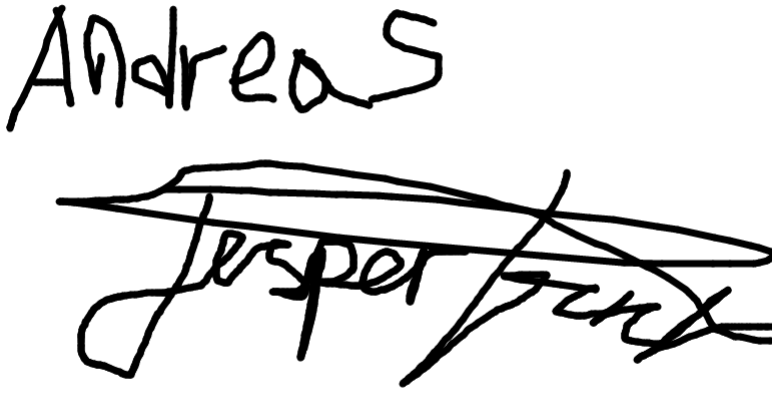
Indlejrede systemer opgave 2

Jesper Samsø Birch (s154366)
Andreas Gramstrup Correia (s153498)

September 2016



Contents

1 Introduktion

Følgende er en rapport der beskriver løsningen af Assignment 2 for indlejrede systemer, efteråret 2016. Rapporten er delt over i 4 dele ud over introduktionen:

- Designdelen, hvor at assembler koden. for programmet der skal køre på CPU'en der er blevet lavet, bliver beskrevet. Ud fra denne vil de instruktioner der er nødvendige for CPU'en at implementerer findes. Der beskrives derudover en ændring af selve programmet der er lavet, og der vil blive set på en compiler fra assembler kode til maskinkode.

- Implementering, hvor at den faktiske CPU vil blive vist og beskrevet. Her vil hvordan den fungerer blive forklaret ud fra de forskellige instruktioner der bruges - dette vil ikke gå i dybden med gezel koden. Der bliver hertil set på nogle valg og konsekvenser deraf, i forhold til designet, samt nogle mulige optimeringer der er lavet, og ikke er lavet. Til sidst vil noget af selve gezel-implementeringen blive diskuteret.

- Resultater, hvor at resultaterne for udførelsen af selve programmet på det opgivede datasæt vil blive beskrevet og illustreret. Der vil derudover blive set på hastigheden og energiforbruget for programmet, og pladsforbruget for CPU'en.

- Konklusionen, der runder det hele op.

Hvad angår selve løsningen, findes de tilhørende filer i bilaget. Koden for selve komponenterne findes i under /CPU/src/, imens at programmet i assemblerkode findes ved placeringen UCI_programs/mwi.uci. Kompileret til hexadecimal maskinkode, er det UCI_programs/mwi.asm. Resultaterne og lignende findes i selve hovedmappen. Programmet som cpu'en kører ligger i UCI_program/program.asm.

For at kører programmet et bestemt antal cycles, er der lavet et lille bash-program der sætter alle filerne for de forskellige komponenter sammen i en fil (mere herom ses under implemetering, gezeldelen), og kører det et bestemt antal cycles. For at køre det, skal følgende kommando derfor køres i hovedmappen for projektet:

bash run-CPU.sh CLOCKS

Hvor CLOCKS erstattes med antallet af clocks man gerne vil simulere.

Hvem der har lavet hvad i rapporten, ses angivet ved navn ude foran de respektive sektioner. Selve koden er udarbejdet sammen, hvor begge har været ligeligt med.

2 Design

2.1 Ændringer i moving window filter - Andreas

Da programmet skal implementeres i en lav energi cpu, er det vigtigt at programmet der implementeres, er så optimeret som muligt til at starte med. Et optimeret program kan gøre det muligt at nedsætte antallet af forskellige operationer cpu'en skal implementerer for at kunne eksekverer det, samt det kan nedsætte tiden programmet tager at eksekvere, hvilket ofte også betyder lavere energi forbrug.

