TSEA83
¬DDR — Rapport

Version b19c3a246d74aeed45de8421e6ea2309db0517ee

Alexander Basa (aleba538) Emil Draws (emidr065)
Hugo Hörnquist (hugho389)

15 juni 2018

Innehåll

Kapitel 1

Inledning

Raporten producerades av ¬DDR (grupp 53) inom ramen för kursen Datorkonstruktion (TSEA83) på Linköpings universitet våren 2018. Syftet med projektet var att konstruera en dator som tog någon form av input och producerade någon form av output. I det här fallet genom ett tangentbord och en datorskärm.

I rapporten går vi igenom funktionaliteten av vår tvåstegspipelinade dator. Den arkitekturen valdes då vi såg en stor fördel med att alla instruktioner skulle köras lika snabbt, vilket underlättar för rytmspel. Även att pipelinade arkitekturer är snabbare var en bidragande faktor i beslutet.

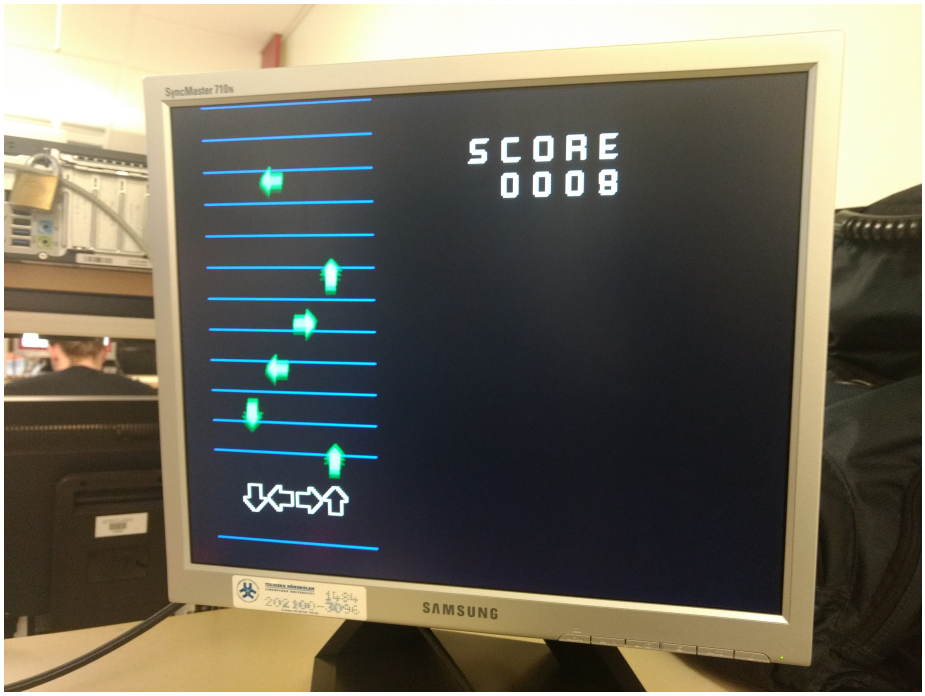
Planen var att vi även skulle ge möjlighet att spela upp musik. Man skulle ha möjligheten att ladda in låtar i minnet skrivna i ett visst format, och dessa låtar skulle även då tolkas som olika nivåer i spelet. Detta togs bort under projektets gång så att vi kunde möta deadline. Eftersom datorn skapades med detta i tanken så finns strukturen där för att enkelt bygga ut dessa funktioner.

Kapitel 2

Spelet

Spelet som körs på datorn är en version av olika rytmbaserade arkadspel[1] där det faller ner pilar på skärmen och spelaren ska försöka trycka på rätt knapp när pilen är inom ett visst område för att få poäng. I figur 2.1 så kan man se hur spelet ser ut då det körs. Spelet i sig är väldigt simpelt och kan även med små ändringar enkelt få ett nytt utseende.

I vårt fall spelas spelet med knapparna `D`, `F`, `J` och `K`. Där knapparna kontrollerar kolumnen på samma plats.



Figur 2.1: Hur spelet kan se ut då det är igång

[1] Ett av de mest kända spelen i den genren är Dance Dance Revolution, vilket är varifrån vi får vårat namn.

2.1 Spelets format

Hur och när pilarna faller hämtar programmet direkt från dataminnet. Formatet för detta ser ut som följande. Varje pil tar upp 2 stycken minnesplatser. Den första anger vilken kolumn pilen ska falla i och den andra anger den tid i millisekunder som pilen ska befinna sig på den plats på skärmen då spelaren bör trycka på tangenten. Detta värde används även för att jämföra ifall spelaren tryckt rätt. Detta upprepas för varje ny pil tills det att ett speciellt värde lästs och spelet vet att nivån är slut. Vilken typ av pil det är som faller vet man inte genom att läsa från dataminnet i det nuvarande formatet utan detta sätts i själva programminnet utifrån vilken kolumn pilen faller i. Detta funkar så länge varje typ av pil faller i sin egen kolumn vilket är fallet i detta spel.

Kapitel 3

Hårdvaran

Konstruktionen är en tvåstegs pipeline design som utför nästan alla tillgängliga instruktioner på en klockcykel. De få instruktioner som tar två klockcykler är de som behöver läsa från dataminnet eller hoppa mellan instruktioner. Se figur 3.1 för ett översiktligt blockschema.

3.1 Kontrollenhet

Kontrollenheten är den centrala komponenten i datorn och det är den som tolkar alla instruktioner. Kontrollenheten hämtar den nuvarande instruktionen från instruktionsregistret och genererar kombinatoriskt de signaler som bestämmer vad alla komponenter ska göra. Förutom att bara tolka operationskoderna i instruktionen så läser den även den givna adressen, vilket gör det möjligt att peka ut olika komponenter beroende på adressen. I vårt system så pekar adresserna ut följande komponenter:

- Bildminne: 4000_{16}–$412b_{16}$
- Senaste tryckta tangent: 5000_{16}
- Millisekundsklocka: 6000_{16}–6001_{16}
- Spriteregister: 7000_{16}–70400_{16}
- Hoppadress för avbrott: 9000_{16}

3.2 Anropsstack

Datorn har en anropsstack (eng. Call Stack) som används för att hålla returadresser för subrutinshopp. Anropsstacken har 32 platser så datorn kan endast göra 32st nästlade hopp.

3.3 Programräknaren PC

Programräknaren (eng. Program Counter) fungerar som en pekare till vilken instruktion som ska utföras näst. Den har tre lägen i konstruktionen:

Läge 1 I läge ett så ökar programräknaren ett steg på varje klocksignal.

Läge 2 I läge två så sätts programräknarens värde till det som ligger på bussen.

Läge 3 I läge tre står programräknaren helt stilla.

3.4 Programminne PMEM

Programminnet (eng. Program Memory) innehåller systemets maskinkod och går inte att skriva till. Komponenten har en insignal från programräknaren vilket är pekaren till den adress vars data ska läggas på komponentens utsignal.

