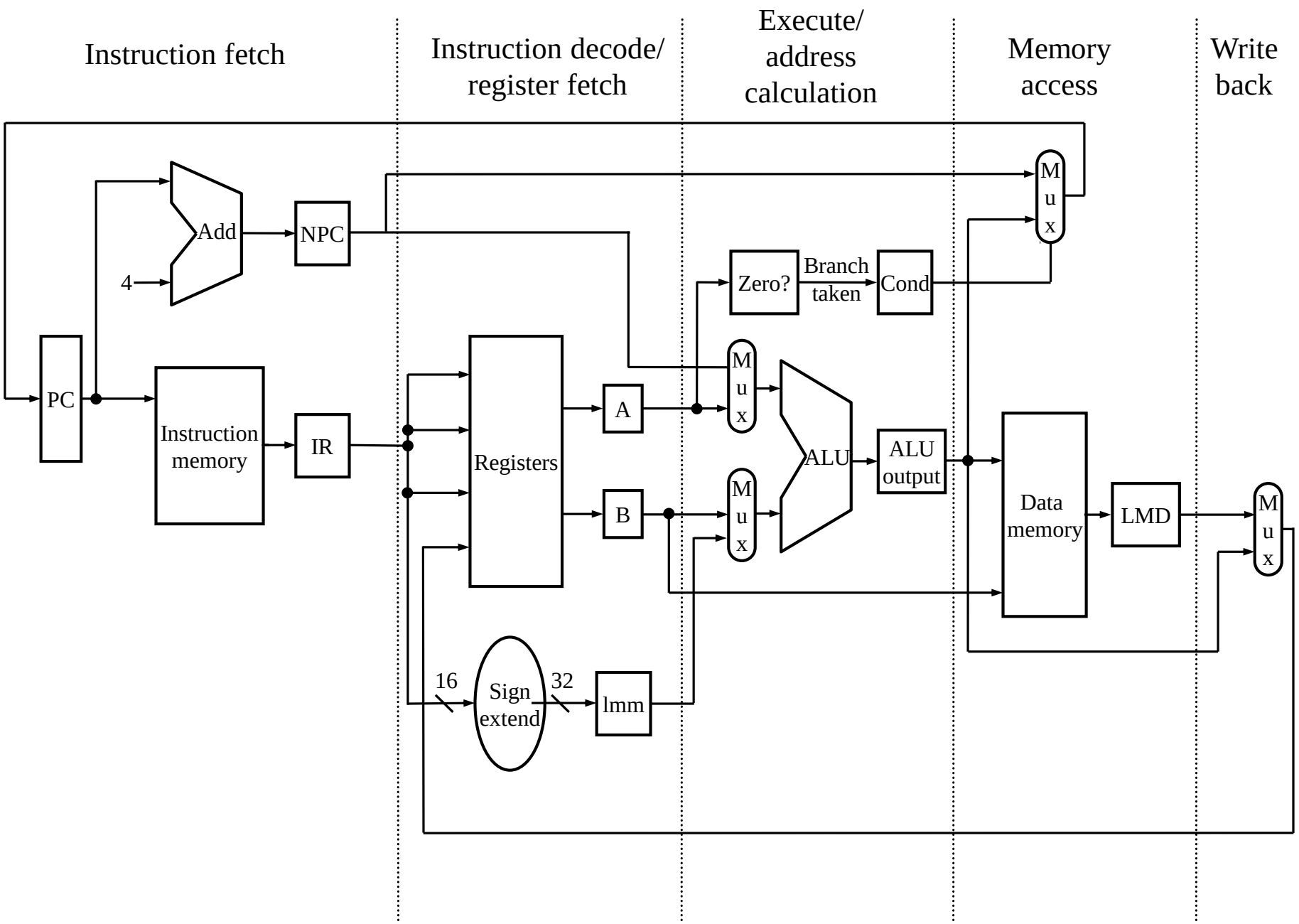
The DLX-560 Instruction Architecture

We have learnt several things in the previous chapter and now we want to apply them by implementing them in an example architecture:
DLX, pronounced as deluxe
What are the specifications?

Specifications of DLX

- GPR-Architecture, load-store (=Register-Register)
- Addressing: Displacement, Immediate, Indirect
- Fast simple instructions (load, store, add, ...)
- 8-bit, 16-bit, 32-bit Integer
- 32-bit, 64-bit Floating-point
- fixed-length instruction format, few formats
- at least 16 GPRs



DLX-Data Path with Clock Cycle

Register

The processor has **32 GPRs.**

Each register is 32-bit long.

They are represented in the form of R0,..,R31.

R0 has the value 0 and cannot be overwritten (Writing to R0 has no effect)

R31 stores the return address of Jump and Link-instructions

Furthermore, **32 FP-Register exists. Each is 32 bit long.**

F0,..,F31

This can be used as individual Single-Register pairwise as Double-Register F0, F2,...,F30.

Between the register of different kinds, special Move instructions exists

Data Type
8-bit bytes.
16-bit half words.
32-bit word.
For Integers. All these are either unsigned Integer or in 2's complement.
32-bit Singles.
64-bit Doubles.
In IEEE Standard 754.

Loading of bytes and half words can either be done with leading 0s (unsigned) or with replication of the sign bit (2’s complement).

Addressing Modes
Displacement and Immediate
Through reasonable usage of R0 and 0, four addressing modes can be realised:

Displacement:	**Load R1, 1000(R2);**
Immediate:	**Load R1, #1000;**
Indirect:	**Load R1, 0(R2);**
Direct:	**Load R1, 1000(R0);**

Instructions Format

I – Instruction

Loads and Stores of bytes, words, half words
All immediate-instructions (rd ← rs1 op immediate)

Conditional Branching (rs1 : register, rd unused)
Jump register, Jump and link register
(rd = 0, rs1 = destination, immediate = 0)

