Designspecifikation Miniräknare

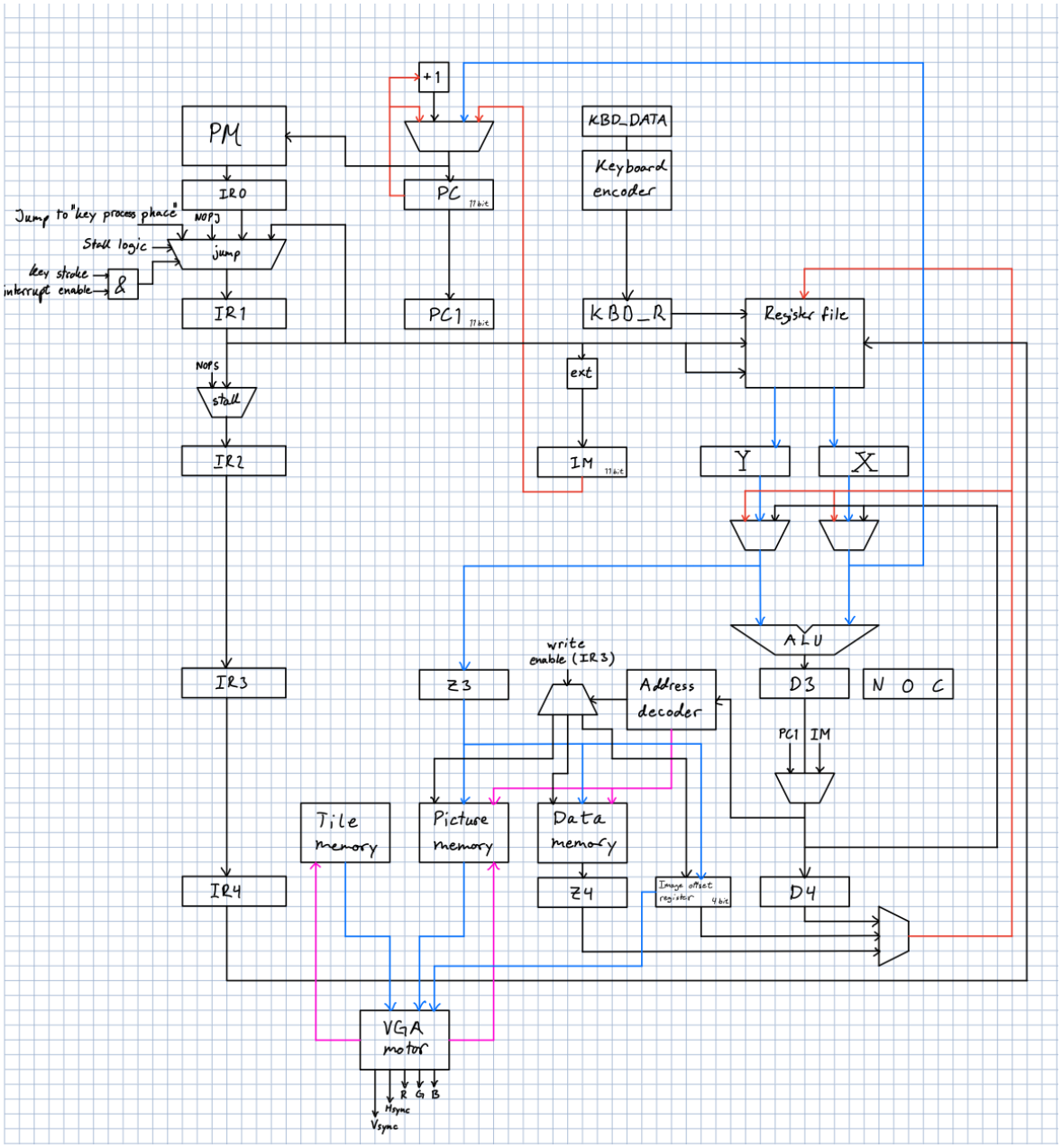
TSEA83, grupp 38

Axel Hellstand - axehe092
Axel Hammarberg - axeha009
Markus Handstedt - marha066
Albin Svärd Gruvell - albsv335

Verison 1.0
9 Mars 2022

1. Sammanfattning

I denna designspec visar vi konstruktionen av en processor, uppbyggd vi en 5-stegs pipeline, se figur 1. Processorn är tänkt att vara anpassad till funktionalitet hos en miniräknare vilket innebär att den har hårdvarustöd för de operationer som redan är implementerade i FPGAn, de resterande delarna anser vi är enklast att implementera i mjukvara för just denna funktionalitet. Utöver processors hårdvarustöd för de önskvärda beräkningarna har processorn även en hantering för ett USB tangentbord som kommunicerar via PS/2(vilket görs via ett instruktionsset och en separat keyboard encoder) samt en VGA-motor med tileminne och bildminne som adresseras via en central adressavkodare. Målet är att instruktioner ska ta en klockcykel och mer avancerad funktionaliteten som division implementerar vi som subrutiner i mjukvara. Endast load och store-instruktionerna interagerar med dataminnet, alla andra instruktioner arbetar mot register.