3.5 Instruktionsregister IR

Datorns pipeline har åstadkommits genom instruktionsregistret samt en mux som finns kopplad mellan instruktionsregistret och programminnet. Instruktionsregistrets uppgift är att hålla den nuvarande instruktionen som utförs. Den existerar för att få pipeline konstruktionen att fungera då instruktionskoden inte kan gå direkt från programminnet till kontrollenheten utan går via en mux som bestämmer vad som ska komma igenom. Detta gör att de instruktioner som inte kan göras klart på en klockcykel får extra klockcyklar genom en väntoperation. För blockschemat över denna delen av hårdvaran se figur 3.2

3.6 Generella register AREG

Systemet har 32st generella register som man kan skriva godtyckliga värden till från programminnet och det är även där som resultaten från systemets ALU sparas. De generella registren är 16 bitar breda.

3.7 ALU och SR Stack

Systemets ALU (Arithmetic Logic Unit) gör de beräkningar som behövs genom de instruktioner som finns tillgängliga (se A.3.3). Indatan till systemets ALU kan komma från endast AREG eller från AREG och den nuvarande instruktionen. Den nuvarande instruktionen har även information om vilken operation som ska utföras. Vid varje beräkning så uppdateras även ett statusregister (SR) som används av kontrollenheten för att se om vissa hopp bör utföras. Systemet har en stack av olika SR för att vid subrutinshopp hantera flera olika register. SR stacken är lika stor som anropsstacken (se 3.2). De flaggor vi har är:

N - Negative, sätts då ALU operationen ger ett negativt resultat

C - Carry, sätts då resultatet av operationen är större än 16 bitar

Z - Zero, sätts då ALU operationen ger ett noll resultat

3.8 Dataminne DMEM

Dataminnet (eng. Data Memory) är systemets generella minne som kan läsas från och skrivas till. Dataminnet har 2^{14} platser och varje plats är 16 bitar bred.

3.9 Grafik

Datorn matar ur sig bild med en upplösning av 640×480 pixlar. För kontroll av dessa finns tiles och sprites. Tiles är statiskt placerade objekt som i vårt fall vardera är 32×32 pixlar.[1] Sprites är istället fritt flytande grafikobjekt som kan finnas vart som på skärmen. Ett antal sprites finns, och är samtliga 32×32 pixlar stora.

3.9.1 VGA-Motor

Denna komponent är den som sköter vad som ritas ut på skärmen. Den arbetar oberoende av resten av datorn så att bilden kan ritas ut även då själva programmet står still. Vad den gör är att den går igenom pixel för pixel och skickar ut signaler till bildminnet (se avsnitt 3.9.3) som sänder tillbaka en signal som talar om vilken plats i tileminnet som färgen för den pixeln finns. Tileminnet är ett minne inbyggt i VGA-motorn som håller reda på utseendet för alla olika tiles i form av färgkodningen för varje individuell pixel. Komponenten tar även emot signaler från spriteregistret (se avsnitt 3.9.2) som talar om ifall det finns en sprite på den pixelns position. Om det gör det så ritar den ut den färgen som spriteregistret sänder istället för den från bildminnet.

3.9.2 Spriteregister

Spriteregistret håller reda på alla sprites, dvs. alla figurer som ska kunna ritas ut vart som helt på skärmen i flytande rörelser. Den har stöd för 16 olika sprites och kan rita ut alla på skärmen samtidigt. Om fler än en sprite finns på en viss position så är det den som hittas på den högsta minnesadressen bland spriteregistren som ritas ut (se avsnitt 3.1).

3.9.3 Bildminne PICTMEM

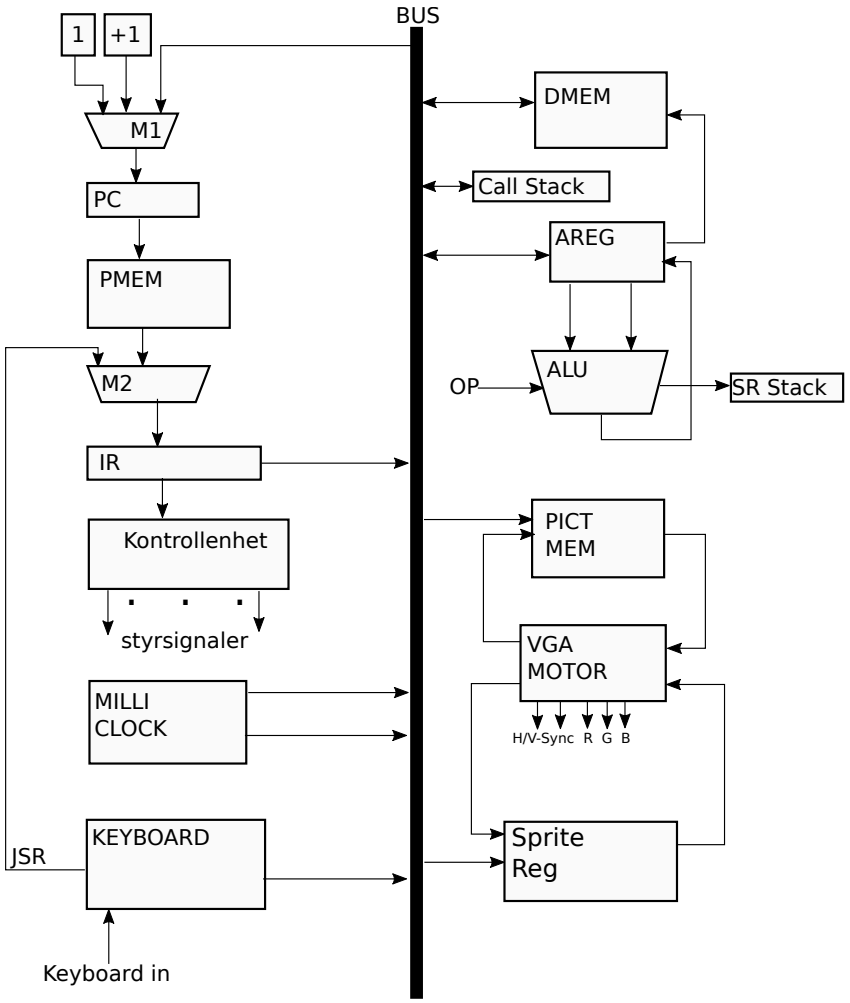
Bildminnet (eng. Picture Memory) är den del i systemet som innehåller information om vilka tiles som befinner sig vart på skärmen. Bildminnet har startpositionen för skärmen inprogrammerat i minnet vid start och dess innehåll kan inte läsas ifrån programmkoden, vilket bara VGA-motorn kan göra. Däremot går det att skriva till den och byta ut vilken tile som befinner sig på vilken position. Bildminnet håller 20×15 tiles i minnet där varje minnesplats refererar till tileminnet i VGA-motorn. Se 3.9.1 för beskrivning av tiles m.m.