R – Instruction

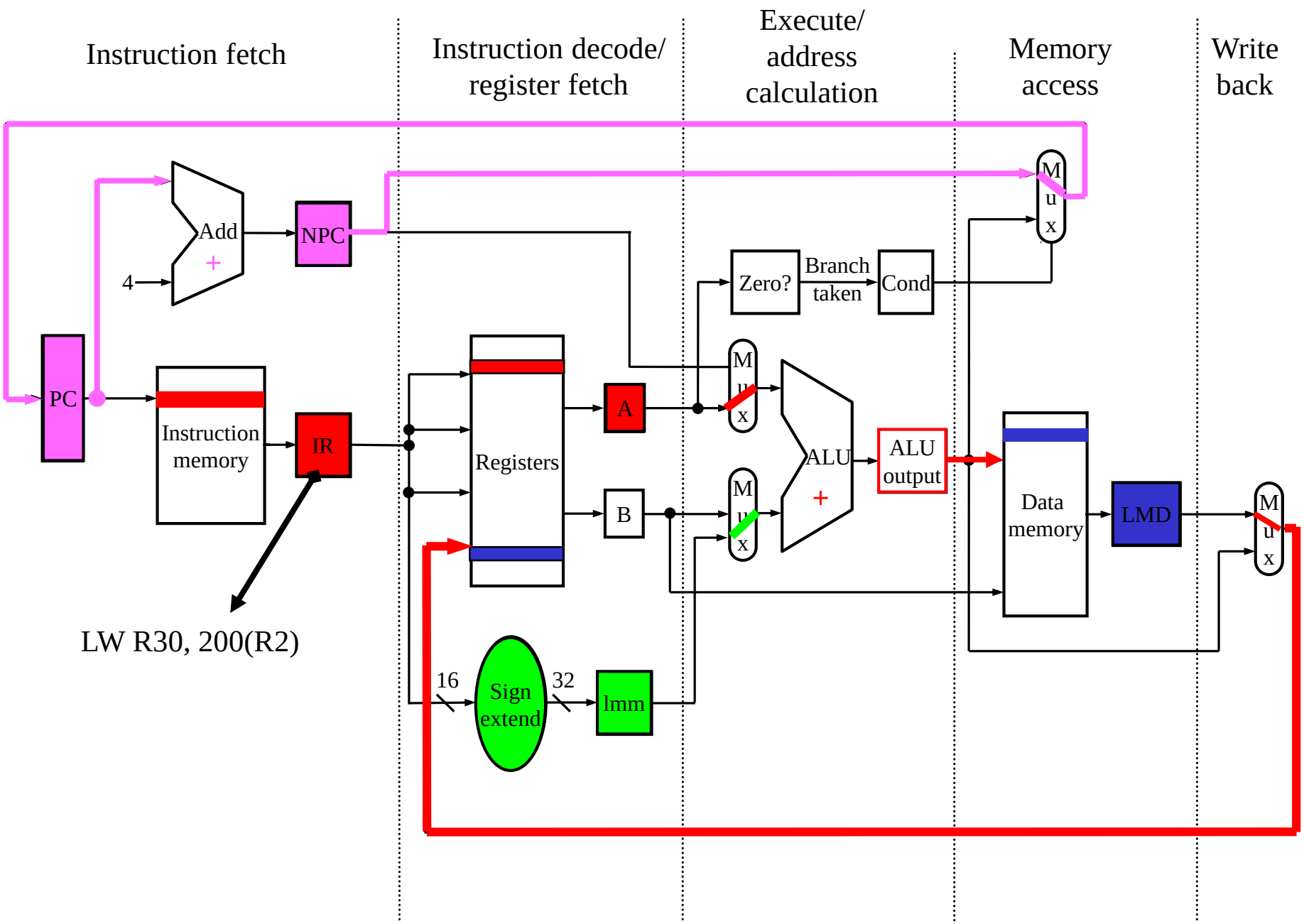
Register-Register ALU Operations: rd ← rs1 func rs2
func (Function) determines what shall be done: Add, Sub, ...
Read/write to special registers and moves

J – Instruction

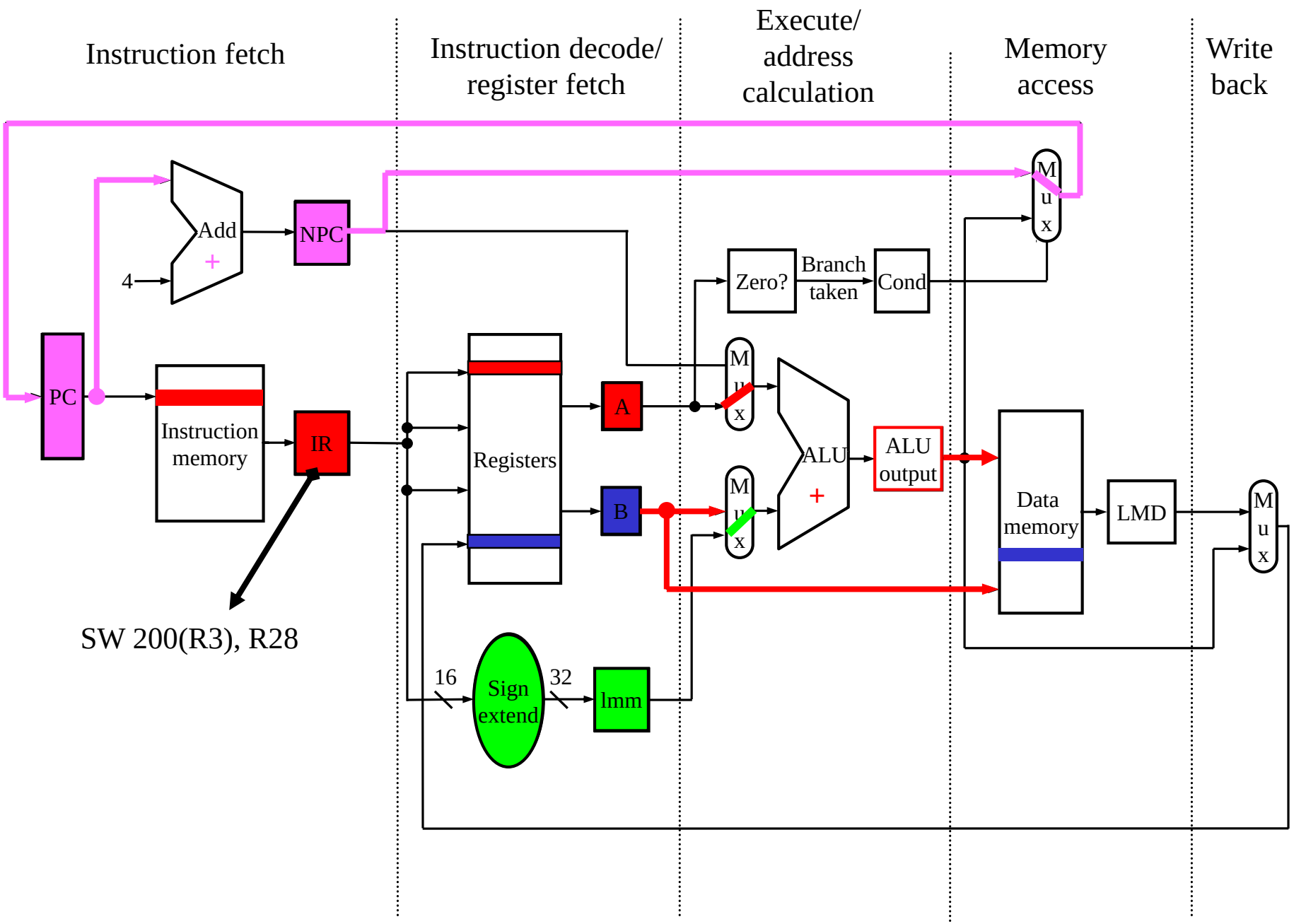
Jump and Jump and link
Trap and Return from exception

