# EE1410: Digitale Systemen

## BSc. EE, 1e jaar, 2012-2013, 10e hoorcollege

Arjan van Genderen, Stephan Wong, Computer Engineering

13-5-2013

# Hoorcollege 10

Boek
"Digital Logic":

- Register Transfer Level (RTL) systemen
  - Finite State Machine model
  - Datapad besturingsmodel
  - RTL model
  - data processing

7.14.4

- Delta I microprocessor
  - architectuur
  - operatie-voorbeelden
  - instructie set
  - instructie decoder
  - programmeervoorbeelden
  - vergelijking met dedicated hardware

7.14.1 tot "Using a Shift…"

7.14.2

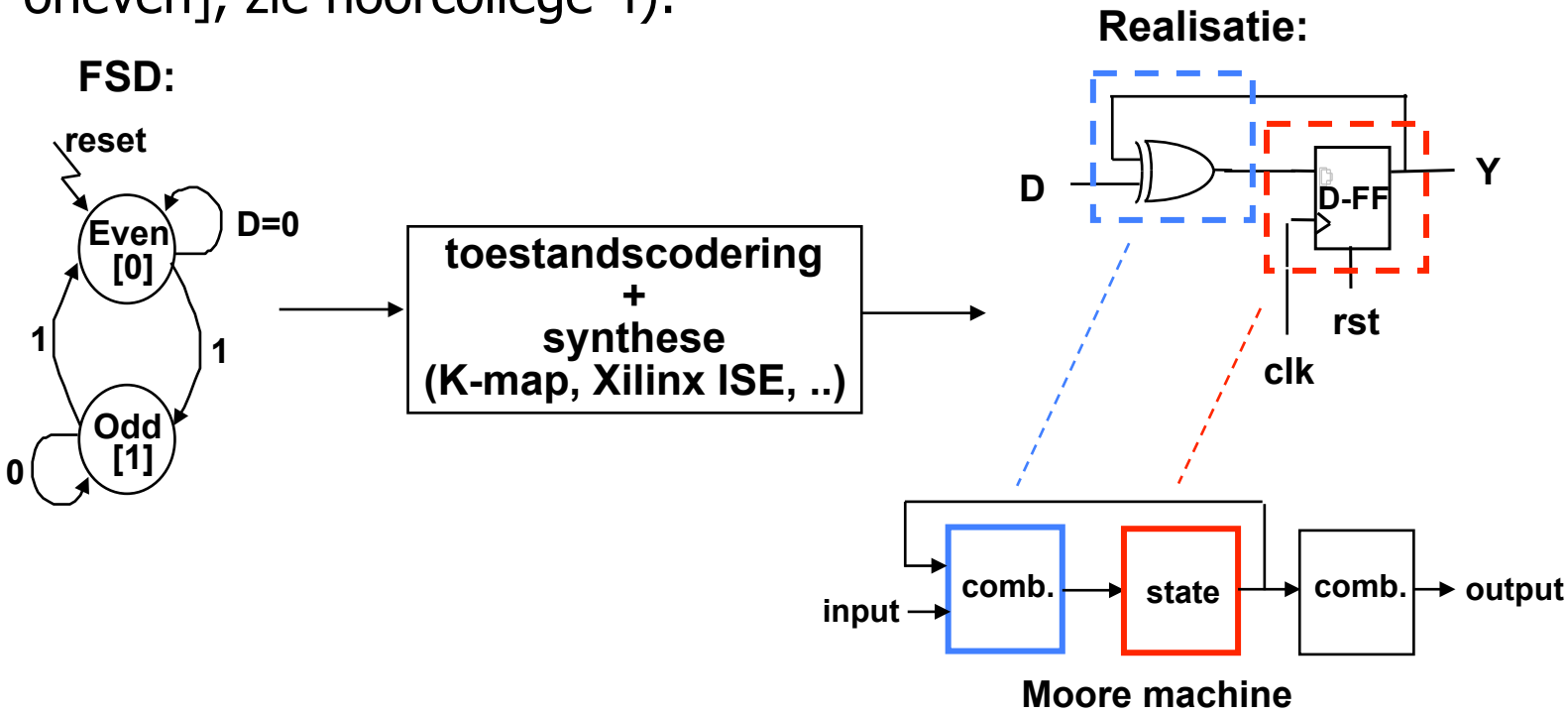
Bestudeer ook het document: RTL_Delta1 (op blackboard)

# Register Transfer Level (RTL) systemen

# Finite State Machine model

Kleine digitale systemen:
standaard opsplitsing in comb. en seq. (Moore/Mealy model) is belangrijk hulpmiddel bij ontwerp.

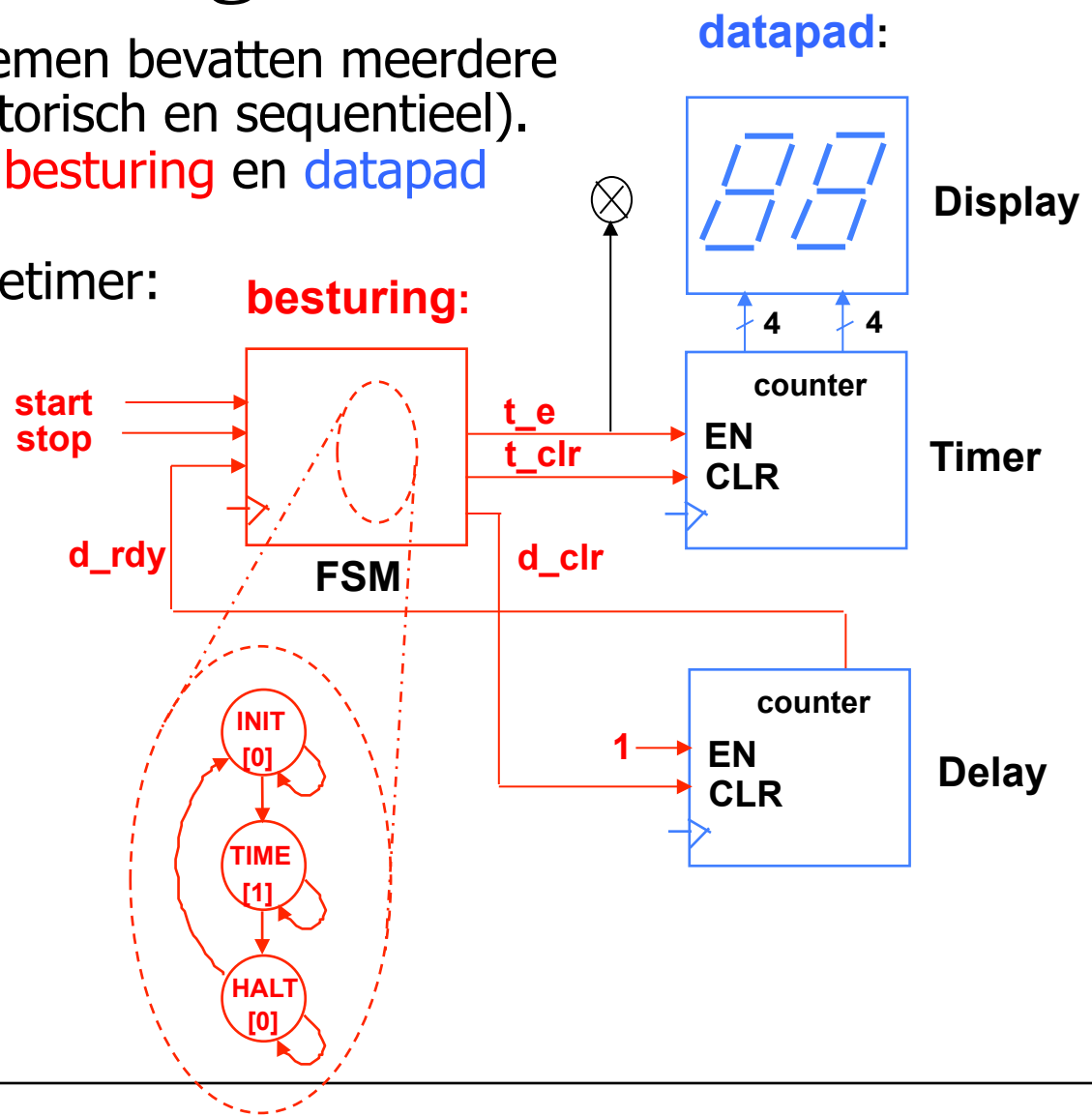
Bijvoorbeeld de parity checker (Y = [ aantal 1-en in invoer D = oneven], zie hoorcollege 4):



# Datapad besturingsmodel

Complexe digitale systemen bevatten meerdere schakelingen (combinatorisch en sequentieel). Goede opsplitsing is in besturing en datapad

Bijvoorbeeld een reactietimer:




