LINKÖPINGS UNIVERSITET

TSEA83

Datorkonstruktion

TI-83 Plus med Z80 på FPGA

Noah Hellman
Yousef Hashem
Dennis Derecichei
Jakob Arvidsson

10 juni 2018

Innehåll

1 Introduktion

En replikation av miniräknaren TI-83 Plus har implementerats i det hårdvarubeskrivande språket VHDL. Designen har sedan programmerats till Spartan 6 FPGA på ett Nexys3-kort kopplat till ett tangentbord och VGA-skärm som emulerar miniräknarens gränssnitt. Minnet från en miniräknare har överförts till minnet på Nexys3-kortet och på så sätt har operativsystemet och diverse spel kunnat köras utan några upptäckta problem.

1.1 Bakgrund

Projektet har utförts av en grupp på fyra studenter som ett delmoment av kursen TSEA83 på Linköpings universitet under större delen av vårterminen 2018. Alla gruppmedlemmar har sedan tidigare läst två förebyggande kurser; digitalteknik, TSEA22 och datateknik, TSEA82. Som projekt i TSEA83 eftersökte gruppen att bygga en existerande allmän processor som kunde köra tidigare skrivna program. Gruppen var sedan tidigare väl bekanta med miniräknaren TI-84 Plus som använder sig av en välkänd processor från 70-talet; Zilog Z80. En replikation av processorn valdes som projektets minimala mål och TI-84-miniräknaren valdes som projektets önskvärda mål. Under projektets gång upptäcktes det att en annan miniräknare, TI-83p hade färre hårdvarukomponenter men nästan identisk funktion vilket ledde till att den implementerades istället.

1.2 Syfte

Syftet med konstruktionen var inlärning av kunskap om hur processorer fungerar och att få en inblick i hur uppbyggnaden av en processor kan se ut. Målet var att skapa en Z80-processor som därefter kunde användas för bygga en TI-miniräknare.

1.3 Referenser och resurser

För att förstå hur Z80:n är uppbyggd och hur den fungerar har boken *Programming the Z80*[17] av Rodnay Zaks varit till väldigt stor nytta. Särskilt kapitel två om processorns organisation. Flera artiklar av Ken Shirriff såsom *How the Z80's registers are implemented*[9] har också varit till stor hjälp. Zilog:s användarmanual[13] har använts väl som referens för hur instruktioner, avbrott och flaggor ska fungera. Tidsdiagrammen har även använts som bas för implementationen. En tabell från ClrHome[3] har använts mycket som snabb referens för alla instruktioners objektkoder.

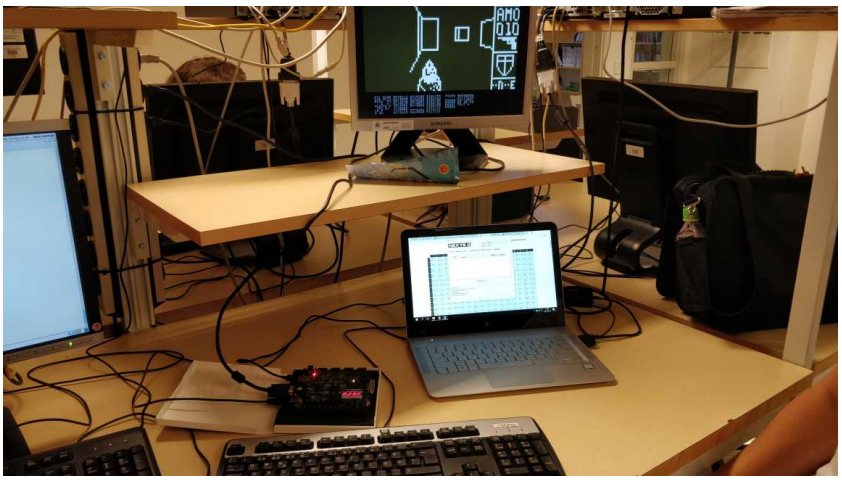
För att förstå hur TI-miniräknarna fungerar har sidor som *Z80 Heaven*[15] och *WikiTI*[14] använts. Sidorna om miniräknarens portar på WikiTI har varit väldigt användbara för att få reda på hur miniräknarens komponenter ska fungera. Källkoden för emulatorer har också varit väldigt användbara för detta syfte. Specifikt har koden till *z80e*[10] och *Tilem2*[12] granskats väl.

Emulatorerna har även använts för att jämföra resultatet med implementationen. Tilem2 har en väldigt användbar debugger som gjorde att en fungerande version kunde köras parallellt med projektet instruktion för instruktion. Det här var oerhört hjälpsamt

i slutet av projektet för att hitta de sista felen och slutligen få operativsystemet att fungera felfritt.

För att skriva kod till processorn har textfiler med instruktioner skrivits med textredigerare och därefter assemblats till objektkod med *z80asm*[16]. *GHDL*[6] tillsammans med *GTKWave*[7] har använts för att simulera VHDL-kod och visa dess vågdiagram.

Digilent Adept har använts för att skicka TI-83p-ROM:en och andra program direkt till minnet på FPGA-kortet. TI-83p-ROM:en har laddats från en TI-83p-miniräknare. Program och spel för miniräknaren har hämtats från *ticalc.org*:s filarkiv[11].



Figur 1: VGA-skärm och tangentbord kopplat till FPGA-kort tillsammans med laptop som används för att programmera FPGA:n samt ladda program till minnet.

2 Konstruktion

Konstruktionen har använt ett Nexys3-kort med en Spartan6-FPGA från Digilent som designen programmerats till. Utöver FPGA:n används även RAM-minnet på kortet samt knappar, strömbrytare och dess 7-segments-display.

Kortet är kopplat till en dator som tillför ström och programmerar bitfilen till FPGA:n på kortet. Datorn används även för att skriva program till RAM. Ett PS/2 tangentbord är kopplat till FPGA-kortets USB-port och en VGA-skärm till dess VGA-port.

2.1 Ladda program

För att ladda program till kortets RAM har programmet "Digilent Adept" använts. För att ladda en fil till minnet väljs RAM under fliken *Memory*. Därefter väljs filen under *Write File To Memory* och skrivs till minnet när *Write* trycks. Därefter kan bitfilen laddas under fliken *Config*. Bitfilen kan väljas och därefter programmeras genom att trycka på *Program*. Processorn kommer då starta och börja exekvera på adress `0x8000`.

Vid laddning till minnet kan en startadress väljas. Om ett program som startar i början av filen ska laddas bör det läggas på adressen `0x7c000` eftersom `0x8000` leder dit med den minnesmappning som är inställd vid uppstart. Om en hel ROM-dump ska laddas såsom TI-83p:s operativsystem ska den läggas på adress `0x00000` eftersom miniräknarens ROM börjar på där. Hur miniräknarens minne ligger i det fysiska minnet beskrivs i sektion 4.2.2 på sidan 33.

Om ett ROM laddas kommer den hamna i halt-läge och vänta på att användaren trycker på ON-knappen. ON-knappen är vänster Ctrl på PS/2-tangentbordet.

2.2 Gränssnitt

VGA-skärmen används för att visa miniräknarens LCD-skärm samt interna signalers värden för debuggingsyfte. De olika värdena inkluderar bland annat processorns interna

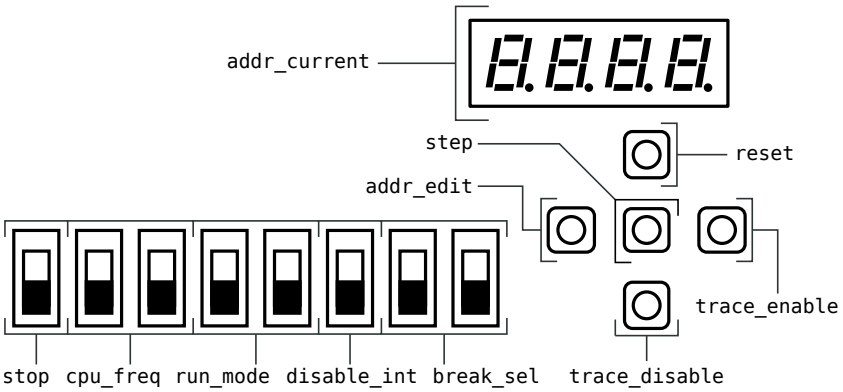
Figur 2: Konstruktionens display vid körning av spelet zDoom.

register, bussar och tillstånd, samt många portar på miniräknaren och den nuvarande minneskartläggningen.

Alla miniräknarens tangenter är kartlagda 1:1 till olika tangenter på tangentbordet. Se tabell B för vilka knappar som är bundna till vilka.

CPU:n kan även kontrolleras direkt med hjälp av instrumenten som befinner sig på kortet. Figur 3 visar vad de olika instrumenten på kortet används till. De olika brytarna justerar bland annat lägen och frekvenser. När alla brytare är nere kör processorn som vanligt på 6 MHz.

`stop`	Aktivera för att stanna CPU:n och TI-ASIC:en.
`cpu_freq`	Välj processorns frekvens enligt `00`: 6 MHz, `01`: 14 MHz, `10`: 1 MHz, `11`: 10 kHz.
`run_mode`	Välj läge enligt `00`: kör processorn normalt, `01`: kör en maskincykel i taget, `10`: kör en instruktion i taget, `11`: kör en T-cykel i taget.



Figur 3: Användning av de olika instrumenten på Nexys3.

`step` Fortsätter processorn efter brytning. Används för att stega igenom instruktioner eller för att fortsätta efter en brytpunkt.

`disable_int` Förhindra avbrott, `int`-signalen går aldrig aktiv om denna är på.

`break_sel` Välj brytpunkt enligt

`00`:	ingen,
`01`:	exekvering från vald adress,
`10`:	läsning från vald adress,
`11`:	skrivning till vald adress.

`addr_edit` En adress kan väljas genom att trycka på `addr_edit`. Därefter väljs siffra med vänster och högerknapp. Adressen visas på 7-segments-displayen. En punkt visas intill den siffra som är vald. Siffran kan ökas eller minskas med upp och nedknapparna. När adressen är inställd trycks mittknappen ner och punkten försvinner.

`reset` Återställer processorn och ASIC:ens alla register och vippor som vid uppstart. Processorn börjar då exekvera från adress `0x8000` igen. Bildminnet i LCD-kontrollern påverkas dock inte.

`trace_enable` Aktiverar spårning av alla hopp som processorn utför. Debug-displayen visar ett `E` om spårningen är på. Varje gång ett hopp utförs stoppas processorn tillfälligt och dess nuvarande adress och dess destinationsadress skrivs till det fysiska minnet på adress `0x88000`. Efter att ett program har kommit fel kan processorn stoppas och minnet läsas med hjälp av programmet Adept. Debug-displayen visar hur många bytes som ska hämtas efter `0x88000`. Därefter kan datan enkelt tolkas med hjälp av `hexdump` med kommandot

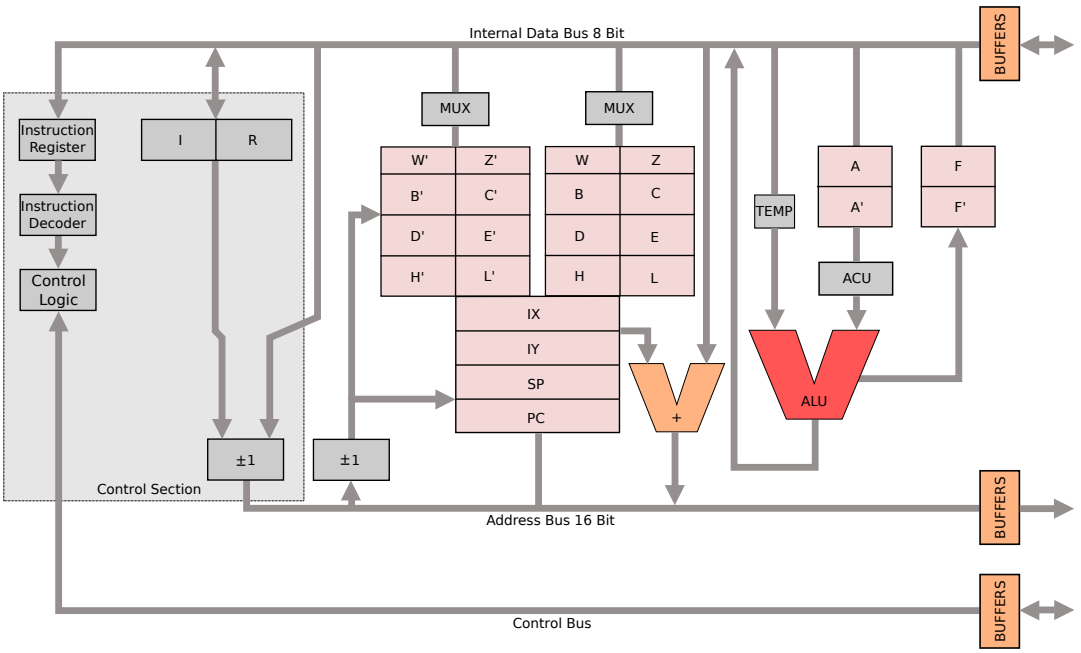
```
hexdump -e '"%06.6_ax: " 1/2 "%04x" " -> " \
            1/2 "%04x" "\n"' trace.bin
```

som exempelvis ger utdatan

```
000000: 0aad -> 0038
000004: 0038 -> 006a
000008: 006c -> 003a
00000c: 0047 -> 00a0
000010: 00a0 -> 07e6
000014: 07ea -> 07f2
  :          :
```

Exemplet är från när operativsystemet precis har startat och hamnat i halt-läge på adress `0x0AAD`. När användaren då trycker på ON-knappen sker ett avbrott och processorn hoppar till adress `0x0038` och påbörjar avbrottsrutinen.

`trace_disable` Stänger av spårning så att processorn inte stoppas vid hopp. Återställer även pekaren som bestämmer var nästa adress ska skrivas till `0x88000`.



Figur 4: Ett approximativt blockschema av Z80-processorns interna organisation. Av *Appaloosa*, CC BY-SA 3.0.

3 Teori

3.1 Processorn Z80

Z80 är en processor från 1976 tillverkad av Zilog. Den är fullt kompatibel med Intel 8080 men har även extra instruktioner och register. Nedan följer en kort beskrivning till vissa aspekt av processorn. Mer detaljerad information om processorn kan hittas i Zilog:s användarmanual.[13]

3.1.1 Register

Processorn har åtta 8-bitars register och tre 16-bitars register. Register med en bokstav är 8 bitar och register med två bokstäver är 16 bitar. Programmerare kan direkt komma åt och använda de flesta av dessa för allmänna behov. Vissa register har dock speciella instruktioner och specifika användningsområden. Par av 8-bitars register kan även användas som 16-bitars register av vissa instruktioner som bland annat `push BC` och `add HL, DE`. Det finns även ett par av register som programmeraren inte har tillgång till. Dessa är `W` och `Z` och används endast internt av processorn.

`A` Kallas även för "ackumulator", det är det primära registret för aritmetiska operationer och för att nå minnet. Registret är direkt kopplat till ALU:n och används ofta som in- och utdata för matematiska instruktioner, till exempel med instruktionen `add a, b` som adderar `A` med `B` och lagrar resultatet i `A`.

bit	7	6	5	4	3	2	1	0
flagga	S	Z	Y	H	X	P	N	C

Figur 5: Flaggornas uppsättning i `F`-registret.

`B` Vissa instruktioner använder `B` som räknare, till exemplet `cpir`.

`C` Används bland annat för att lagra portnumret vid en `in`- eller `ut`-instruktion. Parvis med B används den även som biträknare för vissa instruktioner.

`D`, `E` Används bland annat parvis för att lagra minnesadresser. Till exempel som destinationsadress för `ldir` som flyttar ett block i minnet.