3.10 Tangentbord KBD

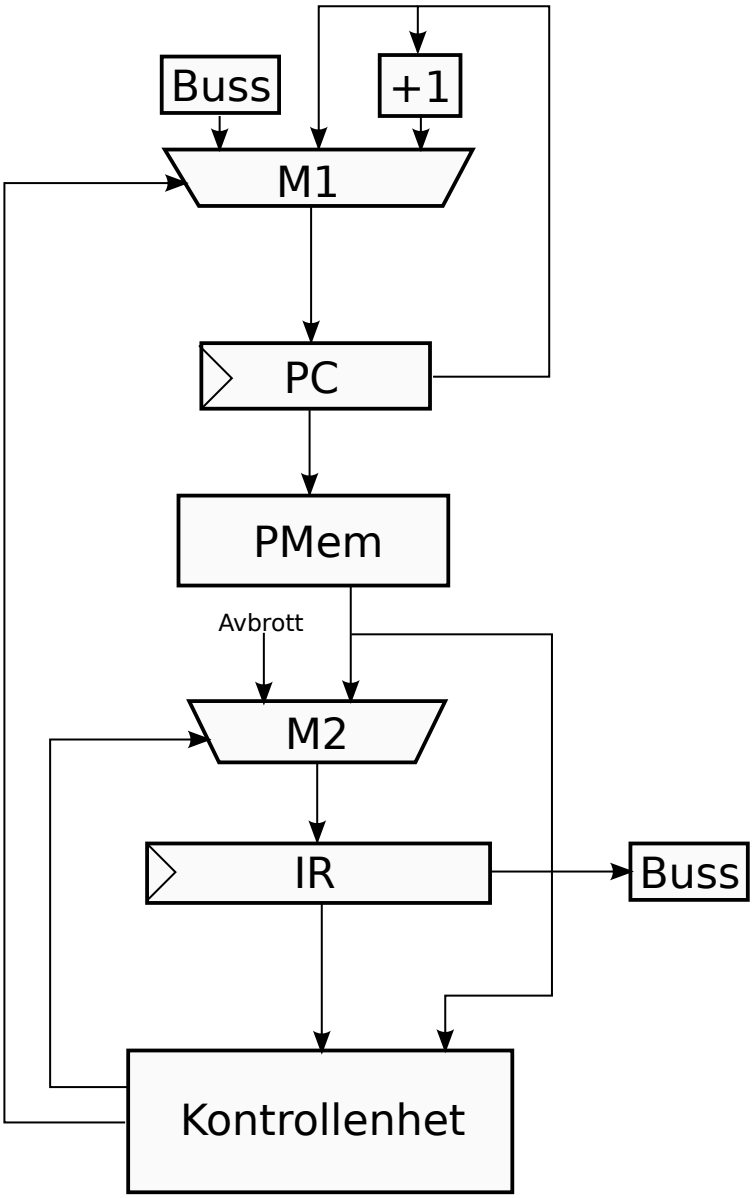
KBD komponenten är den som hanterar insignalerna från ett PS/2-tangentbord. Den senaste tangenten som tryckts sparas alltid på en viss minnesadress och vi

[1] vilket ger oss 20×15 tiles på skärmen

har även en minnesadress reserverad för adressen som ska hoppas till vid ett avbrott som sker i samband med ett tangenttryck(se avsnitt 3.1). Datorn har stöd för att stänga av och på avbrott. Avbrotten stängs av genom att sätta den mest signifikanta biten till hög när man sätter hoppadressen för avbrott. Om avbrotten är aktiverade och en tangent har tryckts så skickas en avbrottssignal till instruktionsregistret (se avsnitt 3.5) som ser till att programmet gör ett subrutinshopp. Själva subrutinen som hoppas till skriver användaren själv i programminnet och sätter sedan dess startadress för avbrottshopp i början av programmet.



Figur 3.1: Översiktligt blockschema av systemets design.



Figur 3.2: Mer detaljerad vy av pipelinen

Kapitel 4

Slutsatser

I slutändan blev projektet inte precis som vi förutspått. Detta berodde på att vi hade valt en för krånglig datorkonstruktion, vilket ledde till att tiden inte räckte till.

I projektets början valde vi att bygga en pipelinad dator, med målet att den skulle vara en bra bit snabbare och göra allting enklare för oss. Nu har vi en ytterst snabb dator, och ett program som inte nyttjar det alls. Vi hinner med ca 100 iterationer av spelloopen per millisekund. Detta borde kunna ha förutsetts. Vi fick därför välja att endast ta med de absolut viktigaste delarna som skulle få datorn att fungera och fick utelämna ljuduppspelning och inläsning till dataminnet.

Att skriva en assembler samt ha stöd för en lite högre nivå av programmering var mycket värt den tid det tog att skapa då detta sparade en enorm mängd tid som kunde ha gått förlorad med att endast skriva programkoden.

Bilaga A

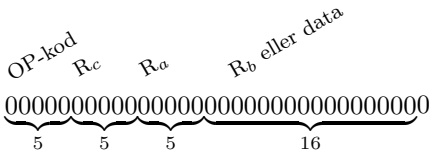
Listningar

A.1 VHDL-Kod

Projektets toppmodul är `projName`. Funktionaliteten för kontrollenheten (se avsnitt 3.1) har implementerats som en del av den.

A.2 Instruktion

Varje instruktion i datorn är 32 bitar lång och är utlagd enligt figur A.1. De 5 först bitarna är operationskoden som avgör vilken typ av operation det är och låter vår instruktionshanterare identifiera den korrekt. De 10 följande bitarna är 2 generella register på 5 bitar vardera (R_c och R_a). De sista 16 bitarna används som antingen data där alla bitar har en funktion eller så används den för att peka på ett tredje register R_b där de 5 bitar längst till höger representerar registret och de andra 11 inte används. Ifall det är ett register eller data i dessa bitar avgörs av operationskoden. Även de bitar som representerar de första 2 registrena behöver inte alltid vara i användning. Däremot så används inte dessa till annat.



Figur A.1: En instruktions delar

A.3 OP-koder

Följande är en lista över de instruktioner som vår maskinkod implementerar. För numeriska representationer av samtliga op-koder, se tabell A.1.

†*: Instruktionen tar 2 klockcykler att utföra.*

A.3.1 Systemkontroll

`HALT` $\longmapsto$ Avbryt programexekeveringen.

`NOOP` $\longmapsto$ Gör ingenting.