Beide subsystemen zijn op zich comb./ sequentieel

# RTL level systemen

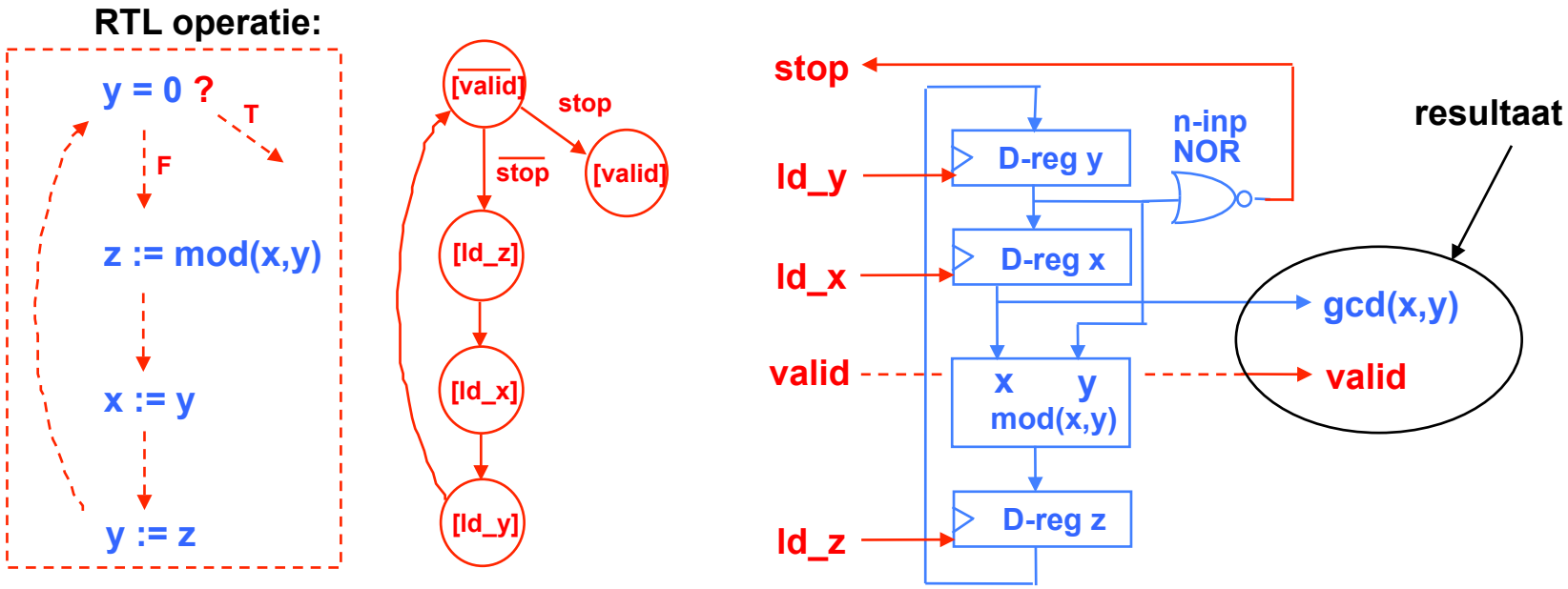
Veel grote dataverwerkende systemen worden op RTL nivo ontworpen (Register Transfer Level) waarbij *datapad* register-gebaseerd is en *besturing* een FSM.

Bijvoorbeeld een GCD-module (Greatest Common Divisor = grootste gemene deler) mbv. algoritme van Euclides:

gcd(x,y) = IF (y = 0) THEN x ELSE gcd(y,mod(x,y))

bv: gcd(24,15) = gcd(15,9) = gcd(9,6) = gcd(6,3) = gcd(3,0) = 3

Een voorbeeld RTL realisatie (D-reg met synchr. load; x en y zijn al in D-reg geladen):



# Data processing

Tot dusverre waren hardware-ontwerpen zeer *applicatie-specifiek*:
- geldt uiteraard mbt. besturing
- maar ook het datapad is applicatie-specifiek

voordeel: optimaal in termen van kosten (HW) en prestatie (snelheid)

nadeel: hogere ontwerpkosten en initiële fabricagekosten,
dus loont alleen bij zeer grote aantallen

Alternatief: standaardisatie van het datapad

- datapad wordt general-purpose RTL circuit (zgn. *data processor*),
- interactie tussen datapad en FSM ("programma") mbv.
  applicatie-*onafhanklijke* stuursignalen (bitcodes), *instructies* genoemd

voordeel: alleen instructiereeks (*programma*) is applicatie-specifiek
(simpeler, flexibeler, "software-oplossing")

nadeel: suboptimaal in hardware-kosten en -prestatie

# De Delta I Microprocessor

# Delta I Specificaties

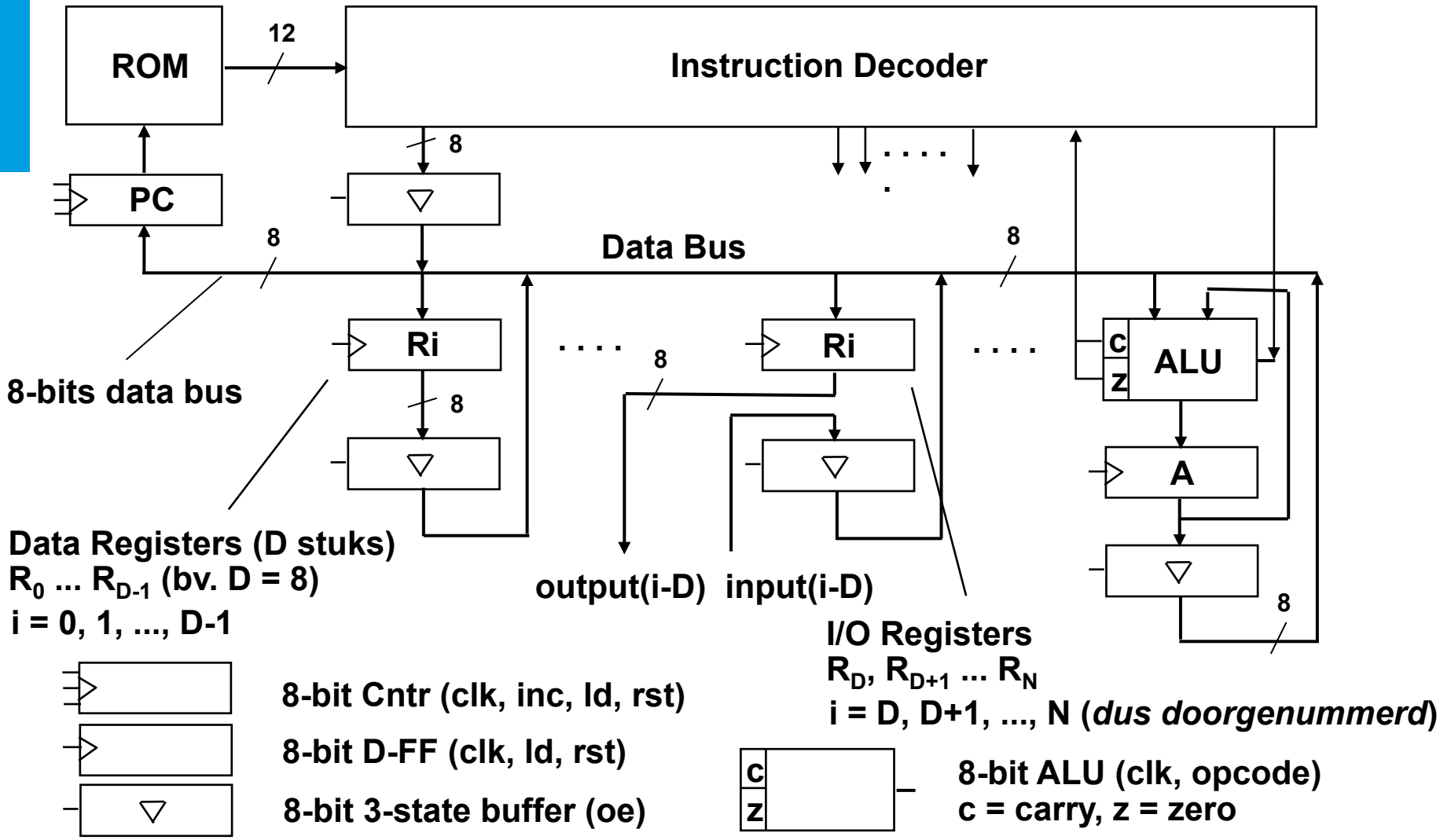
Features:
- 8-bits datapad (transporteert en rekent met 8 bits getallen)
- single cycle operation (1 operatie per actieve klok flank)
- Harvard architectuur: instructie-code en data zijn gescheiden
- gebruikt registers voor I/O (register-mapped I/O)
- simpele instructie set (slechts 16 1-operand instructies)

