Academie voor Technology, Innovation & Society Delft
Academie voor ICT & Media

Digitale Systeem Engineering 2

Week 5 – Eenvoudige microprocessor

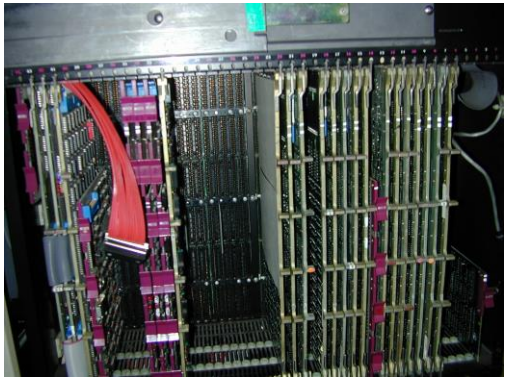
Jesse op den Brouw

DIGSE2/2020-2021

DE HAAGSE HOGESCHOOL

Processor

- Zou het niet mooi zijn om een digitaal systeem te bouwen dat geschikt is voor het uitvoeren van een willekeurig algoritme?
- Denk hierbij aan diverse rekenkundige operaties, zoals vermenigvuldiging, deling of het berekenen van het gemiddelde.
- Zo'n systeem verwerkt data (*to process*) en wordt een processor genoemd.
- Vroegere processors bestonden uit diverse printplaten gekoppeld door een *back plane*.



Processor

- Door de miniaturisering van transistoren op een IC is het mogelijk om een complete processor op een IC te plaatsen.
- Dit wordt een microprocessor genoemd.
- Dankzij VHDL is het relatief eenvoudig een simpele processor te ontwerpen.

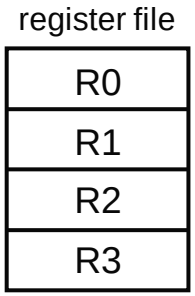
Registers en multiplexers

- Zo'n systeem moet dan registers hebben om (tussen-)resultaten op te slaan en een *rekeneenheid* voor de uitvoeren van de berekeningen.
- Veel microprocessoren hebben een aantal gelijksoortige registers. Deze worden samengenomen in een *register file*.
- De registers moeten op één of andere manier verbonden zijn met de rekeneenheid en die moet het resultaat weer terugleveren aan één van de registers.
- De meeste bewerkingen worden met twee registers uitgevoerd, dus moeten er *multiplexers* gebruikt worden om de juiste registers te koppelen aan de rekeneenheid.

Register File

- De register file bestaat uit vier identieke registers en zijn opgebouwd met flipflops.

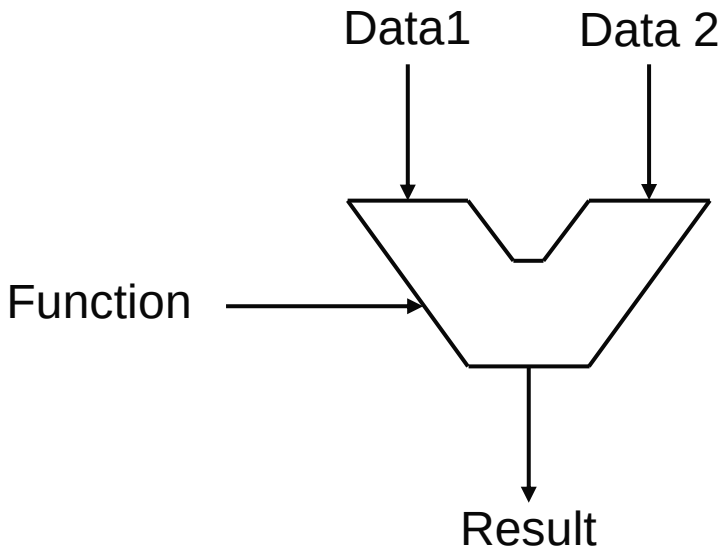
- De registers worden geklokt op hetzelfde kloksignaal en hebben een enable-faciliteit.



ALU

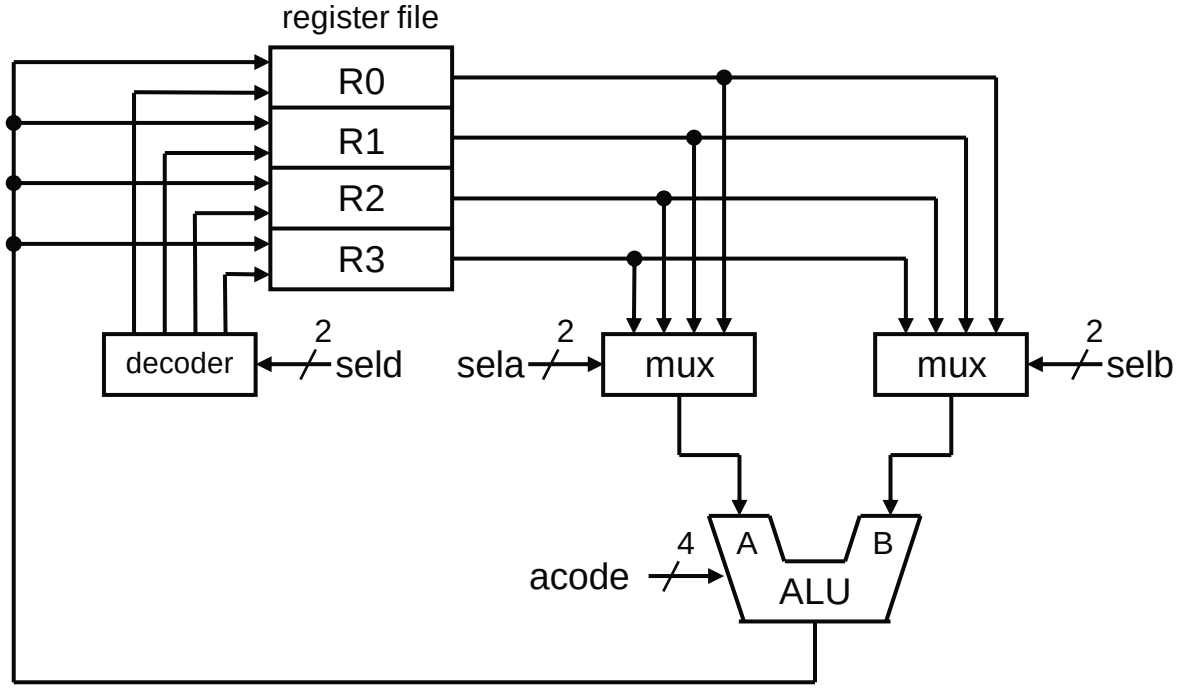
- De ALU (*Arithmetic and Logic Unit*) is een combinatorische schakeling die een aantal rekenkundige en logische bewerkingen kan uitvoeren.

- Bewerkingen: optellen, aftrekken, vermenigvuldigen*, delen*, bitwise AND/OR/XOR, inverse, schuiven, roteren, *pass thru*.



*) complexe ALU's

Datapad microprocessor



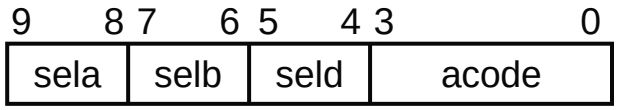
ALU function

0000	nop
0001	pass B*
0010	add
0011	sub**
0100	and
0101	or
0110	exor
0111	not B*
1000	shl B*
1001	shr B*
1010	-
1011	-
1100	-
1101	-
1110	-
1111	-

*via de B-tak van de ALU

** A - B

instruction format



DE HAAGSE HOGESCHOOL

Opgaven

- Waarom zou de nop-operatie handig zijn?
- Waarom zou een pass-operatie handig zijn?
- Zijn er nog meer (eenvoudige) logische operaties te bedenken? Zijn deze operaties ook met de huidige set van logische operaties te realiseren (want dan zijn de nieuwe operaties overbodig...)?
- Geef het bitpatroon dat ervoor zorgt dat de exor van R2 en R3 in R1 terecht komt (snelschrift (zie ook verderop): `exor R1,R2,R3`).
- Beschrijf de werkingen van `and R1,R2,R3` en `and R1,R3,R2`.

DE HAAGSE HOGESCHOOL

Besturing

- Nu willen we het systeem een klein programmaatje laten uitvoeren. Hierin wordt een aantal instructies uitgevoerd.