A.3.2 Register- och minnestillgång

Läser och skriver till och från DMEM, flera av instruktionerna tar dessutom ett M argument, vilket är adresseringsmod. Vilka adresseringsmod som ska finnas är ännu inte bestämt. R_a & R_b är godtyckliga register.

$$\texttt{LOAD}^{\dagger}\ \mathrm{R}_c,\ M,\ \textsc{adr}\Big|\mathrm{R}_a \longmapsto$$
$$\rightarrow \mathrm{R}_c := \mathrm{DMEM}\left(\begin{cases}\textsc{adr} & \text{om } M = \texttt{DIR} \\ \mathrm{R}_a & \text{om } M = \texttt{PTR}\end{cases}\right)$$

$$\texttt{STORE}\ \mathrm{R}_c,\ M,\ \textsc{adr}\Big|\mathrm{R}_a \longmapsto$$
$$\rightarrow \mathrm{DMEM}\left(\begin{cases}\textsc{adr} & \text{om } M = \texttt{DIR} \\ \mathrm{R}_a & \text{om } M = \texttt{PTR}\end{cases}\right) := \mathrm{R}_c$$

`LOADI` $\mathrm{R}_c,\ n \longmapsto \mathrm{R}_c := n$

`STORI` $\textsc{Adr},\ n \longmapsto \mathrm{DMEM}(\textsc{adr}) := n$

A.3.3 Aritmetik

Många av följande sätter flaggor i `SR`. Det är inte i alla fall noterat. Utgå från stycke 3.7 för vilka flaggor som finns, och härled därifrån.

`ADD` $\mathrm{R}_c,\ \mathrm{R}_a,\ \mathrm{R}_b \longmapsto \mathrm{R}_c := \mathrm{R}_a + \mathrm{R}_b$

`SUB` $\mathrm{R}_c,\ \mathrm{R}_a,\ \mathrm{R}_b \longmapsto \mathrm{R}_c := \mathrm{R}_a - \mathrm{R}_b$

`CMP` $\mathrm{R}_a,\ \mathrm{R}_b \longmapsto \mathrm{R}_a - \mathrm{R}_b$
Inget värde returnernas, inget register sätts. Flaggor i `SR` sätts.

`CMPI` $\mathrm{R}_a,\ n \longmapsto \mathrm{R}_a - n$

`CMPJ` $n,\ \mathrm{R}_a \longmapsto n - \mathrm{R}_a$

`ADDI` $\mathrm{R}_c,\ \mathrm{R}_a,\ n \longmapsto \mathrm{R}_c := \mathrm{R}_a + n$

`SUBI` $\mathrm{R}_c,\ \mathrm{R}_a,\ n \longmapsto \mathrm{R}_c := \mathrm{R}_a - n$

`MUL` $\mathrm{R}_c,\ \mathrm{R}_a,\ \mathrm{R}_b \longmapsto \mathrm{R}_c := \mathrm{R}_a \cdot \mathrm{R}_b$

`MULI` $\mathrm{R}_c, \mathrm{R}_a, n \longmapsto \mathrm{R}_c := \mathrm{R}_a \cdot n$

`AND` $\mathrm{R}_c, \mathrm{R}_a, \mathrm{R}_b \longmapsto \mathrm{R}_c := \mathrm{R}_a \wedge \mathrm{R}_b$

`ANDI` $\mathrm{R}_c, \mathrm{R}_a, n \longmapsto \mathrm{R}_c := \mathrm{R}_a \wedge n$

`OR` $\mathrm{R}_c, \mathrm{R}_a, \mathrm{R}_b \longmapsto \mathrm{R}_c := \mathrm{R}_a \vee \mathrm{R}_b$

`ORI` $\mathrm{R}_c, \mathrm{R}_a, n \longmapsto \mathrm{R}_c := \mathrm{R}_a \vee n$

`XOR` $\mathrm{R}_c, \mathrm{R}_a, \mathrm{R}_b \longmapsto Rc := \mathrm{R}_a \veebar \mathrm{R}_b$

`XORI` $\mathrm{R}_c, \mathrm{R}_a, n \longmapsto Rc := \mathrm{R}_a \veebar n$

`NOT` $\mathrm{R}_c, \mathrm{R}_a \longmapsto \mathrm{R}_c := \neg \mathrm{R}_a$

`LSLI` $\mathrm{R}_c, \mathrm{R}_a, n \longmapsto \mathrm{R}_c := \mathrm{R}_a \ll n$

`LSRI` $\mathrm{R}_c, \mathrm{R}_a, n \longmapsto \mathrm{R}_c := \mathrm{R}_a \gg n$

A.3.4 Hopp

Subrutiner är implementerade genom en anropsstack (se avsnitt 3.2). Villkorliga hopp sker då en viss flagga är satt eller inte satt beroende på vilken operation det är som används.

`JMP`† $\text{ADR} \longmapsto PC := \text{ADR}$
Ovillkorligt hopp.

`JEQ`† $\text{ADR} \longmapsto$

$$\rightarrow PC := \begin{cases} \text{ADR} & \text{om } \texttt{Z} \text{ flaggan satt i } \texttt{SR} \\ PC + 1 & \text{övriga fall} \end{cases}$$

`JGE`† $\text{ADR} \longmapsto$

$$\rightarrow PC := \begin{cases} \text{ADR} & \text{om } \texttt{N} \text{ flaggan ej satt i } \texttt{SR} \\ PC + 1 & \text{övriga fall} \end{cases}$$

I praktiken hoppar den om $\mathrm{R}_b \geq \mathrm{R}_a$ vid jämförelse.

`JSR`† $\text{ADR} \longmapsto$
$\rightarrow PC := \text{ADR}, \qquad \text{DMEM}(SP) := \text{ADR}, \qquad \text{INC}(SP)$

Ovillkorligt hopp till subrutin. Lägger returaddress på anroppsstacken.

`RTS`† $\varnothing^1 \longmapsto \quad PC := \text{DMEM}(SP), \quad \text{DEC}(SP)$
Retur från subrutin.

[1] Ingen indata

Tabell A.1: Mappning mellan numeriska och skrivna OP-koder

Hex	Binärt	Instruktion
00_{16}	00000_2	`HALT`
10_{16}	00010_2	`LOAD`
18_{16}	00011_2	`LOADI`
20_{16}	00100_2	`STORE`
28_{16}	00101_2	`STORI`
30_{16}	00110_2	`LSLI`
38_{16}	00111_2	`LSRI`
40_{16}	01000_2	`JSR`
48_{16}	01001_2	`RTS`
50_{16}	01010_2	`ADD`
58_{16}	01011_2	`ADDI`
60_{16}	01100_2	`SUB`
68_{16}	01101_2	`SUBI`
70_{16}	01110_2	`MUL`
78_{16}	01111_2	`MULI`
$B8_{16}$	10111_2	`CMP`
$C8_{16}$	11001_2	`CMPI`
$D8_{16}$	11011_2	`CMPJ`
80_{16}	10000_2	`AND`
88_{16}	10001_2	`ANDI`
90_{16}	10010_2	`OR`
98_{16}	10011_2	`ORI`
$A0_{16}$	10100_2	`XOR`
$A8_{16}$	10101_2	`XORI`
$B0_{16}$	10110_2	`NOT`
$C0_{16}$	11000_2	`JMP`
$D0_{16}$	11010_2	`JEQ`
$E0_{16}$	11100_2	`JGE`
$F8_{16}$	11111_2	`NOOP`
$E8_{16}$	11101_2	*Oanvänd*
$F0_{16}$	11110_2	*Oanvänd*

A.4 Assembly- och maskinkod

A.4.1 Assembly-kod

Assembly-koden är skriven i en lisp-dialekt, vars funktioner utöver möjligheten att skriva direkta instruktioner innefattar `defmacro`, `scheme`, och `defconst`.