Instructions with Memory Access

Instructions		**Name**	**Explanations**
LW	R1,30(R2)	Load word	Regs [R1] $\leftarrow_{32}$Mem [30+Regs [R2]]
LW	R1,1000(R0)	Load word	Regs [R1] $\leftarrow_{32}$ Mem [1000+0]
LB	R1,40(R3)	Load byte	Regs [R1] $\leftarrow_{32}$ (Mem [40+Regs [R3]]$_0$)^{24} # # Mem[40+Regs[R3]]
LBU	R1,40(R3)	Load byte unsigned	Regs [R1] $\leftarrow_{32}$ 0^{24} # # Mem[40+Regs [R3]]
LH	R1,40(R3)	Load half word	Regs [R1] $\leftarrow_{32}$ (Mem[40+Regs [R3]]$_0$) ^{16} # # Mem [40+Regs [R3]] # # Mem[41+Regs [R3]]
LF	F0,50(R3)	Load float	Regs [F0] $\leftarrow_{32}$ Mem [50+Regs [R3]]
LD	F0,50(R2)	Load double	Regs [F0] # #Regs [F1] $\leftarrow_{64}$ Mem[50+Regs [R2]]
SW	500(R4),R3	Store word	Mem [500+Regs [R4]] $\leftarrow_{32}$ Regs [R3]
SF	40(R3),F0	Store float	Mem [40+Regs [R3]] $\leftarrow_{32}$ Regs [F0]
SD	40(R3),F0	Store double	Mem[40+Regs [R3]] $\leftarrow_{32}$ Regs [F0]; Mem[44+Regs [R3]] $\leftarrow_{32}$ Regs [F1]
SH	502(R2),R3	Store half	Mem[502+Regs [R2] $\leftarrow_{16}$ Regs [R31]]$_{16...31}$
SB	41(R3),R2	Store byte	Mem[41+Regs [R3]] $\leftarrow_{8}$ Regs [R2]$_{24...31}$