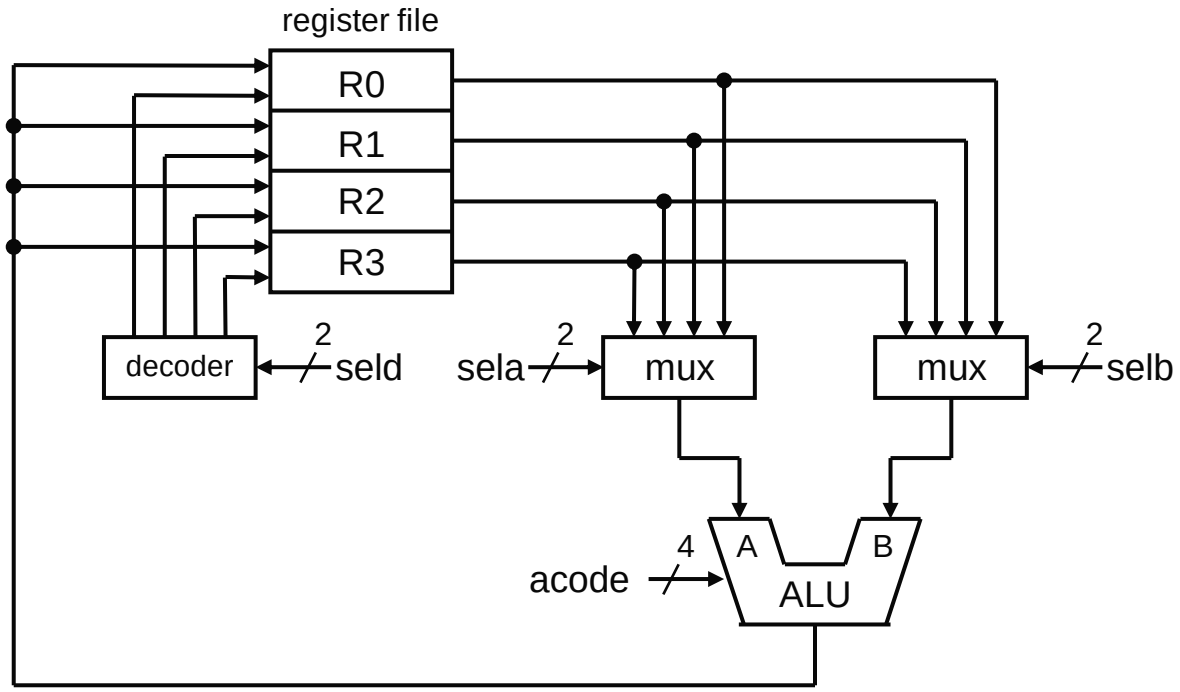
Stap	Instructie	Bitpat. Stap	Bitpatroon instructie
0	R0 = R1 + R2	...0000	01 10 00 0010
1	R3 = R0 – R3	...0001	00 11 11 0011
2	R0 = shl R0	...0010	xx 00 00 1000
3	R2 = not R2	...0011	xx 10 10 0111
4	R1 = R3	...0100	xx 11 01 0001
...	.....	....	.....

DE HAAGSE HOGESCHOOL

Besturing

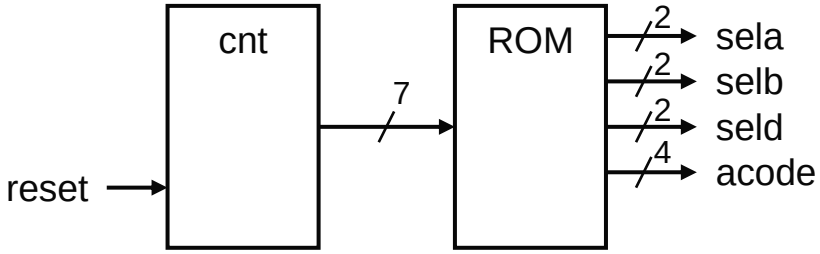
- Aan de tabel kunnen twee zaken worden opgemerkt.
- De stapnummers lopen steeds met één op. Dit is het gedrag van een *teller*.
- De instructie-bits zijn per stap verschillend. Dit wordt gerealiseerd met combinatorische logica. (Het rechter deel van de tabel op de vorige slide is namelijk een waarheidstabel).
- De besturing kan nu worden gerealiseerd met een teller en een ROM.
- De teller heeft 7 telbits, het aantal stappen is 128.

Eerste versie microprocessor



ALU function

0000	nop
0001	pass B
0010	add
0011	sub
0100	and
0101	or
0110	exor
0111	not B
1000	shl B
1001	shr B
1010	-
1011	-
1100	-
1101	-
1110	-
1111	-



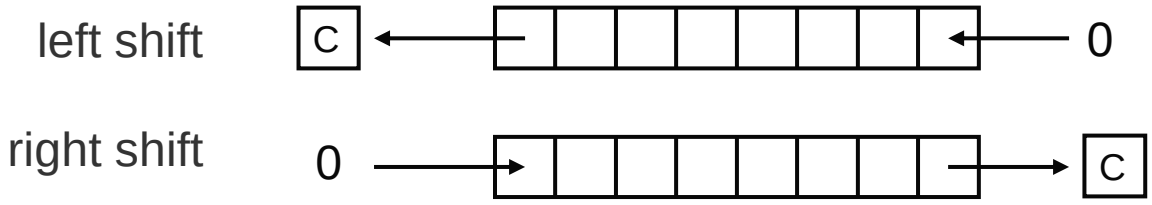
DE HAAGSE HOGESCHOOL

ALU flags

- Soms is niet het resultaat van een bewerking interessant maar wel het effect ervan.
- Denk hierbij aan het testen of A gelijk is aan B. Dat kan eenvoudig worden gedaan door de operatie A – B . Immers A = B kan ook geschreven worden als (A – B) = 0.
- Dit effect wordt opgeslagen in een flipflop, *flag* genoemd.
  - Andere benamingen: Processor Status Word (PSW) , Status Register (SREG), Condition Code Register (CCR).
- Wanneer het resultaat van een bewerking nul is, wordt de *zero flag* (Z) gezet, anders wordt het gewist.

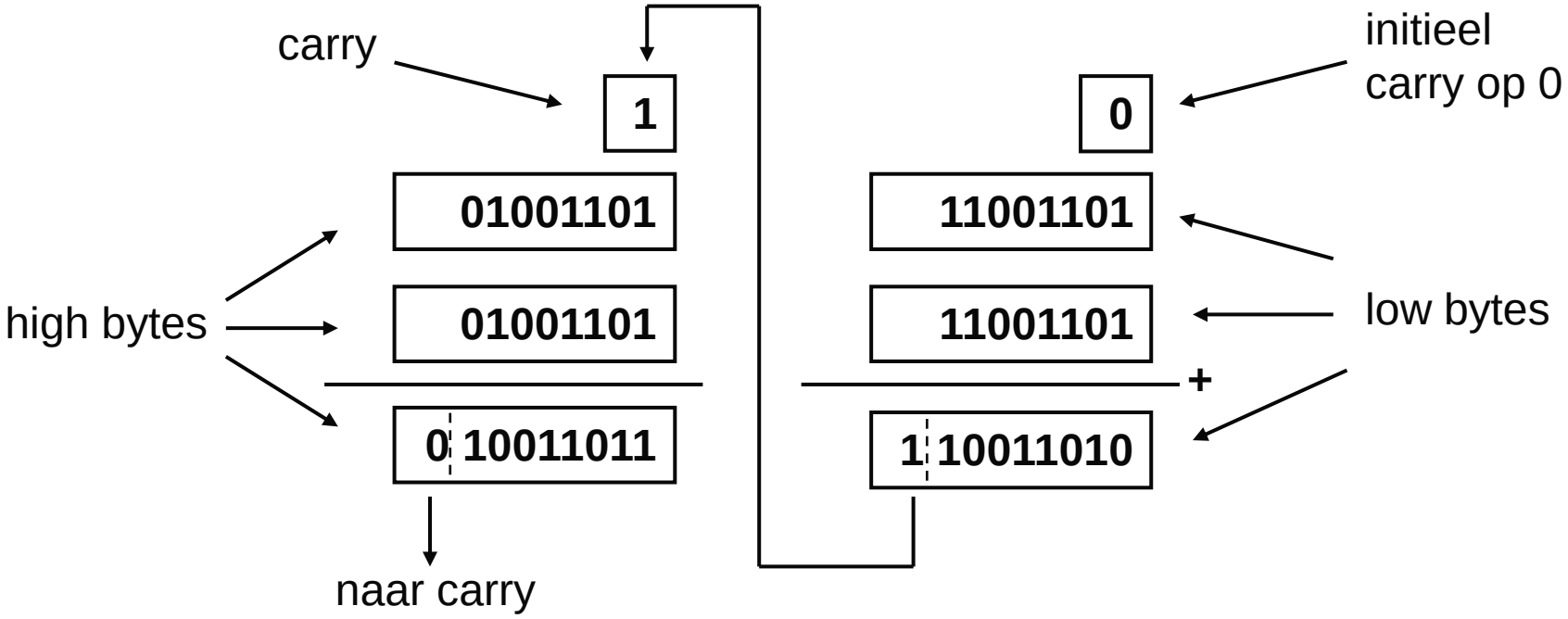
ALU flags

- Bij een optelling van twee 8-bits getallen kan het resultaat groter worden dan 8 bits. Het negende bit wordt opgeslagen in de *carry flag* (C).
- De C-flag geeft *overflow* of *underflow* bij unsigned bewerkingen aan.
- De C-flag wordt ook gebruikt bij schuifoperaties. Het uitgeschoven bit wordt in de C-flag geplaatst. Het ingeschoven bit is 0.