`F` Registret som lagrar flaggorna. Vilken bit som representerar vad visas i figur 5. Under en ALU-instruktion laddas `F` med flaggorna. Det finns inga flyttnings- eller räkneinstruktioner såsom `ld` eller `sub` som använder `F`. `F` ändras endast indirekt via ALU:n. Det är dock möjligt att läsa och skriva till `F` via stacken om `push af` och `pop af` används tillsammans med andra push och pop-instruktioner.

`H`, `L` Register som ofta parvis lagrar en minnesadress. Många instruktioner använder HL för att referera till en plats i minnet. Dessa register kan även växla värden med `DE` genom `ex de, hl`-instruktionen.

`IX`, `IY` Dessa används på liknande sätt som `HL` men istället för att använda adressen som registret pekar på används relativ adressering. Ett exempel är instruktionen `and (ix+d)` som utför `and` på värdet som ligger `d` platser efter adressen som `IX` pekar på.

`SP` Stackpekaren, lagrar adressen till den nuvarande toppen av stacken. Speciella instruktioner som bland annat `push`, `pop` och `call` modifierar `SP` medan programmerarens förmåga att direkt modifiera `SP` är begränsad.

`I` `I` används i samband med avbrottshantering. Registret lagrar de översta åtta bitarna av adressen till en hoppadress för avbrottsläge 2.

`R` Register för memory refresh. Z80:n har inbyggd refresh-hantering för dynamiska minnen.

`PC` Instruktionspekaren, lagrar adressen till minnesplatsen som processorn ska hämta nästa instruktion ifrån. PC ändras med hoppinstruktioner som bland annat `jp`, `call` och `rst`. Ändras även av processorn vid avbrott.

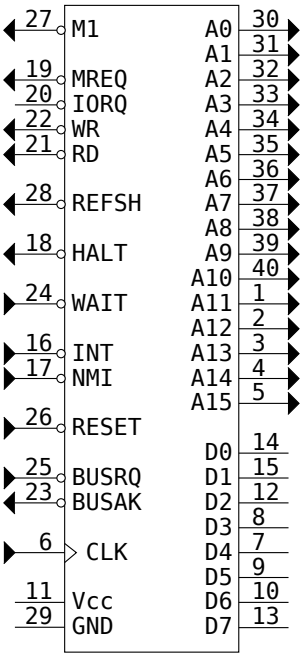
3.1.2 Flaggor

`C`, carry Används framförallt för att indikera minnessiffra vid addition och lån vid subtraktion. Vid aritmetisk operation är det en kopia av den bit 8. För vissa rotationsinstruktioner lagras en av operandens kanter i carry-flaggan.

`N`, subtract	Indikerar att instruktionen är en subtraktion. Används endast av `daa` för att korrigera resultatet till BCD.
`P`, parity/overflow	`P` visar om en aritmetisk overflow har inträffat för aritmetiska instruktioner. För bitinstruktioner som bland annat `xor` visar `P` om resultatet har paritet (jämnt antal nollor). `P` används även för att indikera om `BC` är noll vid blockinstruktioner som `ldir`. Flaggan sätts även till värdet av `IFF` vid exekveringen av `ld a, i` och `ld a, r`.
`X`	Odokumenterad och har ingen funktion, kallas även för `f3`. Generellt så är det en kopia av bit 3 av resultatet.
`H`, half-carry	Indikerar carry för resultatet av de första fyra bitarna. Z80:n har en 4-bitars ALU så den här flaggan kommer därför naturligt.
`Y`	Som `X` men kopia av bit 5.
`S`, sign	Indikerar om resultatet är negativt om tolkat som ett tvåkomplementstal. Det är en kopia av bit 7 i resultatet.
`Z`, zero	Indikerar att resultatet är noll.

3.1.3 Externa bussar

Z80:n har en 8-bitars databuss och en 16-bitars adressbuss. Processorn både skriver och läser till och från databussen men skriver endast till adressbussen. Det finns också en så kallad kontrollbuss som består av flera in- och utsignaler. Insignalerna består av `INT`, `NMI`, `RESET`, `BUSREQ` och `WAIT`. `INT` används för att signalera ett avbrott, `NMI` för ett *nonmaskable interrupt*. `WAIT` används vid minnes- eller IO-operationer för att indikera att indatan ännu inte är redo. `BUSREQ` signalerar till processorn att en enhet vill använda så att processorn sätter databussen och adressbussen till högimpedans. Utsignalerna består av `HALT`, `M1`, `IORQ`, `MREQ`, `RD`, `WR`, `RFSH`, `BUSACK`. `HALT` är aktiv när processorn är i halt-läge. `M1` är aktiv när processorn är i maskincykel ett. Om processorn hämtar från minnet när `M1` är aktiv innebär det att processorn hämtar en instruktion. Med hjälp av `RD` och `WR` signalerar processorn att den ska läsa eller skriva. Samtidigt signalerar processorn om den vill läsa/skriva till minnet eller till en IO-enhet med hjälp av `IORQ` och `MREQ`-signalerna. `IORQ` med `M1` kan även indikera att processorn är redo att läsa från databussen under ett avbrott. `RFSH` används för att skicka refresh-signaler till ett dynamiskt minne. `BUSACK` går aktiv när processorn har lagt hög impedans på bussarna efter en `BUSREQ`-signal.



Figur 6: Z80-processorns in- och utportar. Bild av Sakurambo, GFDL.

3.1.4 IO

Z80-processorn har ett system av portar för att hantera in- och
utdata till andra enheter utöver minne. När ett program vill skicka data till en enhet används en av `out`-instruktionerna. Då placeras portnumret på adressbussens lägre åtta bitar och utdatan läggs på databussen av processorn så att IO-enheten som korresponderar till den porten tar emot den. Under en `in`-instruktion placerar processorn porten på adressbussen men korresponderande IO-enhet placerar sin data på databussen. För att det här systemet ska fungera måste en dedikerad enhet utanför processorn hantera muxning av data till och från rätt enhet utifrån porten på adressbussen.

3.1.5 Avbrott

Vid början av exekveringen av varje instruktion kontrollerar processorn om `INT` eller `NMI` är aktiva. Vid en aktiv `INT`-signal reagerar processorn på olika sätt beroende på vilket läge som är inställt av programmeraren. Processorn har tre olika avbrottslägen som kan väljas med instruktionerna `im 0`, `im 1` och `im 2`. Det finns även två D-vippor `IFF1` och `IFF2`. Processorn svarar endast på en `INT`-signal om `IFF1` är 1. Programmeraren kan välja dess värde med instruktionerna `ei` och `di`. `IFF2` används för att lagra `IFF1`:s värde under en *nonmaskable interrupt*.

Vid avbrott under läge 0 hämtar processorn en 8-bitars instruktion från databussen och exekverar den. Instruktionen kan till exempel vara en `rst 28h` som direkt hoppar till adress `0x0028`. Processorn lägger även `PC` på stacken så att processorn kan återvända till programmet när avbrottsrutinen är färdig och kör `reti`. Under avbrottet återställs även `IFF1` och `IFF2` så att ett nytt avbrott inte kan ske under avbrottsrutinen. Avbrottsrutinen måste själv sätta igång avbrott igen med `ei`. För att ett avbrott inte ska kunna ske mellan `ei` och `reti` så kommer processorn inte acceptera avbrott förrän en instruktion efter att `ei` har använts.

Vid läge 1 hoppar processorn alltid till adress `0x0038`. Det här motsvarar en `rst 38h`-instruktion.

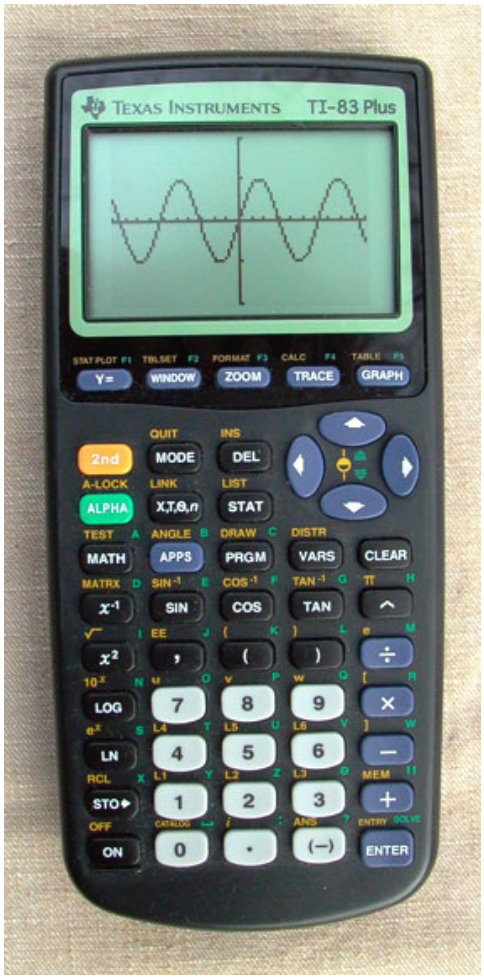
Vid läge 2 kommer processorn att hämta de lägre åtta bitarna av en adress från databussen. I `I`-registret ligger de åtta högre bitarna av adressen till en hoppadress som ligger i minnet. Processorn kommer hämta hoppadressen från denna adress minnet och lägga den i PC så att ett hopp utförs.

Vid en `NMI`-signal kommer processorn reagera precis som under läge 1 men att den hoppar till adress 0x66 istället för 0x38. Dessa avbrott sker oavsett om `IFF1` är 1 eller 0. Vid sådana avbrott återställs `IFF1` precis som innan för att förhindra avbrott men `IFF2` antar nu värdet av `IFF1`. På så sätt kan föregående värdet av `IFF1` återställas när avbrottsrutinen är färdig och exekverar `retn`-instruktionen.

3.2 Miniräknaren TI-83 Plus

Miniräknaren TI-83p är en grafritande miniräknare tillverkad av Texas Instruments. Miniräknaren använder en Z80 processor klockad till 6 MHz, en 96x64 monokrom LCD-skärm. TI-miniräknarna har en inbyggd *application specific integrated circuit* (ASIC) som bland annat muxar data mellan processorn och portarna. TI-83p:an använder sig av en T6A04 LCD kontroller. Detaljerad information om dess funktionalitet finns i dess datablad[4]. Andra delar av TI-83p:ans ASIC är odokumenterade men den innehåller bland annat hårdvarutimers, avbrottshanterare, tangentbordskontroller, minnesmappning och minnesskydd.

TI-83p använder sig inte av alla Z80:ns funktioner. Bland annat används inte *nonmaskable interrupts* eftersom ASIC:en aldrig skickar en `NMI`-signal. Det här innebär även att IFF2 aldrig kommer skilja sig från IFF1 så endast en D-vippa behövs. *Bus request* används aldrig då `BUSRQ` aldrig går aktiv.



Figur 7: En TI-83p-miniräknare från Texas Instruments. Bild av Westernelectric555, public domain.

Vid avbrott lägger ASIC:en ingen data på databussen så avbrottssläge 0 går inte att använda eftersom instruktionen är obestämd. Avbrottsläge 2 går att använda men eftersom databussen bestämmer de 8 sista bitarna på adressen till hoppadressen måste ett helt block av 256 bytes fyllas med hoppadressen. Miniräknarens operativsystem använder endast avbrottsläge 0. Miniräknaren maskar avbrott från olika källor med hjälp av port `03`. Vid ett avbrott måste därefter processorn stänga av avbrottet via port `04` för att det inte ska fortsätta hålla `int`-signalen hög efter att processorn återaktiverar avbrott.

3.2.1 Portar

För att ge en överblick över hur TI-83p kommunicerar med externa enheter följer en lista av miniräknarens alla portar.

`00`: Länkport — Kontrollerar länkporten för kommunikation mellan två TI-miniräknare.

`01`: Tangentbord — Skrivning väljer vilka grupper av tangenter som ska registrera knapptryck. Läsning av ger en mask av alla nedtryckta

	knappar för de nuvarande valda grupperna.
`02`: Status	Läsning ger bland annat batterinivå och visar om ROM är skrivskyddat.
`03`: Avbrottsmask	Maska de fyra avbrottskällorna; nedtryckning av ON-knappen, den första hårdvarutimer:n, den andra timer:n och avbrott från länkporten. Läsning visar hur masken ser ut.
`04`: Minnesläge/avbrott	Vid läsning indikerar de första fyra bitarna vilket typ av avbrott som har skett. Bit 0 visar till exempel att ON-knappen orsakade ett avbrott. Skrivning till porten bestämmer vilket minnesläge som ska användas och frekvensen till miniräknarens hårdvarutimer:s.
`05`: Exekvering/länkdata	Läsning ger den byte som senast mottogs via länkporten. Skriving väljer ut ROM pages som kan maskas med port `16` för att förbjuda eller tillåta exekvering.
`06`: Page A	Välj ut page A. Bit 6 väljer om det är en page från `ROM` eller `RAM`. De lägsta bitarna bestämmer dess nummer. Om till exempel `out (06h),a` exekveras då `A` är `41` så sätts page A till `RAM 1`.
`07`: Page B	Som `06` fast en andra page som kallas `B`.
`10`: LCD-kontroll	Kontrollera LCD-kontrollern. Bland annat hur pekarens position och hur den ska uppdateras vid skrivning av data.
`11`: LCD-data	Skriv eller läs data från pekarens position i LCD:ns bildminne.
`14`: ROM-skrivskydd	Aktivera eller avaktivera modifiering av ROM.
`16`: Exekveringsmask	Välj vilka pages från port `05` som ska tillåta exekvering.

Mer detaljerad information om varje port kan hittas på WikiTI[2].

3.2.2 Tangentbord

Tangentbordet har 50 knappar; 10 rader med 5 knappar vardera. Se figur 7. ON-knappen har särskild funktionalitet. Den skickar ett avbrott till processorn vid nedtryckning. Den används bland annat för att väcka processorn från halt-läge när miniräknaren är avstängd.

De resterande 49 knapparna har delats upp i 6 olika grupper som visas i figur 8. Varje grupp består av åtta knappar. En grupp kan representeras av en byte där varje bit indikerar om en knapp är nedtryckt eller inte. En nolla representerar en nedtryckt

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Grupp 0					UP	RIGHT	LEFT	DOWN
Grupp 1		CLEAR	$\wedge$	$\div$	$\times$	$-$	$+$	ENTER
Grupp 2		VARS	TAN	)	9	6	3	(-)
Grupp 3	STAT	PRGM	COS	(	8	5	2	.
Grupp 4	X,T,Θ,n	APPS	SIN	,	7	4	1	0
Grupp 5	ALPHA	MATH	x^{-1}	x^2	LOG	LN	STO	
Grupp 6	DEL	MODE	2ND	Y=	WINDOW	ZOOM	TRACE	GRAPH

Figur 8: Miniräknarens uppdelning av tangentbordets knappar i grupper.

knapp. Programmeraren kan aktivera grupper genom att skicka en byte till port `01`. En nolla på bit 0 aktiverar grupp 0, en nolla på bit 1 aktiverar grupp 1 och så vidare upp till bit 6. Värdet `0001 1111` aktiverar till exempel grupp 5 och 6 och deaktiverar de resterande grupperna.

När programmeraren läser från port `01` skickas en byte där alla aktiverade grupper har AND:ats. Om till exempel de tidigare grupperna är valda och användaren trycker ner tangenterna DOWN, LN och GRAPH kommer $\texttt{11111011} \cdot \texttt{11111110} = \texttt{11111010}$ tas emot. DOWN-knappen registeras inte eftersom grupp 0 är deaktiverad. Programmeraren vet inte om användaren tryckte på LN eller ZOOM eftersom de använder samma bit. För att veta exakt vilka knappar som är nedtryckta måste en sökning göras genom att gå igenom alla grupper men då endast en grupp är i taget är aktiverad.