`scheme[*]` låter användare köra en godtycklig bit scheme-kod. Om ingen stjärna finns på slutet så förväntas uttrycket evalueras till en lista av assembler-instruktioner, vilka placeras på uttryckets plats. Om en stjärna finns körs koden enbart för sina sidoeffekter.

`defmacro` fungerar liknande, dock med skillnaden att den skapar en form vilken senare expanderas genom ett macroanropp: `(M macro-name arg1 arg2 ...)`.

`defconst` skapar en alternativt namn för ett nummer. Alla bindningar är globala och position i filen spelar ingen roll.

För de instruktioner som nyttjar ett senare men inte ett tidigare register, till exempel `CMP` som saknar R_c behöver det manuellt noteras. Det görs genom att placera `--` på det saknade argumentets plats.

På flertalet ställen förekommer också `(ADDI Ra Rb 0)`. Allt den raden gör är att förflytta värdet från R_b till R_a. `ADDI` används eftersom en flytta-instruktion inte finns. `ADDI` används eftersom en flytta-instruktion inte finns.

```
;;; General notes, keep in mind while programming
;;; JGE hoppar när b >= a
;;; SUB : C = A - B

;;; Standard library for lists, neeed in some macros
(scheme*
 (use-modules (srfi srfi-1)))

;; Increment register by any value
(defmacro inc* (reg n tmp-reg)
  `((ADDI ,tmp-reg ,reg ,n)
    (ADDI ,reg ,tmp-reg 0)))

;; Increment register by one
(defmacro inc (reg tmp-reg)
  `((M inc* ,reg 1 ,tmp-reg)))

;; Decrement register by any value
(defmacro dec* (reg n tmp-reg)
  `((SUBI ,tmp-reg ,reg ,n)
    (ADDI ,reg ,tmp-reg 0)))

;; Decrement register by one
(defmacro dec (reg tmp-reg)
  `((M dec* ,reg 1 ,tmp-reg)))
```

```
;;; Start of actual code

;; reserved regisrers:
;;    R28 - Address to scancode for collumn 0
;;    main loop R0-R12
(defconst note-ptr r31)
(defconst score    r30)
(defconst ptr-offset r29)
;; (defconst - r28)
(defconst key-0 r27)
(defconst key-1 r26)
(defconst key-2 r25)
(defconst key-3 r24)

(defconst note-column  r10)
(defconst note-time    r11)
(defconst current-time r12)

;;; note struct:
;;;    addr0: collumn
;;;    addr1: time

;;; song ends when R31 points to note: collumn 0xFFFF, time 0xFFFF

;;; [Ladda låtdata]

;; "bind" scancodes and store the base address in R28
(loadi key-0 -- 0x23) ; D
(loadi key-1 -- 0x2b) ; F
(loadi key-2 -- 0x3b) ; J
(loadi key-3 -- 0x42) ; K
(loadi R28 -- 0x1000)

;;; Enable keyboard function
(loadi R23 -- kbd-function)
(store R23 dir 0x9000)

;; [Spelloop]
(lbl game-start)

;; get pointer + offset
(add R0 note-ptr ptr-offset)

;; load note and currentTime
(load note-column PTR r0)
(addi R1 R0 1)

(load note-time ptr R1)

;; set current-time register to current time (ms)
```

```
(load current-time dir 0x6000)

(cmpi -- note-column 0xFFFF)
(goto jeq reset-offset)

(cmpi -- note-time 0xFFFF)
(goto jeq reset-offset)

;; if currentTime > noteTime + 100 then goto note-exited-screen
(addi R3 note-time 100)
(cmp -- R3 current-time)
(goto jge note-exited-screen)

;;; if noteTime > 1800 + currentTime: //the note is 2 seconds away
;;;     R29 = 0 //pekaroffset = 0
;; else
;;     R29 += 2
(addi R5 current-time 1800)
(cmp -- R5 note-time)
(goto jge reset-offset)

;;; calculate sprite x,y
;;;     R4 <= X
;;;     R5 <= Y

;; X <= collumn * 32 + 64
(loadi R4 -- 32)
(mul   R3 R4 note-column)
(addi  R4 R3 64)

;; Y <= 416 - (time - current-time) / 4
(sub R6 note-time current-time)
(lsr R8 R6 2)
(loadi R7 -- 416)
(sub R5 R7 R8)

;; R0 = Sprite address <= 0x7000 + offset * 2
;; 2 because offset increments in steps of 2
(loadi R0 -- 2)
(mul R1 R0 ptr-offset)
(addi R0 R1 0x7000)

;; use collumn as sprite type
(store note-column ptr R0)
(addi R1 R0 1)

(store R4 ptr R1) ; store X
(addi R0 R1 1)

(store R5 ptr R0) ; store Y
```

```
(M inc* r29 2 r0)

(cmpi R29 -- 32)
(goto jeq reset-offset)

(goto jmp game-start)

(lbl reset-offset)

(loadi R0 -- 0) ; clear sprites

(cmpi -- R29 32)
(goto jeq reset-offset-end)

;; r1 = (+ 1 (* r29 2))
;; r2 = (+ r1 #x7000)
(muli R3 R29 2)
(addi R1 R3 1)
(addi R2 R1 0x7000)

(store R0 ptr R2)

(M inc* r29 2 r3)

(goto jmp reset-offset)
(lbl reset-offset-end)
(loadi R29 -- 0)

(goto jmp game-start)

;; [Avsluta]

(lbl game-end)
(loadi R0 -- 0)

;; reset x position of first 16 sprites
(scheme
 (map (lambda (n)
        `(STORE r0 dir ,n))
      (iota 16 #x7001 4)))

(scheme (make-list 9 '(HALT))) ; REALLY HALT

(lbl note-exited-screen)

(M inc* note-ptr 2 r0)

(load R0 ptr note-ptr)
(cmpi -- R0 0xFFFF)
```

```
(goto jeq game-end)

(goto jmp game-start)


;;; ------------------- GAME END ---------------------


;;; ------------------- KBD START --------------------

(defconst key-column        R15)
(defconst current-time-kbd R19)
(defconst current-key       R16)
(defconst note-ptr-save     r22)

(scheme*
 (define (key n)
   "Return␣register␣number␣for␣where␣key␣of␣number␣<n>␣has
information␣stored."
   (- 27 n)))

;; Find KeyCollumn
(lbl kbd-function)

;; Disable keyboard function
(loadi R23 -- 0xFFFF)
(store R23 dir 0x9000)

(load current-key dir 0x5000)

(load current-time-kbd dir 0x6000)
(addi note-ptr-save note-ptr 0)

;; check the four columns if this is the one hit
(scheme
 (apply append
  (map (lambda (n)
         `((LOADI key-column -- ,n)
           (CMP -- current-key ,(key n))
           (GOTO JEQ key-found)))
       (iota 4))))

;; No key of interest pressed
(goto jmp kbd-function-end)

(lbl key-found)

(defconst note-column-kbd r13)
(defconst note-time-kbd   r14)
```

```
(load note-column-kbd ptr note-ptr-save)

(addi R20 note-ptr-save 1)
(load note-time-kbd ptr R20)
(cmpi -- R13 0xFFFF)
(goto jeq kbd-function-end)

;; if noteTime >= 100 + currentTime: //the note is 300ms away
;;     goto kbd-function-end
(addi R17 current-time-kbd 100)
(cmp -- R17 note-time-kbd)
(goto jge kbd-function-end)

;; else if noteCollumn == KeyCollumn:
;;     score++
;;     goto kbd-function-end
(cmp -- key-column note-column-kbd)
(goto jeq add-score)

;; else
;;    goto next-key
(goto jmp next-key)

(lbl add-score)

(M inc r30 r18)

;; remove hit note
(addi R20 note-ptr-save 1)
(loadi R19 -- 23)
(store R19 ptr R20)

;;; == print score ==

(defconst temp-score r14)
(defconst   1s r15)
(defconst  10s r16)
(defconst 100s r17)

(addi temp-score R30 0)
(loadi   1s -- 0)
(loadi  10s -- 0)
(loadi 100s -- 0)

;; while tempScore > 0
(lbl score-loop) ; START

(cmpi -- temp-score 0)
(goto jeq score-loop-end)
```

```
(M dec temp-score r19) ; tempScore--

(M inc 1s r19) ; 1s++
(cmpi -- 1s 10)
(goto jeq add-tens)
(goto jmp score-loop) ; else

(lbl add-tens)
(M inc 10s r19) ; 10s++
(loadi 1s -- 0) ; 1s = 0
(cmpi -- 10s 10)
(goto jeq add-hundreds)
(goto jmp score-loop)

(lbl add-hundreds)
(M inc 100s r19) ; 100s++
(loadi 10s -- 0) ; 10s = 0
(cmpi -- 100s 10)
(goto jeq add-hundreds)
(goto jmp score-loop)

(lbl next-key)

;; else
;;     R22 += 2
;;     goto key-found
(M inc* note-ptr-save 2 r18)
(goto jmp key-found)

(lbl score-loop-end) ; END

;; +1 to align with our text tiles
(M inc   1s r19)
(M inc  10s r19)
(M inc 100s r19)

;; Writes score to screen
(store   1s dir 0x404A)
(store  10s dir 0x4049)
(store 100s dir 0x4048)

(lbl kbd-function-end)

;; Enable keyboard function
(loadi R23 -- kbd-function)
(store R23 dir 0x9000)

(rts)
```

