be temporary — the Commodore 64 could bank switch between its I/O devices and regular memory — or permanent.

Port-mapped I/O often uses a special class of CPU instructions specifically for performing I/O. This is found on Intel microprocessors, with the IN and OUT instructions. These instructions can read and write one to four bytes (outb, outw, outl) to an I/O device. I/O devices have a separate address space from general memory, either accomplished by an extra "I/O" pin on the CPU's physical interface, or an entire bus dedicated to I/O. Because the address space for I/O is isolated from that for main memory, this is sometimes referred to as isolated I/O.

[http://en.wikipedia.org/wiki/Memory-mapped_I/O]

NIOS: Common Register Usage in C

Reg	Name	Normal usage
r0	zero	0x0000_0000
r1	at	Assembler Temporary
r2		Return Value (least-significant 32 bits)
r3		Return Value (most significant 32 bits)
r4		Register Arguments (First 32 bits)
r5		Register Arguments (Second 32 bits)
r6		Register Arguments (Third 32 bits)
r7		Register Arguments (Fourth 32 bits)
r8		**Caller**-Saved
r9		General-Purpose Registers
r10		
r11		
r12		
r13		
r14		
r15		

Reg	Name	Normal usage
r16		**Callee**-Saved
r17		General-Purpose Registers
r18		
r19		
r20		
r21		
r22		
r23		
r24	et	Exception Temporary
r25	bt	Break Temporary
r26	gp	Global Pointer
r27	sp	Stack Pointer
r28	fp	Frame Pointer
r29	ea	Exception Return Address
r30	ba	Break Return Address
r31	ra	Return Address

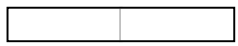
Data Types in Instructions

NIOS II registers hold 32-bit (4-byte) words.

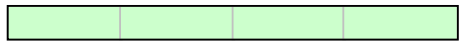
Other common data sizes include byte and halfword.

Byte = 8 bits - example: **ldb / ldbu** - load byte extend signed/unsigned

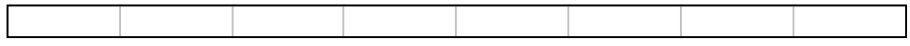
Halfword = 2 bytes example: - **ldh / ldhu** - load halfword extend signed/unsigned



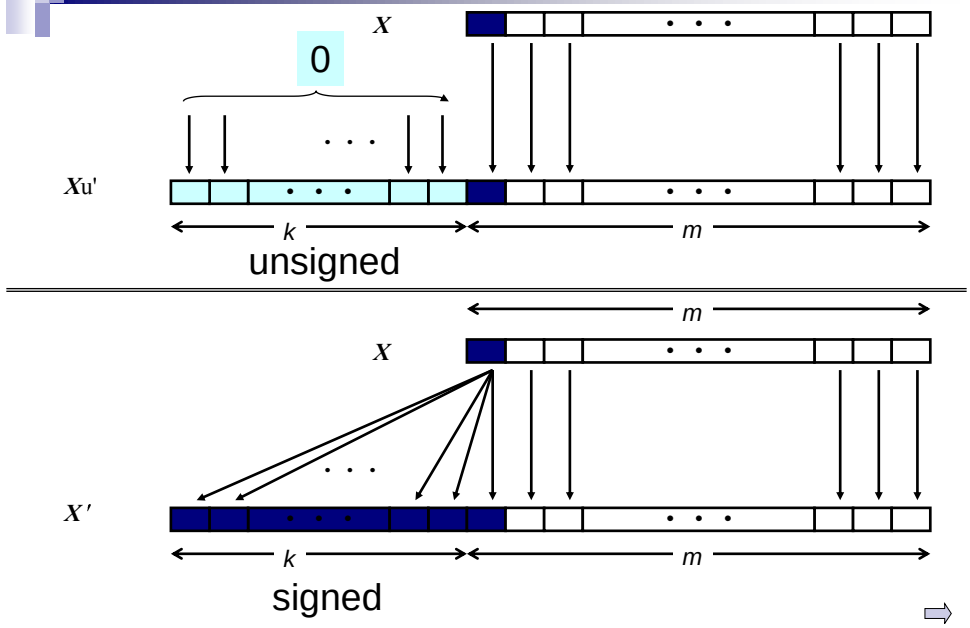
Word = 4 bytes - example: - **ldw** - load word