For at lave det mest optimale mwi filter er det nødvendigt at lave nogle små ændringer til at originale mwi filter program. De ændringer der er foretaget, ses beskrevet og sammenlignet med det tidligere program(vist som pseudokode), nedenfor:

2.1.1 Originale program

```
N = 30
sum = 0
index = 0

arr[N]

mwiFilter(x)
    index++
    if index == N
        index = 0

    lastVal = arr[index]
    arr[index] = x

    sum += x - lastVal
    return sum / N
```

Det originale mwi filter som c programmet brugte, kan ses ovenfor som pseudokode. Det bruger et cirkulært array til at gemme de sidste 30 værdier i. Programmet gemmer summen af arrayet i variablen "sum". Når metoden mwiFilter køres, indsættes den nye værdi, og summen af arrayet opdateres, ved at tage forskellem imellem den værdi i arrayet som overskrives og den nye værdi, og lægge den til "sum". Derefter divideres summen af arrayet med 30, hvorefter den værdi returneres.

2.1.2 Optimerede program

```
N = 32
```

```
sum = 0
index = 0
ALLOWED_BITS = N - 1
arr[N]

mwiFilter(x)
    index++
    index &= ALLOWED_BITS


    lastVal = arr[index]
    arr[index] = x

    sum += x - lastVal
    return sum >> 5
```

For at kunne fjerne if blokken, og derved også den eneste conditional branch, er længden af arrayet arr blevet sat til 32, da man så i stedet kan bruge bitwise and til at bestemme indeksets interval, hvilket if blokken netop blev brugt til før. Dette er sandt fordi det maksimale indeks er $N - 1$, så når indekset er N så vil bitwise and med N - 1 gøre at index = 0, da $N \ \& \ N - 1 = 0$. Dette er dog kun sandt når N er en toerpotens. Da kodens eneste conditional branch er fjernet, er det nu ikke nødvendigt for cpu'en at kunne conditional branche, hvilket forventes at simplificere cpu'ens design, og dermed gøre den hurtigere og kræve mindre energi. Årsagen til at man kan gøre dette, uden at det ændrer resultatet af udregningerne, ses i næste sektion.

Da der nu divideres med en toerpotens, er det muligt at erstatte divisionen med bitshift mod højre. Division er en meget dyr operation, som kan forlænge kredsløbets kritiske vej en del. Dette fordi det forventes at den kritiske vej vil gå igennem en ALU, der vil stå for de aritmetiske udregninger i CPU'en. Derfor er det en klar fordel at erstatte divisionen med bitshift. At dividere med 32 er det samme som at bitshifte $log_2(32) = 5$ til højre, hvilket gøres i stedet.

Herunder kan forskellem imellem det originale og optimere program ses, hvor røde linjer markere linjer der er blevet slettet og grønne linjer markerer linjer der er tilføjet.

2.1.3 Originale/Optimerede programs outputs forskel

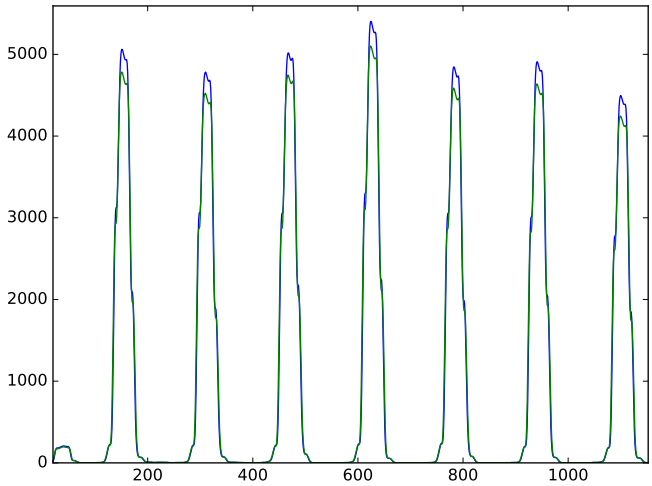
På grund af at det optimeret program tager gennemsnittet af de sidste 32 tal, i stedet for kun de sidste 30 tal, som det originale program gør, er der en forskel i de to programmers output. Årsagen til at man bruger 30 datapunkter til at starte med, er fordi at det skal svarer til bredden af et det størst mulige QRS kompleks[1]. Dette er krævet for at algoritmen dur. Der er selvfølgelig noget

```
N = 30
sum = 0
index = 0

arr[N]

mwiFilter(x)
    index++
    if index == N
        index = 0

    lastVal = arr[index]
    arr[index] = x

    sum += x - lastVal
    return sum / N
```

```
N = 32
sum = 0
index = 0
ALLOWED_BITS = N - 1
arr[N]

mwiFilter(x)
    index++
    index &= ALLOWED_BITS


    lastVal = arr[index]
    arr[index] = x

    sum += x - lastVal
    return sum >> 5
```