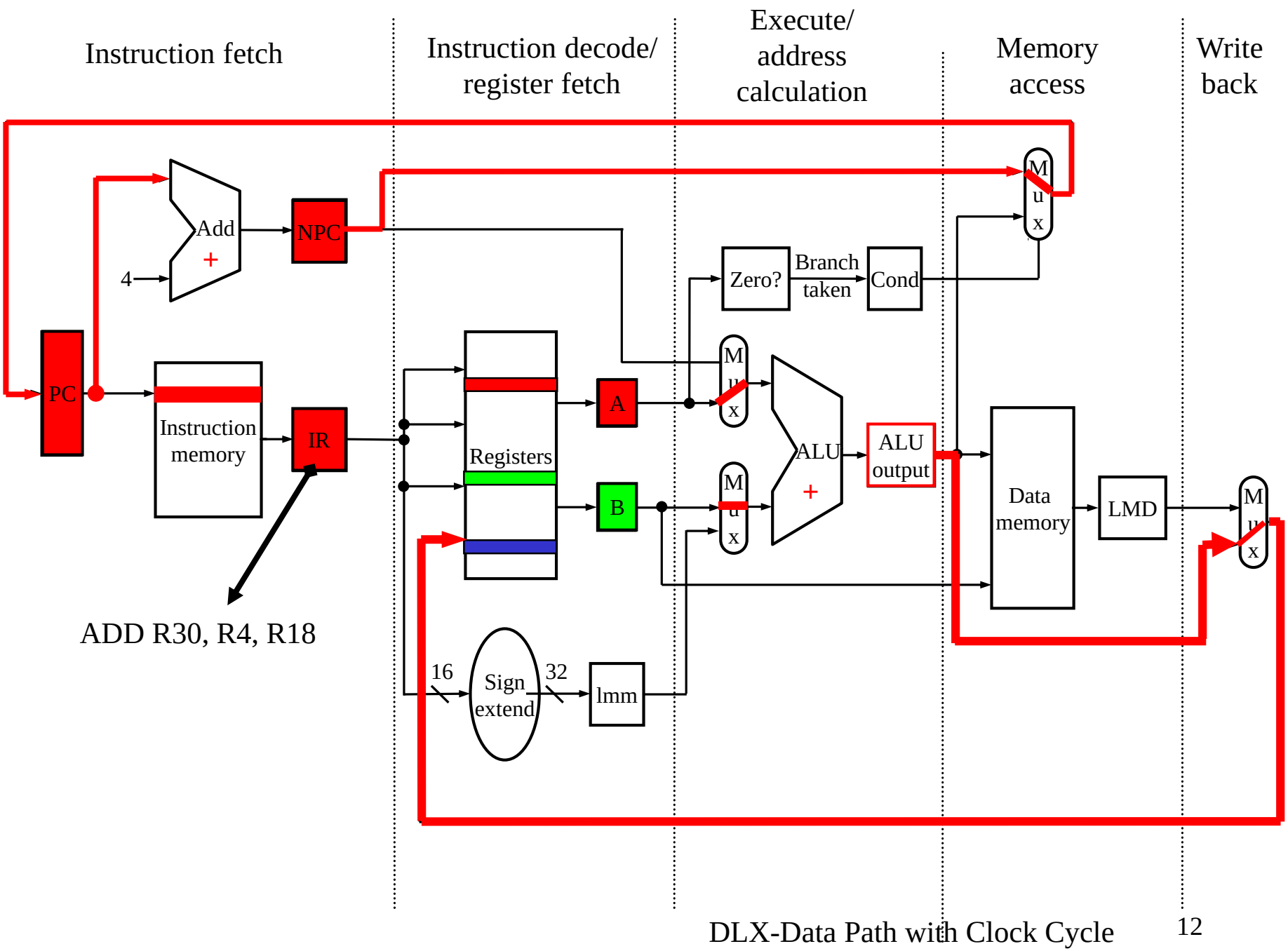
DLX-Data Path with Clock Cycle



DLX-Data Path with Clock Cycles

ALU-Instructions

Instructions		**Name**	**Explanations**
SUB	R1, R2, R3	Subtract	Regs[R1] ← Regs[R2] – Regs[R3]
ADDI	Rl, R2, #3	Add immediate	Regs [Rl] ← Regs [R2]+3
LHI	R1, #42	Load high immediate	Regs [R1]←42 # $\#0^{16}$



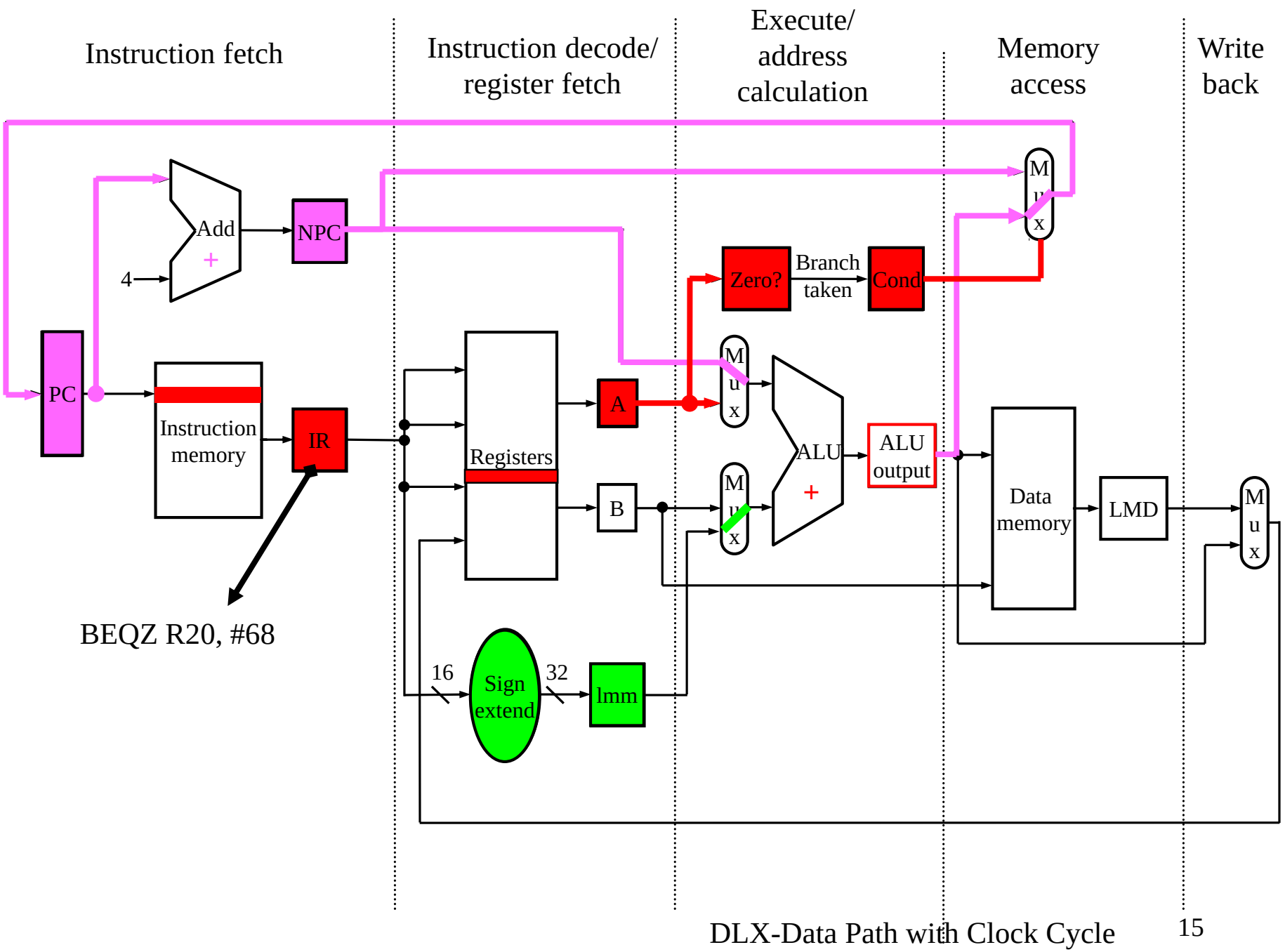
DLX-Data Path with Clock Cycle

Compare-Instructions

Instructions		**Name**	**Explanations**
SLLI	R1,R2,#5	Shift left logical immediate	Regs [R1] ← Regs [R2]<<5
SLT	R1, R2, R3	Set less than	if (Regs[R2]<Regs[R3]) Regs [R1] ←1

Jump Instructions

Instructions		Name	Explanations
J	name	Jump	PC ← name; ((PC+4) - 2^{25}) ≤ name < ((PC+4)+2^{25})
JAL	name	Jump and link	Regs[R31]←PC+4; PC ← name; ((PC+4) – 2^{25}) ≤ name < ((PC+4) +2^{25})
JALR	R2	Jump and link register	Regs [R31] ← PC+4; PC ← Regs [R2]
JR	R3	Jump register	PC ← Regs [R3]
BEQZ	R4,name	Branch equal zero	if (Regs [R4] =0) PC←name; ((PC+4) – 2^{15}) ≤ name < ((PC+4) +2^{15})
BNEZ	R4,name	Branch not equal zero	if (Regs [R4] 0) ≠ PC←name; ((PC+4) – 2^{15}) < name < ((PC+4) +2^{15})



DLX-Data Path with Clock Cycle

The syntax for the PC relative jump instructions is a little bit confusing, because the parameter stated as an operand is to be understood as displacement to the PC.