3.2.3 Minne

Z80-processorn har en databuss på 16 bitar. Den kan därmed peka ut 65536 platser eller 64 KiB av minne. TI-83p har däremot ett 512 KiB ROM och ett 32 KiB RAM. För kunna komma åt allt har de 64KiB av det virtuella minnet delats upp i fyra 16 KiB pages. Varje page refererar till en lika stor page i det fysiska minnet. Dessa pages kan bytas ut för att komma åt en annan del av det fysiska minnet. ROM är uppdelat i 32 pages (`ROM 00`...`ROM 1F`) och RAM är uppdelat i 2 pages (`RAM 0` och `RAM 1`).

TI-83p använder två olika lägen vilkas layouts kan ses i figur 9. Läget och page A, B väljs med port `04`, `06` respektive `07`. [5]

start	page	slut
0000	ROM 00	3FFF
4000	MEMPAGE A	7FFF
8000	MEMPAGE B	BFFF
C000	RAM 0	FFFF

(a) läge 0

start	page	slut
0000	ROM 00	3FFF
4000	MEMPAGE A (jämn)	7FFF
8000	MEMPAGE A	BFFF
C000	MEMPAGE B	FFFF

(b) läge 1

Figur 9: Layout av virtuellt minne för de två lägena.

3.2.4 LCD-kontroller

TI-83p-miniräkaren använder en TOSHIBA T6A04 som LCD-kontroller. Kontrollern har ett inbyggt RAM på 960 bytes som lagrar 64 rader med 120 pixlar vardera. Eftersom skärmen är monokrom används endast en bit per pixel. LCD-skärmen visar endast de första 96 minnets kolumner, hela utrymmet kan dock användas för lagring.

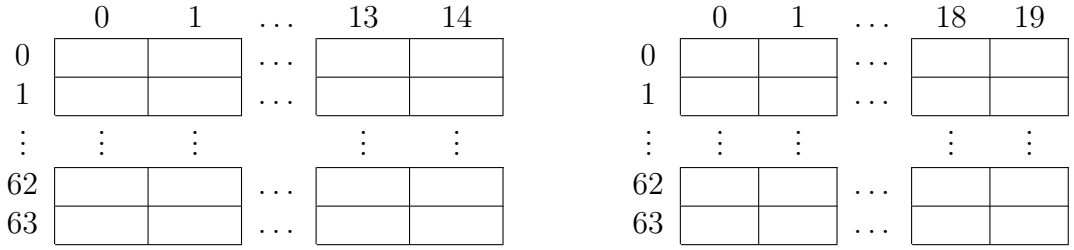
Minnet är uppdelat i pages, storleken på varje page kan vara sex eller åtta bitar. De två olika lägena visas i figur 10. En skrivning eller läsning sker alltid med en page i taget. Om läget är inställt på sex bitar används endast de sex lägre bitarna av den data som skickas till porten.

Kontrollern har en inbyggd pekare som lagrar `X` och `Y`-värdet av page. `X` är vilken rad page:n ligger på och `Y` pekar ut kolumnen. Vid läsning kommer page:n som pekaren refererar till skickas och vid skrivning kommer denna page skrivas över. De olika lägena gör att pekaren kommer referera till olika platser i minnet för lika värden på `Y`. Om `Y` exempelvis är 14 i 8-bitarsläge kommer page:n längst till höger vara vald. Men om `Y` är 14 i 6-bitarsläge kommer den valda page:n ligga några pages in från höger kant.

Pekarens värde kan väljas manuellt av programmeraren med port `11`. Men vid varje läsning eller skrivning till dataporten `11` kommer pekaren antingen öka eller minska `X` eller `Y` ett steg. Pekarens riktning ställs också in med port `10`.

LCD-kontrollern håller även koll på ett tredje värde; `Z`. Det här värdet avgör vilken rad `X` ska börja räkna ifrån vid läsning och skrivning. Om `X` och `Y` exempelvis är noll och `Z` är `31` kommer den första page:n på den mittersta raden (31) väljas. Eftersom `X` börjar på `Z` kan värdet överstiga antalet av skärmens rader. Då börjar den om på noll. Raden väljs därmed egentligen genom att beräkna $\texttt{X} + \texttt{Z} \mod 64$.

`Z` räknar inte upp eller ner vid läsning/skrivning utan sätts manuellt av programmeraren via port `10`. Syftet med `Z` är att möjliggöra vertikal scrollning utan att behöva flytta varje rad i minnet. Det här används bland annat av operativsystemet för hemskärmen. Då ökar `Z` med antalet pixlar för höjden av en rad och den översta raden skrivs över med den nya raden som då hamnar längst ner på skärmen eftersom den går runt.



(a) 8-bitars kliv, 15 pages per rad. (b) 6-bitars kliv, 20 pages per rad.

Figur 10: Uppdelning av pages för de två lägena.

3.3 Specifikationer

VGA

För att få ut en bild till VGA-skärmen har data skickats enligt VGA-standarden. Information och tidsintervaller finns i manualen för FPGA-kortet Nexys3 på sidan 15.[8]

PS/2

För att ta emot knapptryck från ett tangentbord har PS/2-standarden använts vid kommunikation. Hur standarden fungerar är beskrivet i Nexys3-manualen på sidan 13.[8]

Externt minne

Ett 16MB RAM, modell Micron M45W8MW16 har använts asynkront som primärminne för miniräknaren. Dess funktionalitet och protokoll kan hittas i dess manual.[1]

4 Hårdvara

Konstruktionen är uppdelad i tre delar; processorn, TI-ASIC:en och kontroller för externa enheter. Dess sammansättning kan ses i blockschema C.1. I blockschemat syns även att det finns olika klockor som genereras. Processorn kan köra på olika frekvenser medan ASIC:en alltid kör på 33 MHz och VGA-motorn på 25 MHz. Resterande komponenter använder systemklockan på 100 MHz. Klockorna kan även stängas av vid bland annat brytpunkter som använts för debugging.

Det finns en gemensam 8-bitars databuss som processorn, ASIC:en och även minnet delar på. Kontrollbussen och adressbussen från processorn går direkt till ASIC:en som hanterar dem först innan något går till de externa kontrollerna. Adressen behöver till exempel översättas innan den når det fysiska minnet. I ASIC:en sker även hanteringen av alla de olika portarna i miniräknaren. ASIC:en talar i sin tur med de externa komponenterna och skickar indata via portarna.

4.1 Z80

Z80:n kan delas upp i tre sektioner; kontroll, register och ALU. De är anslutna med varandra via databussen, addressbussen och en del tunnlar som alla kan ses i blockschema C.2. Kontrollsektionen har även kontrollsignaler till alla delar av processorn i form av ett långt kontrollord. Kontrollordet kallas för `cw` i blockschemat. Alla signaler i kontrollordet finns med i blockschemat förutom muxadresser till adress- och databussen samt alla lässignaler för alla register.

4.1.1 Kontrollsektion

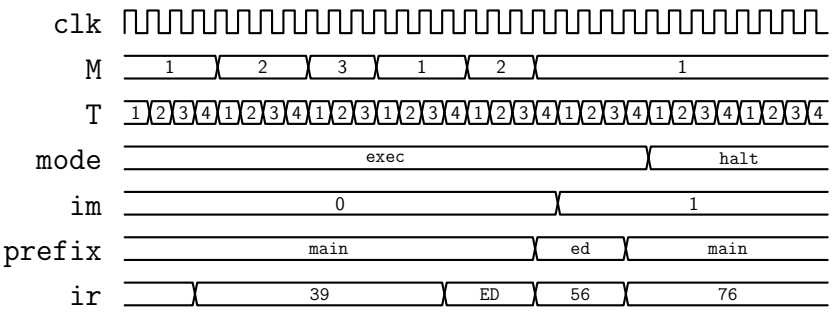
Kontrollsektionen består av tre komponenter; instruktionsregistret, tillståndsmaskinen och instruktionsdekodaren. Tillståndsmaskinen lagrar och ändrar hela processorns tillstånd, förutom alla registervärden. Utifrån processorns tillstånd och instruktionregistrets värde aktiverar instruktionsdekodaren, som endast är ett stort kombinatoriskt nät, olika kontrollsignaler.

Instruktionsregistret `IR`

`IR` är det register som instruktioner direkt laddas till när de läses från minnet. Här lagras de fram till att nästa instruktion laddas in. Om en instruktion har ett prefix lagras först prefixet i `IR` och skrivs kort därefter över med ett nytt prefix eller en instruktion.

Tillstånd

Tillståndsmaskinen har fem olika tillståndsvariabler; läget `mode`, avbrottsläget `im`, instruktionsprefixet `prefix`, maskincykeln `M` och T-tillståndet `T`. Tillståndsmaskinen lagrar dessa och sänder dem till instruktionskodaren. I blockschema C.2 är alla dessa signaler inkluderade i signalen `cpu_state`.



Figur 11: Tillstånd samt värdet av `IR` under exekveringen av programmet `add hl,sp; im 1; halt`

`T` är ett heltal mellan 1 och 6 som avgör vilken klockpuls av en maskincykel processorn är i. En maskincykel kan bestå av 3 till 6 klockpulser. Med andra ord ökas `T` med 1 varje klockpuls förutom när processorn byter maskincykel, då sätts `T` till 1 igen.

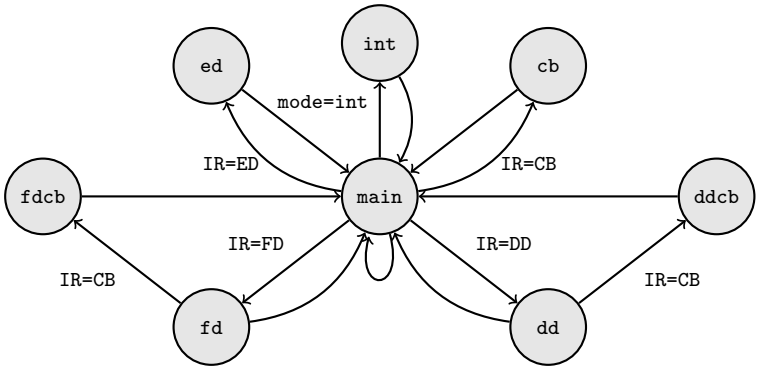
`M` är ett heltal mellan 1 och 5 men som avgör vilken del av en instruktion processorn består i. Vid instruktioner som består av flera ord kommer maskincykeln gå tillbaka till 1 vid varje hämtning. Det här är för att göra instruktionsdekodaren enklare. Instruktionshämtning sker därmed endast på maskincykel 1. Det innebär dessutom att liknande instruktioner med olika prefix kan återanvändas eftersom allas exekveringsfas börjar på maskincykel 1. Till exempel instruktionerna `add hl,sp` (39) och `add ix,sp` (DD39) är identiska förutom att de har förskjutna maskincyklar och destinationsregistret har olika addresser. Deras maskincyklar visas i figur 12.

Genom att sätta maskincykeln till 1 vid andra hämtningen hamnar maskincyklarna i fas och instruktionsdekodaren kan ge identiska kontrollsignaler för båda instruktioner utöver addressen till destinationsregistret.

Prefix-tillståndet tillsammans med `IR` används för att avgöra vilken instruktion som ska exekveras. Det finns åtta olika prefix; `main` för huvudinstruktioner, `ed` för utökade instruktioner, `cb` för bitinstruktioner, `dd/fd` för IX/IY-instruktioner, `ddcb/fdcb` för IX/IY-bitinstruktioner samt `int` som sätts vid avbrott. Vid processorns start och efter varje instruktion sätts prefixet till `main`. Om `IR` därefter laddas med en prefix-byte kommer prefix-tillståndet att övergå till motsvarande prefix och maskincykeln sätts till 1 så att hämtningsfasen börjar om på nytt. Därefter kan ett nytt prefix eller en instruk-

maskincykel	1	2	3	4
`add hl,sp`				
prefix	`main`			
fas	hämta 39	addera låga	addera höga	färdig
`add ix,sp`				
prefix	`main`	`dd`		
fas	hämta DD	hämta 39	addera låga	addera höga

Figur 12: tidsdiagram för instruktionerna `add hl,sp` och `add ix,sp` utan korrigering av maskincyklar.



Figur 13: Tillståndsdiagram för prefixtillståndet.

tion laddas. Om en instruktion laddas påbörjas dess exekvering direkt. I ovan exempel är prefix-byten `DD` och instruktions-byten `39`.

Det finns dessutom ett prefix som inte går att komma åt med ett prefix i instruktionen. Det är `int`-prefixet. `int`-prefixet sätts alltid efter att en instruktion har avslutats om ett avbrott har skett. Om prefixet är `int` kommer instruktionsdekodaren inte exekvera någon instruktion utefter vad `IR` innehåller. Istället kommer den då alltid att köra en avbrottssekvens utefter vad `im`-tillståndet är. `im`-tillståndet avgör det nuvarande avbrottsläget och kan väljas av programmeraren med `im 0`, `im 1` och `im 2`-instruktionerna.

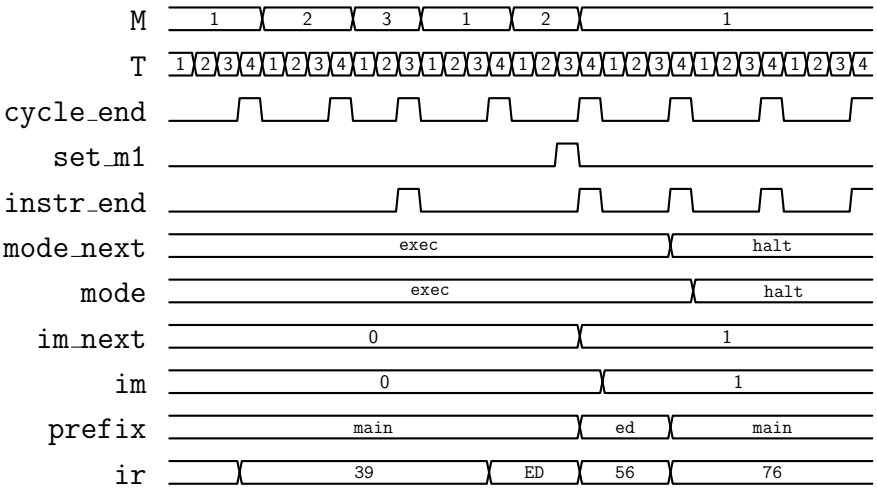
Prefixet uppdateras med lässignalen till `IR`. Det nästa prefixet beror på det föregående värdet i `IR` som kan ses i figur 11 vid övergången från `main` till `ed`.

Det sista tillståndet i processorns tillståndsmaskin är `mode` som kan anta de tre olika tillstånden `exec`, `halt` och `interrupt`. Normalt sätt är processorn i `exec`. Då kommer instruktionsdekodaren exekvera instruktioner utefter `IR` prefixet och även `IM`. I `halt` däremot kommer processorn fortsätta exekvera instruktioner men den hämtar inte nya instruktioner under maskincykel 1. Normalt sätt hamnar processorn i `halt` från `HALT`-instruktionen. När `HALT` instruktionen avslutas övergår processorn till `halt`-läge och maskincykel 1 men eftersom den inte hämtar en ny instruktion kommer den exekvera `HALT` om och om igen tills processorn får ett avbrott eller återställs. Det sista läget är `interrupt` som väljs när ett avbrott sker. Anledningen till att det finns både ett `interrupt`-läge och ett `int`-prefix är för att även om ett avbrott har skett kan en överlappande instruktion behöva exekvera färdigt innan avbrottsekvensen körs. `interrupt`-läget förhindrar behöver förhindra att en ny instruktion som inte ännu ska exekveras hämtas och att programräknaren inte ökas innan den ska sparas till stacken.

Kontrollsignaler

Kontrollsignalerna är uppdelade i tre grupper; extern kontrollbus, tillståndskontroll samt interna kontrollsignaler.