variation inden for bredden af disse, ligesom der er inde for pulsen, hvorfor man skal have den større. Det må dog ikke blive for stort, så det begynder at inkluderer T komplekset[1]. At forlængde den med 2 datapunkter svarer til at forlænge den med 2/250=1/125 del af et sekund, eller 0,008 sekunder, og i forhold til den bredde angivet i algoritmen, svarer det til en forlængelse på 2/30, hvilke er ca. 7 procent. Da man kun ændrer den en smule, og kun gør den bredere, burde det ikke ændrer meget på resultatet, og filteret burde ikke ramme T komplekset, når det er over QRS komplekset. Ved en test af det (ved at ændrer på c programmet og se om det så dur), kan det ses, at den mest synlige forskel på 30 og 32 datapunkter er, at intensiteten er en smule lavere i tilfældet med 32. Den overordnede form holdes stadig. Ved en kørsel af vores program med det nye filtrer, findes samtidigt også de samme r peaks. Derfor vil der blive gået ud fra at det virker.

Nedenunder kan ses en en graf over data'en fra det originale program, og det optimeret, over et udvalgt område. Det ses at der er en forskel, men at den generelle form nogenlunde holdes.



2.2 Assemblerkode - Jesper og Andreas

Ud fra pseudokoden fra det optimeret program, som kan findes i sektion 2.1.2, er nedenstående assemblerkode skrevet. Pseudokoden på venstre side svarer til assemblerkoden på højre side. Der antages at personen der læser dette har nogenlunde styr på normal assembler syntaks, og dermed ud fra dette og pseudokoden, kan forstå hvad de enkelte instruktioner gør. Bemærk at ved en LOAD eller STORE instruktion, svarer det højre mest ord (enten MEM eller CACHE her), hvilket target instruktionerne har.

```
N = 32
sum = 0                     //R3 sum
index = 0                   //R1 index
ALLOWED_BITS = N - 1        ADDI R2 R0 31 //ALLOWED_BITS
arr[N]
                            ADDI R4 R0 1   //load index
                            //R5 loaded value
                            //R6 former mwi value
                            //R7 new mwi result

                            #LOOP
                            //load new sensor value from memory
mwiFilter(x)                LOAD R5 R4 MEM

                            // do mwi filter on sensor value
    index++                 ADDI R1 R1 1
    index &= ALLOWED_BITS   AND R1 R1 R2

    lastVal = arr[index]    LOAD R6 R1 CACHE
    arr[index] = x          STORE R5 R1 CACHE

    sum += x - lastVal      SUB R6 R5 R6
                            ADD R3 R3 R6

    return sum >> 5         SRI R7 R3 5

                            //increment sensor load index and restart
                            ADDI R4 R4 1
                            JMP LOOP
```

Det bemærkes her to ting: resultaterne gemmes ikke til nogle steder i hukommelsen, og i assemblerkoden angives lokationerne CACHE og MEM. Det første gøres ikke, da der ikke er angivet at man skal det, og dermed heller ikke hvortil. Det ville dog være en nemmere måde at tjekke om ens resultater er korrekte, hvis man kunne gemme dem et sted i hukommelsen, men ændringer i hukommelsen gemmes ikke imellem kørsler. Dermed bliver det ligemeget at gøre, hvorfor det ikke tilføjet.