Multibyte optelling

- De carry flag wordt gebruikt bij multi-byte bewerkingen. Zo kunnen 16- en 32-bit optellingen worden gedaan met een 8-bit processor.



Bewerken flags / enable registers

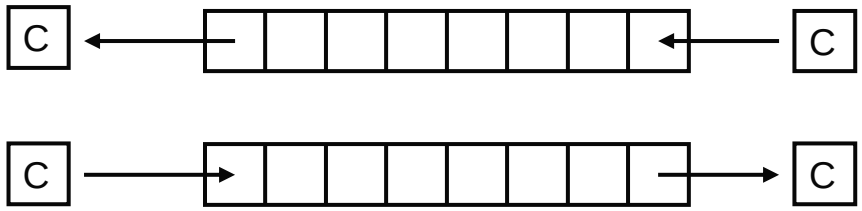
- Soms is het noodzakelijk om de flags op een logische 0 of 1 te zetten. Denk hierbij aan (het begin van) een multibyte optelling.
- De flags kunnen via de ALU gemanipuleerd worden.
- De ALU krijgt twee nieuwe operaties:

  schrijf flags uit B (wrf B)
  lees flags (rdf)

- Omdat het resultaat van een test niet moet worden opgeslagen wordt de decoder uitgebreid met een *enable*. Hierdoor wordt geen van de registers geladen met het resultaat.

Opgaven

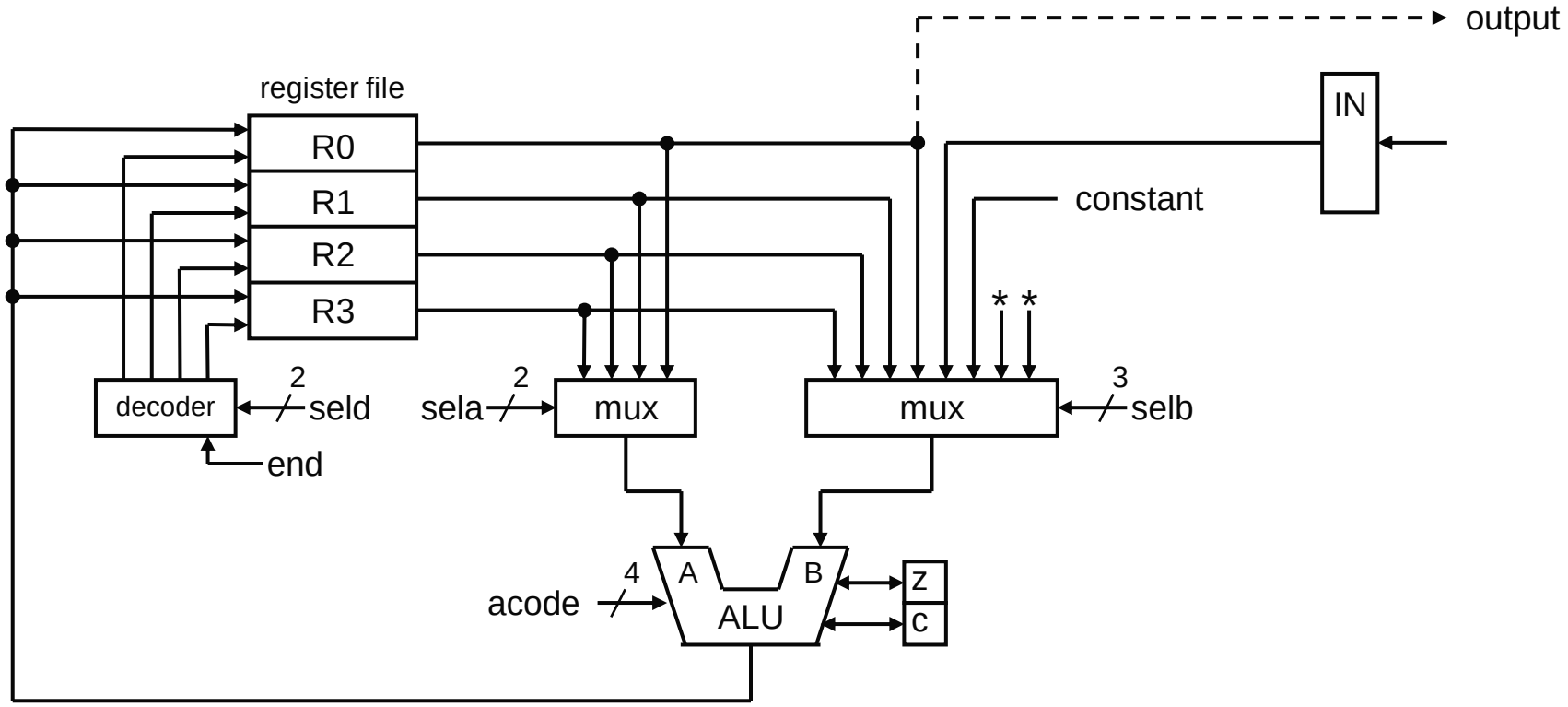
- Eerder zijn de schuifoperaties aan bod geweest. Hierbij werd het hoogstwaardige of laagstwaardige bit in de C-flag geplaatst en werd een logische 0 ingeschoven. In plaats van deze 0 zou ook de C-flag kunnen worden ingeschoven. Beschrijf het effect van deze nieuwe operatie. (Merk op dat alle registers en flags flankgestuurd zijn, dus de huidige waarde van de C-flag wordt ingeschoven en de huidige hoogstwaardige of laagstwaardige bit wordt uitgeschoven).

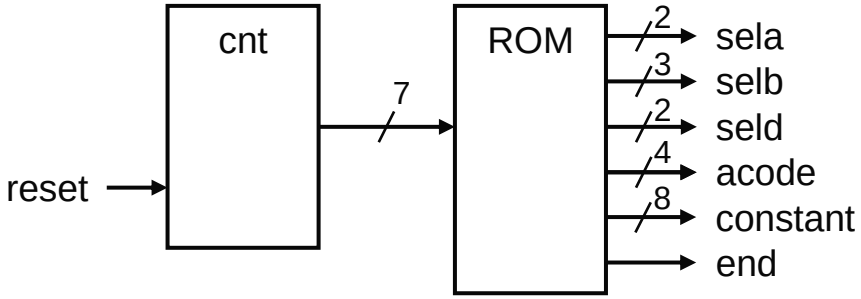


Nog meer invoer en uitvoer

- In versie één is het niet mogelijk om data van buiten af in te laden.
- Daarnaast is het niet mogelijk om een constante te laden. Bedenk dat de constante 0 veel gebruikt wordt (tellers op nul).
- Het datapad moet aangepast worden zodat de twee invoermogelijk-heden kunnen worden gebruikt.
- De multiplexer aan kant B van de ALU wordt uitgebreid van vier naar acht ingangen. Het aantal selectiesignalen wordt dan drie.
- De inhoud van R0 wordt naar buiten uitgevoerd.