Den externa kontrollbussen har sex utsignaler som instruktionsdekodaren aktiverar; `M1`, `MREQ`, `IORQ`, `RD` och `WR`. `RD` och `WR` används för att signalera läsning eller skrivning till och från antingen minnet eller IO-portar. Processorn signalerar vilket med hjälp av



Figur 14: Programmet från figur 11 med kontrollsignalerna för tillstånden synliga.

`MREQ` och `IORQ`. `M1` används för att signalera om CPU:n är i maskincykel 1.

Kontrollsignalerna för tillståndsmaskinen avgör processorns nästa tillstånd. Signalerna är `set_m1`, `cycle_end`, `instr_end`, `mode_next` och `im_next`. I blockschema C.2 är dessa samlade under signalen `ctrl`.

`set_m1` används för syftet beskrivet ovan; att synkronisera liknande instruktioner för att kunna återanvända hårdvara i instruktionsdekodaren till flera instruktioner. Den används endast efter hämtning av prefix så att varje instruktion alltid börjar exekvera under `M1T4`

`cycle_end` används för att signalera att den nuvarande cykeln är avslutad. I slutet av varje maskincykel går `cycle_end` aktiv så att tillståndsmaskinen återställer `T`-tillståndet till `T1` och ökar `M` med ett. På liknande sätt går `instr_end` alltid aktiv i slutet av en instruktion. När den går aktiv återställs maskincykeln till `M1`, `mode` antar värdet av `mode_next` och `im` antar värdet av `im_next`. Notera dock att instruktionen fortsätter att exekvera under nästa maskincykel efter att `instr_end` har gått aktiv. Först när `IR` har laddats med nästa instruktion (T4) är det nästa instruktion som exekveras. `mode_next` och `im_next` vanligtvis antar det nuvarande värdet av `mode` respektive `im` och antar endast ett annat värde då `instr_end` går aktiv om ett byte ska ske.

Processorn har även en grupp kontrollsignaler för alla andra komponententer inuti processorn som alla tillhör ett långt kontrollord. För att ge en överblick inför de resterande två sektionerna följer en kort beskrivning av varje signal i kontrollordet.

`dbus_src` Komponentens utdata som ska muxas till databussen. Kan komma från `dbufi`, registerfilen, `TMP` eller ALU:n samt en sträng av nollor.

`abus_src` Källan till adressbussen; registerfilen, adressadderaren eller `rst`-adressen.

`rf_daddr` Adressen till ett 8-bitars register i registerfilen vars värde läggs som utdata mot databussen.

`rf_addr` Adressen till ett 16-bitars register i registerfilen vars värde läggs som utdata mot adressbussen och adressadderaren.

`rf_rdd`	Läs från databussen och skriv värdet till registret som `rf_daddr` pekar på.
`rf_rda`	Läs från adressbussen och skriv värdet till registret som `rf_aaddr` pekar på.
`rf_swp`	Utför ett byte i registerfilen. Signalen kan anta något av `none`, `af`, `reg`, `dehl` och `afwz`. `af` växlar värdena på `AF` och `AF'`. `reg` växlar värdena på `BC` med `BC'`, `DE` med `DE'` och `HL` med `HL'`. `dehl` växlar värdena av `DE` och `HL`. `afwz` växlar värdena mellan `AF` och `WZ`.
`rf_ldpc`	Ladda värdet av registret som `rf_aaddr` pekar på till `PC` i registerfilen.
`f_rd`	Läs flaggor till `F` som ligger i registerfilen.
`pv_src`	Var värdet av `P`-flaggan ska komma ifrån.
`ir_rd`	Läs från databussen till instruktionsregistret.
`addr_op`	Operation som ska utföras på adressbussens värde innan det går till registerfilen. Kan vara `none`, `inc` eller `dec`.
`rst_addr`	Bitar 3-5 av adressen vid en `rst`-instruktion.
`iff_next`	Nästa värde av `iff`, D-vippan för avbrott.
`alu_op`	Instruktionen som ska utföras av ALU:n.
`act_rd`	Läs värdet av `A` i registerfilen till `ACT`.
`tmp_rd`	Läs värdet från databussen till `TMP`.
`data_rdi`	Läs från den externa databussen till buffern `dbufi`.
`data_rdo`	Läs från databussen till buffern `dbufo`.
`data_wro`	Skriv värdet från `dbufo` till den externa databussen.
`addr_rd`	Läs från adressbussen till buffern `abuf`.

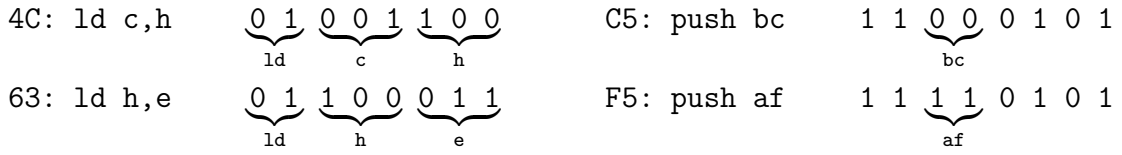
4.1.2 Registersektion

Registersektionen består av tre komponenter; en registerfil, en adressadderare och en adressinkrementerare. Blockschema C.2 visar hur de är sammankopplade.

Registerfil

Nästan alla register i processorn har implementerats med hjälp av ett litet RAM. Flera register behöver sällan läsas eller skrivas till samtidigt så en lässignal och en väg till databussen för varje register är onödigt. Som nämnt tidigare kan ett par av två 8-bitars register användas som ett 16-bitars register. Vid läsning och skriving till registerfilens RAM kan antingen 8- eller 16-bitars kliv användas. För att referera till ett 8-bitars register används ingången `daddr` som är en adress på fem bitar. För att referera till ett 16-bitars register används `aaddr` som är en adress på fyra bitar. `aaddr` är identisk till `daddr` förutom att den saknar bit 0 som avgör om det är den höga eller låga byte:n. Figur 15 visar alla registers externa adresser, de adresser som instruktionsdekodaren använder.

Dessa adresser har bland annat valts utefter kodningen i Z80:s instruktioner för att förenkla instruktionsdekodaren. Instruktioner som refererar till `B`, `C`, `D`, `E`, `H`, `L`, `A`, `BC`, `DE`, `HL` och `AF` använder alltid samma kodning i en del av instruktionen. De tre minst signifikanta bitarna för 8-bitars register är tagna direkt från instruktionen och på liknande sätt de två sista bitarna för 16-bitars register. Ett exempel är instruktionerna `ld c,h` och `ld h,e` samt `push bc` och `push af`:



daddr	hög	låg	daddr
00 00000	B	C	00001 01
02 00010	D	E	00011 03
04 00100	H	L	00101 05
06 00110	A	F	00111 07
08 01000	W	Z	01001 09
10 01010	SPh	SPl	01011 11
12 01100	IXh	IXl	01101 13
14 01110	IYh	IYl	01111 15
16 10000	I	R	10001 17
18 10010	PCh	PCl	10011 19

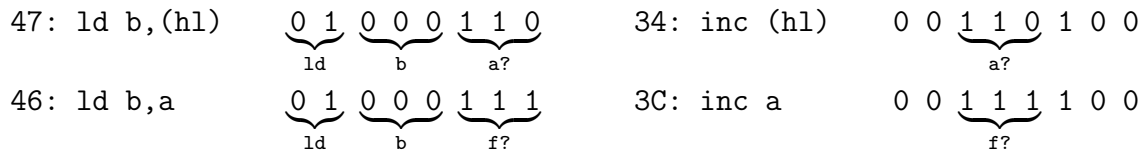
(a) 8-bitars kliv.

aaddr	register
00 0000	BC
01 0001	DE
02 0010	HL
03 0011	AF
04 0100	WZ
05 0101	SP
06 0110	IX
07 0111	IY
08 1000	IR
09 1001	PC

(b) 16-bitars kliv.

Figur 15: Registrernas externa adresser för registerfilen med 8- eller 16-bitars *stride* eller kliv.

Det finns däremot undantag där instruktionens kodning har ändrats. Se objektkoderna för instruktionerna `ld b,(hl)`, `ld b,a`, `inc (hl)` och `inc a`:



Dessa instruktioner refererar inte till `A` och `F` som koderna föreslår. Instruktioner som har adressen för vårt `A` hänvisar egentligen till `(hl)`, platsen i minnet som `HL` pekar på. Instruktioner som använder adressen för vårt `F` refererar egentligen till `A`. Anledningen till det här är att om instruktionernas kodning används kommer `AF` lagras som `FA` istället. När `AF` ska hanteras måste då registerfilen byta plats på den höga och låga byte:n vid läsning och skrivning. Lösningen till det här är att översätta `111` till `110` och `110` till `111` direkt i instruktionsdekodaren. På så sätt kan `A` och `F` enkelt hanteras som antingen två 8-bitars register eller ett 16-bitars register. Notera att vid instruktioner som refererar till `(hl)` eller i vårt fall `F` kommer värdet inte att hämtas från registerfilen och därmed används inte adressen. Adressen till `F` används endast vid `push` och `pop` eftersom där måste en byte i taget överföras till eller från stacken i minnet.

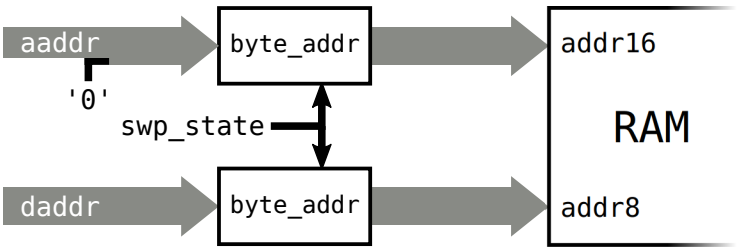
Ett alternativ för att implementera registerfilen för det här är att helt enkelt skapa ett RAM med den struktur och de externa adresserna som i figur 15a. Då skrivs eller hämtas en byte från byte:n som `daddr` pekar på. Där `aaddr` pekar på skrivs eller hämtas två bytes istället.

Det uppstår dock ett problem när `ex` instruktionerna ska användas. Det finns tre sådana instruktioner; `ex af,af'`, `ex de,hl` och `exx`. `exx` växlar värdet på BC med BC', DE med DE' och även HL med HL'. Dessa antar att det finns en kopia av vissa register som kan lagra originalregistrets värde temporärt. Implementationen ser därför ut som i figur 16 istället. Notera att det inte finns ett primärregister och ett sekundärt "prim"-register utan istället två identiska kopior av varje register. När ett byte sker ändras endast pekaren till det registret till den andra kopian. Båda kopior kan därmed agera som både primärregister och prim-register. Det här leder till att inga inga värden flyttas vid ett byte.

Registerfilen måste därmed lagra pekare som avgör vilket register som ska användas. Totalt behövs vippor för att hålla koll på alla växlingar:

adress	hög	låg	adress	adress	hög	låg	adress
00 00000	B	C	00001 01	14 01110	A	F	01111 15
02 00010	B	C	00011 03	16 10000	W	Z	10001 17
04 00100	D	E	00101 05	18 10010	SPh	SPl	10011 19
06 00110	D	E	00111 07	20 10100	IXh	IXl	10011 21
08 01000	H	L	01001 09	22 10110	IYh	IYl	10111 23
10 01010	H	L	01011 11	24 11000	I	R	11001 25
12 01100	A	F	01101 13	26 11010	PCh	PCl	11011 27

Figur 16: Registernas interna organisation i registerfilens RAM.



Figur 17: Två identiska kombinatoriska nät används för att översätta adresserna `baddr` och `daddr` till de interna adresserna.

`reg` Avgör om `BC`, `DE` och `HL` ska hänvisas till den första eller andra kopian av registret. Det här motsvarar bit 1 i adressen. Den skiftas när `rf_swp` är `reg`.

`af` Avgör om `AF` ska hänvisas till första eller andra kopian på liknande sätt.

`dehl0` Avgör om `DE` och `HL` ska hänvisa till sig själva eller varandra när `reg` är noll. Ändras endast då `rf_swp` är `dehl` och `reg` är noll. Om endast en vippa användes skulle även `DE'` och `HL'` byta plats från programmerarens sida.

`dehl1` Som `dehl0` fast då `reg` är ett.

`afwz` Avgör om `AF` och `WZ` ska hänvisa till sig själva eller varandra. Denna används endast internt så det är okej att den även ändrar `AF'` eftersom den byter tillbaka innan instruktionen är avklarad.

Det här är implementerat med hjälp av ett kombinatoriskt nät som översätter adressen till dess nuvarande position i registerfilens RAM utifrån vad D-vipporna har för nuvarande värde.

Adressinkrementerare

Mellan adressbussen och ingången till registerfilen sitter en inkrementerare som kan antingen inkrementera, dekrementera eller behålla adressbussens nuvarande värde. Med hjälp av denna kan en adress från registerfilen läggas på adressbussen och sedan antingen inkrementeras eller dekrementeras och därefter laddas till samma register. På det här viset stegas `PC` och `SP` upp eller ner. Det går däremot inte att hänvisa till två olika adresser i registret. Det finns även `inc` och `dec`-instruktioner för 16-bitars register som denna används till.

Adressadderare

Det finns även en adderare som tar adressen `aaddr` pekar på och adderar den med värdet på databussen och därefter skickar resultatet till databussen. Det här används för relativa adresseringar som till exempel `jr d` som hoppar `d` steg i programmet och `cp (iy+d)` som jämför värdet `d` steg efter värdet som `IY` pekar på i minnet med `A`.

4.1.3 ALU-sektion

ALU-sektionen hanterar matematiska beräkningar samt hantering av flaggor. Registerna `ACT` och `TMP`, ALU:n själv, en mux till `P`-flaggan samt en komparator för addressindatan tillhör ALU-sektionen. Hur de är sammankopplade visas i mitten av blockschema C.2.

`ACT` och `TMP`

Det primära syftet av registerna `ACT` och `TMP` att lagra operanderna under ALU:ns operation så att resultatet direkt kan läggas på databussen. De två operanderna till ALU:n måste alltid laddas till `ACT` och `TMP`. `ACT` kallas för den temporära ackumulatorn och kan endast ladda värden från `A` i registerfilen. Vissa instruktioner använder endast en operand, i så fall laddas den till `TMP` och `ACT`:s värde har ingen påverkan på resultatet. `TMP` kan ladda ett godtyckligt värde eftersom den är ansluten till databussen. `TMP` kan även skriva till databussen och kan användas för temporär lagring, därav namnets ursprung.

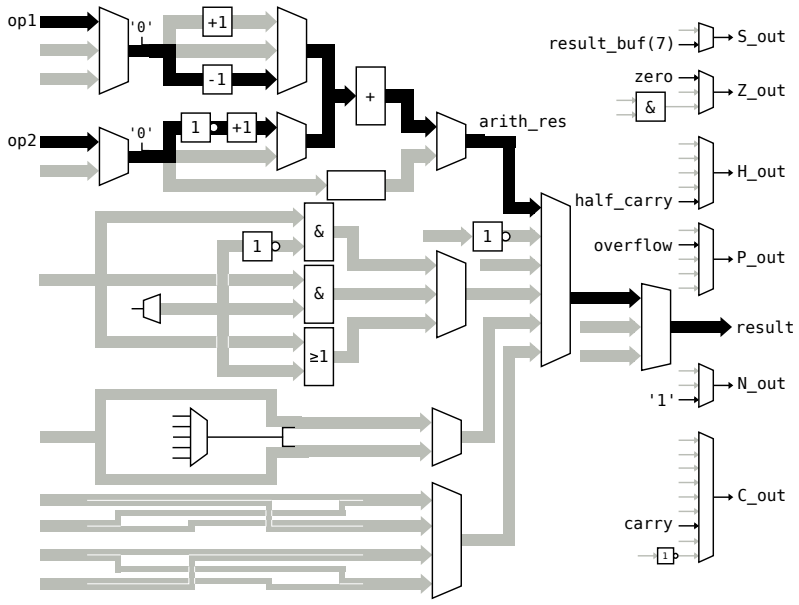
Flaggor

Flaggorna lagras i registerfilen men dess beräkning sker i ALU-sektionen. Vid en ALU-instruktion får ALU:n alltid in de tidigare flaggorna direkt från `F` i registerfilen. Utifrån de tidigare flaggorna och resultatet beräknas de nya flaggorna inom ALU:n. Under klockintervallet som ALU:n används läggs de nya flaggorna på indatan till `F` och lässignalen för `F` går aktiv. Alla flaggorna går dock inte direkt från ALU:n. `P`-flaggan använder inte alltid den beräknade `P`-flaggan som kommer från ALU:n. Vissa instruktioner sätter den till värden som ALU:n inte har någon kännedom om. Dessa är det nuvranande värdet av `IFF` och om adressindatan till registerfilen är noll. `IFF`-värdet används endast för instruktionerna `ld a,i` och `ld a,r`. Komparatorn används endast av blockinstruktionerna för att indikera om `BC`-1 är noll. De resterande instruktionerna använder `P`-flaggan som ALU:n har räknat ut.