The first row should be:

J offset means PC <--- PC+4 + offset with -2^{25} <= offset < $+2^{25}$

That would mean that one would have to state the displacement at relative addresses explicitly in assembler programs. That makes the maintenance of such programs unbelievably difficult. Hence, it is allowed to introduce names for the effective addresses. These are written like labels in the assembler program. Of course, the compiler uses the actual offsets but a program written with those labels is much simpler to understand for the reader.

Floating Point Instructions

Instructions		**Name**	**Explanations**
ADDS	F2, F0, F1	Add single precision floating point numbers	Regs[F2] ← Regs[F0] + Regs[F1]
MULTD	F4, F0, F2	Multiply double precision floating point numbers	Regs[F4]##Regs[F5] ← Regs[F0]##Regs[F1] * Regs[F2]##Regs[F3]

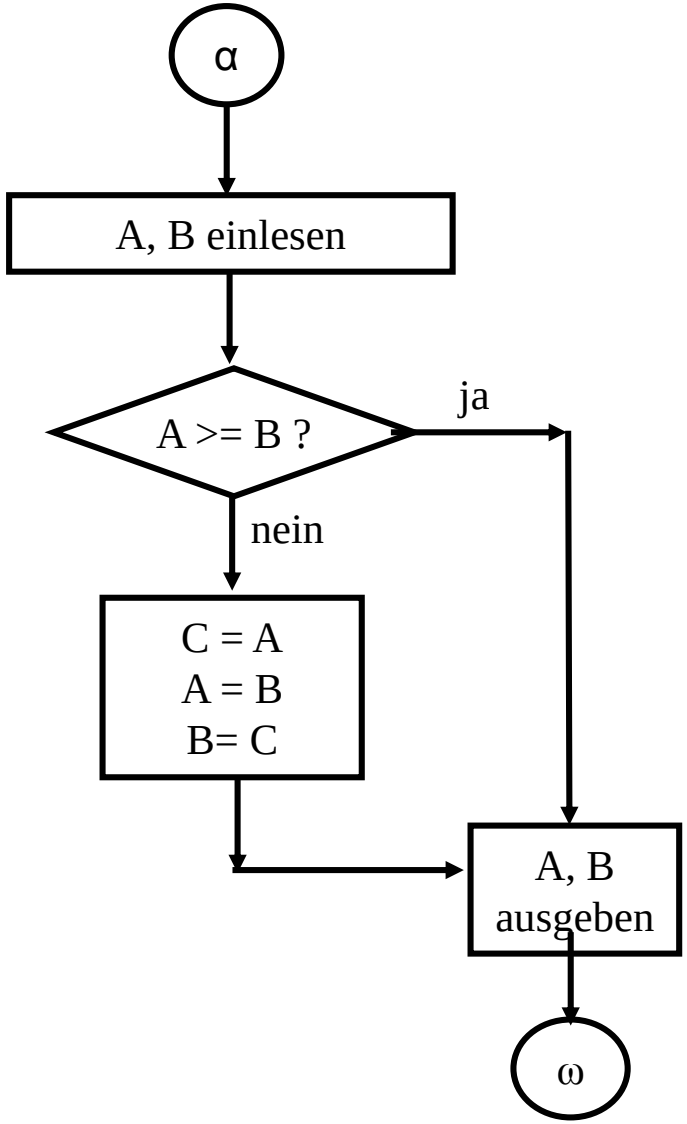
Instruction type/opcode	Instruction meaning
Data transfers	**Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR**
LB,LBU,SB	Load byte, load byte unsigned, store byte
LH, LHU, SH	Load half word, load half word unsigned, store half word
LW, SW	Load word, siore word (to/from integer registers)
LF, LD, SF, SD	Load SP float, load DP float, store SP float, store DP float
MOVI2S, NOVS2I	Move from/to GPR to/from a special register
MOVF, MOVD	Copy one FP register or a DP pair to another register or pair
MOVFP2I,MOVI2FP	Move 32 bits from/to FP registers to/from integer registers
Arithmetic/logical	**Operations on integer or logical data in GPRs; signed arithmetic trap on overflow**
ADD, ADDI, ADDU, ADDUI	Add, add immediate (all immediates are 16 bits); signed and unsigned
SUB, SUBI, SUBU, SUBUI	Subtract, subtract immediate; signed and unsigned
MULT,MULTU,DIV,DIVU	Multiply and divide, signed and unsigned; operands must be FP registers; all operations take and yield 32-bit values
AND,ANDI	And, and immediate
OR,ORI,XOR,XORI LHI	Or, or immediate, exclusive or, exclusive or immediate
LHI	Load high immediate – loads upper half of register with immediate
SLL, SRL, SRA, SLLI, SRLI, SRAI	Shifts: both immediate (S l) and variable form (S); shifts are shift left logical, right logical, right arithmetic
S_, S_ I	Set conditional: ”_ ” may be LT, GT, LE, GE, EQ, NE
Control	**Conditional branches and jumps; PC-relative or through register**
BEQZ,BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC+4
BFPT,BFPF	Test comparison bit in the FP status register and branch; 16-bit offset from PC+4
J, JR	Jumps: 26-bit offset from PC+4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC+4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
RFE	Return to user code from an exception; restore user mode
Floating point	**FP operations on DP and SP formats**
ADDD,ADDF	Add DP. SP numbers
SUBD,SUBF	
MULTD,MULTF	Multiply DP, SP floating point
DIVD, DIVF	Divide DP, SP floating point
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F, CVTI2D	Convert instructions: CVTx2y converts from type x to type y, where x and y are I (integer), D (double precision), or F (single precision). Both operands are FPRs.
_D,_ F	DP and SP compares: ”_ ” = LT, GT, LE, GE, EQ, NE; sets bit in FP status register

Beispiele für Assembler-Programmierung

Sortieren zweier Zahlen A und B.
A steht an Adresse 1000, B an Adresse 1004
Die größere Zahl soll am Ende in 1000 stehen, die kleinere in 1004

Input: Natürliche Zahlen A und B
Output: A und B in der Reihenfolge der Größe
Methode:

```
IF A < B Then
{           C = A
            A = B
            B = C  }
Endif
Output A
Output B
```



Wir wollen dafür ein Assemblerprogramm schreiben. Wir setzen voraus, dass die Operanden an den Adressen 1000 und 1004 im Speicher stehen und das Ergebnis sortiert an den selben Adressen entstehen soll:

Register:

R1 :	A
R2 :	B
R3 :	Hilfsregister
R4:	Hilfsregister

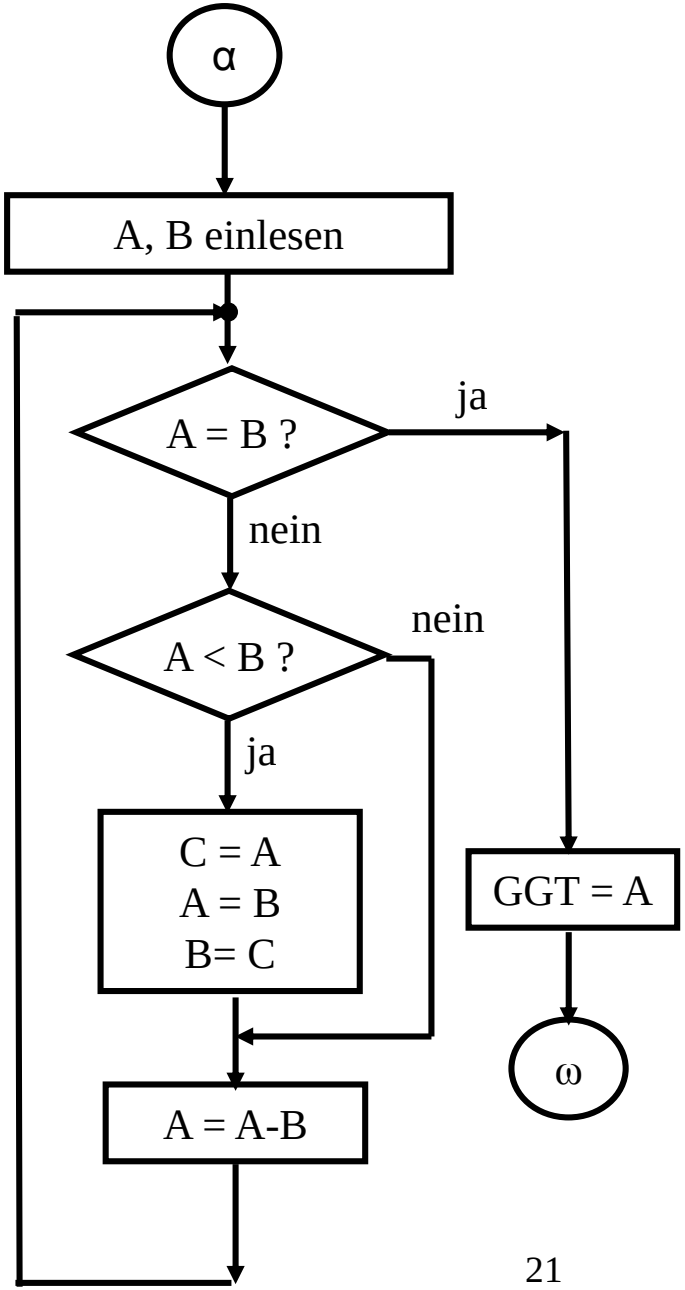
```
Start    LW      R1, 1000(R0)    /Lade erste Zahl nach R1
         LW      R2, 1004(R0)    /Lade zweite Zahl nach R2
         SUB     R4, R1, R2      /R4 negativ falls R2>R1
         SLT     R3, R4, R0      /R3 ungleich 0, falls R4 negativ
         BEQZ    R3, Ausgabe     /Zahlen bereits in der richtigen Reihenfolge
         ADD     R4, R1, R0      /Vertauschen: R4 := R1
         ADD     R1, R2, R0      /Vertauschen: R1 := R2
         ADD     R2, R4, R0      /Vertauschen: R2 := R4

Ausgabe  SW      1000(R0), R1    /Ausgabe des Maximums
         SW      1004(R0), R2    /Ausgabe des Minimums
         TRAP                    /Ende des Programms
```

Berechnung des GGT zweier Zahlen A und B.

Input: Natürliche Zahlen A und B
Output: GGT(A,B)
Methode:

```
While A <> B Do
        {       If  A<B Then
                {  C = A
                A = B
                B = C  }
                Endif
                A = A – B
        }
GGT = A
```



Wir wollen dafür ein Assemblerprogramm schreiben. Wir setzen voraus, dass die Operanden an den Adressen 1000 und 1004 im Speicher stehen und das Ergebnis an der Adresse 1008 entstehen soll:

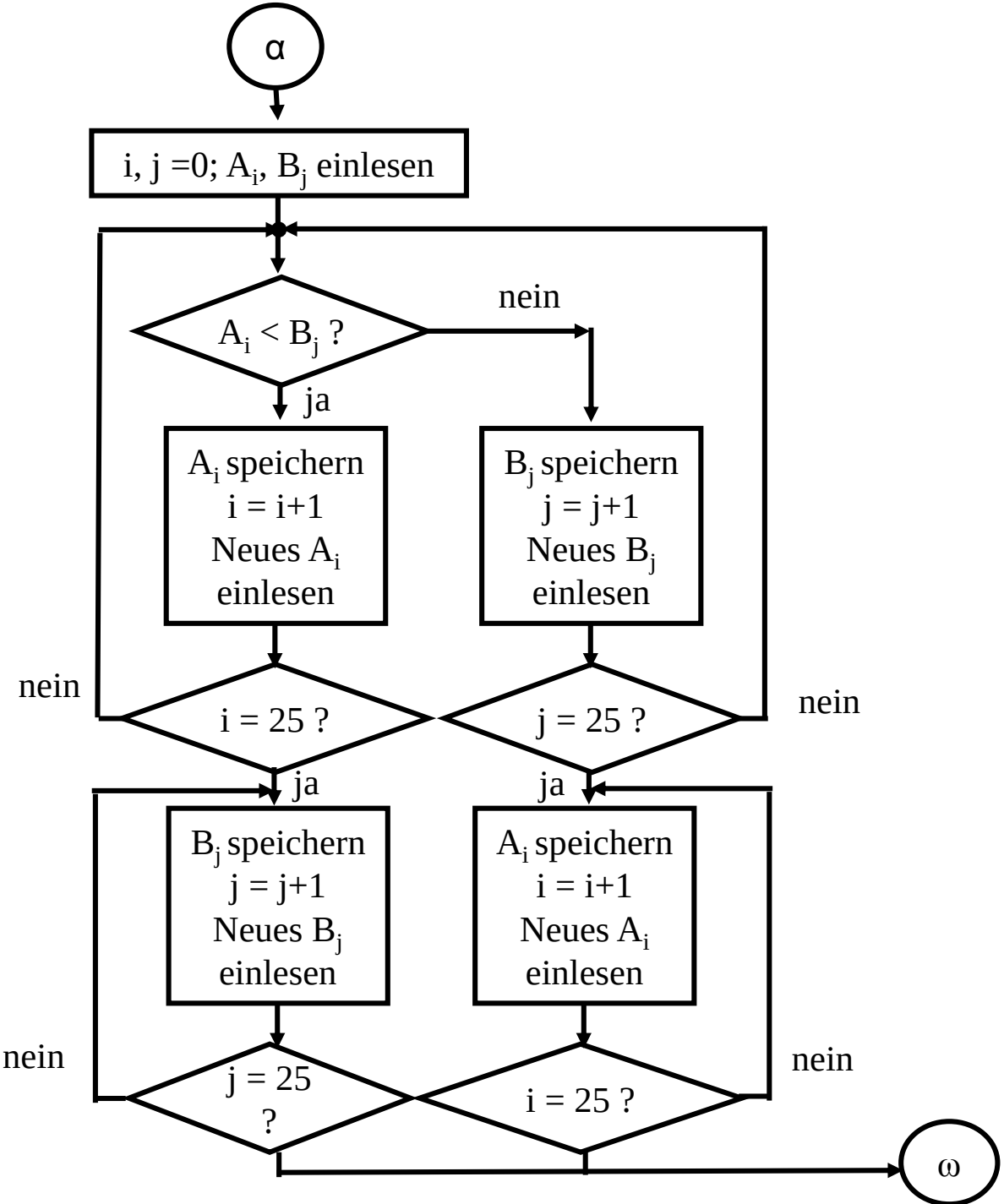
Register:
R1 : A
R2 : B
R3 : Hilfsregister

```
Start   LW      R1, 1000(R0)
        LW      R2, 1004(R0)
Loop    SEQ     R3, R1, R2
        BNEZ    R3, Ende
        SLT     R3, R1, R2
        BEQZ    R3, Weiter
        ADD     R3, R1, R0
        ADD     R1, R2, R0
        ADD     R2, R3, R0
Weiter  SUB     R1, R1, R2
        J       Loop
Ende    SW      1008(R0), R1
        TRAP
```

Beispiele für Assembler-Programmierung

Mergen zweier sortierter Listen der Länge 25. Die eine ist ab Adresse 1000 im Speicher, die andere ab Adresse 1100. Die sortierte Gesamtliste soll ab Adresse 2000 in den Speicher geschrieben werden. Sortiert heißt: Das kleinste Element steht in der Zelle mit der kleinsten Adresse und von da an aufsteigend.

Input: Zwei sortierte Listen A_i und B_j
Output: Eine sortierte Gesamtliste
Methode:
Das jeweils kleinste Element der einen Liste wird mit dem jeweils kleinsten Element der zweiten Liste verglichen. Das kleinere von beiden wird in die Gesamtliste gespeichert. Ein neues nunmehr kleinstes Element wird aus der Liste geladen, aus der das eben gespeicherte Element kam.
Sobald eine der Listen verbraucht ist, speichert man die restlichen Elemente der anderen Liste in der Reihenfolge, in der sie bereits sind.


α
i, j =0; Ai, Bj einlesen
Ai < Bj ?
nein
ja
Ai speichern
i = i+1
Neues Ai
einlesen
Bj speichern
j = j+1
Neues Bj
einlesen
nein
i = 25 ?
j = 25 ?
nein
ja
ja
Bj speichern
j = j+1
Neues Bj
einlesen
Ai speichern
i = i+1
Neues Ai
einlesen
nein
j = 25
?
i = 25 ?
nein
ω

Register:

R1 : Zeiger auf die Speicherzellen von A

R2 : Zeiger auf die Speicherzellen von B

R3 : Zeiger auf die Speicherzellen der Ergebnisliste

R4 : A_i

R5: B_j

R6 : Hilfsregister

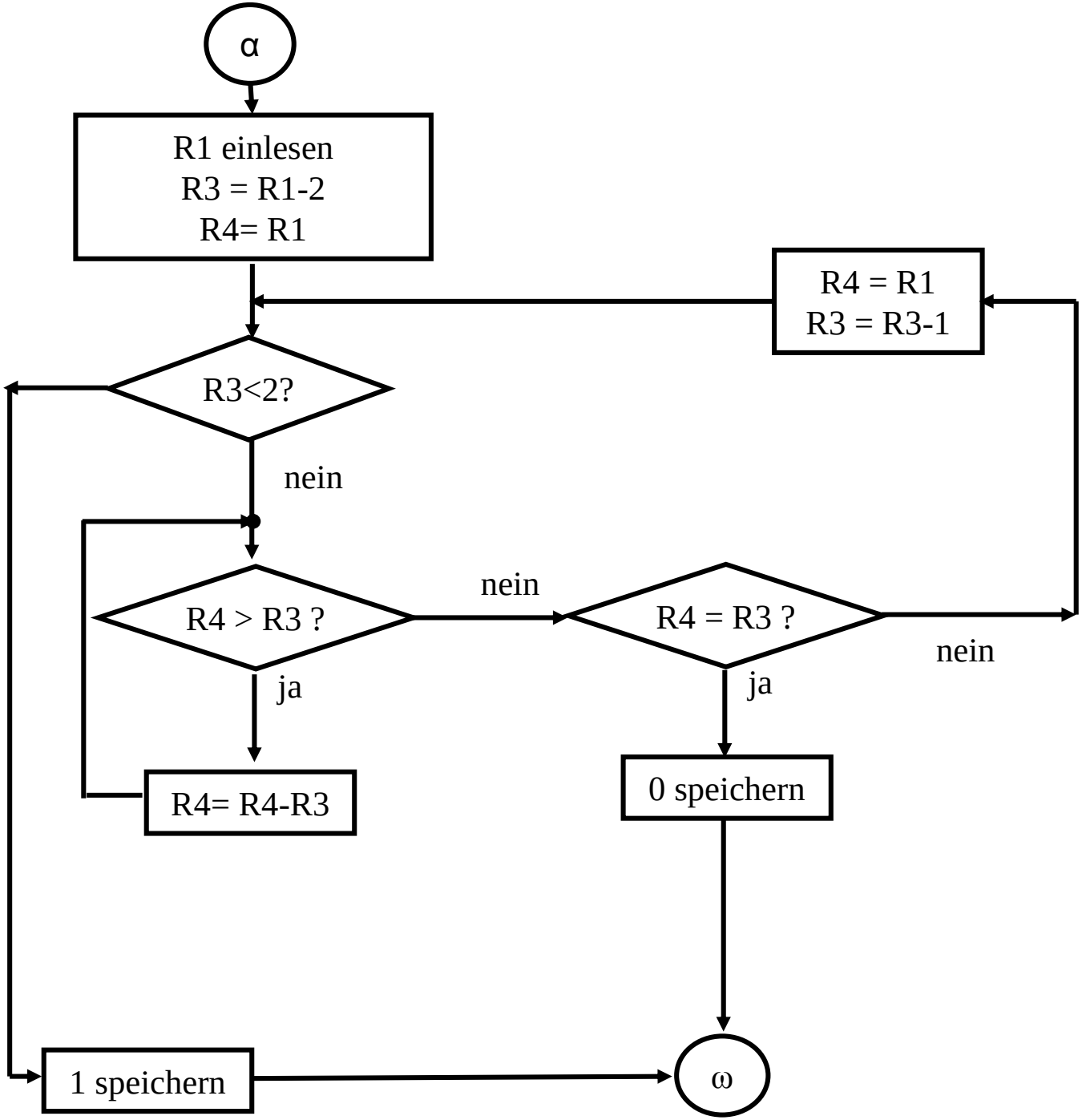
```
START   ADD     R1, R0, R0          / Initialisieren von R1 mit 0
        ADD     R2, R0, R0          / Initialisieren von R2 mit 0
        ADD     R3, R0, R0          / Initialisieren von R3 mit 0
        LW      R4, 1000(R1)        / Laden von A0
        LW      R5, 1100(R2)        / Laden von B0

LOOP    SGT     R6, R4, R5          / Welches ist das Kleinere?
        BEQZ    R6, R4KL            / Springen, falls R4 kleiner
        SW      2000(R3), R5        / Speichern von Bj
        ADDI    R3, R3, #4          / R3 zeigt auf die nächste freie Zelle
        ADDI    R2, R2,#4           / R2 zeigt auf nächstes Bj
        LW      R5, 1100(R2)        / Laden des nächsten Bj
        SUBI    R6, R2, #100        / prüfen, ob die Liste B schon leer
        BEQZ    R6, ARAUS           / wenn ja, Restliste A rausschreiben
        J       LOOP                / wenn nein, neuer Vergleich
```

```
R4KL    SW      2000(R3), R4    / R4 enthält das nunmehr kleinste Elt.
        ADDI    R3, R3, #4      / Zeiger auf die nächste freie Zelle
        ADDI    R1, R1,#4       / R1 zeigt auf  neues A_i
        LW      R4, 1000(R1)    / Laden des nächsten A_i
        SUBI    R6, R1, #100    / prüfen, ob die Liste A bereits leer
        BEQZ    R6, BRAUS       / wenn ja, Sprung nach BRAUS
        J       LOOP            / wenn nein, erneuter Vergleich
ARAUS   SW      2000(R3), R4    / Ab hier wird der Rest von A
        ADDI    R3, R3, #4      / herausgeschrieben
        ADDI    R1, R1,#4       / R1 zeigt auf nächstes A_i
        LW      R4, 1000(R1)    / Laden des nächsten A_i
        SUBI    R6, R1, #100    / prüfen, ob alle Elemente von A bereits
        BNEZ    R6, ARAUS       / verarbeitet sind
        J       ENDE
BRAUS   SW      2000(R3), R5    / ab hier wird der Rest von B
        ADDI    R3, R3, #4      / herausgeschrieben
        ADDI    R2, R2,#4       / R2 zeigt auf nächstes B_j
        LW      R5, 1100(R2)    / Laden des nächsten B_j
        SUBI    R6, R2, #100     / prüfen, ob alle Elemente von B bereits
        BNEZ    R6, BRAUS        / verarbeitet sind
ENDE    HALT
```

Testen einer Zahl auf Primalität

Die Zahl steht in Adresse 1000 im Speicher. Wenn es eine Primzahl ist, soll nach dem Programm eine 1 in Adresse 1004 stehen, wenn sie teilbar ist eine 0.


α
R1 einlesen
R3 = R1-2
R4= R1
R4 = R1
R3 = R3-1
R3<2?
nein
R4 > R3 ?
nein
R4 = R3 ?
nein
ja
ja
R4= R4-R3
0 speichern
1 speichern
ω

Register:

R1 : Auf Primalität zu prüfende Zahl
R2 : Konstante 2
R3 : Durchläuft alle kleineren Zahlen und prüft ob sie R1 teilen
R4 : Kopie von R1 für Teilbarkeitstest
R5: Hilfsregister

```
START       ADDI   R2, R0, #2        / Initialisieren von R2 mit 2
            LW     R1, 1000(R0)      / Laden der zu testenden Zahl
            ADD    R4, R0, R1        / Kopieren von R1
            SUBI   R3, R4,#2         / Erster Teilerkandidat
LOOP1       SLT    R5, R3, R2        / Ist R3 kleiner als 2?
            BNEZ   R5, PRIMZAHL      / R1 hat keine nichttrivialen Teiler
LOOP2       SGT    R5, R4,R3         / ist R4>R3?
            BEQZ   R5, GLEICHTEST    / wenn nicht, muss R4=R3 geprüft werden
            SUB    R4, R4, R3        / R4 um R3 verringern
            J      LOOP2             / Nächster Durchlauf der inneren Schleife
GLEICHTEST  SEQ    R5, R4, R3        / Ist R3 gleich R4
            BNEZ   R5, NICHTPRIM     / Dann teilt R3 die Zahl in R1
            ADD    R4, R0, R1        / Setzten von R4 auf ursprünglichen Wert
            SUBI   R3, R3,#1         / verringern von R3 um 1
            J      LOOP1             / neuer Test auf Teilbarkeit
PRIMZAHL    ADDI   R5, R0, #1        / Erzeugen einer 1 in R5
            SW     1004(R0), R5      / Schreiben der 1 in 1004
            J      ENDE              / ENDE
NICHTPRIM   SW     1004(R0), R0      / Schreiben der 0 nach 1004
ENDE        HALT
```