Tweede versie microprocessor





* reserved for future use

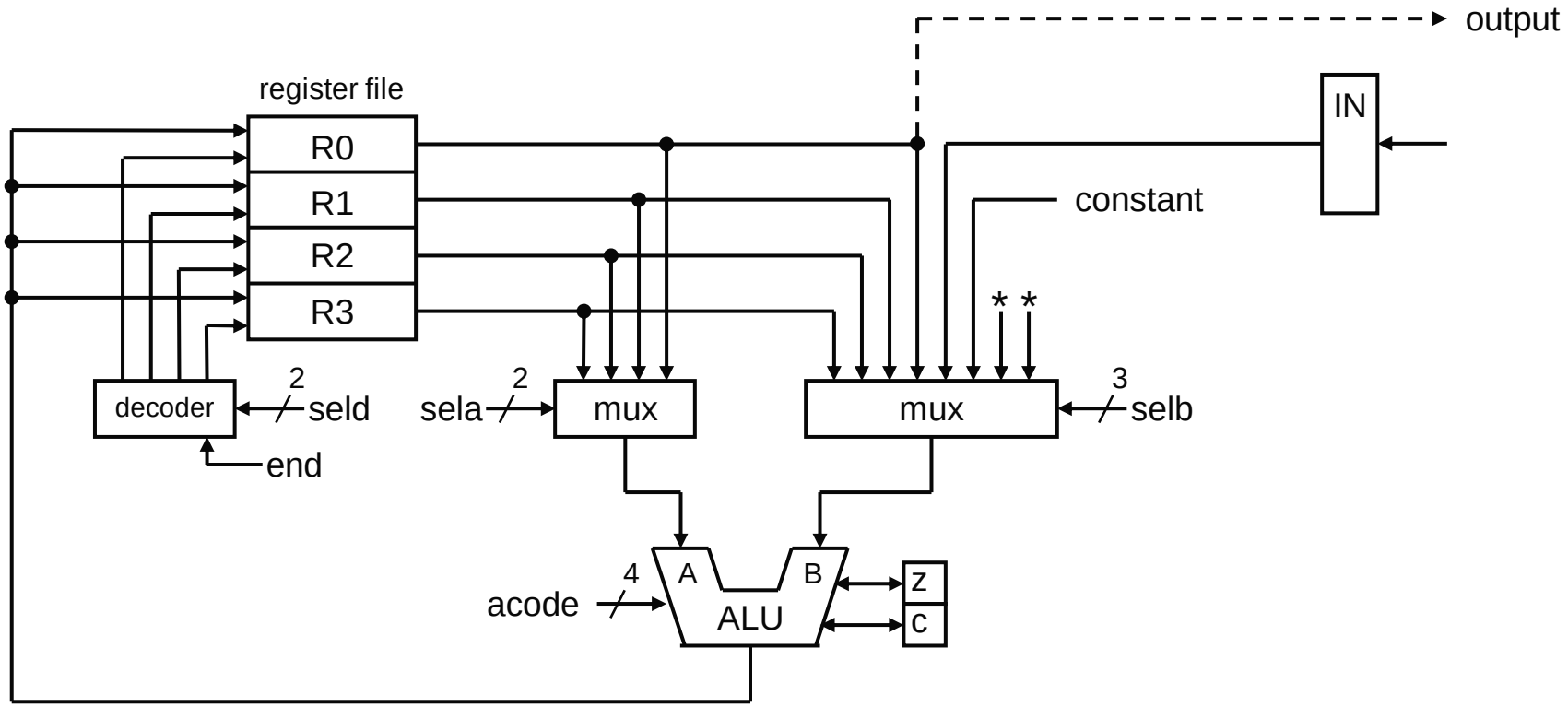
Programmateller

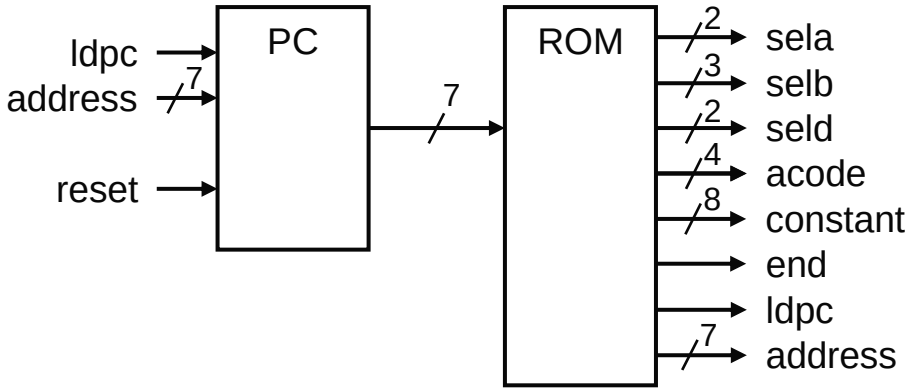
- De teller die bijhoudt welke instructie wordt uitgevoerd heet *programmateller*.
- Andere namen:
- Program Counter (Motorola, AVR, ARM, afgekort tot PC)
- Instruction Pointer (Intel x86/x64, afgekort tot IP)

Springen

- Eén van de nadelen bij het beschreven systeem is dat het systeem eeuwig door blijft gaan met het sequentieel uitvoeren van instructies.
- Wanneer we een aantal instructies steeds moeten herhalen is dit een probleem. We hebben dan een oneindig grote ROM nodig (en teller) die steeds eenzelfde reeks instructies herhalen.
- Slimmer is om de teller opnieuw te laten beginnen op een bepaald punt. We moeten **springen** in het programma. Dit kan elegant worden opgelost door de PC te laden met een getal, waar vanaf de teller opnieuw moet beginnen. We noemen zo'n getal een **adres**.
- In feite is het inbouwen van een sprongopdracht geen moeilijke klus. De ROM krijgt een extra uitgang die naar een stuuringang gaat op de teller, de zogenaamde *ldpc*-ingang. Het benodigde adres wordt ook in de ROM opgeslagen.

Derde versie microprocessor





DE HAAGSE HOGESCHOOL

Conditioneel springen

- We kunnen het nog interessanter maken door te springen als er een bepaalde conditie voorkomt, bijvoorbeeld het resultaat van de laatste berekening is nul (Z-flag = 1).
- We kunnen dan beslissingen maken, bv: als R0 = R2, dan ….
- We gebruiken hiervoor de flags en een nieuw stukje logica, dat de load-ingang van de teller aanstuurt.
- Noot: het engelse woord voor springen is *jump*. Het woord *branch* wordt veel gebruikt, dit betekent vertakken.