ALU

ALU:n tar emot två 8-bitars operander, de nuvarande flaggorna och en instruktion — samt en bit-select som används av endast vissa instruktioner, de så kallade bitinstruktionerna. Med dessa indata avgörs därefter ett 8-bitars resultat och en ny uppsättning av flaggor under samma klockintervall. ALU:n är inte klockad och består endast av ett kombinatoriskt nät. Hela ALU:ns uppbyggnad visas i blockschema C.3. Gråskalan på bussarna indikerar vilken bitbredd de har; 7, 8 eller 9 bitar. Inom ALU:n refereras operanderna till `op1` och `op2`. `op1` kommer från `ACT` och `op2` kommer från `TMP`. De olika flaggornas fulla namn och dess användning beskrivs i sektion 3.1.2 på sidan 10.

ALU:n räknar ut flera delresultat och resultat men använder sig av instruktionen för att muxa vilka av dessa resultat som ska användas. En av muxarna använder dessutom sig av den nuvarande `C`-flaggan men de resterande muxarna använder endast instruktionen som adress.



Figur 18: Resultat och delresultat som muxas då `sbc`-instruktionen utförs och `C`-flaggan är satt.

Den översta delen i blockschemat visar beräkningen för aritmetiska instruktioner. Aritmetiska instruktioner inkluderar bland annat `add`, `sub`, `inc`, `sbc` (subtract with carry), `cp` (compare) och `daa` (decimal adjust after addition), Figur 18 visar exempelvis vilka delresultat som väljs ut när instruktionen `sbc` körs. `sbc`-instruktionen utför beräkningen `op1-op2-C` och skickar ut det som resultat. I exemplet är `C`-flaggan satt till 1 vid instruktionen start. För aritmetiska instruktioner använder sig ALU:n av en enda adderare men med olika möjliga operander. I exemplet kommer `op1` väljas och subtraheras med ett eftersom `C` är satt. Den första operanden till adderaren blir då `op1-1`. För den andra operanden väljs `op2` och dess negation eftersom `sbc` utför en subtraktion. Den andra operanden är då `-op2` och adderaren kommer ge `op1-1-op2` vilket är det slutgiltiga resultatet som därefter muxas till utdatan för resultatet.

En annan grupp av instruktioner är bitinstruktionerna. De hanteras under aritmetiken och skapar `bit_res` signalen. Denna grupp inkluderar `res`, `set` och `bit`-instruktioner. Dessa instruktioner nollställer en bit, sätter en bit, respektive testar en bit. Dessa instruktioner använder sig av `bit_select` som kommer direkt från instruktionen i `IR`. Ett exempel på en instruktion är `set 6,e` som sätter bit 6 i `E` till ett.

```
F3: set 6,e          1 1  1 1 0  0 1 1
                     set    6      e
```

För bitinstruktionerna skapas först en mask med hjälp av `bit_select`. I fallet då `bit_select` är 6 skapas masken `0100 0000`. Bit 6 är ett och de resterande bitarna är noll. För att sätta bit 6 i ett godtyckligt tal `OR`:as tal med masken. För att nollställa bit 6 `AND:as` talet med inversen av masken. För att testa bit 6 `AND`:as talet med masken och resultatet jämförs med noll.

Nästa grupp av instruktioner är rotations- och skiftinstruktionerna. Dessa är bland annat `rr` (rotate right), `sla` (shift left arithmetic) och `rlc` (rotate left with carry). Dessa skiftar talet antingen ett steg åt höger eller vänster. Den nya biten som tillkommer,

`edge`, skiljer får olika värden beroende på instruktionen. Vid rotation kan biten som kom ut från andra sidan användas, vid skiftning kan till exempel noll tillkomma. De olika instruktioner skiljer också med vilket värde som hamnar i den nya carry-flaggan. Alla dessa instruktioners funktion beskrivs i användarmanualen[13] i kapitlet som börjar på sidan 204.

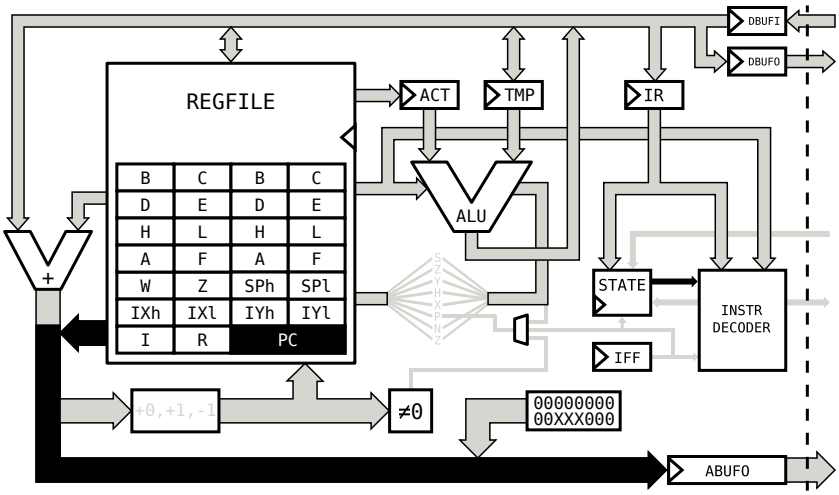
Den sista delen av ALU:n används för endast två instruktioner; `rld` och `rrd`. Dessa utför en rotation inom ett 16-bitars tal. Talet består av `A` som de högre bitarna och värdet som `HL` pekar på i minnet som de lägre bitarna. Instruktionerna roterar fyra bitar i tagen höger för `rrd` och vänster för `rld`. Eftersom resultatet är ett 16-bitars tal kan ALU:n inte räkna ut det under ett klockintervall. Istället räknas talet ut i två steg; först de lägre åtta bitarna och därefter de högre åtta bitarna.

Det finns en annan grupp av instruktioner som också använder ALU:n i två steg. Dessa är `add`, `adc` och `sbc` för 16-bitars register, till exempel `add ix, bc`. Dessa tar två 16-bitars tal och ger ett nytt 16-bitars tal. I fallet för `add` sker här först en vanlig `add`-instruktion för de lägre åtta bitarna av varje register. Under det andra steget körs istället en `adc`-instruktion för de högre åtta bitarna. `adc` använder carry:n från det första steget, på så sätt försvinner inget värde om det första steget ger ett tal större än 256.

4.1.4 Instruktionsförlopp

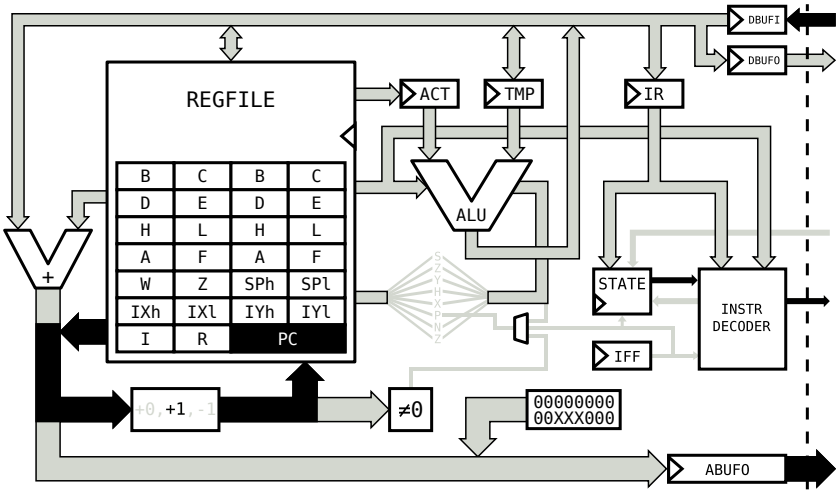
För att sammanfatta processorns implementation följer ett exempel av hela händelseförloppet för en instruktion. Instruktionen är `add a,d` och den adderar `A` med `D` och lagrar resultatet i `A`.

`add a,d:` `M1T1`



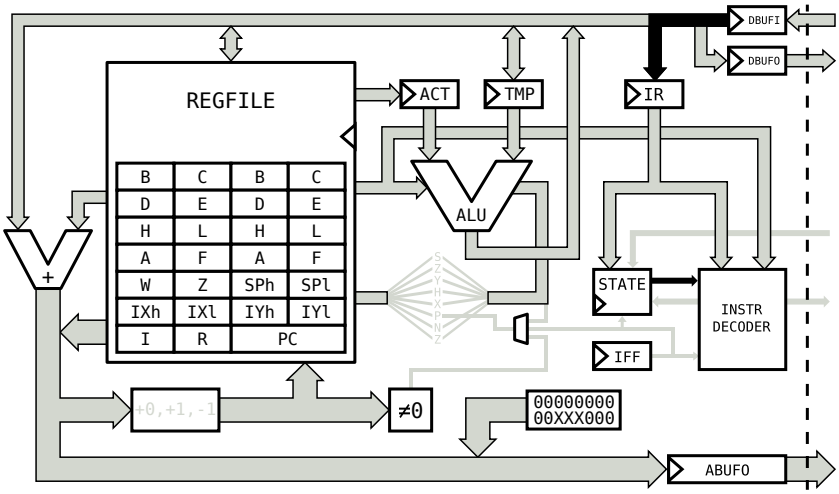
Processorn börjar på maskincykel 1 och T-state 1. Det första som måste ske är att instruktionen hämtas från minnet. Adressen till `PC` väljs i registerfilen. Registerfilens utdata muxas till adressbussen. När `PC` väl ligger på databussen kan det läsas till adressbuffern.

`add a,d:` `M1T2`



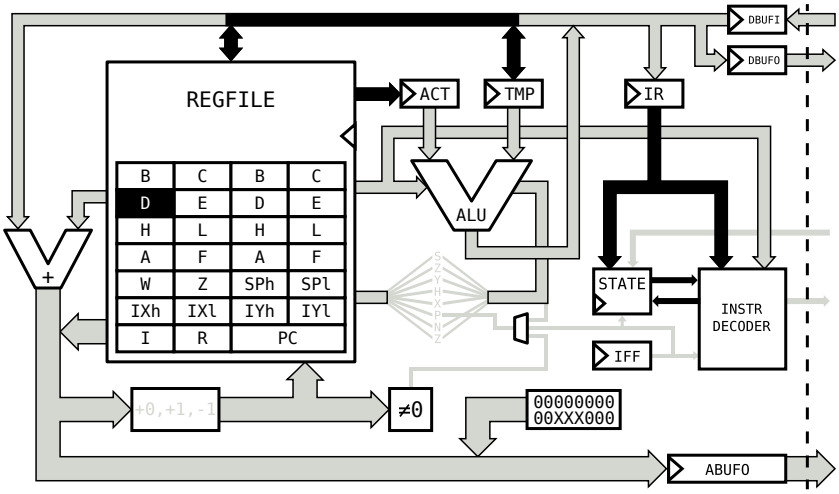
Inför T2 har adressbuffern laddat adressen och kan nu skicka den till minnet. I och med det skickas en lässignal så att instruktionen i minnet ligger på den externa databussen i slutet av klockintervallet. Därifrån läses den till processorns indatabuffer `dbufi`. Under samma klockintervall behålls `PC` på adressbussen för att inkrementeras så att PC nu pekar på nästa instruktion.

`add a,d:` `M1T3`



Nu återstår bara att läsa instruktionen från databuffern till `IR` där instruktionsdekodaren kan avläsa den.

`add a,d:` `M1T4`



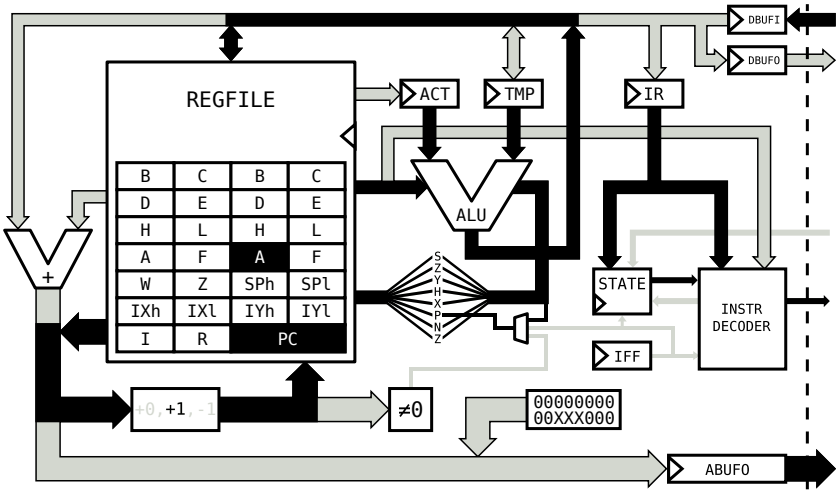
Först nu ligger objektkoden för `add a,d` i `IR` och processorn kan börja exekvera instruktionen. Det första som sker är att `TMP` laddas med `D` och `ACT` med `A`. `A` har en egen buss

och kan därmed läsas parallellt med ett annat värde som går via databussen. Instruktionsdekodaren signalerar nu att masknincykeln är färdig och att hämtningen av nästa instruktion kan påbörja. `add a,d`-instruktion är dock inte färdig riktigt ännu.

`add a,d`: M2T1

Precis som under `M1T1` läggs `PC` på adressbussen och laddas till adressbuffern. `PC` pekar dock nu på instruktionen som ligger på platsen efter `add a,d`.

`add a,d`: M2T2



Under tiden som instruktionshämtningen sker nere på adressbussen kan `add a,d`-instruktionen avslutas. Nu beräknar ALU:n `A+B` och lägger det på databussen. Därifrån läses det av registerfilen till `A`. Registerfilen läser även in de nya flaggorna från en dedikerad buss. Under tiden som ALU:n beräknar laddas den nya instruktionen till `dbufi`.

`add a,d`: M2T3

Precis som under `M1T3` läses den nya instruktionen från databuffern in till `IR`. `add a,d` är härmed avslutad och under nästa klockintervall kommer nästa instruktion att börja exekvera.