Doel van het ontwerp: *extreem simpel* uit didaktische overwegingen

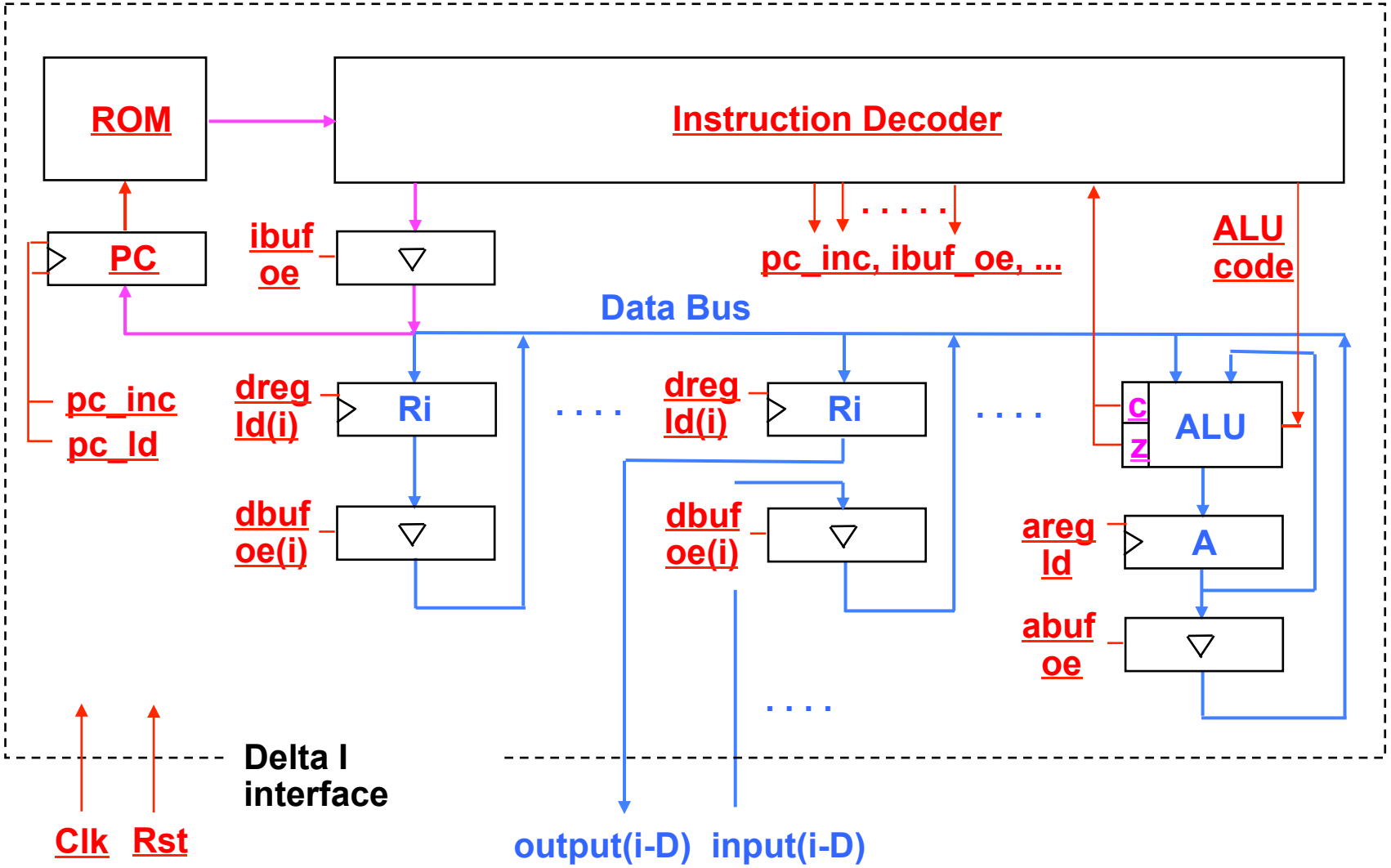
Hardware (RTL level):
- 1 $2^8$ x 12 bits ROM (tbv. het programma)
- 1 8-bits upcounter (program counter (PC), adresseert ROM)
- 1 8-bits ALU (de rekenkern van de processor)
- 1 8-bits register + 3-state buffer (tbv. de ALU accumulator)
- N 8-bits registers + 3-state buffers, afhankelijk van applicatie (N = 0...256)
- wat besturingscombinatoriek (instruction decoder)