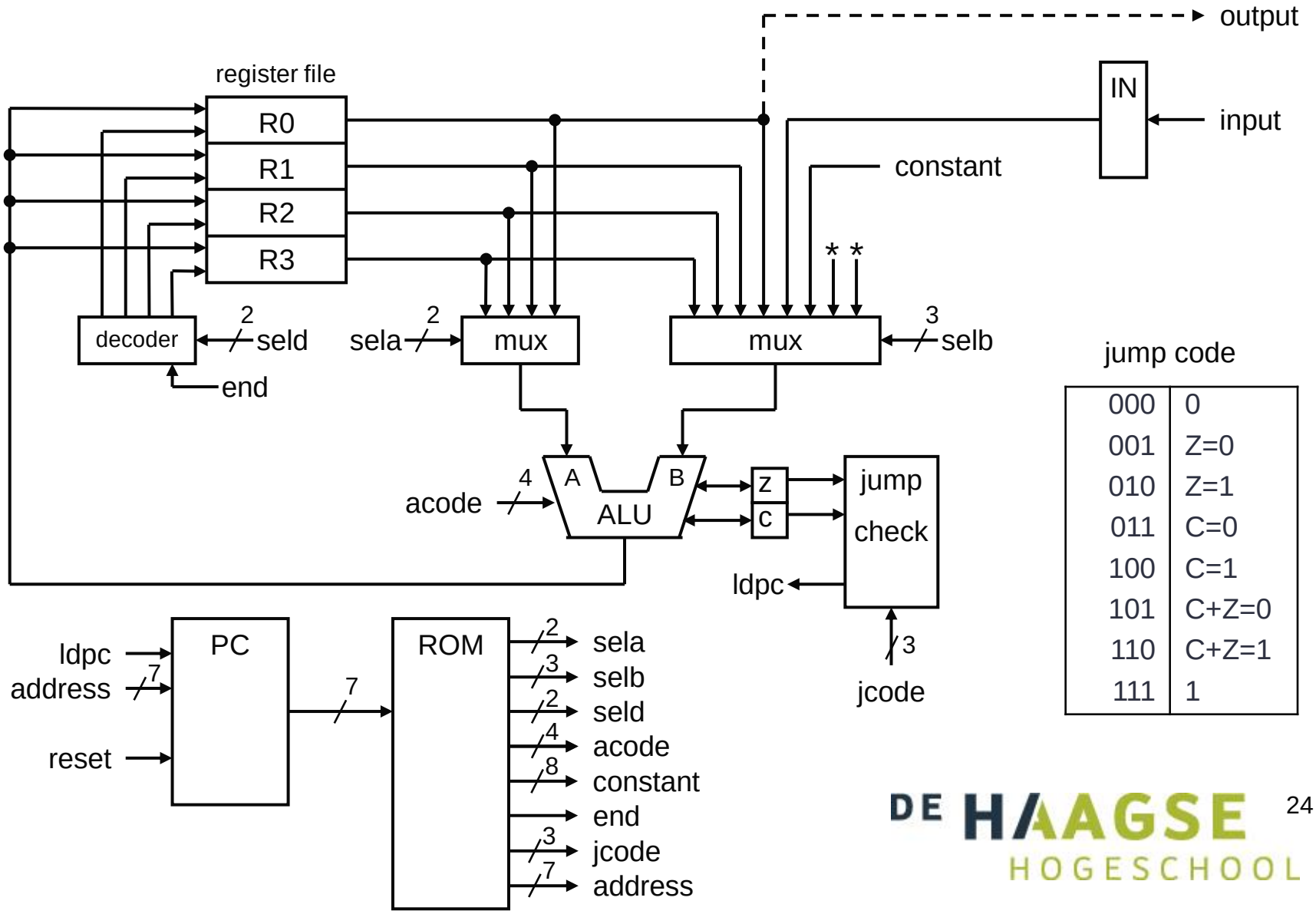
Beslissen

- Alle beslissingen zoals A = B of A > B kunnen worden omgewerkt naar een aftrekoperatie en een test op de flags.

$A \neq B$	$(A - B) \neq 0$	$Z = 0$
$A = B$	$(A - B) = 0$	$Z = 1$
$A \geq B$	$(A - B) \geq 0$	$C = 0$
$A < B$	$(A - B) < 0$	$C = 1$
$A > B$	$(A - B) > 0$	$C+Z = 0$
$A \leq B$	$(A - B) \leq 0$	$C+Z = 1$

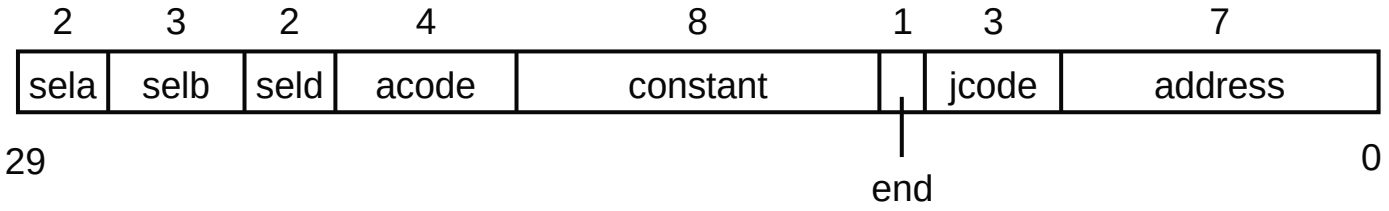
- Het resultaat van de aftrekoperatie wordt *niet* opgeslagen; alleen de flags worden aangepast. Het betreft hier *unsigned* getallen.

Vierde versie microprocessor



Instructieformaat

- De bitpatronen die de instructies vormen zijn 30 bits breed.
- In totaal zijn er 128 instructies mogelijk (adres is 7 bits).
- De complete ROM beslaat dan 3840 bits aan data.
- Hieronder het instructieformaat:



Software

- De microprocessor is nu gereed, dat wil zeggen: de hardware is af.
- Om het systeem nu te gebruiken moeten de bitpatronen van de instructies in de ROM worden vastgelegd.
- Deze ROM is eventueel te herprogrammeren (PROM, Flash-ROM, ...) waardoor het systeem een andere functie uitvoert.
- De *hardware*-ontwikkeling wordt nu verlaten en gaat nu over in *software*-ontwikkeling.

Voorbeeld vermenigvuldigen

- Vermenigvuldigen kan ook door herhaald optellen:

```
uitkomst := 0;
for i := 1 to 11 do
    uitkomst := uitkomst + 13;
end for;
```

- Omzetten naar variabelen en while-lus met een afteller:

```
a := 13; b := 11;
uitkomst := 0;
while (b ≠ 0) do
    uitkomst := uitkomst + a;
    b := b - 1;
end while;
```

Omzetten naar registers

- Merk op de een 8x8 bit vermenigvuldiging een 16 bit resultaat oplevert. Het resultaat moet in twee registers worden opgeslagen/bewerkt.

- Omzetten naar registers (*register mapping*):

```
R0 := 13; R1 := 11;
R3\R2 := 0;                    // 16-bit result
while (R1 ≠ 0) do
    R3\R2 := R3\R2 + R0;
    R1 := R1 – 1;
end while;
```

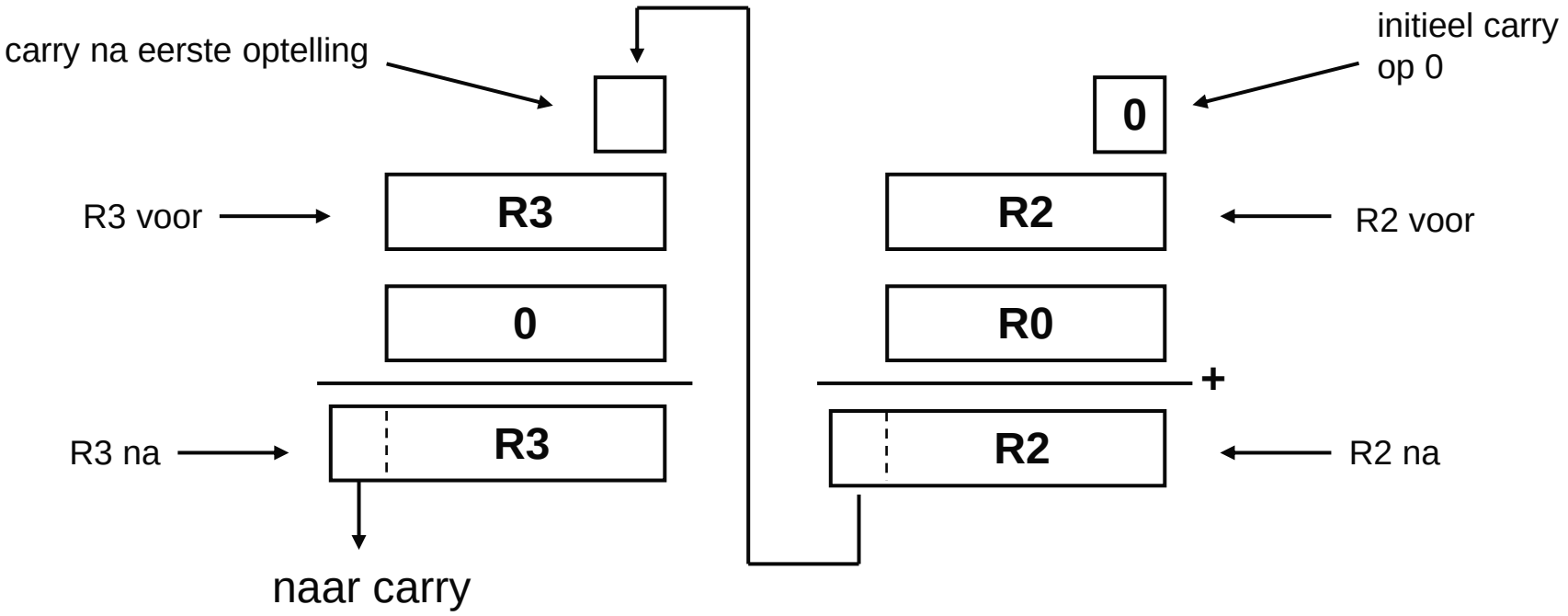
Aanpassen while-lus

- De while-lus moet worden aangepast aan de hardware, dus testen en springen. Testen gebeurt op de inverse relationele operator.

```
R0 := 13;
R1 := 11;
R3\R2 := 0;
opnieuw: if (R1 = 0) goto klaar;
         R3\R2 := R3\R2 + R0;
         R1 := R1 – 1;
         goto opnieuw;
klaar:   goto klaar;
```

Multibyte-optelling

- Het resultaat wordt in twee registers (R3, R2) opgeslagen. Daar moet de inhoud van register R0 bij opgeteld worden. Dat moet in twee optel-slagen.