Doubleword = 8 bytes - user defined instructions only



Unsigned/Signed Extension



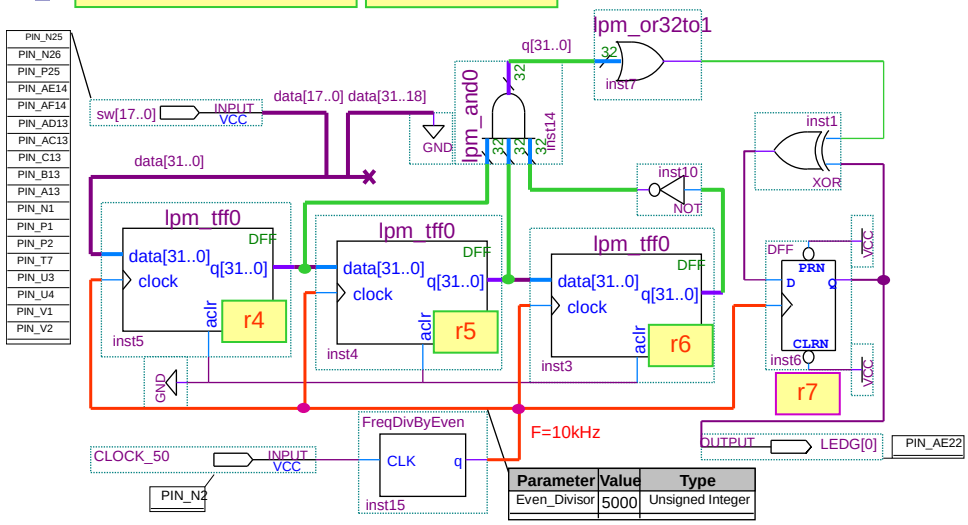
Sign Extension Example in C

```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 C4 92	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

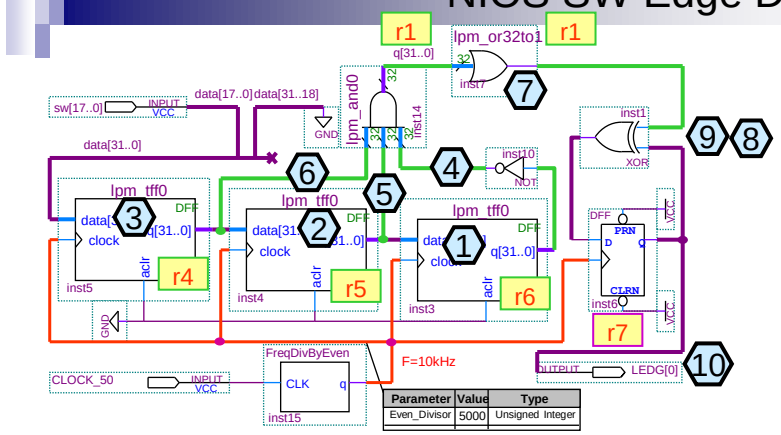
NIOS SW Edge Detector

r8 - address base | r1 - accumulator



97

NIOS SW Edge Detector



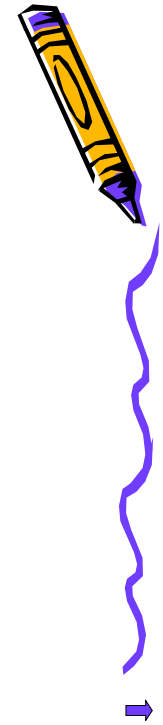
LOOP:

1. mov r6, r5
2. mov r5, r4
3. ldwio r4, SW(r8)
4. nor r1, r6, r0
5. and r1, r1, r5
6. and r1, r1, r4
7. beq r1, r0, LOOP
8. addi r1, r0, 1
9. xor r7, r7, r1
10. stwio r7, LEDG(r8)

    jmpi LOOP

Some GNU assembler directives

.include	insert another file
.text	this directive tells the assembler to place the following statements at the end of the code section
.global	this directive makes the symbol visible to the program loading instructions into memory.
.data	this directive tells the assembler to place the following statements at the end of the data section.
.equ	this directive set the value of a symbol.
.word	this directive is used to set the content of memory locations to the specified value.
.end	Marks the end of current assembly program.



Initialization

```
.equ IOBASE, 0x10000000
.equ LEDG, 0x10
.equ SW,   0x40

.text  -- section of code
    .global _start – entry point
_start:
   movia    r8, IOBASE
    mov     r7, zero     /* zero = r0 = 0 */
```

100

Edge Detector – Full Source Code