De processor inclusief ROM past in een eenvoudige FPGA

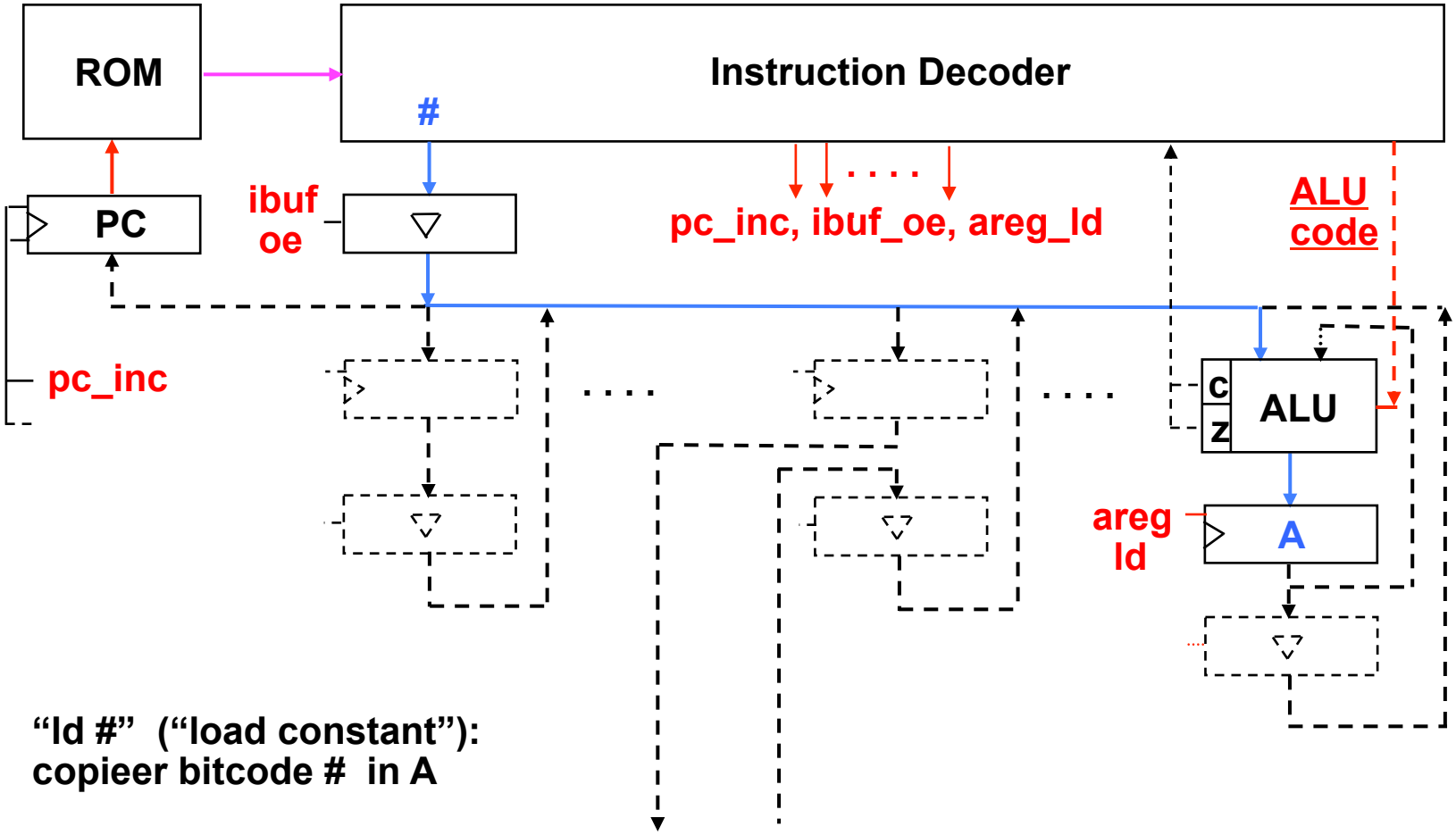
# Delta I Circuit Overzicht



# Delta I besturing vs datapad



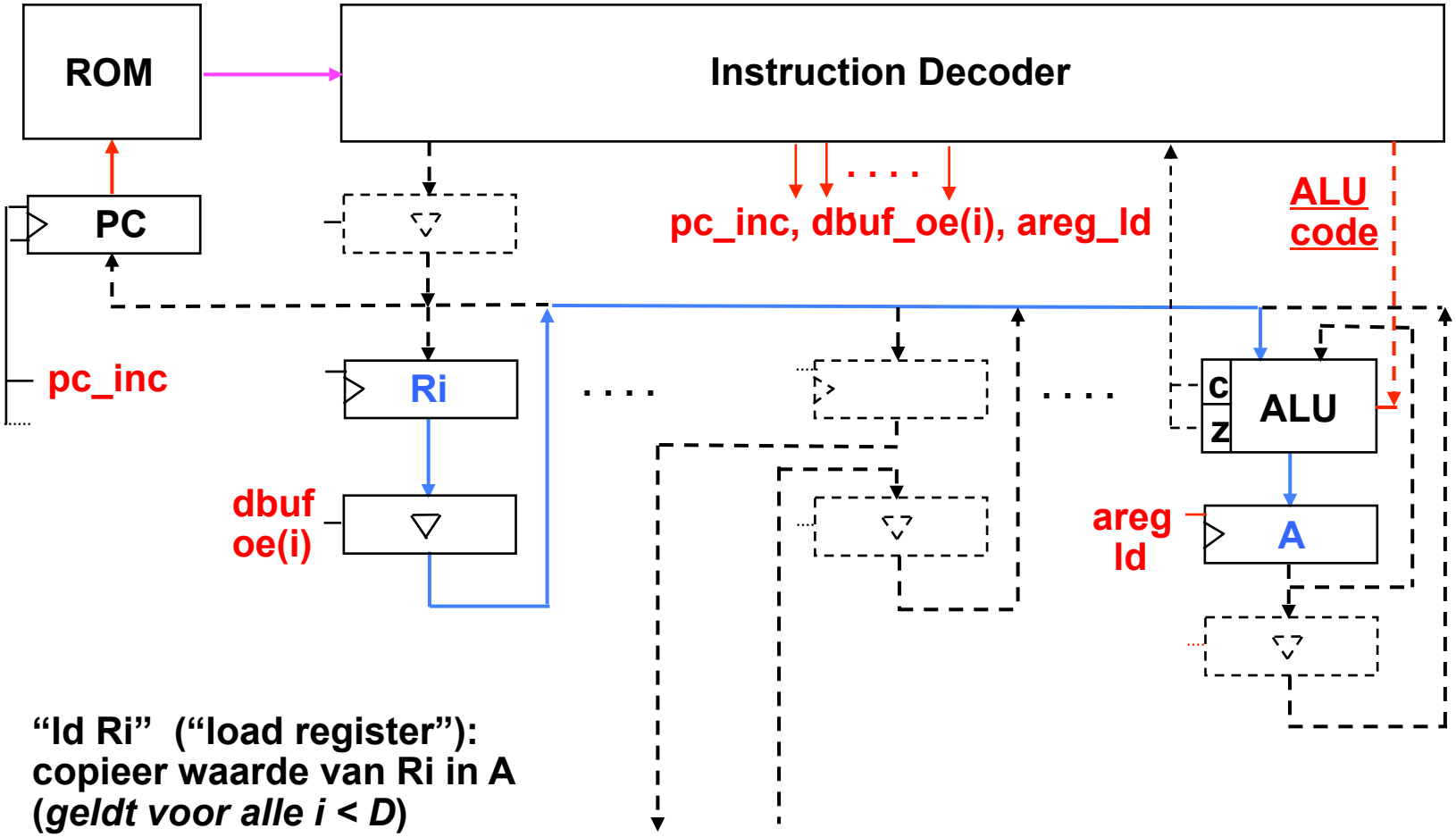
# Delta I “ld #” operatie




**“ld #” (“load constant”): copieer bitcode # in A**

**vb: ld $FF_H$**

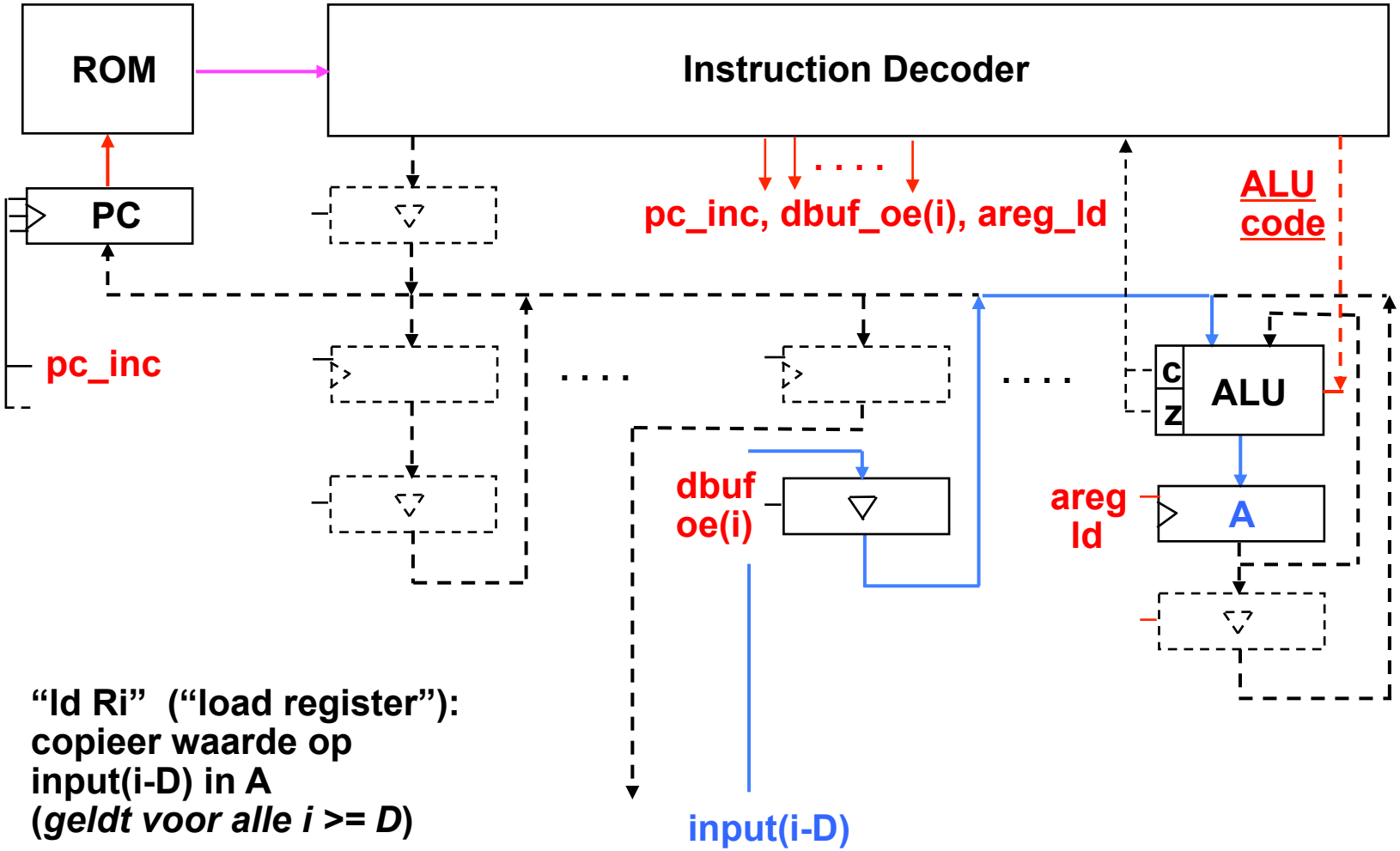
# Delta I “ld Ri” operatie (1)