(noop)

A.4.2 Maskinkod

Assemblykoden assembleras ner till VHDL-kod. Blocket ovan blir följande:

```
-- JUMP LABELS:
--   185: kbd-function-end
--   176: score-loop-end
--   172: next-key
--   164: add-hundreds
--   156: add-tens
--   144: score-loop
--   135: add-score
--   120: key-found
--    97: kbd-function
--    89: note-exited-screen
--    63: game-end
--    60: reset-offset-end
--    48: reset-offset
--     7: game-start
-- MACHINE CODE:
    0 => B"00011_11011_00000_0000000000100011_0", -- LOADI 27,  0, ( 3|35)
    1 => B"00011_11010_00000_0000000000101011_0", -- LOADI 26,  0, (11|43)
    2 => B"00011_11001_00000_0000000000111011_0", -- LOADI 25,  0, (27|59)
    3 => B"00011_11000_00000_0000000001000010_0", -- LOADI 24,  0, ( 2|66)
    4 => B"00011_11100_00000_0001000000000000_0", -- LOADI 28,  0, ( 0|4096)
    5 => B"00011_10111_00000_0000000001100001_0", -- LOADI 23,  0, ( 1|97)
    6 => B"00100_10111_00000_1001000000000000_0", -- STORE 23,  0, ( 0|36864)
    7 => B"01010_00000_11111_0000000000011101_0", -- ADD    0, 31, (29|29)
    8 => B"00010_01010_00001_0000000000000000_0", -- LOAD  10,  1, ( 0|0)
    9 => B"01011_00001_00000_0000000000000001_0", -- ADDI   1,  0, ( 1|1)
   10 => B"00010_01011_00001_0000000000000001_0", -- LOAD  11,  1, ( 1|1)
   11 => B"00010_01100_00000_0110000000000000_0", -- LOAD  12,  0, ( 0|24576)
   12 => B"11001_00000_01010_1111111111111111_0", -- CMPI   0, 10, (31|65535)
   13 => B"11010_00000_00000_0000000000110000_0", -- JEQ    0,  0, (16|48)
   14 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0|0)
   15 => B"11001_00000_01011_1111111111111111_0", -- CMPI   0, 11, (31|65535)
   16 => B"11010_00000_00000_0000000000110000_0", -- JEQ    0,  0, (16|48)
   17 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0|0)
   18 => B"01011_00011_01011_0000000001100100_0", -- ADDI   3, 11, ( 4|100)
   19 => B"10111_00000_00011_0000000000001100_0", -- CMP    0,  3, (12|12)
   20 => B"11100_00000_00000_0000000001011001_0", -- JGE    0,  0, (25|89)
   21 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0|0)
   22 => B"01011_00101_01100_0000011100001000_0", -- ADDI   5, 12, ( 8|1800)
   23 => B"10111_00000_00101_0000000000001011_0", -- CMP    0,  5, (11|11)
   24 => B"11100_00000_00000_0000000000110000_0", -- JGE    0,  0, (16|48)
   25 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0|0)
   26 => B"00011_00100_00000_0000000000100000_0", -- LOADI  4,  0, ( 0|32)
```

```
27 => B"01110_00011_00100_0000000000001010_0", -- MUL    3,  4, (10/10)
28 => B"01011_00100_00011_0000000001000000_0", -- ADDI   4,  3, ( 0/64)
29 => B"01100_00110_01011_0000000000001100_0", -- SUB    6, 11, (12/12)
30 => B"00111_01000_00110_0000000000000010_0", -- LSR    8,  6, ( 2/2)
31 => B"00011_00111_00000_0000000110100000_0", -- LOADI  7,  0, ( 0/416)
32 => B"01100_00101_00111_0000000000001000_0", -- SUB    5,  7, ( 8/8)
33 => B"00011_00000_00000_0000000000000010_0", -- LOADI  0,  0, ( 2/2)
34 => B"01110_00001_00000_0000000000011101_0", -- MUL    1,  0, (29/29)
35 => B"01011_00000_00001_0111000000000000_0", -- ADDI   0,  1, ( 0/28672)
36 => B"00100_01010_00001_0000000000000000_0", -- STORE 10,  1, ( 0/0)
37 => B"01011_00001_00000_0000000000000001_0", -- ADDI   1,  0, ( 1/1)
38 => B"00100_00100_00001_0000000000000001_0", -- STORE  4,  1, ( 1/1)
39 => B"01011_00000_00001_0000000000000001_0", -- ADDI   0,  1, ( 1/1)
40 => B"00100_00101_00001_0000000000000000_0", -- STORE  5,  1, ( 0/0)
41 => B"01011_00000_11101_0000000000000010_0", -- ADDI   0, 29, ( 2/2)
42 => B"01011_11101_00000_0000000000000000_0", -- ADDI  29,  0, ( 0/0)
43 => B"11001_11101_00000_0000000000100000_0", -- CMPI  29,  0, ( 0/32)
44 => B"11010_00000_00000_0000000000110000_0", -- JEQ    0,  0, (16/48)
45 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
46 => B"11000_00000_00000_0000000000000111_0", -- JMP    0,  0, ( 7/7)
47 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
48 => B"00011_00000_00000_0000000000000000_0", -- LOADI  0,  0, ( 0/0)
49 => B"11001_00000_11101_0000000000100000_0", -- CMPI   0, 29, ( 0/32)
50 => B"11010_00000_00000_0000000000111100_0", -- JEQ    0,  0, (28/60)
51 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
52 => B"01111_00011_11101_0000000000000010_0", -- MULI   3, 29, ( 2/2)
53 => B"01011_00001_00011_0000000000000001_0", -- ADDI   1,  3, ( 1/1)
54 => B"01011_00010_00001_0111000000000000_0", -- ADDI   2,  1, ( 0/28672)
55 => B"00100_00000_00001_0000000000000010_0", -- STORE  0,  1, ( 2/2)
56 => B"01011_00011_11101_0000000000000010_0", -- ADDI   3, 29, ( 2/2)
57 => B"01011_11101_00011_0000000000000000_0", -- ADDI  29,  3, ( 0/0)