4.2 TI-ASIC

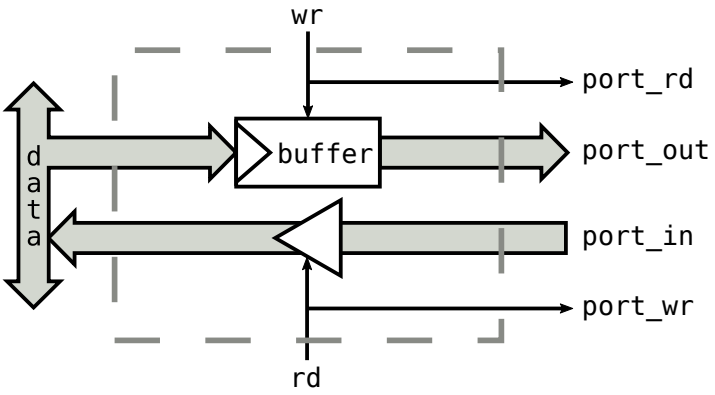
TI-ASIC:en innehåller alla de kontrollkomponenter som är del av TI:s miniräknare utanför processorn. In till ASIC:en är databussen direkt kopplad från processorn. Med hjälp av in/out instruktioner och adresser kan datan muxas till eller från rätt port. De olika portarna leder därefter till en eller flera olika komponenter. Blockschema C.4 ger en överblick av denna struktur samt vilka komponenter som hanterar vilken in- och utdata.

4.2.1 ASIC/PIO

ASIC/PIO:ns uppgift är att delegera databussen samt läs- och skrivsignalerna till rätt port. Data som går till portarna hanteras annorlunda från data som kommer från portarna. Data som kommer från portarna läses alltid från porten men läggs endast på databussen när den specifika porten ska läsas ifrån. Å andra sidan skrivs alltid datan från en buffer till porten men buffern läser endast från databussen när en skrivning till porten sker. Figur 19 visar hur varje port är uppbyggd. Läs och skrivsignalerna går endast vidare till den port som är vald med de fem lägsta bitarna av adressbussen. Lässignalen `rd` aktiveras vid en `in`-instruktion och `wr` aktiveras vid en `out`-instruktion. Åtta av tio egentliga portar för TI-83p har implementerats. De portar som saknas hanterar länkporten, exekveringsmask samt skrivskydd för ROM.

In- och utsignalerna går vidare till de komponenter som hanterar porten, som kan ses i blockschema C.4. Flera komponenter kan ta emot data från en port vid en `out`-instruktion men endast en komponent skriver tillbaka data vid en `in`-instruktion. Läs och skrivsignalerna skickas även till komponenterna eftersom vissa reagerar på läsningar och skrivningar. Vissa portar skickar direkt tillbaka datan från buffern vid en läsning och behöver inte ta emot indata från en annan komponent.

I praktiken speglar TI-miniräknaren flera portadresser till en och samma port. Det här löses genom att aktivera lässignalen och skrivsignalen till "originalporten" om någon av spegelbilderna ligger på adressbussen. Eftersom fem bitar väljer port kan det finnas upp till 32 portar. Egentligen finns endast tio portar men nästan alla 32 adresser pekar till någon av dessa.



Figur 19: Implementation av en port.

4.2.2 Minneskontroller

TI-83p använder sig av två separata minnen men den här implementation använder endast ett stort kontinuerligt minne. ROM har lagts som de första 512 KiB och RAM har lagts som de 32 KiB direkt efter ROM. `RAM 1` ligger dock före `RAM 0`. Den fysiska placeringen kan ses i figur 20.

startadress	page	slutadress
00000	ROM 00	03FFF
04000	ROM 01	07FFF
⋮	⋮	⋮
78000	ROM 1E	7BFFF
7C000	ROM 1F	7FFFF
80000	RAM 1	83FFF
84000	RAM 0	07FFF
88000	oanvänd	8BFFF
⋮	⋮	⋮

Figur 20: Placering av pages i det fysiska minnet.

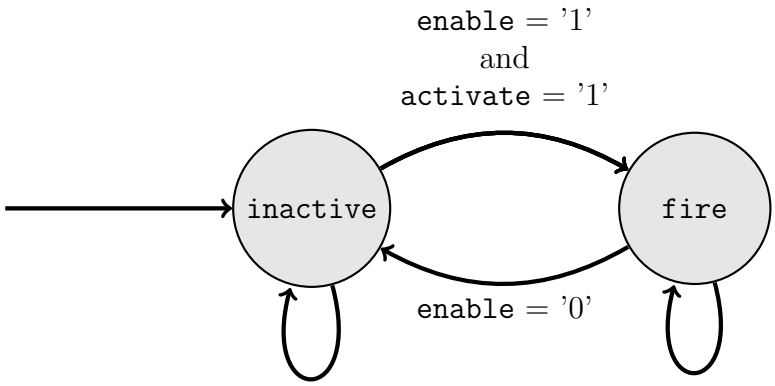
Minneskontrollerns uppgift är att översätta den virtuella adressen från processorn till den fysiska adressen utefter läget samt page A, B och därefter skicka den till det fysiska minnet. Implementationen fungerar genom att först bestämma A och B utefter den nuvarande buffern till port `06` och `07` och därefter muxa de till de fyra olika platserna utefter det nuvarande läget. Därefter muxas en av dessa fyra pages utefter de högsta bitarna i den virtuella adressen. Sedan läggs bitarna i page:n ihop med de låga bitarna från den virtuella adressen för att få den exakta fysiska adressen.

4.2.3 Avbrott

Konstruktionen använder sig av avbrott från tre källor; hårdvarutimer 1, hårdvarutimer 2 och ON-knappen. TI-83p har även ett avbrott från länkporten men denna är ej implementerad. TI-ASIC:en har en avbrottshanterare som tar emot signaler från ON-knappen och ASIC:ens hårdvarutimers och därefter avgör om `int`-signalen ska gå aktiv utefter avbrottsmasken.

Signalen från ON-knappen kommer direkt från tangentbordskodaren. Om knappen är nedtryckt är signalen låg, annars är den hög. Signalerna för timers:en genereras i en egen komponent. Komponenten består av två räknare som alltid räknar ner. När en räknare kommer till noll skickas en puls till avbrottshanteraren och räknaren laddas om med sitt startvärde. Startvärdet beror på vilken frekvens som är inställd och frekvensen på räknaren som är 33 MHz. Vid start ska timer 1 ha frekvensen 118 Hz, då måste startvärdet vara $\frac{33\,\text{MHz}}{118\,\text{Hz}} = 282486$. Då kommer räknaren skicka en puls 118 gånger per sekund. Frekvensen på timers:en går att välja med genom port `04`.

Varje avbrottskälla hanteras individuellt i avbrottshanteraren. Det finns två signaler som avgör om ett avbrott ska avfyras; `enable` och `activate`. `enable` är biten för av-



Figur 21: Tillståndsdiagram för avbrottshantering av en enskild källa.

brottskällan i avbrottsmasken i port 03. activate är en signal från själva avbrottskällan, till exempel pulsen från en timer. Om activate går hög måste enable vara hög för att avbrottet ska avfyras. När avbrottet väl ha avfyrats kommer det vara aktivt till att biten i masken nollställs. Port 04 visar vilka avbrott som är aktiva. Det räcker med att ett avbrott är aktivt för att int ska gå hög. De tre källorna OR:as. Programmeraren kan först kolla vilket avbrott som är källan och därefter inaktivera just det avbrottet innan avbrott för processorn sätts på igen med ei-instruktionen.

4.2.4 LCD-kontroller

Implementationen av LCD-kontrollern består av räknare och register för X, Y och Z samt ett block-RAM på 7680 bitar. Tre D-vippor används även för att hålla koll på läget (wl) och pekarens rörelseinställningar; upp/ner (up) för X/Y (counter. Det finns även ett register som lagrar bildminnets utdata för läsning från processorn.

X består av en upp/ner räknare på sex bitar. Den går upp till 63 och sedan går runt till noll igen vid uppräkning. Likaså går den från noll till 63 vid nedräkning. Dess CE-ingång är kopplad till läs/skriv-signalen från port 11 AND:ad med D-vippan som väljer mellan X och Y. På så sätt räknar X endast om X är vald samt att porten skrivs eller läses ifrån. U/D-ingången som väljer riktning är direkt kopplad till D-vippan som väljer riktning. Räknaren har även en indata port som kommer från utdatan till port 10. Om port 10 skrivs till med de två högsta bitarna som "10"kommer räknaren laddas med de 6 lägre bitarna.

Y består av en räknare på fem bitar och fungerar på liknande sätt som räknaren för X. Y ska dock räkna olika högt beroende på vilket läge som är valt. Vid 6-bitarsläge ska Y loopa mellan 0-19 och vid 8-bitarsläge mellan 0-14. Här måste dataingången och CLR-ingången tillsammans med komparatorer användas för att åstadkomma loopning vid randerna.

Bildminnet är ett block-RAM med två 1-bitars dataportar, en för läsning/skrivning av processorn och en för läsning av VGA-motorn. Bildminnet går på systemklockan som ligger på 100 MHz. VGA-motorn läser en pixel i taget, den skickar rad och kolumn för en pixel och förväntar sig dess färg samma klockintervall. Det är möjligt endast på grund av att VGA-motorn går på 25 MHz. Vid läsning ger bildminnet pixeln ett klockintervall

senare men eftersom bildminnet går på systemklockan hinner rätt värde nå VGA-motorn i god tid.

Från processorns sida är tidsrestriktionen ännu mer avslappnad. Processorn går som högst på 14 MHz och intervallet mellan två läsningar eller skrivningar kan som minst vara 10 T-states. Dock måste flera läsningar göras eftersom antingen ska sex eller åtta bitar läsas. För läsning av processorn läser LCD-kontrollern alltid från bildminnet, en bit i taget. När den kommer till den sista biten börjar den om på den första. Bitarna lagras i en 8-bitars buffer på rätt plats. Beroende på läget läses sex eller åtta bitar. Den här buffern är dock inte registret som processorn läser ifrån. När processorn läser förväntar den sig värdet där pekaren pekade på sist en läsning gjordes. Processorn läser värdet från registret och i och med läsningen antar registret det nuvarande värdet från den buffern.

Skrivning fungerar på liknande sätt. Då stannas läsningen och dataporten används för skrivning istället. Kontrollern får in en page via port `11` och sätter igång en räknare. Bit för bit skrivs page:n till bildminnet. När räknaren har nått bit 5 eller bit 7 är den färdig och dataporten överlåts åter för läsning.

7C000	AA
7C001	BB
7C002	CC
7C003	DD
7C004	EE
7C005	FF

(a) Miniräknaren.

3E000	BB	AA
3E001	DD	CC
3E002	FF	EE

(b) Det externa minnet.

Figur 22: Två olika synvinklar för samma block av minne.

4.3 Externa kontroller

4.3.1 Minnesgränssnitt

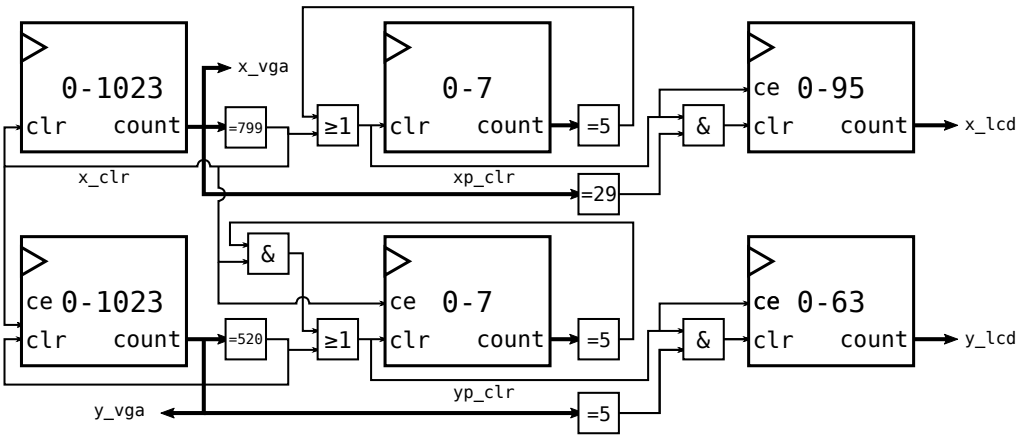
Minnesgränssnittet ansvarar för att översätta TI-ASIC:ens läs- och skrivsignaler till det externa minnets kontrollsignaler. Minneskontrollern i TI-ASIC:en skickar endast en läs- eller skrivsignal till gränssnittet. Det externa minnet har däremot fler kontrollsignaler som bland annat `chip_enable`, `output_enable` och `write_enable`. Alla kontrollsignaler för minnet kan ses i dess manual.[1]

Gränssnittet utför även en annan viktig uppgift. Databussen till minnet är 16 bitar och minnesadressen pekar ut 16-bitars platser. Z80-processorn lagrar förstås endast åtta bitar i taget eftersom dess databuss är åtta bitar. Ett alternativ är att endast använda en av bytes:en i varje plats eftersom det finns tillräckligt med utrymme. Ett problem med detta är att mjukvaran Adept skriver och läser till båda bytes och det leder till att varannan byte missas när processorn läser. Istället ser gränssnittet till att adressen halveras och datan skrivs eller läses ifrån antingen övre eller undre delen av platsen i minnet. Vid skrivning används `upper_byte`- och `lower_byte`-signalerna för att förhindra att den andra byte:n i platsen skrivs över. Figur 22 visar de två olika uppläggen av minnet som gränssnittet måste översätta emellan.

4.3.2 Tangentbordskodare

Tangentbordskodaren har i uppgift att översätta knapptrycken på PS/2-tangentbordet till miniräknarens format som beskrivs i sektion 3.2.2 på sidan 14. Vilken knapp på PS/2 tangentbordet som motsvarar vilken på miniräknaren visas i tabell B.

Tangentbordet skickar endast signaler vid nedtryckning och uppsläpp av tangenter. Därför måste tangentbordskodaren lagra en matris av vippor som håller koll på varje tangents nuvarande status. När en knapp trycks ner sätts motsvarande vippa till noll och när en knappen släpps sätts vippan till ett igen. Denna matris är direkt kopplad till tangentbordskontrollern i TI-ASIC:en som AND:ar de grupper som programmeraren har aktiverat.



Figur 23: Räknarna inom VGA-motorn.

4.3.3 VGA-motor

VGA-motorns primära uppgift är att visa upp miniräknarens bildminne på VGA-skärmen. VGA-motorn skickar färgen av en pixel i taget varje klockintervall. Den börjar från det övre högra hörnet och går rad för rad neråt. När pekaren når det nedre högra hörnet har det gått en sextiodels sekund och den börjar om för att rita nästa bild. För att hålla koll på vilken pixel på skärmen den är på används två räknare används, en för raden (`y_vga`) och en för kolumnen (`x_vga`). Utifrån pekarens värde bestäms även hsync och vsync-signaler. Mer information om hur VGA-standarden fungerar hittas i Nexys3-manualen.[8]

Eftersom VGA-skärmen är 640×480 och miniräknarens LCD-skärm är 96×64 pixlarna i bildminnet som störst vara 6×6 pixlar för att hela LCD-skärmen ska rymma. Om pixelbredden var en potens av två skulle det gå att avgöra vilken pixel från bildminnet som skulle ritas ut utifrån de högre bitarna i `x_vga` och `y_vga`. Istället har fyra ytterligare räknare använts. Två av räknarna (0-7) håller koll på vilken rad och kolumn pekaren är på inom varje LCD-pixel, de andra två (0-95, 0-63) håller koll på vilken LCD-pixel pekaren är på. Bildminnesräknarna räknar på pixelräknarnas `clr`-signal och räknar därmed endast upp var sjätte pixel på VGA-skärmen.

Figur 24: VGA-skärmens upplägg av bildminne och display för debugging samt blanksignal utanför bilden.

Bildens position på skärmen ligger vågrätt centrerad sex pixlar från övre kant. För att bildminnesräknarna ska hamna i fas nollställs `x_lcd` då `x_vga` är 29 och `y_lcd` när `y_vga` är 5. För att pixelräknarna ska hamna i fas nollställs de vid `x_clr` och `y_clr`.

VGA-motorn visar även upp en display för debugging. Då skickas `x_vga` och `y_vga` till en komponent som har tillgång till alla debug-signaler som ska visas. Med hjälp av ett minne av karaktärer bestäms om pixeln ska vara på eller av utifrån pekarens värde.