Derudover er der det andet. MEM er det normale target for hukommelsen, imens CACHE er en ekstra tilføjelse ved implementeringen i denne rapport. Det virker som et lille stykke hukommelse der er hurtigere at tilgå end MEM, så det gør udførelsen af programmet hurtigere. Detaljerne findes i implementeringsdelen, men uden CACHE, vil man kunne lave assemblerkoden således:

```
ADDI R1 R0 10048 //STARTINDEX
ADDI R2 R0 10079 //ALLOWED_BITS
//R3 SUM
```

```
ADDI R4 R0 1   //LOADINDEX
//R5 LOADEDVALUE
//R6 FORMERVALUE
//R7 DIFFERENCE
#LOOP

LOAD R5 R4 MEM

ADDI R1 R1 1
AND R1 R1 R2
LOAD R6 R1 MEM
SUB R6 R5 R6
ADD R3 R3 R6
STORE R5 R1 MEM
SRI R7 R3 5

ADDI R4 R4 1
JMP R0 LOOP
```

Ovenstående program er beregnet til at køre over 10000 datapunkter i hukommelsen hvilket er derfor startindekset er så stort. Selve platformen understøtter ikke en hukommelse på mere end 4kb, så ram blokkens hukommelse er øget til 11kb og adressen til hukommelsen er ændret fra 12 til 14, så det nye ram kan tilgås.

Begge programmer er testet, og begge virker. Ud fra assembler koden angivet, kan man se hvilke instruktioner der er nødvendige at implementerer. I hex format, som skal bruges til faktisk at køre programmet, ser programmet ud således:

```
0 6800001f
1 70000001
2 d6000002
3 64800001
4 24a00000
5 d8800004
6 5ae00000
7 de00000
8 a2900004
9 9d800005
a 72000001
b e0000002
```

Hvorfor det ser ud på den måde, kan ses under afsnittet "Implementering af instruktioner", hvor der bliver forklaret hvilke bits fra instruktionerne, svarer til hvad. Det er derefter bare et spørgsmål om at kompilere det til maskinkode.

2.3 Instruktionsdesign

Ud fra det optimerede program kan de forskellige instruktioner, som CPU'en skal have, for at kunne udføre programmet, ses. Registrene der benævnes er der hvor data midertididgt gemmes. Disse går som følger, angivet med en af de steder operationen skal bruges:

2.3.1 Tilføjede instruktioner - Jesper og Andreas

- ADDI
  $N = 32$
  For at kunne loade en konstant ind i et register på en nem måde, eller en nem måde at addere et register og en konstant, er denne instruktion tilføjet. Dette er også en simpel måde at lave en incrementer og decrementer på, da man så skal addere med 1 eller -1.

- AND
  $index = ALLOWED\_BITS$ Pga. ovenstående linje er en AND instruktion på to registre tilføjet.

- LOAD
  $lastVal = arr[index]$ For at det skal være muligt for cpu'en at tilgå data der ligger uden for dens registre, hvilket skal gøres for at kunne tilgå sensor data, samt at bruge gemt information fra mwi filtret, er denne instruktion tilføjet.

- STORE
  $arr[index] = x$
  For at kunne outputte information til et display (som skal gøres i virkeligheden) og for at kunne gemme mwi filter data, er en STORE operation lavet.

- SUB
  $sum+ = x - lastVal$
  For at kunne trække værdien i to registre fra hinanden, som skal gøres i ovenstående linje, er denne instruktion tilføjet.

- ADD
  $sum+ = x - lastVal$ For at kunne addere ovenstående, er en ADD operation tilføjet.

- SRI
  $returnsum >> 5$ For at kunne udføre ovenstående operation er en shift right intermediate (SRI) blevet tilføjet. Der tages kun intermediate, da der kun skal laves en shift right operation med et konstant antal bits.

- JMP
  Da programmet skal køre mere end en gang, er en JMP operation tilføjet som er i stand til at hoppe til en hvilket som helst linje i programmet.

I denne opgave, er der blevet bedt om at lave en CPU, der specifikt kan eksekverer en implementering af moving window filteret, skrevet som et program. Den behøvedes ikke at kunne mere. Derfor er det kun ovenstående instruktioner der er implementeret. Dette har nogle generelle konsekvenser i forhold til hvad CPU'en kan. Den er bl.a. ikke Turing-komplet, da den ikke kan lave conditional branching, så den kan ikke løse alle problemer der kan løses af en normal computer.