**“ld Ri” (“load register”): copieer waarde van Ri in A (*geldt voor alle i < D*)**

**vb: ld R0**

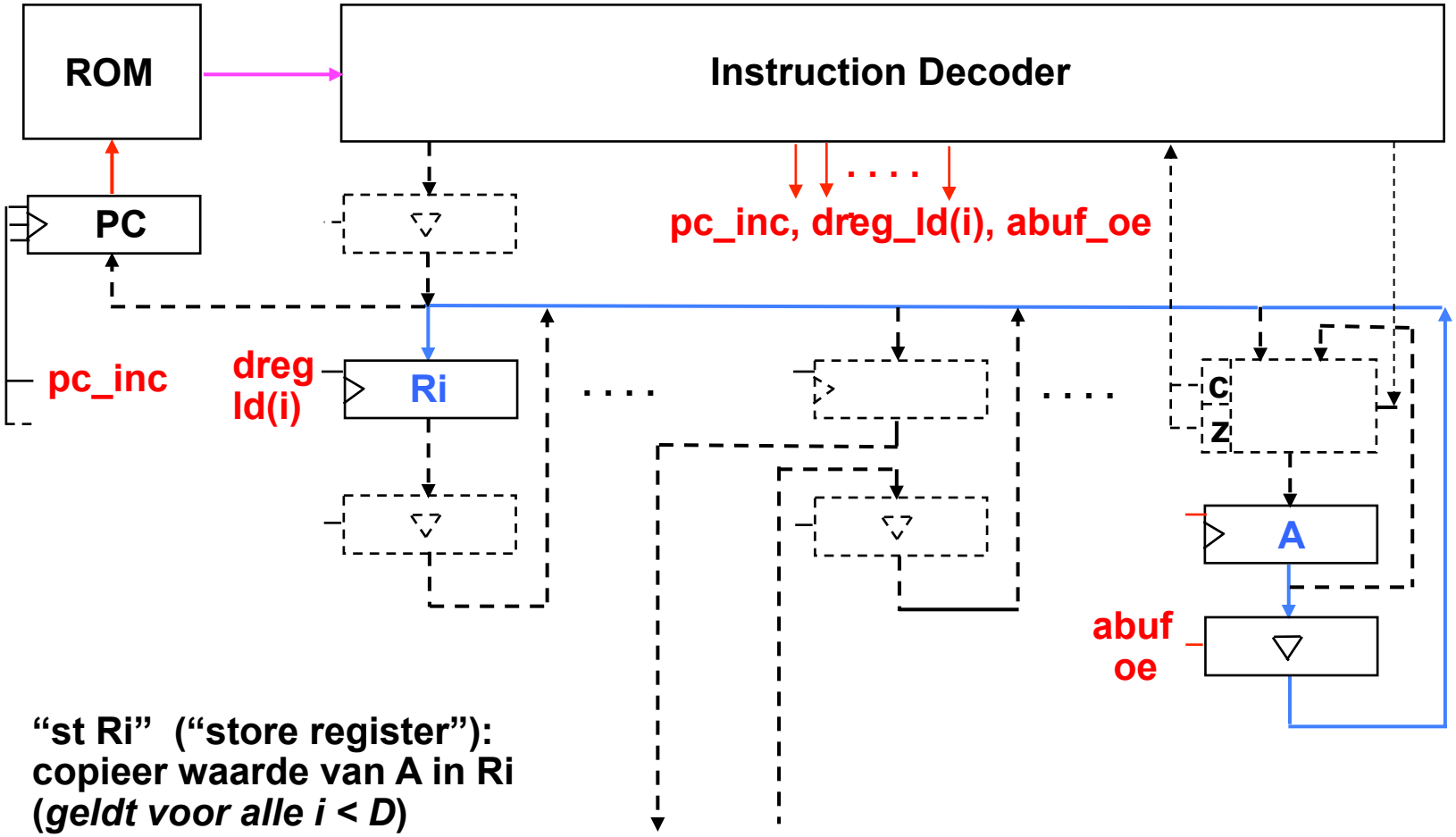
# Delta I “ld Ri” operatie (2)




**“ld Ri” (“load register”): copieer waarde op input(i-D) in A (*geldt voor alle i >= D*)**

**vb: ld R255**

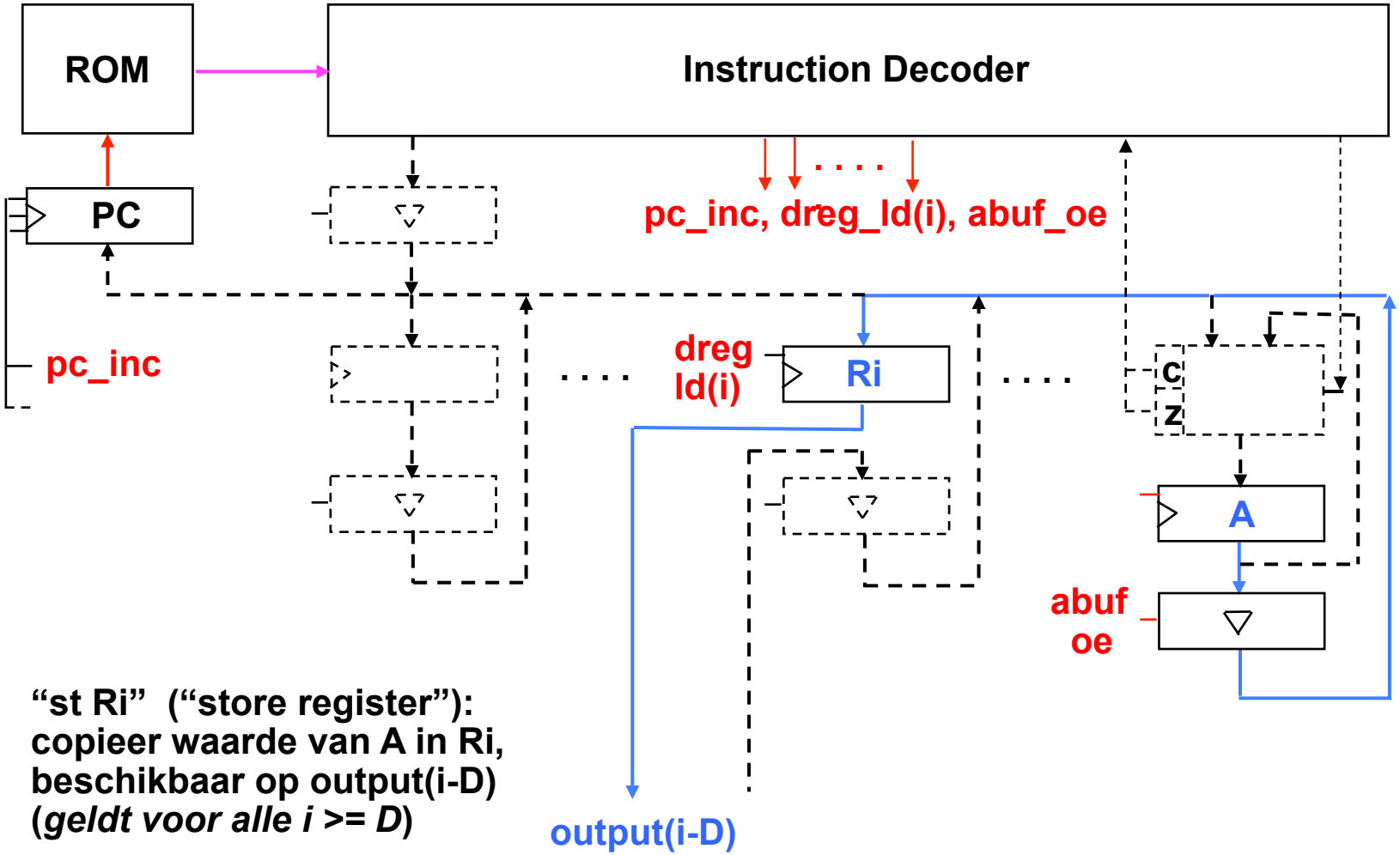
# Delta I “st Ri” operatie (1)




**“st Ri” (“store register”):**
**copieer waarde van A in Ri**
**(*geldt voor alle i < D*)**

**vb: st R0**

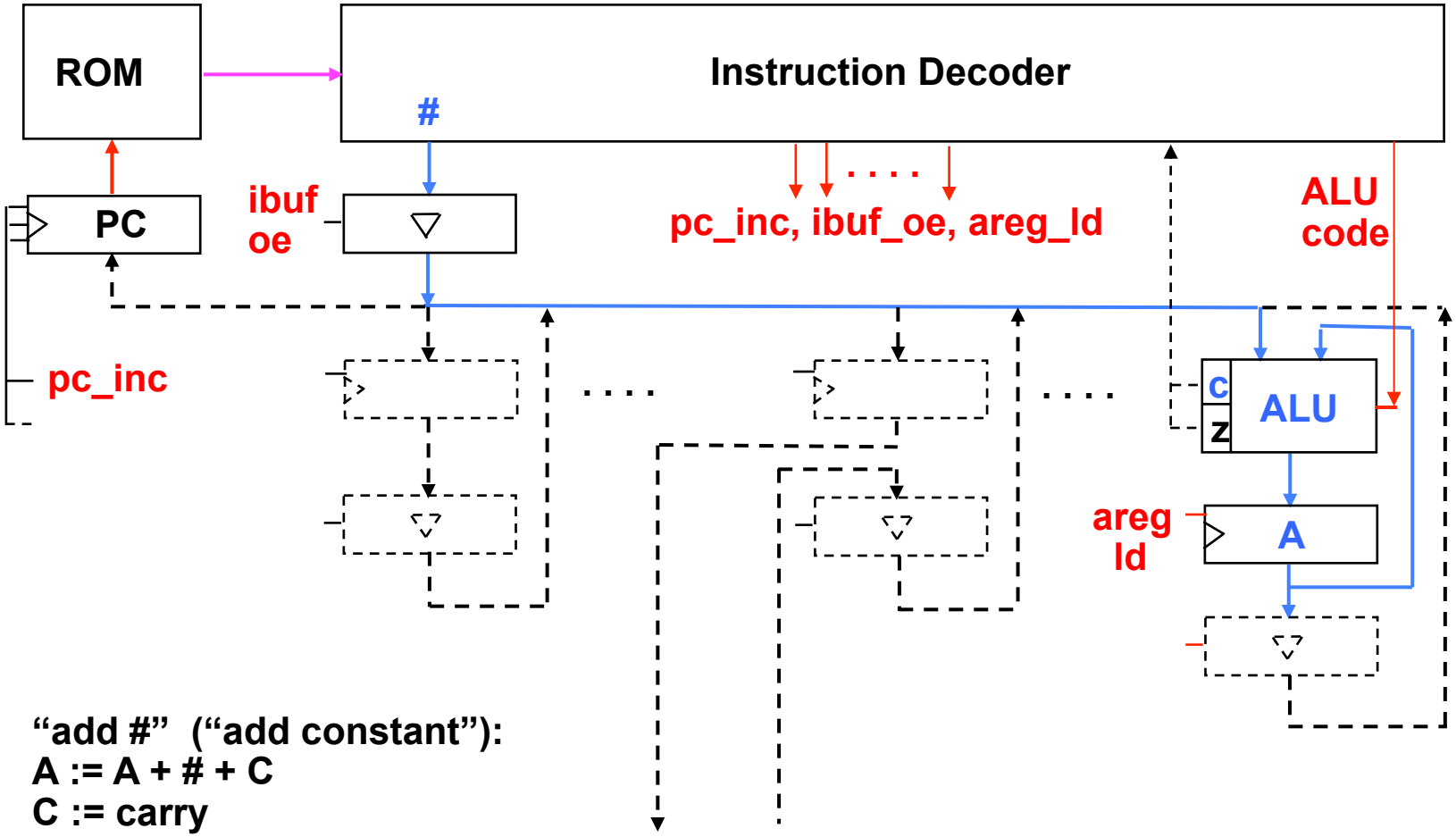
# Delta I “st Ri” operatie (2)