Figur 1, 5-stegs pipelinead processor

2. VGA-motor, Bild och Tile-minne

Bildminnet innehåller information om vilken tile som befinner sig på vilken position på skärmen, medans tileminnet innehåller information om hur dessa tiles ser ut. Bildminnet kan skrivas till med hjälp av adressavkodaren och båda minnen kan läsas från av VGA-motorn. VGA-motorn jobbar sedan med denna information som finns i minnena för att rita ut rätt på skärmen med färg, Vsync och Hsync. Inuti VGA-motorn finns logik som översätter x och y position som lagras internt till motsvarande position i bildminnet. Denna logik använder sig av bildförskuktningsregister som kan läsas och skrivas till med hjälp av load och store instruktioner. Värdet i bildförskuktningsregister bestämmer hur många rader förskjutna alla koordinater är. Det gör det möjligt att flytta rader med gamla resultat högre upp utan att behöva läsa från bildminnet.

3. Adressavkodaren + minnen

När datorn ska skriva till i något av data eller bild-minne så måste adressen först skickas genom adressavkodaren som avgör till vilken av de tre minnena adressen tillhör. Adressavkodaren styr sedan en demultiplexer för att välja rätt minne att skicka vidare datan ifrån vid en store-instruktion. Detta tillåter skrivning till alla minnen (utan PM, som är separat från DM och är för nu read-only) utan speciella instruktioner. Även bildförskuktningsregistret kan kommas åt med hjälp av adressavkodaren. Adressavkodaren gör det också möjligt att läsa från minnen utan speciella instruktioner, men bildminnet kan inte läsas ifrån av pipelinen. Det läses istället endast VGA-motorn för att rita ut bilden. Dataminnet och bildförskuktningsregistret kan läsas från. Z4-registret kan ses som att ingå i dataminnet, då blockram inte kan läsas asynkront, och läsningen ska ske på en klockcykel. Därför ger adressavkodaren också styrsignaler som är med i logiken för muxen som går till registerfilen / data forwarding-muxarna.

4. Programmräknaren

Programräknaren arbetar med att hålla koll på vilken adress i programminnet som nästa instruktion ligger på. I PC ligger adressen till den senaste instruktionen. PC1 används för att kunna hoppa effektivt då det är aktuellt. Den adress som ska hoppas till hamnar då i PC2. PC kan i sin tur ställas till tre olika saker vid klockflank. PC kan både behålla sitt värde, ökas på med 1 eller sättas till en lämplig adress för hopp (PC2). När ett hopp sker kommer pipelinen stallas en klockcykel.

5. ALU

ALUn är kärnan för de beräkningarna som själva miniräknaren ska kunna genomföra. De mest grundläggande operationerna som ALUn ska klara av är till en början addition av två register, subtraktion mellan två register eller multiplikation av två register. Vi ser även att ALUn ska klara av att utföra dessa grundläggande operationer fast med signed och unsigned värden. För övrigt vill vi även till en början att ALUn ska klara av vissa logiska operationer så som att genomföra en **and** för innehållet i två register eller **or**. Som i tidigare labbar kommer vi använda ALUn för att sätta de flaggor som vi ser att vi har användning för, vilket just nu är carry-flaggan (c) för en möjlig utökning av **add** till en **addc** funktion. Overflow-flaggan(v) för till exempel **muls**. Samt negative-flaggan(n) för att avgöra om resultatet är negativt eller ej, något som våra branch instruktioner kommer att nyttja. ALU:n stödjer också addition och subtraktion med carry, för att arbeta med större tal än 16 bitar.

6. Tangentbord

I datorn finns en flagga K som styr om ett tangentbordstryck kan tas emot. K = 1 innebär att datorn väntar på ett tangenttryck. Då kommer pipen stallas, endast nop-instruktioner skickas in och PC ökar inte. Den sätts av WAIT_K instruktionen. När ett knapptryck sker avkodas det av keyboard decodern och skrivs till KBD_R. Sedan sätts K till 0 och nästa instruktion åker in in pipen som vanligt. Om K = 0 kommer knapptrycket ignoreras. Detta fungerar bra då nästan alla knapptryck behöver spara undan ett värde för senare behandling, och sedan fortsätta vänta på knapptryck. Att hantera knapptryck samtidigt som en beräkning pågår är inte ett önskat beteende.

7. Instruktionsregister + Styrsignaler

Eftersom vår processor ska ha en 4:a stegs pipeline finns det register IR1 till IR4 där instruktioner. Dessa fyra register är 16 bitar breda och är kopplade till instruktionsavkodande kombinatorik som ger styrsignaler åt de olika delarna i pipelinen, som att sätta Write enable i minnet när en load-instruktion kommit till rätt del, eller att välja rätt operation på ALU:n.

8. Subrutiner

För att göra mjukvaran lättare att programmera så finns det JMPI som hoppar till en specifik plats i programminnet, och JMP som hoppar till en address i ett register. Detta gör att man kan skapa subrutiner. Dessa subrutiner får programmeraren själv bestämma vart dem hoppar tillbaka till. Det som bör vara den smartaste lösningen är att varje subrutin tar emot ett register som innehåller tillbaka adressen vilket subrutinen då kan återvända till när den är klar.