58 => B"11000_00000_00000_0000000000110000_0", -- JMP    0,  0, (16/48)
59 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
60 => B"00011_11101_00000_0000000000000000_0", -- LOADI 29,  0, ( 0/0)
61 => B"11000_00000_00000_0000000000000111_0", -- JMP    0,  0, ( 7/7)
62 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
63 => B"00011_00000_00000_0000000000000000_0", -- LOADI  0,  0, ( 0/0)
64 => B"00100_00000_00000_0111000000000001_0", -- STORE  0,  0, ( 1/28673)
65 => B"00100_00000_00000_0111000000000101_0", -- STORE  0,  0, ( 5/28677)
66 => B"00100_00000_00000_0111000000001001_0", -- STORE  0,  0, ( 9/28681)
67 => B"00100_00000_00000_0111000000001101_0", -- STORE  0,  0, (13/28685)
68 => B"00100_00000_00000_0111000000010001_0", -- STORE  0,  0, (17/28689)
69 => B"00100_00000_00000_0111000000010101_0", -- STORE  0,  0, (21/28693)
70 => B"00100_00000_00000_0111000000011001_0", -- STORE  0,  0, (25/28697)
71 => B"00100_00000_00000_0111000000011101_0", -- STORE  0,  0, (29/28701)
72 => B"00100_00000_00000_0111000000100001_0", -- STORE  0,  0, ( 1/28705)
73 => B"00100_00000_00000_0111000000100101_0", -- STORE  0,  0, ( 5/28709)
74 => B"00100_00000_00000_0111000000101001_0", -- STORE  0,  0, ( 9/28713)
75 => B"00100_00000_00000_0111000000101101_0", -- STORE  0,  0, (13/28717)
76 => B"00100_00000_00000_0111000000110001_0", -- STORE  0,  0, (17/28721)
```

```
 77 => B"00100_00000_00000_0111000000110101_0", -- STORE  0,  0, (21/28725)
 78 => B"00100_00000_00000_0111000000111001_0", -- STORE  0,  0, (25/28729)
 79 => B"00100_00000_00000_0111000000111101_0", -- STORE  0,  0, (29/28733)
 80 => B"00000_00000_00000_0000000000000000_0", -- HALT   0,  0, ( 0/0)
 81 => B"00000_00000_00000_0000000000000000_0", -- HALT   0,  0, ( 0/0)
 82 => B"00000_00000_00000_0000000000000000_0", -- HALT   0,  0, ( 0/0)
 83 => B"00000_00000_00000_0000000000000000_0", -- HALT   0,  0, ( 0/0)
 84 => B"00000_00000_00000_0000000000000000_0", -- HALT   0,  0, ( 0/0)
 85 => B"00000_00000_00000_0000000000000000_0", -- HALT   0,  0, ( 0/0)
 86 => B"00000_00000_00000_0000000000000000_0", -- HALT   0,  0, ( 0/0)
 87 => B"00000_00000_00000_0000000000000000_0", -- HALT   0,  0, ( 0/0)
 88 => B"00000_00000_00000_0000000000000000_0", -- HALT   0,  0, ( 0/0)
 89 => B"01011_00000_11111_0000000000000010_0", -- ADDI   0, 31, ( 2/2)
 90 => B"01011_11111_00000_0000000000000000_0", -- ADDI  31,  0, ( 0/0)
 91 => B"00010_00000_00001_0000000000011111_0", -- LOAD   0,  1, (31/31)
 92 => B"11001_00000_00000_1111111111111111_0", -- CMPI   0,  0, (31/65535)
 93 => B"11010_00000_00000_0000000000111111_0", -- JEQ    0,  0, (31/63)
 94 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
 95 => B"11000_00000_00000_0000000000000111_0", -- JMP    0,  0, ( 7/7)
 96 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
 97 => B"00011_10111_00000_1111111111111111_0", -- LOADI 23,  0, (31/65535)
 98 => B"00100_10111_00000_1001000000000000_0", -- STORE 23,  0, ( 0/36864)
 99 => B"00010_10000_00000_0101000000000000_0", -- LOAD  16,  0, ( 0/20480)
100 => B"00010_10011_00000_0110000000000000_0", -- LOAD  19,  0, ( 0/24576)
101 => B"01011_10110_11111_0000000000000000_0", -- ADDI  22, 31, ( 0/0)
102 => B"00011_01111_00000_0000000000000000_0", -- LOADI 15,  0, ( 0/0)
103 => B"10111_00000_10000_0000000000011011_0", -- CMP    0, 16, (27/27)
104 => B"11010_00000_00000_0000000001111000_0", -- JEQ    0,  0, (24/120)
105 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
106 => B"00011_01111_00000_0000000000000001_0", -- LOADI 15,  0, ( 1/1)
107 => B"10111_00000_10000_0000000000011010_0", -- CMP    0, 16, (26/26)
108 => B"11010_00000_00000_0000000001111000_0", -- JEQ    0,  0, (24/120)
109 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
110 => B"00011_01111_00000_0000000000000010_0", -- LOADI 15,  0, ( 2/2)
111 => B"10111_00000_10000_0000000000011001_0", -- CMP    0, 16, (25/25)
112 => B"11010_00000_00000_0000000001111000_0", -- JEQ    0,  0, (24/120)
113 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
114 => B"00011_01111_00000_0000000000000011_0", -- LOADI 15,  0, ( 3/3)
115 => B"10111_00000_10000_0000000000011000_0", -- CMP    0, 16, (24/24)
116 => B"11010_00000_00000_0000000001111000_0", -- JEQ    0,  0, (24/120)
117 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
118 => B"11000_00000_00000_0000000010111001_0", -- JMP    0,  0, (25/185)
119 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
120 => B"00010_01101_00001_0000000000010110_0", -- LOAD  13,  1, (22/22)
121 => B"01011_10100_10110_0000000000000001_0", -- ADDI  20, 22, ( 1/1)
122 => B"00010_01110_00001_0000000000010100_0", -- LOAD  14,  1, (20/20)