```
LOOP:
        mov   r6, r5
        mov   r5, r4
        ldwio  r4, SW(r8)                    LD r1,28(r0); ST r1,r4
        nor    r1, r6, r0   /* r1:=not r6*/
        and    r1, r1, r5
        and    r1, r1, r4
        beq    r1, r0, LOOP
        addi   r1, r0, 1    /* r1:=1*/
        xor    r7, r7, r1
        stwio  r7, LEDG(r8)
        br     LOOP
.end
```

101

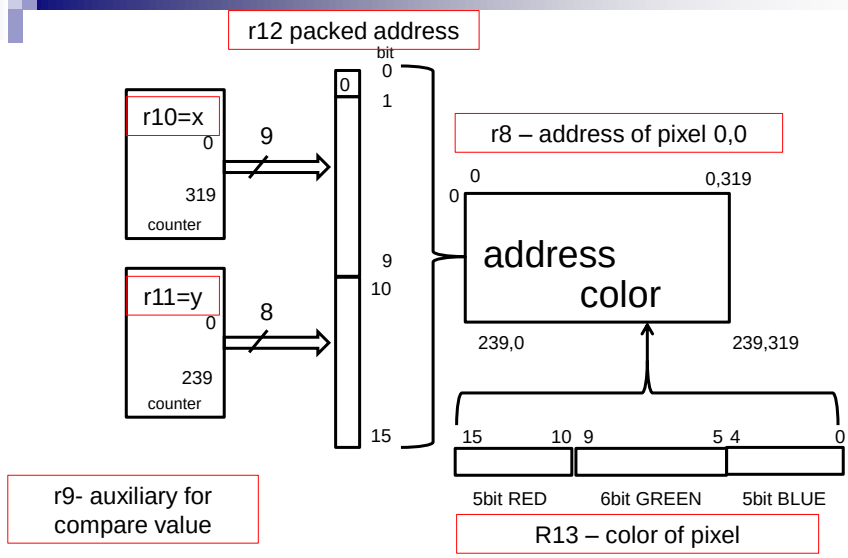
VHDL 32 bit edge detector

```
library ieee; use ieee.std_logic_1164.all;
entity re32d2 is port(data   : in std_logic_vector(31 downto 0);
                      clk    : in std_logic;  q     : out std_logic);
end entity;
architecture rtl of re32d2 is
signal r15, r16, r17: std_logic_vector(31 downto 0); --the shift register
signal r18: std_logic:='0';  --output flip-flop
begin
  process (clk)
  begin
      if (rising_edge(clk)) then
          if((r15 and r16 and (not r17))/=X"00000000") then  r18<=not r18;
          end if;
          r17<=r16; r16<=r15; r15<=data; --shift
      end if;
   end process;
   q<=r18; --write output
end rtl;
```

102

VGA DEMO

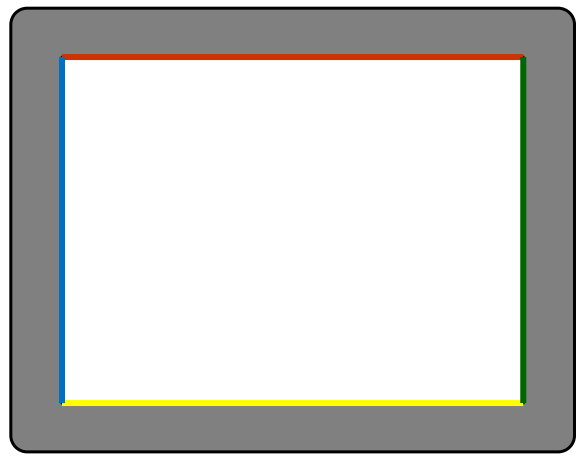
Registers in VGA demo



DEMO VGA program

```
.equ PIXELBUF,0x8000000  /* end = 0x803BE7E */
.equ PIXEL_X_END,319  /* width 0..319 */
.equ PIXEL_Y_END,239  /* height 0..239 */

.equ RED,  0xF800   /* 5 bit red */
.equ GREEN, 0x7E0   /* 6 bit green */
.equ BLUE, 0x1F     /* 5 bit blue */

.equ YSHIFT,9
.equ XYSHIFT,1

.text

        .global _start
_start:
        movia   r8, PIXELBUF    /* video memory */
        mov          r10, r0                          /* X */
        mov          r11, r0        /* Y */
        movia   r13, BLUE

LOOP:
   /* pack y-x into address in PIXELBUF */
        slli    r12, r11, YSHIFT /* shift left logical */
        or      r12, r12, r10
        slli    r12, r12, XYSHIFT
        add     r12, r12, r8   /* add PIXEL BUFFER */

        /* we create color frame for proving
           correct ranges of x-y loop */
        movia   r13, BLUE          /* 0 column blue  */
        beq     r10, r0, NOWHITE /* branch if r10=r0 */
        movia   r13, GREEN
        movia   r9, PIXEL_X_END    /* x-last column green */
        beq     r10, r9, NOWHITE

        movia   r13, RED       /* 0 line red  */
        beq     r11, r0, NOWHITE
        movia   r13, RED | GREEN
        movia   r9, PIXEL_Y_END             /* the last line yellow */
        beq     r11, r9, NOWHITE

        movia   r13, RED | GREEN | BLUE/* white color */

NOWHITE:
        sth     r13, 0(r12)       /* store pixel */
        addi    r10, r10, 1       /* increment x */
        movia   r9, PIXEL_X_END
        bleu     r10, r9, LOOP
        mov      r10, r0         /* next line, x=0 */
        addi     r11, r11, 1     /* increment line */
        movia   r9, PIXEL_Y_END
        bleu       r11, r9, LOOP
STOP:
        br STOP   /* You flag is finished. */
.end
```

Result on VGA Display



106