5 Slutsats

Projektets mål uppnåddes; en replikation av en Z80-processor har byggdes och användes för att bygga en TI-miniräknare.

5.1 Slutmål

Slutmålet var att bygga en TI-84 Plus men en TI-83p visade sig vara väldigt lik uti-från men ha färre saker som behövde implementeras inuti. TI-84p har till exempel en dedikerad komponent för att räkna ut MD5-hashes som TI-83p hanterar med mjukvara. TI-84p har även en andra uppsättning av timers som går efter en kristall och kan justeras mer utförligt av programmeraren. Jämfört med TI-83p har TI-84p dessutom ett väldigt stort antal portar som gruppen var osäkra på vilka som var nödvändiga och om några kunde anta konstanta värden. I efterhand verkar beslutet att istället göra en TI-83p bra eftersom det sparade mycket tid. Med den tid som projektet gick var det dessutom svårt att hinna med och få TI-83p att fungera helt.

5.2 Tillvägagång

I början av projektet skedde allt arbete med simulering. Det första som antogs var processorns ALU och registerfil för att därefter kunna testa utföra enkla instruktioner. Mycket tid gick åt till att göra ALU:n i början, med tiden blev den enklare och enklare. Något som har varit till oerhört stor nytta vid designen av ALU:n är enhetstester. För varje instruktion som implementerades skrevs ett antal enhetstester i en testbänk efter manualen och emulatorernas beteende. På så sätt kunde ALU:n justeras till att testerna gick igenom och instruktionen var rätt implementerad. Senare vid varje ändring kördes alla tester och om något gick fel visade testet vilken instruktion, med vilka operander och vad som gick fel, resultatet eller någon flagga. Utan dessa test hade det varit väldigt lätt att misstag vid ändringar gick oupptäckta tills långt senare.

När ALU:n och registerfilen fungerade påbörjades instruktionsdekodaren och de andra delarna av processorn. Ett simulerat minne byggdes också för att lagra program. Då kunde processorn börja köra enkla program som lästes från minnet. Därefter utökades processorn med fler instruktioner och testning med FPGA:n påbörjades. I det här stadiet var enhetstester också oerhört användbara. Nu skrevs tester med Z80-assembly kördes på processorn. Vid en implementation av en instruktion skrevs ett program i en textfil som använde instruktionen och testade resultatet. Därefter användes ett skript för att assembla programmet, simulera projektet med GHDL och ladda programmet från filen till det simulerade minnet. Då kunde vågorna från simulationen ses med hjälp av GTKWave. Programmen som skrevs utförde flera tester i rad, om något test misslyckades hoppade programmet till en rutin som lagrade en felkod i ackumulatorn och halt:ade processorn. Om alla test gick igenom lagrades en annan kod och processorn halt:ade. På så sätt kunde ett stort test köras för att testa flera instruktioner i taget för att snabbt hitta misstag vid ändringar.

Det här var smidigt för att testa fel på individuella instruktioner, men det var svårt att använda för att felsöka problem som uppstod när operativsystemet och andra program så småningom testades. Dessa kör miljontals instruktioner och använder mycket loopar vilket inte går att simulera inom en rimlig tid. Dessa var tvungna att köras på FGPA:n. För att kunna felsöka program på FPGA:n skapades en display på skärmen som visade värdet av alla register. Dessa gör inte så stor nytta när processorn kör i 6 MHz så brytpunkter och instruktionsstegning implementerades också. På så sätt gick det relativt fort att stega igenom programmet tills något gick fel. Fel hittades genom att jämföra resultatet med en emulator. När man gick för långt kunde man spara adressen, sätta en brytpunkt och starta om. Det här ledde till att rätt många fel med processorn upptäcktes och fixades. Vissa problem uppstod dock rätt långt in i programmet och var inte alls lätta att hitta snabbt på det här sättet.

Vid ett tillfälle var det tydligt att något gick fel eftersom processorn hamnade på en adress som emulatorn aldrig kom till. Då implementerades ett sätt att spåra alla hopp som utförs av processorn. Spåraren aktiverades och det felaktiga programmet kördes. Processorn kom till fel ställe och nådde en `halt`. Då sparades alla hopp till en fil och med binärsökning kunde det exakta stället där processorn kommit fel hittas snabbt. Denna metod gjorde att felsökningen gick oerhört fort i slutet av projektet och alla kvarstående fel löstes under ett par dagar.

5.3 Utökningar

Många av TI-miniräknarna använder sig av Z80-processorn men med små skillnader på räknarna som också skulle kunna implementeras. De komponenter som är gemensamma kan separareras från de komponenter som skiljer. För varje miniräknare kan dessa komponenter implementeras. En av dessa miniräknare som kan implementeras är förstås TI-84p.

Z80-processorn skulle dessutom kunna användas för att bygga andra datorer som inte är TI-miniräknare. Processorn är fullt kompatibel med Intel 8080. Datorer som använder den skulle också kunna byggas. Exempelvis skulle en dator som kan köra CP/M-operativsystemet kunna byggas.

En annan möjlig utökning är att emulera miniräknarens LCD-skärm bättre. Vår VGA-skärm har nu endast två olika färger men LCD-skärmens pixlar har olika intensitet. När en pixel aktiveras går den kontinuerligt från släckt till tänd och inte instant som i vår implementation. Det här används av många spel för att skapa gråskala genom att snabbt sätta på och stänga av pixlar. Det här skulle kunna implementeras genom att skapa ett andra bildminne som lagrar varje pixels intensitet. Med en viss frekvens kan pixlars intensitet inkrementeras eller dekrementeras beroende på om pixeln håller på att släckas eller tändas.

Referenser

[1] *Async/Page/Burst CellularRAM Memory*. MT45W1MW16BDGB. Micron. 2005.

[2] *Category:83Plus:Ports - WikiTI*. URL: http://wikiti.brandonw.net/index.php?title=Category:83Plus:Ports.

[3] ClrHome. *Z80 instruction set*. URL: http://clrhome.org/table/ (hämtad 2018-05-14).

[4] *COLUMN AND ROW DRIVER LSI FOR A DOT MATRIX GRAPHIC LCD*. T6A04. TOSHIBA. Mars 2001.

[5] Dr. D'nar. *83Plus:Memory Mapping - WikiTI*. 2013. URL: http://wikiti.brandonw.net/index.php?title=83Plus:Memory_Mapping (hämtad 2018-06-07).

[6] *GHDL*. URL: https://github.com/ghdl/ghdl.

[7] *GTKWave*. URL: http://gtkwave.sourceforge.net/.

[8] *Nexys3 Board Reference Manual*. Digilent, Inc. Nov. 2011.

[9] Ken Shirriff. *Down to the silicon: how the Z80's registers are implemented*. 2014. URL: http://www.righto.com/2014/10/how-z80s-registers-are-implemented-down.html.

[10] KnightOS Team. *z80e*. URL: https://github.com/KnightOS/z80e.

[11] *ticalc.org*. URL: https://www.ticalc.org/pub/83plus.

[12] *Tilem2*. URL: http://lpg.tTcalc.org/prj_tilem.

[13] *User Manual*. Z80 CPU. Rev. 11. Zilog, Inc. 2016.

[14] Brandon Wilson. *WikiTI*. URL: http://wikiti.brandonw.net.

[15] *Z80 Heaven*. URL: http://z80-heaven.wikidot.com/ (hämtad 2018-05-14).

[16] *z80asm*. URL: http://www.nongnu.org/z80asm.

[17] Rodnay Zaks. *Programming the Z80*. SYBEX Inc, 1981.

A Bilaga. VHDL-kod

A.1 Filstruktur

Källkodens filer är ungefär strukturerade utefter komponenternas hierarki i konstruktionen. Högst upp är filen `comp.vhd` som innehåller huvudkomponenten, den som laddas på FPGA:n. Den här filen hämtar komponenter från de sex delmapparna; `dbg`, `ext`, `pkg`, `prm`, `ti` och `z80`. `dbg` innehåller alla komponenter som används för debugging. `ext` innehåller alla externa kontroller. `pkg` innehåller VHDL-paket för att deklarera typer och konstanter som används av flera olika komponenter. `prm` innehåller allmänna primitiva komponenter som kan användas var som helst i projektet, såsom register och räknare. `ti` innehåller TI-ASIC:en och alla dess komponenter, `z80` likaså för processorn.

A.2 Instruktionsdekodare

Hur instruktionsdekodaren är implementerad har inte diskuterats så mycket i hårdvarusektionen. Det är för att instruktionsdekodaren syntetiseras endast till en stor uppslagstabell av kontrollord. Kontrollordet är dock rätt stort och det finns många tillstånd för processorn vilket ger en väldigt stor uppslagstabell. Det skulle gå att manuellt skriva in värdet för varje signal i kontrollordet för varenda möjliga tillstånd hos processorn. Det är dock inte praktiskt och inte heller enkelt att felsöka eller att anpassa om förändringar på processorn görs.

VHDL:s `process`-sats har istället använts för att skapa det här stora kombinatoriska nätet. I början av processen sätts alla kontrollsignaler till deras neutrala värden. Därefter skrivs de över sekvensiellt inom processen. Då behövs endast de signaler som instruktionen använder ändras. Instruktioner är implementerade som funktioner som tar emot en ram av alla kontrollsignaler och returnerar en ny ram med modifierade kontrollsignaler. Den nya ramen skriver därefter över den tidigare ramen. På det här viset kan varje instruktion beskrivas och därefter kan syntesverktyget bygga ett kombinatoriskt nät utefter ramens värde i slutet av processen.

Ett exempel på en hur en funktion skrivs visas i figur 25. Alla instruktioner som laddar ett 8-bitars register till ett annat 8-bitars register använder den här funktionen. Utifrån funktionen är det lätt att följa instruktionens förlopp. Under M1T4 laddas källregistrets värde till `TMP`. Under M1T5 laddas `TMP` till destinationsregistret. Därefter är instruktionen färdig.

En annan viktig uppgift av instruktionsdekodaren är att ta reda på vilken instruktion som ska utföras. Det här åstadkoms genom att titta på en uppdelning av den nuvarande instruktionen i `IR`. Se en av instruktionenerna som använder ovan funktion; `ld c,h`:

Här har instruktionen delats upp i tre delar; `x`, `y` och `z`. `x` berättar att det är en `ld r,r`-instruktion, `y` berättar att registret `c` är destinationen och `z` berättar att `h` är källan. Vi

```
function ld_r_r(state : id_state_t; f_in : id_frame_t;
                dst, src : std_logic_vector(4 downto 0))
return id_frame_t is variable f : id_frame_t; begin
    f := f_in;
    case state.m is
    when m1 =>
        case state.t is
        when t4 =>
            f.cw.rf_daddr := src;       -- select src reg
            f.cw.dbus_src := rf_o;      -- place regfile output on databus
            f.cw.tmp_rd := '1';         -- read from databus to tmp
        when t5 =>
            f.cw.rf_daddr := dst;       -- select dst reg
            f.cw.dbus_src := tmp_o;     -- place tmp output on databus
            f.cw.rf_rdd := '1';         -- read from databus to selected reg
            f.ct.cycle_end := '1';      -- end machine cycle
            f.ct.instr_end := '1';      -- end instr, begin with next
        when others => null; end case;
    when others => null; end case;
    return f;
end ld_r_r;
```

Figur 25: Funktionen för alla `ld r,r`-instruktioner.

vet då att vi ska utföra funktionsutropet

```
f := ld_r_r(state, f, r(y), r(z));
```

där `r` är en tabell av adresser till 8-bitars register. En annan användbar uppdelning syns i bland annat `adc` och `sbc` för 16-bitars register:

Här har `y`-variabeln delats upp i två variabler `p` och `q`. I det här fallet pekar `p` ut ett register och `q` avgör om det är en `adc`- eller `sbc`-instruktion. Dessa fem variabler tillsammans med prefixet används för att avgöra exakt vilken instruktion som ska exekveras.

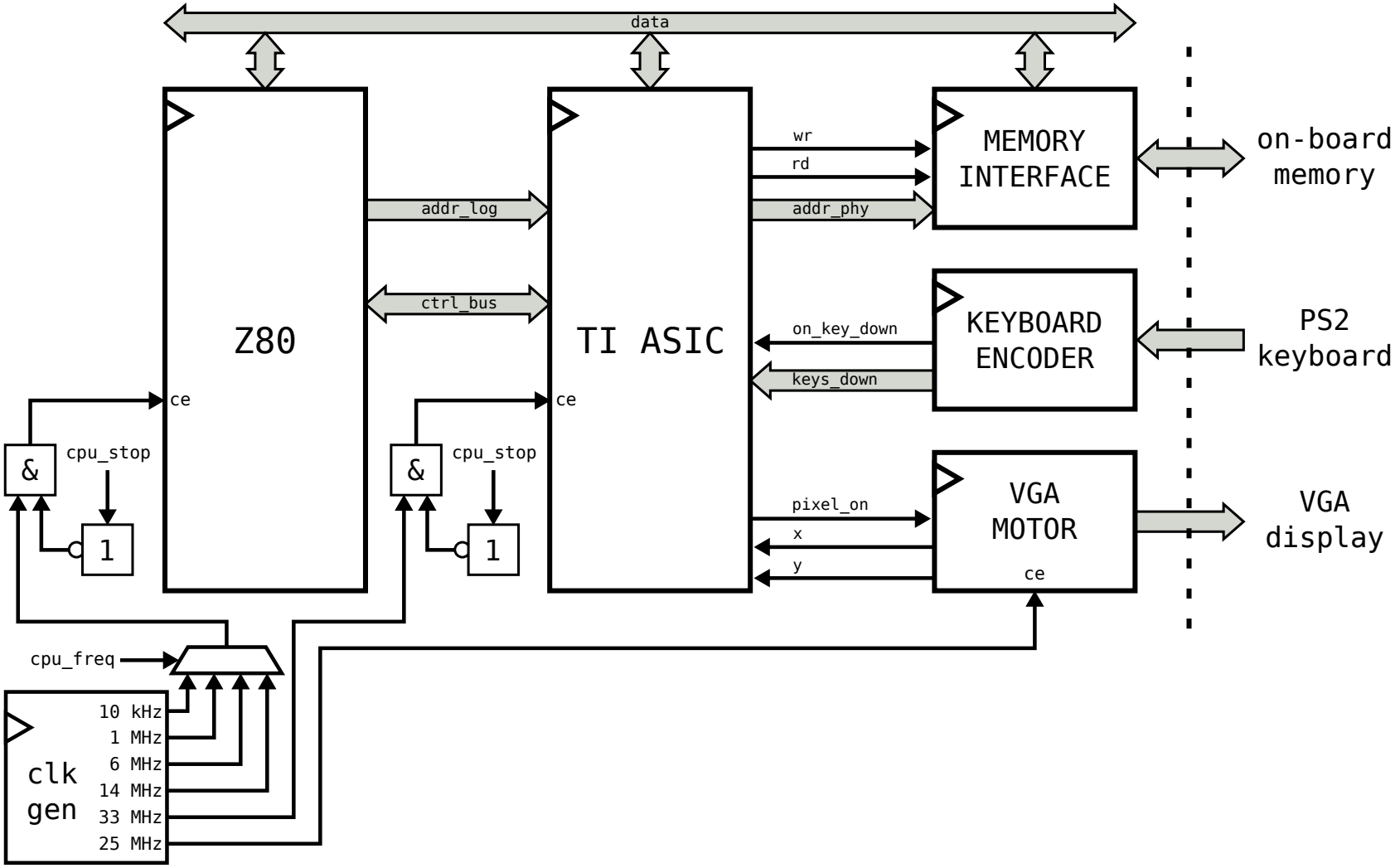
B Bilaga. Tangentbordslayout

TI-83p	alpha	2nd	PS/2
GRAPH	F5	TABLE	F5
TRACE	F4	CALC	F4
ZOOM	F3	FORMAT	F3
WINDOW	F2	TBLSET	F2
Y=	F1	STAT PLOT	F1
2ND			TAB
MODE		QUIT	1
DEL		INS	2
STO	X	RCL	X
LN	S	e^x	S
LOG	N	10^x	N
x^2	I	$\sqrt{}$	I
x^{-1}	D	MATRIX	D
MATH	A	TEST	N
ALPHA		A-LOCK	Caps Lock
0	SPACE	CATALOG	KP0
1	Y	L1	KP1
4	T	L4	KP4
7	O	u	KP7
,	J	EE	J
SIN	E	SIN^{-1}	E
APPS	B	ANGLE	B
X,T,θ, n		LINK	3