Compleet programma

- De test op 0 moet gesplitst worden in twee instructies: één die de Z-flag op 1 zet (als het resultaat 0 is) en één die conditioneel springt (de PC wordt geladen met een adres) als de Z-flag 1 is.

```
R0 := 13;
R1 := 11;
R2 := 0;   R3 := 0;
opnieuw: zero := (R1 - 0) = 0;          // set/reset Z flag
         if zero = 1 goto klaar;        // test Z flag
         carry := 0;
         R2 := R2 + R0 (+ carry);       // compute low byte
         R3 := R3 + 0 (+ carry);        // compute high byte
         R1 := R1 – 1;
         goto opnieuw;
klaar:   goto klaar;
```

DE HAAGSE HOGESCHOOL

Stappenplan

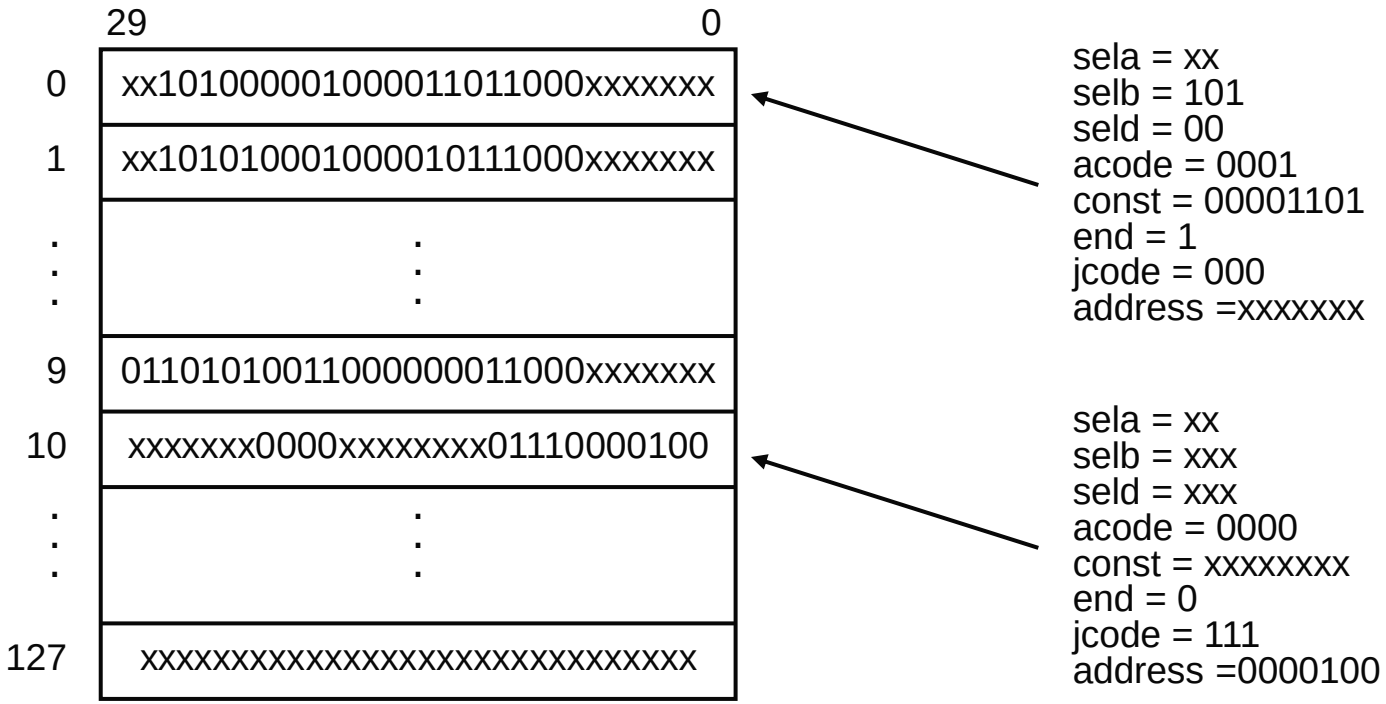
- Het programma wordt dan:

0: laad register 0 met de constante 13
1: laad register 1 met de constante 11 (teller)
2: laad register 2 met de constante 0 (lage byte uitkomst)
3: laad register 3 met de constante 0 (hoge byte uitkomst)
4: is register 1 al 0 (zero flag = 1)?
5: als ja, dan spring naar stap 11 (programma klaar)
6: zet de carry flag op 0
7: verwerk lage byte (R2 + R0 + carry)
8: verwerk hoge byte (R3 + 0 + carry)
9: verlaag de teller (register 1) met 1
10: spring naar stap 4
11: plaats de uitkomst op de uitgang (alleen lage byte)
12: spring naar stap 12 (wacht voor altijd)

DE HAAGSE HOGESCHOOL

Instructie-ROM

- De instructies van de vorige slide moeten in bitpatronen worden omgezet waarmee het datapad van de processor te besturen is. Deze bitpatronen moeten in de ROM geplaatst worden.



Programma in assembler

- Het programma in *assembler-taal*:

```
 0:   ld   R0,#13      ; load multiplicand (constant 13)
 1:   ld   R1,#11      ; load multiplier (constant 11)
 2:   ld   R2,#0       ; clear result registers
 3:   ld   R3,#0       ; or use ld R3,R2
 4:   cmp  R1,#0       ; is multiplier already 0 ?
 5:   je   @11         ; yes it is, so jump!
 6:   ld   F,#0        ; clear the flags (clears carry)
 7:   add  R2,R2,R0    ; add multiplicant (low byte)
 8:   add  R3,R3,#0    ; process carry in high byte
 9:   sub  R1,R1,#1    ; multiplier one off
10:   jmp  @4          ; and again
11:   ld   R0,R2       ; result to output*
12:   jmp  @12         ; hold your horses!
```

DE HAAGSE HOGESCHOOL

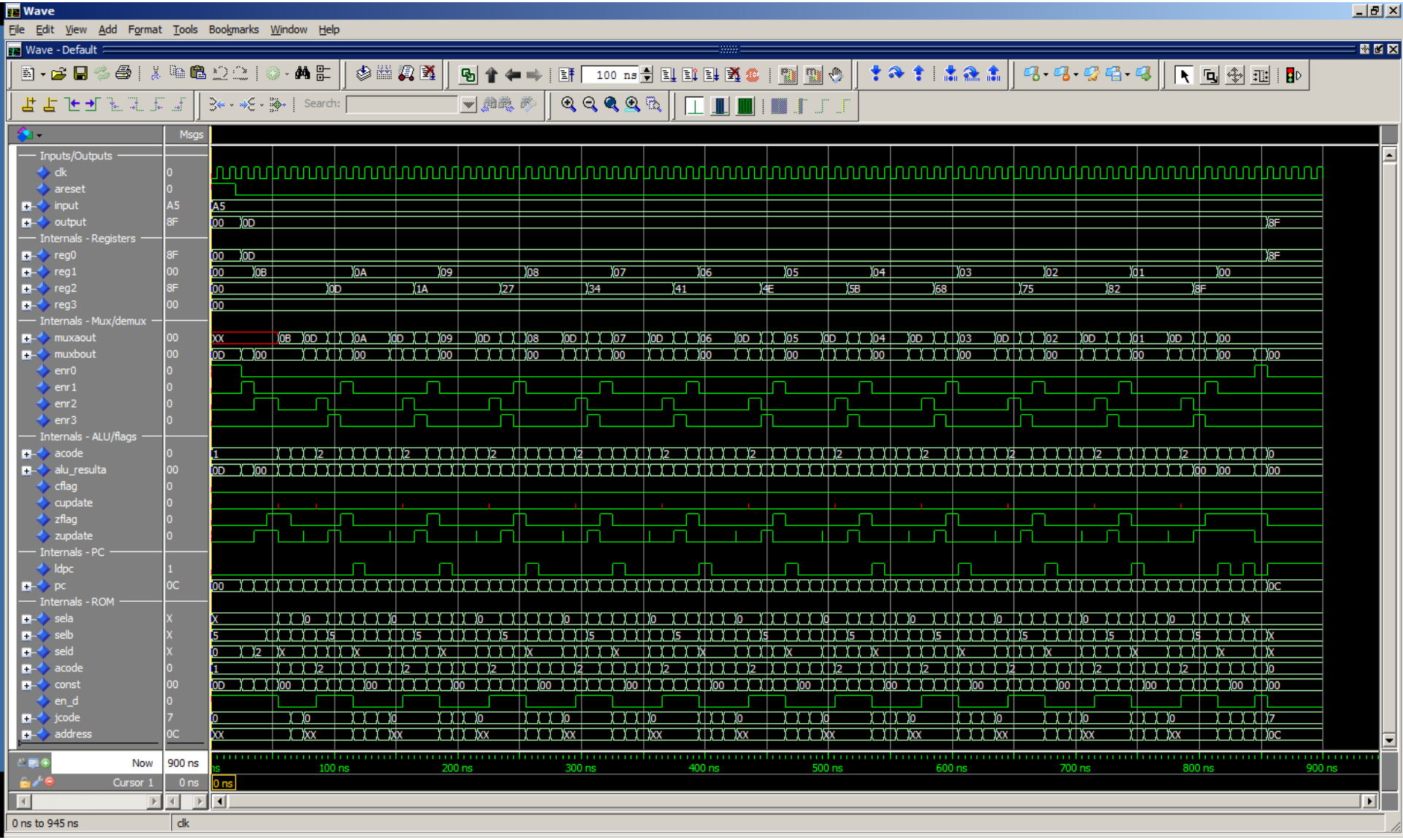
Uitleg mnemonics en operands

- De mnemonics en operands:

`ld`	Load register with register or constant
`cmp`	Compare register with register or constant (subtract)
`je`	Jump to address if equal (Z-flag = 1)
`add`	Add register/constant to register
`sub`	Subtract register/constant from register
`jmp`	Jump to address (always, unconditionally)
`#`	Constant value (ld, add, sub, cmp)
`Rx`	Register X (ld, add, sub, cmp)
`@`	Address (je, jmp)
F	Flags

DE HAAGSE HOGESCHOOL

Simulatieresultaat voorbeeld



VHDL-code – decoder

```
-- Selection of the register to write back the result.
decoderd : process (seld, en_d) is
begin
    enr0 <= '0';
    enr1 <= '0';
    enr2 <= '0';
    enr3 <= '0';
    if en_d = '1' then
        case seld is
            when "00" => enr0 <= '1';
            when "01" => enr1 <= '1';
            when "10" => enr2 <= '1';
            when "11" => enr3 <= '1';
            when others => null;
        end case;
    end if;
end process decoderd;
```

VHDL-code – register file

```
-- The Register File contains four registers and loads a
-- specified register if the enable is active.
regfile : process (clk, areset, enr0, enr1, enr2, enr3) is
begin
    if areset = '1' then
        reg0 <= data_zeros;
        reg1 <= data_zeros;
        reg2 <= data_zeros;
        reg3 <= data_zeros;
    elsif rising_edge(clk) then
        if enr0 = '1' then reg0 <= alu_result; end if;
        if enr1 = '1' then reg1 <= alu_result; end if;
        if enr2 = '1' then reg2 <= alu_result; end if;
        if enr3 = '1' then reg3 <= alu_result; end if;
    end if;
end process regfile;
```

DE HAAGSE HOGESCHOOL

VHDL-code – multiplexer A

```
-- Selection of the A input of the ALU
muxa : process (sela, reg0, reg1, reg2, reg3) is
begin
    case sela is
        when "00" => muxaout <= reg0;
        when "01" => muxaout <= reg1;
        when "10" => muxaout <= reg2;
        when "11" => muxaout <= reg3;
        when others => null;
    end case;
end process muxa;
```

VHDL-code – multiplexer B

```
-- Selection of the B input of the ALU
muxb : process (selb, reg0, reg1, reg2, reg3, input, const) is
begin
    case selb is
        when "000" => muxbout <= reg0;
        when "001" => muxbout <= reg1;
        when "010" => muxbout <= reg2;
        when "011" => muxbout <= reg3;
        when "100" => muxbout <= input;
        when "101" => muxbout <= const;
        --when "110" => muxaout <= (others => '-');
        --when "111" => muxaout <= (others => '-');
        when others => muxbout <= (others => '-');
    end case;
end process muxb;
```

VHDL-code – flags

```
-- The Zero and Carry flags. They are really flip-flops and
-- depend on the outcome of the ALU result.
flags : process (clk, areset) is
begin
    if areset = '1' then
        cflag <= '0';
        zflag <= '0';
    elsif rising_edge(clk) then
        cflag <= cupdate;
        zflag <= zupdate;
    end if;
end process flags;
```

DE HAAGSE HOGESCHOOL

VHDL-code – ALU (part I)

```
alu : process (acode, muxaout, muxbout, zflag, cflag) is
variable alu_result_int : data_type1;
variable z_int, c_int : std_logic;
begin
    alu_result_int := (others => '0');
    z_int := zflag;
    c_int := cflag;
    case acode is
        when alu_nop =>
            alu_result_int := '0' & muxbout;
        when alu_passb =>
            alu_result_int := '0' & muxbout;
        when alu_add =>
            alu_result_int := ('0' & muxaout) + ('0' & muxbout) + ("" & cflag);
            c_int := alu_result_int(8);
        when alu_sub =>
            alu_result_int := ('0' & muxaout) + ('0' & not muxbout) +
                                                        ("" & not cflag);
            c_int := not alu_result_int(8);
        when alu_and =>
            alu_result_int := '0' & (muxaout and muxbout);
        when alu_or =>
            alu_result_int := '0' & (muxaout or muxbout);
```

VHDL-code – ALU (part II)

```
when alu_exor =>
    alu_result_int := '0' & (muxaout xor muxbout);
when alu_notb =>
    alu_result_int := '0' & (not muxbout);
when alu_shlb =>
    alu_result_int := muxbout & '0';
    c_int := alu_result_int(8);
when alu_shrb =>
    alu_result_int := "00" & muxbout(7 downto 1);
    c_int := muxbout(0);
when alu_rolb =>
    alu_result_int := muxbout & cflag;
    c_int := alu_result_int(8);
when alu_rorb =>
    alu_result_int := '0' & cflag & muxbout(7 downto 1);
    c_int := muxbout(0);
when alu_wrfb =>
    alu_result_int := ('0' & muxbout);
    c_int := alu_result_int(0);
    z_int := alu_result_int(1);
```

DE HAAGSE HOGESCHOOL

VHDL-code – ALU (part III)

```
        when alu_wrfb =>
            alu_result_int := ('0' & muxbout);
            c_int := alu_result_int(0);
            z_int := alu_result_int(1);
        when alu_rdf =>
            alu_result_int := (0 => cflag, 1 => zflag, others => '0');
        when others =>
            alu_result_int := ('0' & muxbout);
    end case;
    if acode /= alu_rdf and acode /= alu_wrfb and acode /= alu_nop then
        if alu_result_int(7 downto 0) = data_zeros then
            z_int := '1';
        else
            z_int := '0';
        end if;
     end if;
     alu_result <= alu_result_int(7 downto 0);
     zupdate <= z_int;
     cupdate <= c_int;
end process alu;
```

DE HAAGSE HOGESCHOOL

VHDL-code – jump checker

```
-- The Conditional Jump Checker. This process determines if
-- a certain condition is met and sets the PC Load signal accordingly.
jumpcheck : process (jcode, zflag, cflag) is
begin
    case jcode is
        when jump_jn =>   ldpc <= '0';
        when jump_jnz =>  ldpc <= not zflag;
        when jump_jz =>   ldpc <= zflag;
        when jump_jnc =>  ldpc <= not cflag;
        when jump_jc =>   ldpc <= cflag;
        when jump_jncz => ldpc <= not (cflag or zflag);
        when jump_jcz =>  ldpc <= cflag or zflag;
        when jump_j =>    ldpc <= '1';
        when others =>    null;
    end case;
end process jumpcheck;
```

VHDL-code – program counter

```
-- The Program Counter. It can be loaded which
-- effectively jumps to another program location.
-- Program starts a address 0 on a reset.
regpc : process (clk, areset) is
begin
    if areset = '1' then
        pc <= address_zeros;
    elsif rising_edge(clk) then
        if ldpc = '1' then
            pc <= address;
        else
            pc <= pc + 1;
        end if;
    end if;
end process regpc;
```

VHDL-code – ROM decoding

```
-- Instruction ROM decoding is nothing more then rewiring
-- the instruction bits to the control inputs. Only VHDL
-- makes such a trouble to do this the simple way...
instr_rom : process (pc) is
begin
    sela    <= rom(to_integer(pc))(29 downto 28);
    selb    <= rom(to_integer(pc))(27 downto 25);
    seld    <= rom(to_integer(pc))(24 downto 23);
    acode   <= rom(to_integer(pc))(22 downto 19);
    const   <= unsigned(rom(to_integer(pc))(18 downto 11));
    en_d    <= rom(to_integer(pc))(10);
    jcode   <= rom(to_integer(pc))(9 downto 7);
    address <= unsigned(rom(to_integer(pc))(6 downto 0));
end process instr_rom;
```