**“st Ri” (“store register”): copieer waarde van A in Ri, beschikbaar op output(i-D) (*geldt voor alle i >= D*)**

**vb: st R8**

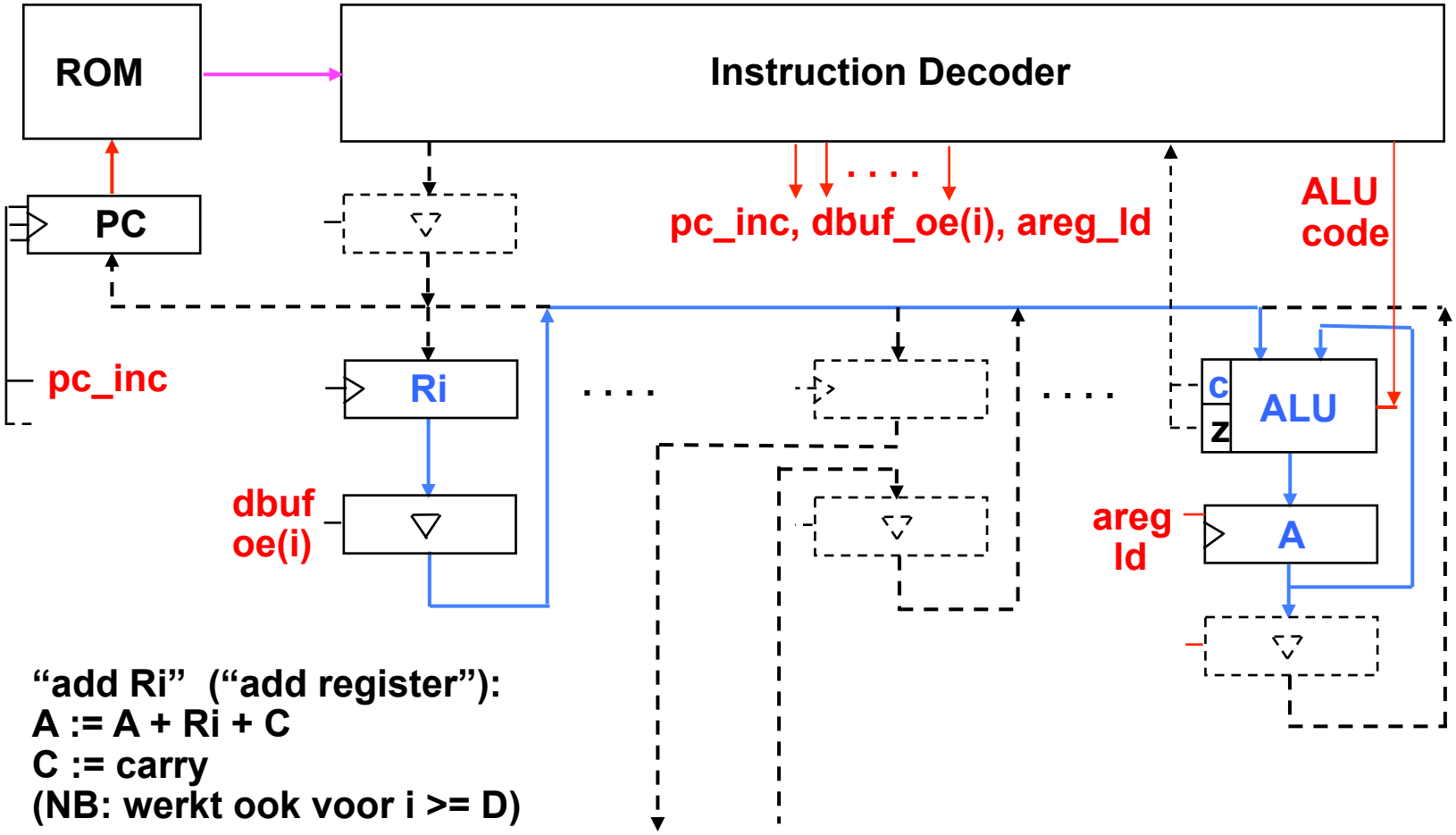
# Delta I “add #” operatie




**“add #” (“add constant”):**
**A := A + # + C**
**C := carry**

**vb: add $FF_H$**
**(NB: vergelijkbare versies voor andere ALU operaties)**

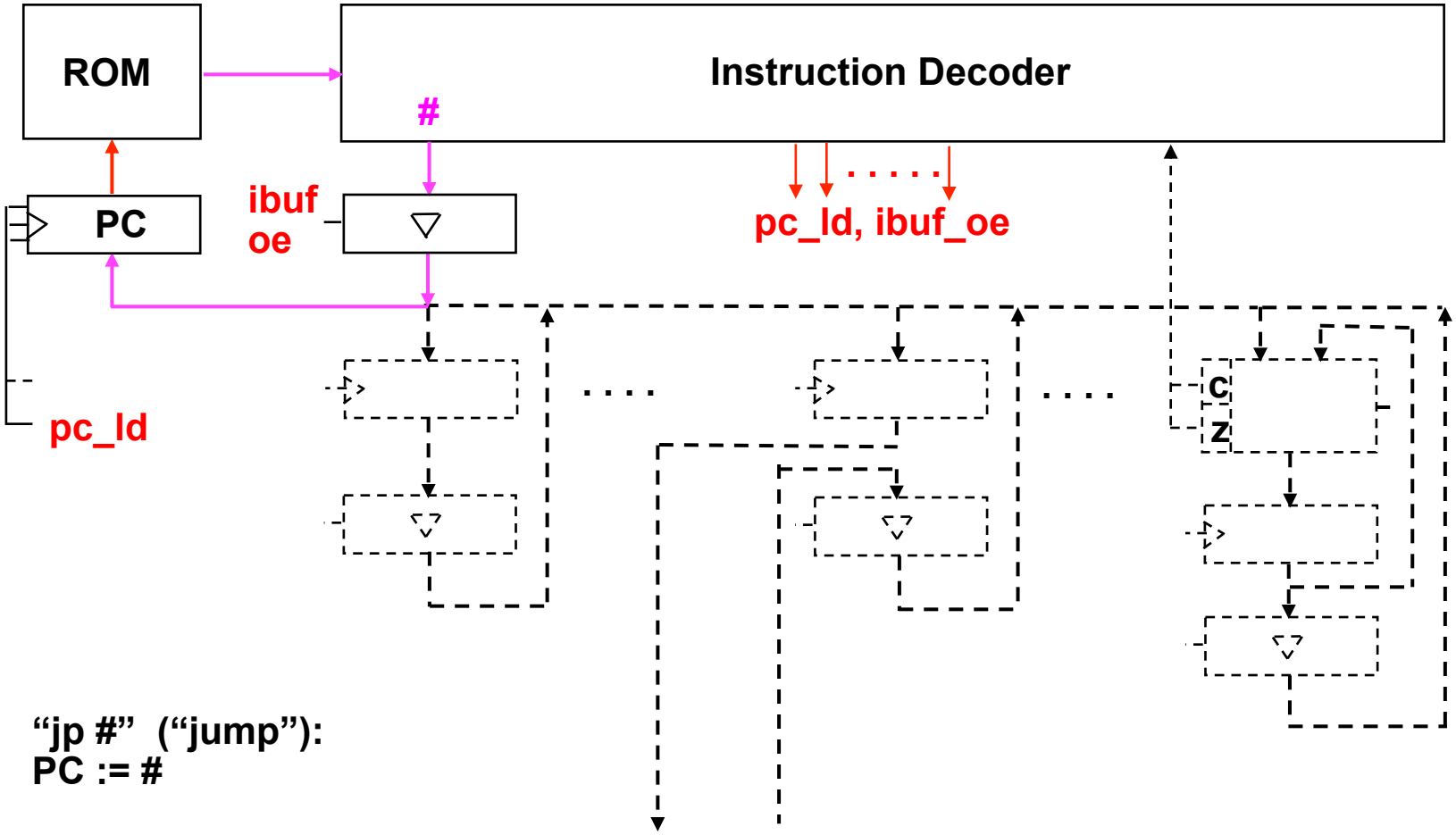
# Delta I “add Ri” operatie




**“add Ri” (“add register”):**
**A := A + Ri + C**
**C := carry**
**(NB: werkt ook voor i >= D)**

**vb: add R0 (of ook: add R8)**
**(NB: vergelijkbare versies voor andere ALU operaties)**

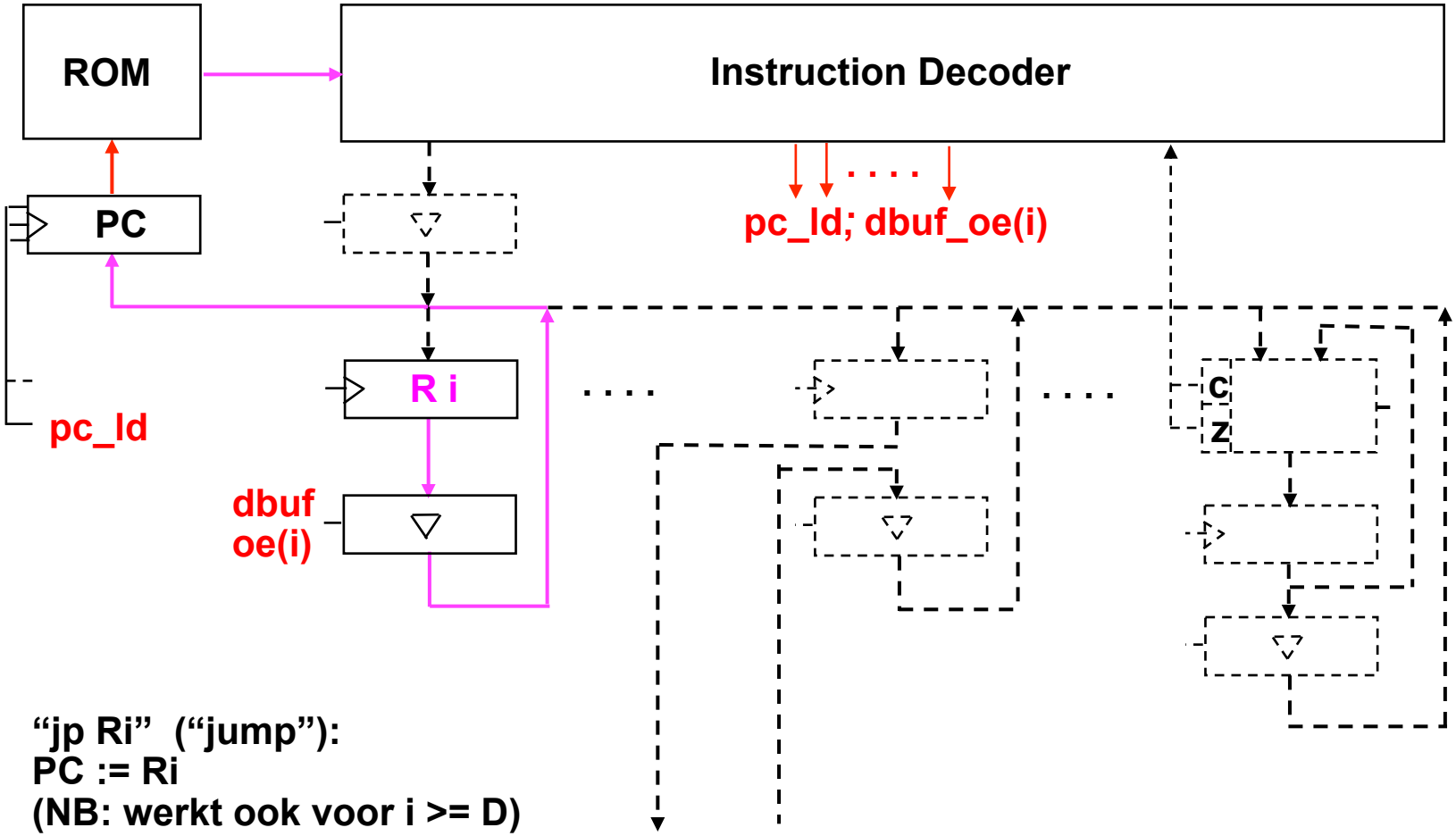
# Delta I “jp #” operatie




**“jp #” (“jump”):**
**PC := #**

**vb: jp 0**

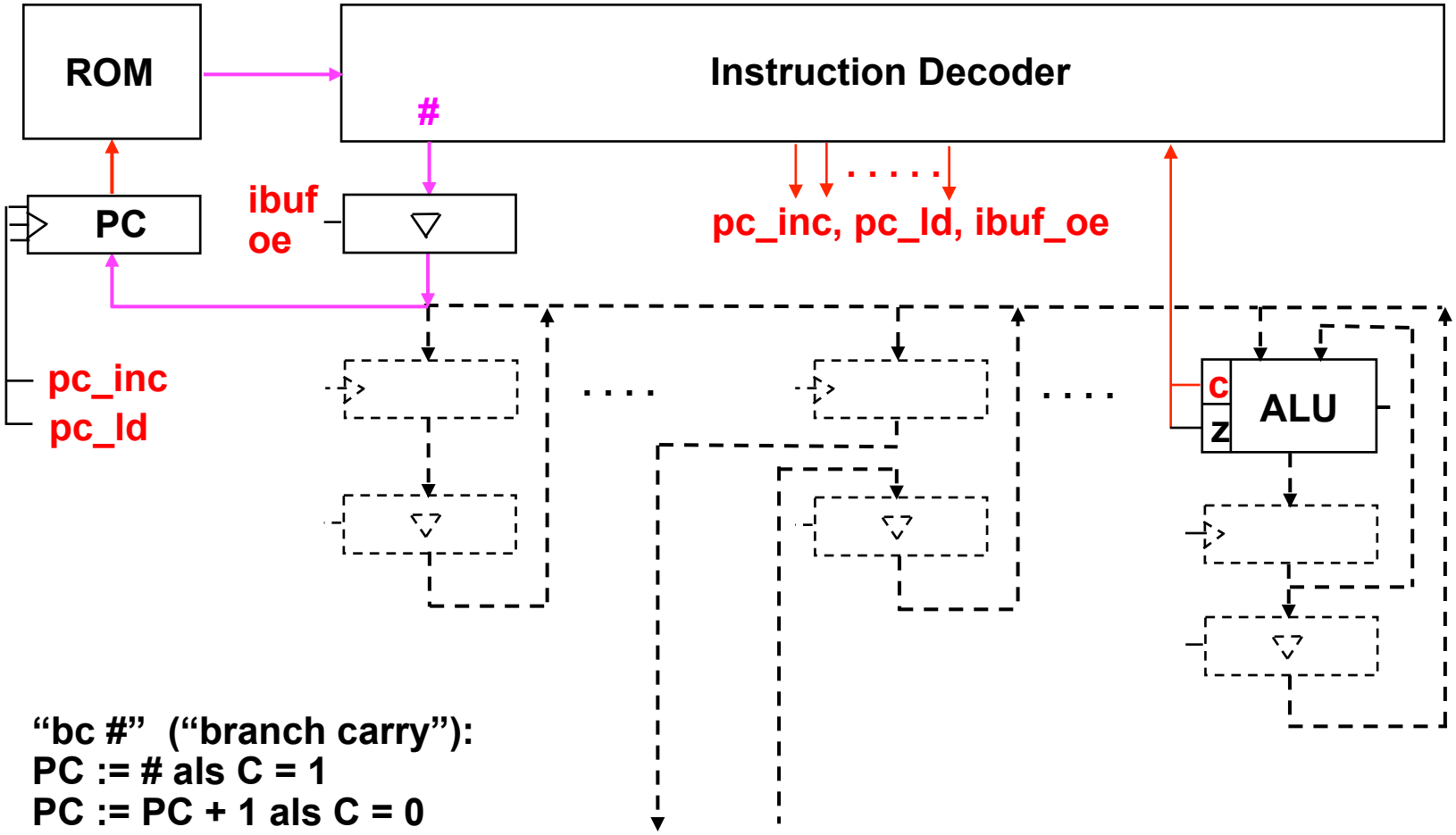
# Delta I “jp Ri” operatie




**“jp Ri” (“jump”):**
**PC := Ri**
**(NB: werkt ook voor i >= D)**

**vb: jp R0 (ook: jp R8)**

# Delta I “bc #” operatie




**“bc #” (“branch carry”):**
**PC := # als C = 1**
**PC := PC + 1 als C = 0**

**vb: bc 0**
**(NB: vergelijkbare versie voor Z (bz “branch zero”))**

# Delta I Instructie-set

| instr. | werking | toelichting |
|---|---|---|
| nop | no operation | voor vertraging of reservering |
| set c | C := 1 | zet de carry op 1 |
| clr c | C := 0 | zet de carry op 0 |
| ld # | A := # | load constant |
| ld $R_i$ | A := $R_i$ | load register |
| st $R_i$ | $R_i$ := A | store register |
| add # | A := (A + # + C) mod 256 | C := (A + # + C >= 256) |
| add $R_i$ | A := (A + $R_i$ + C) mod 256 | C := (A + $R_i$ + C >= 256) |
| xor # | A := A XOR # | bitwise XOR |
| xor $R_i$ | A := A XOR $R_i$ | bitwise XOR |
| and # | A := A AND # | bitwise AND, Z := (A AND # = 0) |
| and $R_i$ | A := A AND $R_i$ | bitwise AND, Z := (A AND $R_i$ = 0 |
| jp # | PC := # | onvoorwaardelijke sprong |
| jp $R_i$ | PC := $R_i$ | onvoorwaardelijke sprong |
| bz # | Z => PC := # | Z' => PC := PC + 1 |
| bc # | C => PC := # | C' => PC := PC + 1 |

**Tenzij anders vermeld geldt voor iedere instructie:**
**PC := PC + 1 en Z, C blijven ongewijzigd**

# Delta I Instruction Decoder

| Instr. name | Instruction Opcode | Operand | pc inc | pc ld | ibuf oe | areg ld | abuf oe | dreg ld | dbuf oe | ALU code |
|---|---|---|---|---|---|---|---|---|---|---|
| nop | 0000 | - - - - - - - - | 1 | 0 | 0 | 0 | 0 | 0 | 0 | - - - |
| set c | 0001 | - - - - - - - - | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 001 |