9. Minnesstorlekar

Bildminnet ska innehålla 15 * 20 tiles, 5 bitar stora => 1500 bits (blockram)
Tileminnet ska innehålla 20 olika tiles, med 8*8 pixlar och 3 bitar per pixel som väljer färg från en färgpalett => 3840 bits. (blockram)
Programminnet uppskattar vi räcker med 2048 instruktioner * 16 bitars instruktioner = 32768 bits (blockram)
Dataminnet räknar vi räcker med 2048 rader på 16 bitar var => 32768 bits. (blockram)
Load och store läser / skriver till 16 bitar med start på den angivna adressen. Anledningen till att datorminnet har 16-bitars rader är för att BCD-talen som lagras är 16 bitar stora.
Registerfilen innehåller 16 register på 16 bitar => 256 bits. (luttar)
Att vi valt 2048 rader innebär att hela minnerna kan addresseras med 11 bitar, vilket är hur stora immediate värden vi kan ladda in med 16 bitars instruktionsord.

10. BCD-Räkning

ALU:n kan utöver att räkna med binära tal också räkna med BCD-tal. Instruktionerna BCD_ADD och BCD_SUB fungerar som vanliga ADD och SUB, förutom att C-flaggan sätts som om talen vore BCD. Hantering av negativa tal, decimaltal samt multiplikation och division av BCD-tal görs med hjälp av mjukvara. Eftersom räknaren ska kunna räkna med större tal än 0-9 kommer BCD-tal sparas i dataminnet först med 1 rad som beskriver tecken, antal tecken och decimalpunktens plats, och sedan alla BCD-talen på rad. Varje BCD-tal 0-9 tar alltså 16 bitars plats, vilket innebär att 12 bitar blir oanvända. Detta gör dock att beräkningarna går snabbare och lättare att implementera, och då vi tänkt begränsa mängden input som kan ges till två rader på bildskärmen = 40 tecken = 640 bitar är den extra minnesanvändningen inte alls ett problem. Alla tal ses på som decimaltal, men decimalpunkter i slutet av talet skickas inte till bildminnet. Talen kan adderas iterativt, och negativa tal hanteras genom att välja mellan BCD_SUB och BCD_ADD och eventuellt byta tecken på resultatet.

11. Instruction set

NOP			00000
ADD	rx, ry	rx <= rx+ ry	10000
ADDC	rx, ry	rx <= rx + ry + C	10001
CMP	rx, ry	FLAGS as if rx - ry	00001
BCD_ADD	rx, ry	rx <= rx + ry	10010
BCD_ADDC	rx, ry	rx <= rx + ry + C	10011
BCD_SUB	rx, ry	rx <= rx - ry	10100
BCD_SUBC	rx, ry	rx <= rx - ry + C	10101
MOV	rx, ry	rx <= ry	10110
MOVI	rx, I	rx <= I	10111
MOV0	I	r0 <= I	00101
SUB	rx, ry	rx <= rx - ry	11000
SUBC	rx, ry	rx <= rx - ry + C	11001
MUL	rx, ry	rx <= rx * ry	11010
MULS	rx, ry	rx <= rx * ry	11011
LOAD	rx, ry	rx <= DM(ry)	00010
LOAD_PC	rx	rx <= PC	00111
STORE	rx, ry	DM(rx) <= ry	00011
JMPI	I	PC <= I	01000
JMP	rx	PC <= rx	01001
JMPEQ	I	PC <= I if Z	01010
JMPN	I	PC <= I if N	01011
JMPGE	I	PC <= I if not N	01100
SHR	rx	rx <= rx >> 1	11100
SHL	rx	rx <= rx << 1	11101
SWAP	rx	rx(HIGH) <= rx(LOW) rx(LOW) <= rx(HIGH)	00110
AND	rx, ry	rx <= rx & ry	11110
OR	rx, ry	rx <= rx | ry	11111
WAIT_K		PC <= PC tills K	00100

Det är mycket möjligt att de ändras om vi kommer på ett bättre sätt att uppnå detta.

12. Intruktionsuppbyggnad

5 bitar indentifikation

4 bitar rx	eller	11 bitar Immediate value	eller	4 bitar rx
4 bitar ry				7 bitar Immediate value

Att vi har endast 16 bitars instruktionsord innebär att vi inte kan ha immediate tal som är tillräckligt stora för att ladda ett fullt register. Därför har vi infört instruktionerna MOV0 och SWAP. Detta gör att vi lätt kan ladda den höga och låga delen av r0 och därefter flytta, addera med mera det till valfri plats.

13. Halvtidsmål

Vårt halvtidsmål är att hårdvaran av datorn är gjord och fungerar. Detta inkluderar både att alla instruktioner går att köra samt att alla delar av hårdvaran fungerar. Alltså ska datorn gå att köra om man bara sätter in önskad mjukvarukod i programminnet.