Det gælder også at den heller ikke en ordentlig divisionsmekanisme (gange tages som ligegyldigt, da den meget hurtigt kan fremstilles kode der gør dette - hvis man da har conditional branching), men kun divions med toerpotenser. Det gør den meget infleksibel, men også hurtigere. Primært det sidste har betydning, da gezel gør så den netop kører instruktioner over en clockcycle. Ved en implementering af division, hvor man ikke selv laver komponentet og bruger pipelining, men bruger gezels egen division, vil den forlænge tiden for en cycle betydeligt, da division er betydeligt langsommere end plus, minus, bitshift og bitwise anding, hvilket er de andre aritmetiske operationer der bruges. Samtidigt kræver den også mere energi, da den er mere kompliceret, hvorfor det på alle punkter er bedre ikke at have den med, hvis den ikke skal bruges.

Da der kun bitshiftes med en bestemt konstant, ville det også være muligt at fastsætte dette direkte i instruktionen, så shift right kommandoen, altid bitshifter 5 gange til højre. Detter er dog ikke gjort, da det potentielt er muligt at en mere fleksibel sri kommando kan gøre design fasen hurtigere.

2.3.2 Instruktions opbygning - Andreas

Nedenstående er beskrevet under antagelsen om at man har et komponent Register, der indeholder 8 registre til midlertidigt hukommelse (hvoraf register 0 altid er ligmed 0). Det antages at dets output a og b, er koblet til en ALU og andre komponenter, til faktiske udregninger og instruktionsudførsel.

Alle instruktioner har en længde på 32 bits. Der blev fundet at 16 bit var for lidt, men med 32 bits per instruktion, er der rigeligt med bits tilgængeligt. De 3 første bits(set fra venstre side af) er opkoden, som gør det muligt for cpu'en at se hvilken instruktion den nu skal køre. Der er 3, da der er 8 forskellige instruktioner. På samme måde, da der er 8 registrer i Register, angives et register med 3 bits. De tre næste bits repræsentere hvilket register i Register outputtet fra hukommelsen/bussen, eller ALU'en skal gemmes i. De næste 3 bits repræsentere hvilket register svarer til output a i Register. De næste tre bits står for det samme, men for b i stedet for a. De sidste 20 bits bliver brugt til at repræsentere et tal, hvis instruktioner har brug for det.

Nedenfor er en tabel som viser hvilke bits hver instruktion bruger. Bits som er konstante skrives med deres konstante værdier. Bits som bruges af instruktioner er markeret med `*`. Bits som ikke bruges af en instruktion er markeret med `x`.

navn	op	storeToReg	a	b	remainder
ADD	000	***	***	***	xxxxxxxxxxxxxxxxxxx
AND	001	***	***	***	xxxxxxxxxxxxxxxxxxx
SUB	010	***	***	***	xxxxxxxxxxxxxxxxxxx
ADDI	011	***	***	xxx	**********
SRI	100	***	***	xxx	**********
STORE	101	xxx	***	***	xxxxxxxxxxxxxxxxxxx
LOAD	110	***	***	xxx	xxxxxxxxxxxxxxxxxxx
JMP	111	xxx	xxx	xxx	**********

2.3.3 Instruktions specifikationer - Andreas

Ud fra ovenstående tabel kan der laves en lidt mere præcis instruktionsbeskrivelse:

- ADDI: Ligger summen af a og remainder ind i storeToReg.
- SUB: Ligger forskellen imellem a og b ind i storeToReg.
- AND: Ligger bitwise and imellem a og b ind i storeToReg.
- LOAD: Indlæser data fra target lig med remainder med adresse a ind i storeToReg.
- STORE: Skriver data fra a til addresse b i target lig med remainder.
- ADD: Ligger summen af a og b ind i storeToReg.
- SRI: Shifter a remainder gange til højre, og ligger resultatet ind i storeToReg.
- JMP: Hopper til linje nummer angivet ved remainder.

Det er så disse instruktioner der vil blive implementeret.