| clr c | 0010 | - - - - - - - - | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 010 |
| ld # | 0011 | # | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 000 |
| ld $R_i$ | 0100 | i | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 000 |
| st $R_i$ | 0101 | i | 1 | 0 | 0 | 0 | 1 | 1 | 0 | - - - |
| add # | 0110 | # | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 011 |
| add $R_i$ | 0111 | i | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 011 |
| xor # | 1000 | # | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 100 |
| xor $R_i$ | 1001 | i | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 100 |
| and # | 1010 | # | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 101 |
| and $R_i$ | 1011 | i | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 101 |
| jp # | 1100 | # | 0 | 1 | 1 | 0 | 0 | 0 | 0 | - - - |
| jp $R_i$ | 1101 | i | 0 | 1 | 0 | 0 | 0 | 0 | 1 | - - - |
| bz # | 1110 | # | Z’ | Z | 1 | 0 | 0 | 0 | 0 | - - - |
| bc # | 1111 | # | C’ | C | 1 | 0 | 0 | 0 | 0 | - - - |

**vb: de bewerking R8 := R0 + 64 correspondeert met de 3 volgende instructies:**
**ld R0 = 0100 00000000 ; add $40_H$ = 0110 01000000 ; st R8 = 0101 00001000**

# Delta I Programmeervoorbeeld

**Berekening van R0 := R1 + 15:**

| address | bitcode | instructie | effect na volgende klokflank |
|---|---|---|---|
| 0000 | 0010 00000000 | clr c | C := 0, PC := PC + 1 |
| 0001 | 0100 00000001 | ld R1 | A := R1, PC := PC + 1 |
| 0010 | 0110 00001111 | add 15 | A := A + 15, PC := PC + 1 |
| 0011 | 0101 00000000 | st R0 | R0 := A, PC := PC + 1 |
| 0100 | 1100 00000000 | jp 0 | PC := 0 |

**Manier waarop men in de praktijk een programma opschrijft (assembly code):**

```
label     mnemonic        commentaar
begin:    clr c           ; bereid optelling voor
          ld R1           ; R0 = R1 + 15
          add 15
          st R0
          jp begin        ; herhaal programma
```

# Delta I: gcd-applicatie (1)

```
; Delta I configuratie:
;
; D = 4 data-registers:        R0 (x), R1 (y), R2 (z), R3 (return adres)
; 3 I/O-registers:             R4 (x), R5 (y), R6 (gcd resultaat)
;
; Algoritme:
;
;           als (y = 0) dan
;                     klaar: x is gcd
;           anders
;                     z := x mod y
;                     x := y
;                     y := z
;                     herhaal algoritme
```

Vergelijkbaar C programma:

```
int x, y, z;

fscanf (fp0, "%d", &x);
fscanf (fp1, "%d", &y);

while(y){
    z = x % y;
    x = y;
    y = z;