VHDL-code – ROM table

```
subtype  instr_type is std_logic_vector (29 downto 0);
type  rom_table is array (0 to 127) of instr_type;
-- Program ROM starts here
constant rom : rom_table := (
    "--10100001000011011000-------", -- Load R0 with constant 13
    "--10101000100010111000-------", -- Load R1 with constant 11
    "--10110001000000001000-------", -- Load R2 with constant 0
    "--10111000100000001000-------", -- Load R3 with constant 0
    "01101--0011000000000000-------", -- Compare R1 with constant -
    "-------00000000000000100001011", -- Jump if equal (R1 == 0) to
                                      -- address 11
    "00101001100------000000-------", -- Load flags with zeros
    "00010100010--------1000-------", -- Add: R2 := R2 + R0 + carry
    "11101110010000000001000-------", -- Add: R3 := R3 + 0 + carry
    "01101010011000000011000-------", -- Sub: R1 := R1 - 1
    "-------0000--------01110000100", -- Jump to address 4
    "--010000001--------1000-------", -- Load R0 with contents of R2
    "-------0000--------01110001100", -- Jump to address 12 (itself)
    others => "------------------------------"); -- Rest is don't care
```


DE HAAGSE HOGESCHOOL

Opgaven

- Stel dat het voorbeeldprogramma niet vanaf adres 0 geplaatst is, maar vanaf adres 5 (het programma zelf blijft ongewijzigd). Kan het programma daar zomaar “neergezet” worden? Motiveer.

- In het programma wordt direct na de twee add-instructies een sub-instructie uitgevoerd. Maar de add- en sub-instructies verwerken ook de carry-flag. Moet dan vóór de sub-instructie de flags niet gewist worden?

- De instructie `not Rx,#const` “bestaat” niet. Is deze instructie wel te maken?

- Stel dat er helemaal geen not-instructies zouden bestaan. Kan dan toch de NOT-operatie worden uitgevoerd/verwerkt? (ja dat kan). Motiveer.

Instructies

- Op de volgende slides worden de instructies in tabelvorm weergegeven. De instructies beginnen met een *mnemonic* en worden gevolgd door eventuele *operands*.

Instructie	Flags	Commentaar
ld Rx,#const	Z C	Laad Rx met de constante #const
ld Rx,Ry	Z C	Laad Rx met inhoud Ry
ld Rx,F	-	Laad Rx met flags
ld F,Rx	Z C	Laad Flags met Rx
ld Rx,IN	Z C	Laad Rx met externe input
nop	-	No operation

Instructies

Instructie	Flags	Commentaar
and Rx,Ry,Rz	Z	Logische AND: Rx := Ry AND Rz
or Rx,Ry,Rz	Z	Logische OR: Rx := Ry OR Rz
exor Rx,Ry,Rz	Z	Logische EXOR: Rx := Ry EXOR Rz
not Rx,Ry	Z	Logische NOT: Rx := NOT Ry
and Rx,Ry,#const	Z	Logische AND: Rx := Ry AND #const
or Rx,Ry,#const	Z	Logische OR: Rx := Ry OR #const
exor Rx,Ry,#const	Z	Logische EXOR: Rx := Ry EXOR #const

Instructies

Instructie	Flags	Commentaar
shl Rx,Ry	Z C	Logische schuif links: Rx := SHL Ry
shr Rx,Ry	Z C	Logische schuif rechts: Rx := SHR Ry
rol Rx,Ry	Z C	Logisch roteren links: Rx := ROL Ry
ror Rx,Ry	Z C	Logisch roteren rechts: Rx := ROR Ry
add Rx,Ry,Rz	Z C	Optellen: Rx := Ry + Rz + C
add Rx,Ry,#const	Z C	Optellen: Rx := Ry + #const + C
sub Rx,Ry,Rz	Z C	Aftrekken: Rx := Ry – Rz – C
sub Rx,Ry,#const	Z C	Aftrekken: Rx := Ry – #const – C

Instructies

Instructie	Flags	Commentaar
cmp Rx,#const	Z C	Vergelijk: Rx - #const – C (resultaat niet opgeslagen)
cmp Rx,Ry	Z C	Vergelijk: Rx – Ry – C (resultaat niet opgeslagen)
j/jmp @adres	-	Spring naar adres @adres
jz/je @adres	-	Spring naar adres @adres als Z = 1 (=)
jnz/jne @adres	-	Spring naar adres @adres als Z = 0 (≠)
jc/jl @adres	-	Spring naar adres @adres als C = 1 (<)
jnc/jge @adres	-	Spring naar adres @adres als C = 0 (≥)
jcz/jle @adres	-	Spring naar adres @adres als C+Z = 1 (≤)
jncz/jg @adres	-	Spring naar adres @adres als C+Z = 0 (>)

Opgave

- Ontwerp een programma dat de input eenmaal inleest en het aantal enen bepaalt in de ingelezen input. Hieronder is de pseudocode gegeven.

```
R3 := 0;                        // result
R1 := 8;                        // counter
R2 := input;                    // read input once
do                              // loop
     R2 := shr(R2);              // shift right R2, low ...
                                 //    ... bit in carry
     R3 := R3 + 0 + carry;       // add carry
     R1 := R1 – 1;               // ...
while (R1 ≠ 0);                 // as long as R1 ≠ 0
```

ALU instructietabel

acode	instr	flags	Comment
0000	nop	-	no operation (flags untouched)
0001	pass B	Z C	pass (B branch), Z and C affected
0010	add	Z C	arithmetic addition, Z and C affected
0011	sub	Z C	arithmetic subtraction, Z and C affected
0100	and	Z	logical AND, Z affected
0101	or	Z	logical OR, Z affected
0110	exor	Z	logical EXOR, Z affected
0111	not B	Z	logical NOT (B branch), Z affected
1000	shl B	Z C	logical shift left (B branch), Z and C affected
1001	shr B	Z C	logical shift right (B branch), Z and C affected
1010	rol B	Z C	rotate left through C (B branch), Z and C affected
1011	ror B	Z C	rotate right through C (B branch), Z and C affected
1100	wrf B	Z C	write flags (B branch), Z and C affected
1101	rdf	-	read flags (flags untouched)
1110	-	-	Reserved
1111	-	-	reserved

Jump tabel - Mux/decoder tabellen

jcode	test	flags	name	alt	comment
000	false	0	jn	jn	Jump never
001	≠	Z=0	jnz	jne	Jump not zero/jump not equal
010	=	Z=1	jz	je	Jump zero/jump equal
011	≥	C=0	jnc	jge	Jump not carry/jump greater or equal
100	<	C=1	jc	jl	Jump carry/jump less
101	>	C+Z=0	jncz	jg	Jump not carry or zero/jump greater
110	≤	C+Z=1	jcz	jle	Jump carry or zero/jump less or equal
111	true	1	j	jmp	Jump (always)

sela	select
00	R0
01	R1
10	R2
11	R3

selb	select
000	R0
001	R1
010	R2
011	R3
100	IN
101	constant
110	*
111	*

seld	select
00	R0
01	R1
10	R2
11	R3

DE HAAGSE HOGESCHOOL

Literatuur

- Digital System Design with VHDL – Mark Zwoliński, 2nd Ed, 2004
- Fundamentals of Digital Logic with VHDL Design – Stephen Brown, 3rd Ed, 2009
- Digital Logic and Computer Design, M. Morris Mano

Op Internet is genoeg te vinden over het ontwerpen/beschrijven van een microprocessor.
Een paar voorbeelden:

- http://opencores.org/usercontent,doc,1262702554
- http://people.tamu.edu/~akshitdayal/468/MIPS-Implementation.pdf
- http://www.nt-nv.fh-koeln.de/Labor/VhdlEnglish/Kap8/k832.html
- http://www.applelogic.org/Processors.html
- http://en.wikipedia.org/wiki/Status_register

Academie voor Technology, Innovation & Society Delft

Academie voor ICT & Media

De Haagse Hogeschool, Delft
+31-15-2606311
J.E.J.opdenBrouw@hhs.nl
www.dehaagsehogeschool.nl

DE HAAGSE HOGESCHOOL