2.4 Kompiler - Andreas

For at simplificere processen med at lave assembler kode om til maskinkode, er der blevet lavet et program i Java, som er i stand til at gøre denne proces automatisk. Programmet tager en tekst fil med assembler kommandoer og giver en fil tilbage i hex med linje nummer. Kompileren kan findes i følgende github `https://github.com/TheAIBot/UCIC`. Instruktionerne kan ses implementeret her `https://github.com/TheAIBot/UCIC/blob/master/UCI/src/UCI/UCICompiler.java#L30`

Som et eksempel på hvordan kompileren fungere, så er den i stand til at tage denne linje `"ADD R1 R1 R3"`, som ligger summen af R1 og R3 ind i R1, og konvertere den til det korrekte hexadecimal program `"0 4a00000"`.

Kort eksempel på hvad kompileren giver tilbage når nedenstående fibonacci program kompileres.

Assembler	Maskinkode
`ADDI R2 R0 1`	`0 68000001`
`#LOOP`	
`//calculate first fibb number`	
`ADD R1 R1 R2`	`1 4a00000`
`//calculate second fibb number`	
`ADD R2 R2 R1`	`2 9100000`
`JMP LOOP`	`3 e0000001`

For at gøre kompilering af programmet nemmere, er et lille bash program i UCI_program mappen kaldet compile.sh lavet. Programmet tager to argumenter. Det første er stien til det program man gerne vil kompilere og det andet er navnet på filen, som det kompilerede program skal gemmes til. Et eksempel på hvordan programmet bruges kan ses herunder.

bash compile.sh fibb.uci fibb.asm

Bemærk at vores dokumenter med assemblerkode har filtypen .uci, og når det er kompileret til hexadecimaler, har det filtypen .asm.

3 Implementering

I forhold til implementeringen af CPU'en der kan kører programmet, vil der blive set på nedenstående:

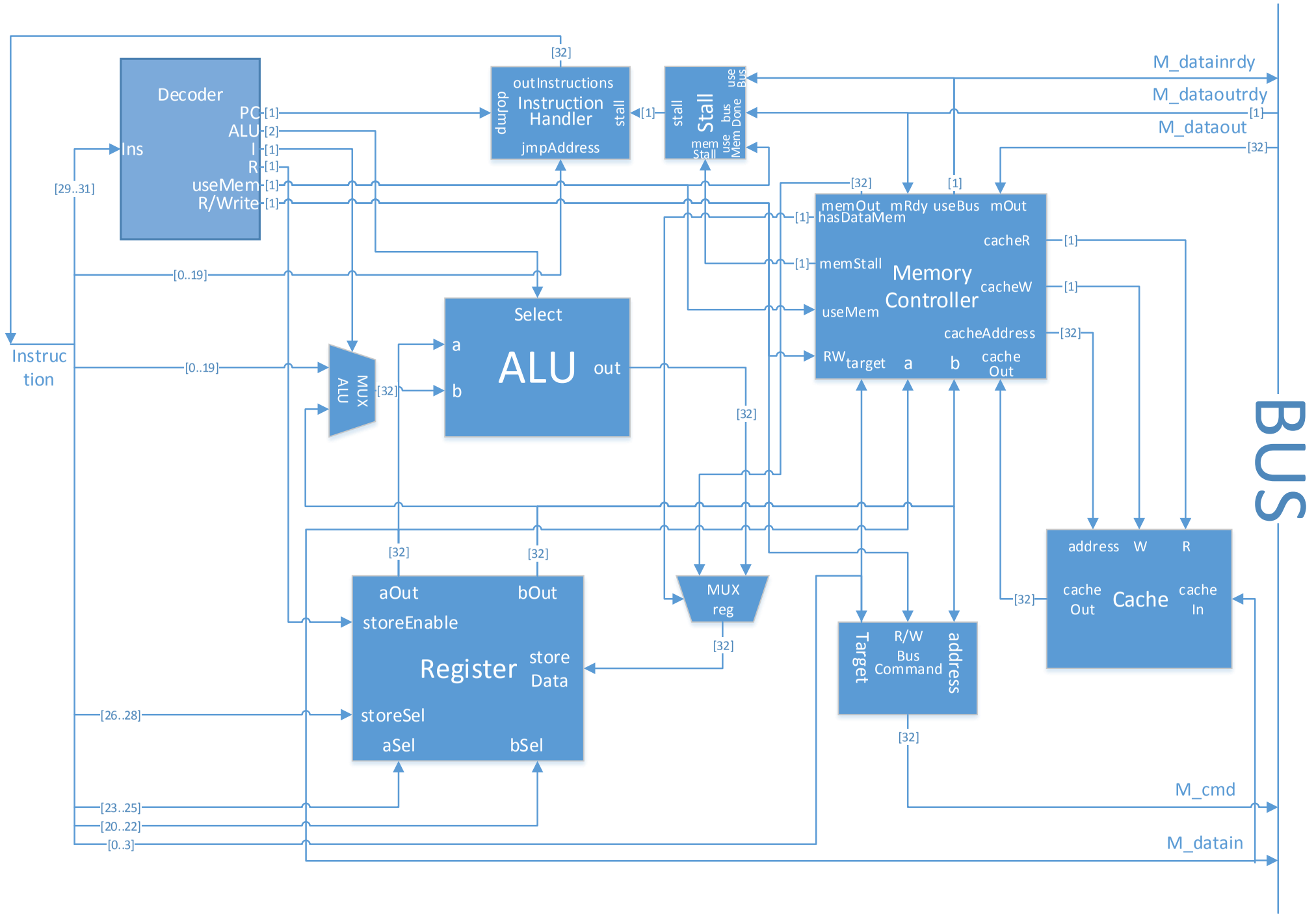
- Diagrammet for CPU'en (med kort forklaring).
- Instruktionstiderne, så det kan ses hvor lang tid hver instruktion tager i cycles.
- Hvordan CPU'en fungerer, ud fra forklaringer af hvordan instruktionerne håndteres af den.
- Mulige optimeringer som ikke er blevet lavet.
- Gezel implementationen af CPU'en.