123 => B"11001_00000_01101_1111111111111111_0", -- CMPI   0, 13, (31/65535)
124 => B"11010_00000_00000_0000000010111001_0", -- JEQ    0,  0, (25/185)
125 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
126 => B"01011_10001_10011_0000000001100100_0", -- ADDI  17, 19, ( 4/100)
```

```
127 => B"10111_00000_10001_0000000000001110_0", -- CMP    0, 17, (14/14)
128 => B"11100_00000_00000_0000000010111001_0", -- JGE    0,  0, (25/185)
129 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
130 => B"10111_00000_01111_0000000000001101_0", -- CMP    0, 15, (13/13)
131 => B"11010_00000_00000_0000000010000111_0", -- JEQ    0,  0, ( 7/135)
132 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
133 => B"11000_00000_00000_0000000010101100_0", -- JMP    0,  0, (12/172)
134 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
135 => B"01011_10010_11110_0000000000000001_0", -- ADDI  18, 30, ( 1/1)
136 => B"01011_11110_10010_0000000000000000_0", -- ADDI  30, 18, ( 0/0)
137 => B"01011_10100_10110_0000000000000001_0", -- ADDI  20, 22, ( 1/1)
138 => B"00011_10011_00000_0000000000010111_0", -- LOADI 19,  0, (23/23)
139 => B"00100_10011_00001_0000000000010100_0", -- STORE 19,  1, (20/20)
140 => B"01011_01110_11110_0000000000000000_0", -- ADDI  14, 30, ( 0/0)
141 => B"00011_01111_00000_0000000000000000_0", -- LOADI 15,  0, ( 0/0)
142 => B"00011_10000_00000_0000000000000000_0", -- LOADI 16,  0, ( 0/0)
143 => B"00011_10001_00000_0000000000000000_0", -- LOADI 17,  0, ( 0/0)
144 => B"11001_00000_01110_0000000000000000_0", -- CMPI   0, 14, ( 0/0)
145 => B"11010_00000_00000_0000000010110000_0", -- JEQ    0,  0, (16/176)
146 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
147 => B"01101_10011_01110_0000000000000001_0", -- SUBI  19, 14, ( 1/1)
148 => B"01011_01110_10011_0000000000000000_0", -- ADDI  14, 19, ( 0/0)
149 => B"01011_10011_01111_0000000000000001_0", -- ADDI  19, 15, ( 1/1)
150 => B"01011_01111_10011_0000000000000000_0", -- ADDI  15, 19, ( 0/0)
151 => B"11001_00000_01111_0000000000001010_0", -- CMPI   0, 15, (10/10)
152 => B"11010_00000_00000_0000000010011100_0", -- JEQ    0,  0, (28/156)
153 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
154 => B"11000_00000_00000_0000000010010000_0", -- JMP    0,  0, (16/144)
155 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
156 => B"01011_10011_10000_0000000000000001_0", -- ADDI  19, 16, ( 1/1)
157 => B"01011_10000_10011_0000000000000000_0", -- ADDI  16, 19, ( 0/0)
158 => B"00011_01111_00000_0000000000000000_0", -- LOADI 15,  0, ( 0/0)
159 => B"11001_00000_10000_0000000000001010_0", -- CMPI   0, 16, (10/10)
160 => B"11010_00000_00000_0000000010100100_0", -- JEQ    0,  0, ( 4/164)
161 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
162 => B"11000_00000_00000_0000000010010000_0", -- JMP    0,  0, (16/144)
163 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
164 => B"01011_10011_10001_0000000000000001_0", -- ADDI  19, 17, ( 1/1)
165 => B"01011_10001_10011_0000000000000000_0", -- ADDI  17, 19, ( 0/0)
166 => B"00011_10000_00000_0000000000000000_0", -- LOADI 16,  0, ( 0/0)
167 => B"11001_00000_10001_0000000000001010_0", -- CMPI   0, 17, (10/10)
168 => B"11010_00000_00000_0000000010100100_0", -- JEQ    0,  0, ( 4/164)
169 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
170 => B"11000_00000_00000_0000000010010000_0", -- JMP    0,  0, (16/144)
171 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
172 => B"01011_10010_10110_0000000000000010_0", -- ADDI  18, 22, ( 2/2)
173 => B"01011_10110_10010_0000000000000000_0", -- ADDI  22, 18, ( 0/0)
174 => B"11000_00000_00000_0000000001111000_0", -- JMP    0,  0, (24/120)
175 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0/0)
176 => B"01011_10011_01111_0000000000000001_0", -- ADDI  19, 15, ( 1/1)
```

```
177 => B"01011_01111_10011_0000000000000000_0", -- ADDI  15, 19, ( 0|0)
178 => B"01011_10011_10000_0000000000000001_0", -- ADDI  19, 16, ( 1|1)
179 => B"01011_10000_10011_0000000000000000_0", -- ADDI  16, 19, ( 0|0)
180 => B"01011_10011_10001_0000000000000001_0", -- ADDI  19, 17, ( 1|1)
181 => B"01011_10001_10011_0000000000000000_0", -- ADDI  17, 19, ( 0|0)
182 => B"00100_01111_00000_0100000001001010_0", -- STORE 15,  0, (10|16458)
183 => B"00100_10000_00000_0100000001001001_0", -- STORE 16,  0, ( 9|16457)
184 => B"00100_10001_00000_0100000001001000_0", -- STORE 17,  0, ( 8|16456)
185 => B"00011_10111_00000_0000000001100001_0", -- LOADI 23,  0, ( 1|97)
186 => B"00100_10111_00000_1001000000000000_0", -- STORE 23,  0, ( 0|36864)
187 => B"01001_00000_00000_0000000000000000_0", -- RTS    0,  0, ( 0|0)
188 => B"11111_00000_00000_0000000000000000_0", -- NOOP   0,  0, ( 0|0)
```