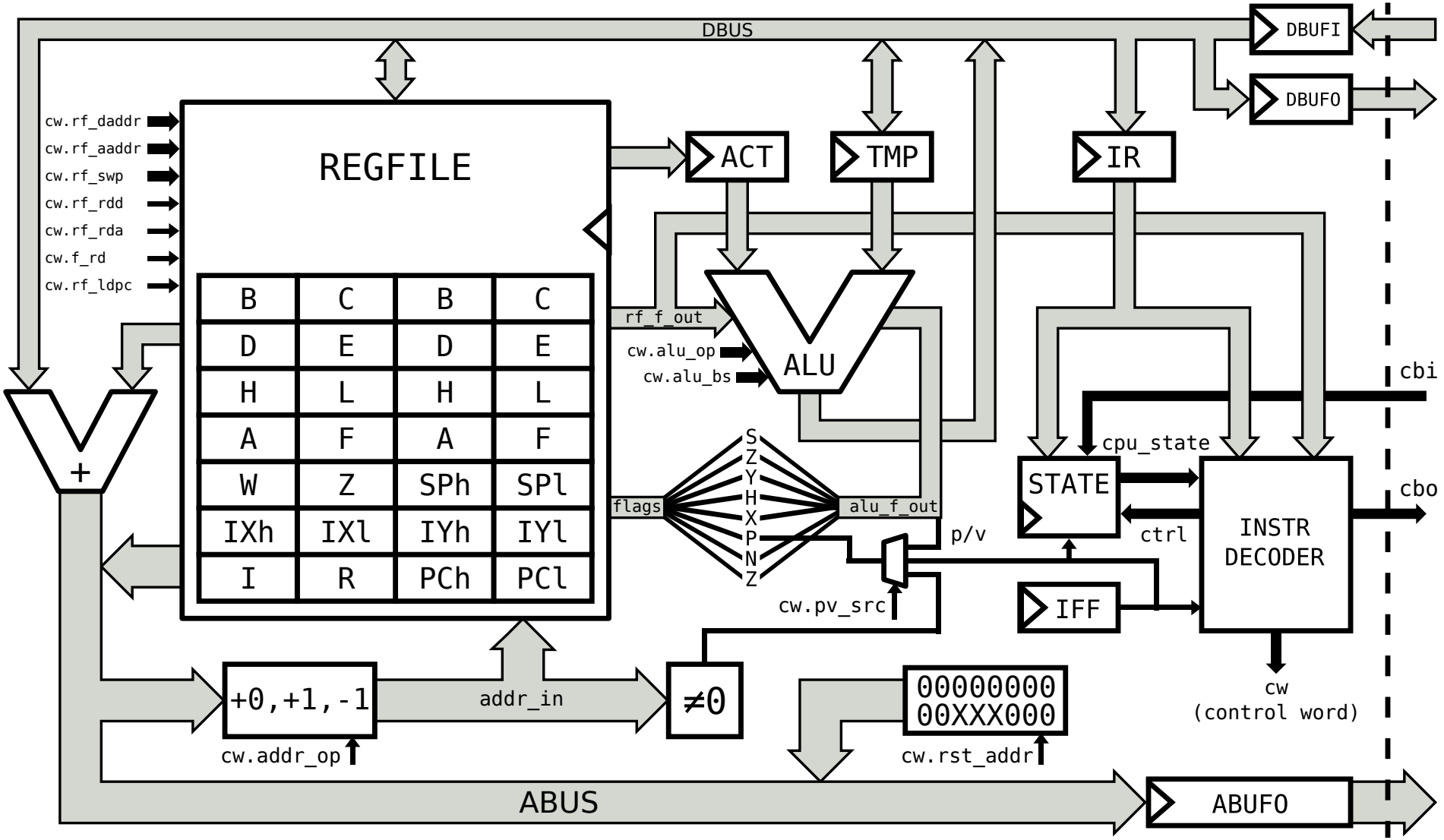
TI-83p	alpha	2nd	PS/2
.	:	i	KP_DEL
2	Z	L2	KP2
5	U	L5	KP5
8	P	v	KP8
(	K	{	9
COS	F	COS^{-1}	F
PRGM	C	DRAW	F
STAT		LIST	4
(-)	?	ANS	-_
3	Θ	L3	KP3
6	V	L6	KP6
9	Q	w	KP9
)	L	}	9
TAN	G	TAN^{-1}	N
VARS		DISTR	5
ENTER	SOLVE	ENTRY	Enter
+	"	MEM	KP_PLUS
−	W	]	KP_MINUS
×	R	[	KP_MULTI
÷	M	e	KP_DIV
∧	H	π	^~
CLEAR			Num Lock
DOWN			End
LEFT			Delete
RIGHT			PgDn
UP			HOME
ON		OFF	Ctrl

C Bilaga. Blockscheman

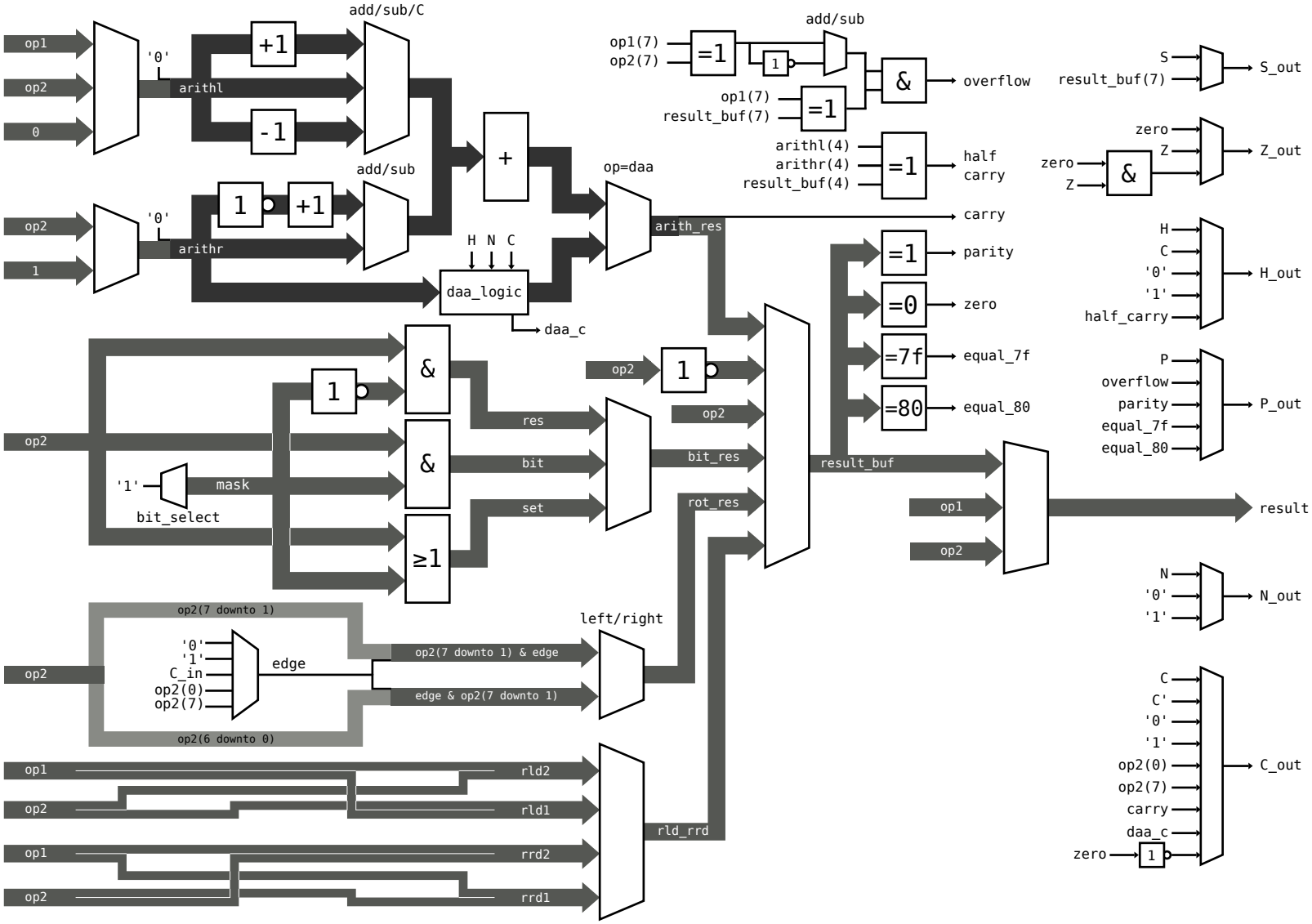
C.1 Huvudhierarki



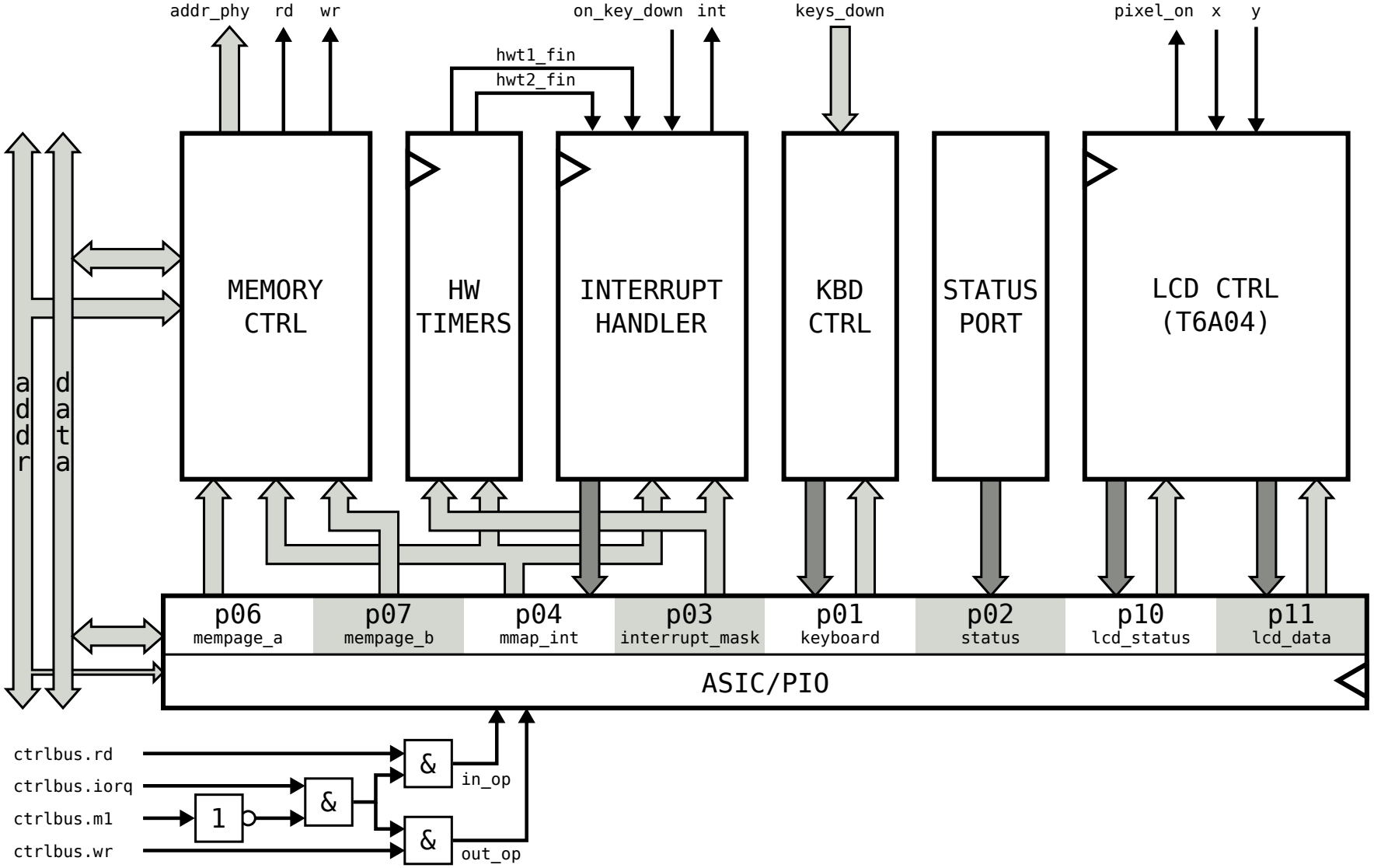
C.2 Z80


DBUS
DBUFI
DBUFO
REGFILE
cw.rf_daddr
cw.rf_aaddr
cw.rf_swp
cw.rf_rdd
cw.rf_rda
cw.f_rd
cw.rf_ldpc
B C B C
D E D E
H L H L
A F A F
W Z SPh SPl
IXh IXl IYh IYl
I R PCh PCl
ACT
TMP
IR
rf_f_out
cw.alu_op
cw.alu_bs
ALU
+
S
Z
Y
H
X
P
N
Z
flags
alu_f_out
p/v
cw.pv_src
cbi
cpu_state
STATE
ctrl
INSTR
DECODER
cbo
IFF
cw
(control word)
+0,+1,-1
addr_in
≠0
00000000
00XXX000
cw.addr_op
cw.rst_addr
ABUS
ABUFO

C.3 ALU


op1
op2
0
'0'
arithl
+1
-1
add/sub/C
1
+1
add/sub
'0'
arithr
op2
1
+
op=daa
H N C
daa_logic
daa_c
arith_res
carry
op1(7)
op2(7)
=1
1
add/sub
&
overflow
op1(7)
result_buf(7)
=1
arithl(4)
arithr(4)
result_buf(4)
=1
half
carry
S
result_buf(7)
S_out
zero
Z
Z_out
zero
Z
&
H
C
'0'
'1'
half_carry
H_out
P
overflow
parity
equal_7f
equal_80
P_out
=1
parity
=0
zero
=7f
equal_7f
=80
equal_80
op2
1
op2
bit_res
result_buf
op1
op2
result
&
1
op2
res
&
bit
'1'
mask
bit_select
set
≥1
rot_res
op2(7 downto 1)
'0'
'1'
C_in
op2(0)
op2(7)
edge
op2
op2(7 downto 1) & edge
edge & op2(7 downto 1)
op2(6 downto 0)
left/right
op1
op2
rld2
rld1
op1
op2
rrd2
rrd1
rld_rrd
N
'0'
'1'
N_out
C
C'
'0'
'1'
op2(0)
op2(7)
carry
daa_c
zero
1
C_out

C.4 TI-ASIC


addr_phy
rd
wr
on_key_down
int
keys_down
pixel_on
x
y
hwt1_fin
hwt2_fin
MEMORY CTRL
HW TIMERS
INTERRUPT HANDLER
KBD CTRL
STATUS PORT
LCD CTRL (T6A04)
addr
data
p06 mempage_a
p07 mempage_b
p04 mmap_int
p03 interrupt_mask
p01 keyboard
p02 status
p10 lcd_status
p11 lcd_data
ASIC/PIO
ctrlbus.rd
ctrlbus.iorq
ctrlbus.m1
ctrlbus.wr
1
&
&
&
in_op
out_op