}

fprintf (fp2, "%d", x);
```

# Delta I: gcd-applicatie (2)

```
gcd:    ld R4          ; x := input(0)
        st R0
        ld R5          ; y := input(1)
        st R1
loop:   ld R1          ; test y = 0
        and 11111111b
        bz done        ; y = 0 dus zijn we klaar
        ld after       ; R3 := terugkeeradres na executie van mod routine
        st R3
        jp mod         ; spring naar mod routine => z := mod(x,y)
after:  ld R1          ; x := y
        st R0
        ld R2          ; y := z
        st R1
        jp loop        ; herhaal algoritme
done:   ld R0          ; output(2) := x
        st R6
        jp gcd         ; doe een nieuwe gcd
```

# Delta I: gcd-applicatie (3)

```
; Routine: bereken z := x mod y
;
; Algoritme:
;
;           z := x
;           herhaal z := z - y totdat z < 0
;           z := z + y

mod:        ld R0               ; z := x
            st R2
subtr:      ld R1               ; A := y
            xor 11111111b       ; A := y'
            set c               ; A := y' + 1 = -y
            add R2              ; A := z - y        (wat zijn de eisen aan x en y?)
            st R2               ; z := z - y
            and 10000000b       ; z < 0 ?
            bz subtr            ; nee, herhaal subtr
            ld R2               ; ja, A := z
            clr c               ; geen verrassingen
            add R1              ; A := z + y
            st R2               ; z := z + y
            jp R3               ; return naar adres in R3
```

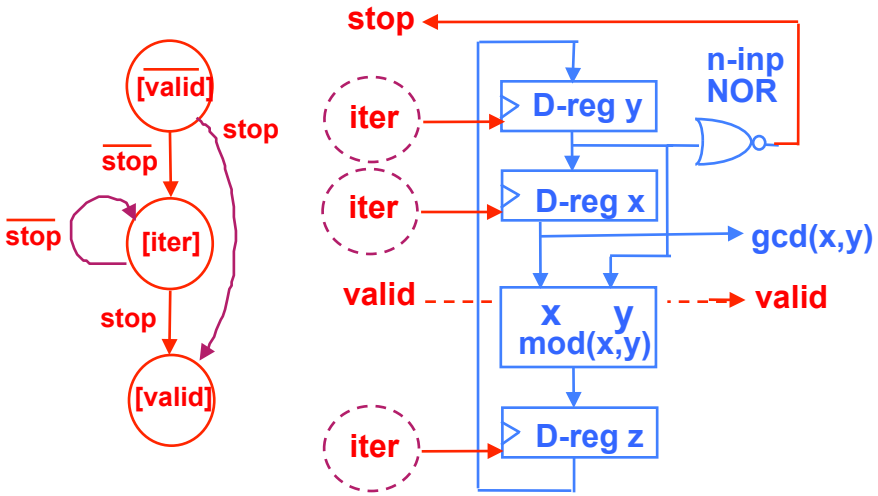
# Hardware versus software

**Vergelijking mbt. gcd-applicatie (1 gcd iteratie, stel mod functie kost 1 klokcyclus):**

**Vorige hardware-oplossing: 4-state FSM => 4 cycli per iteratie**

**Delta I software-oplossing: 9 cycli/instructies wanneer de "mod" operatie ≡ {ld after, st R3, jp mod}, wordt geteld als 1 instructie**

**De snelste HW-oplossing kan echter itereren in 1 cyclus!**




**Snelheidsverhouding 9:1 en bovendien zijn de hardwarekosten lager….**

**Afweging tussen SW-oplossing en HW-oplossing is niet triviaal en zeker applicatie-specifiek**

# Samenvatting

- Wat is het datapad besturingsmodel?
- Wat zijn RTL systemen?
- Wat zijn processoren?
- Hoe werken processoren?
- Wat is een software-oplossing?
- Wat zijn de verschillen tussen SW- en HW-oplossingen?

Deze week: bekendmaking VHDL opdracht kwartaal 4
Vanaf volgende week vrijdag: werkcolleges
Vrijdag 7 juni: vragencollege