Det er kun i det sidste punkt der vil blive set på den faktiske implementering i gezel. De andre holder sig til at tale om forbindelsen af forskellige komponenter og de ting de gør. Selv i det punkt vil der ikke blive gået i detaljerne med noget specifikt komponent, men der vil blive lavet overordnede bemærkninger derom,

og der vil blive kommenteret på specielle dele. Det gøres på den måde, da hvad gezel koden gør og hvorfor, burde være klar fra beskrivelsen af CPU'en ud fra instruktionerne. Dette især fordi CPU'en er så simpel som den er.

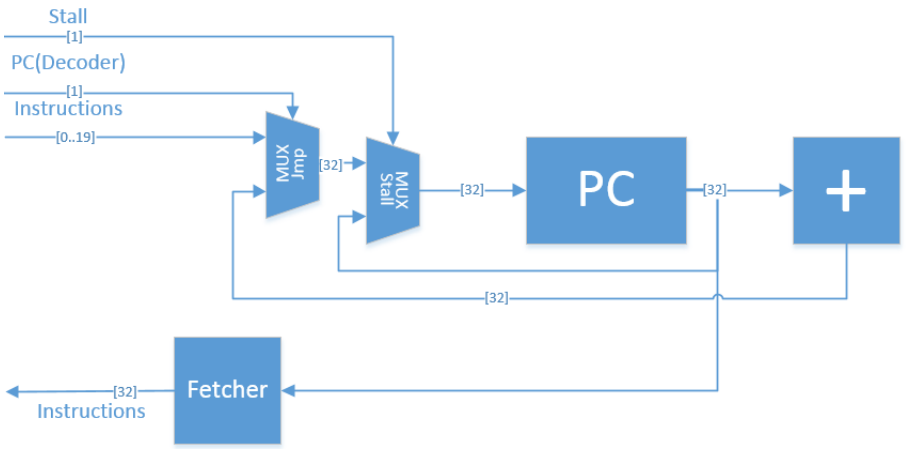
3.1 CPU diagram - Jesper

Det endelige design af CPU'en lavet, der kan udføre de angivede instruktioner, ses nedenfor:


Decoder
Ins
PC
ALU
I
R
useMem
R/Write
[29..31]
[1]
[2]
[1]
[1]
[1]
[1]
[32]
outInstructions
Instruction Handler
drop
stall
jmpAddress
[1]
stall
Stall
use Bus
bus Done
use Mem
mem Stall
M_datainrdy
M_dataoutrdy
M_dataout
[1]
[32]
[32]
[1]
memOut
mRdy
useBus
mOut
hasDataMem
cacheR
memStall
Memory Controller
cacheW
useMem
cacheAddress
RW
target
a
b
cache Out
[1]
[1]
[1]
[32]
[0..19]
[0..19]
Instruc tion
Select
a
b
ALU
out
MUX ALU
[32]
[32]
[32]
[32]
[32]
MUX reg
[32]
aOut
bOut
storeEnable
Register
store Data
storeSel
aSel
bSel
[26..28]
[23..25]
[20..22]
[0..3]
Target
R/W
Bus Command
address
[32]
address
W
R
cache Out
Cache
cache In
[32]
M_cmd
M_datain
BUS

På hver forbindelse er der indikeret størrelsen af signalet i bits, og hvis kun en del af et signal, se f.eks. a signalet fra instruktionerne, skal bruges, indikeres det med [fra..til] i forhold til bitsne i signalet.

Det gælder også at InstructionHandleren er bygget op af følgende komponenter (der var ikke plads i diagrammet ovenfor, hvis det skulle holdes inden for en A4 side):



Altså, den virker som en udvidet udgave af en PC og Fetcher. Bemærk at de i gezel koden ikke helt står opskrevet på den måde, men det burde ikke være noget problem at finde rundt i den - flere kommentarer herom i afsnittet om gezel koden. På diagrammet kan der også ses implementeringen af cachen som en del af designet, som benævnt før, vist ved sit eget komponent.

3.2 Instruktionstider - Andreas

navn	cycles
ADD	1
AND	1
SUB	1
ADDI	1
SRI	1
STORE	2 eller 8
LOAD	2 eller 8
JMP	1

Alle instruktioner, undtagen LOAD og STORE, tager 1 cycle da cpu'en blev designet med 1 instruktion/cycle i tanken. Dette har nogle ulemper i forhold til den kritiske vej, da den potentielt set godt kunne være kortere, hvis en instruktion kunne strække sig over flere cycles, f.eks. med pipelining. Grunden til at LOAD og STORE tager mere end en cycle, forklares i afsnittet nedenfor om memory operationer. Tiden på 2 er for tilgang til cachen, og på 8 er for tilgangen til busen.

3.3 Instruktionsbeskrivelsen og CPU implementering

Hvordan CPU'en fungerer, kan blive forklaret ud fra hvordan de instruktioner der kan gives til den, bliver eksekveret. Instruktionerne kan generelt set blive kategoriseret som en af fire typer:

- Direkte aritmetik. Herunder ADD, SUB og AND.
- Intermediate aritmetik, som set med ADDI og SRI. Meget lig direkte aritmetik.
- Branching, som set med JMP.
- Memory-operation. Herunder STORE og LOAD til bussen eller cachen.

Inden for de opgivede typer, er instruktionerne meget lig hinanden (se f.eks. sub og add), hvorfor de vil blive forklaret ud fra disse. En forklaring af opbygning, og hvorfor de fungerer på den måde som de gør, ses nedenfor:

3.3.1 Direkte aritmetik - Jesper

Operationerne fungerer overordnet set ved at ALU'en får givet et signal om hvilken type operation der skal laves (ADD, SUB, etc.) ud fra dekoderen, som selv aflæser det fra opkoden fra en given instruktion. Herefter udføres operationen på de givne input a og b. a kommer direkte fra registret, imens b kommer fra en mux, MUXAlu, således at der kan vælge imellem det andet output fra Register, eller en intermediate værdi fra instruktionen.

Outputet fra ALU'en bliver så sent til registreret, så hvis en LOAD eller STORE operation ikke bruges, tages den som input. Dette ordnes med muxen angivet i diagrammet.

Der gives ikke nogle flag for status af output fra ALU'en, f.eks. om det er lig 0, som set med normale ALU'er, og den beskrevet i opgavebeskrivelsen. Dette er ikke med, da det ikke er nødvendigt i forholde til programmet, primært fordi der ikke bruges, og dermed ikke er implementeret, conditional branching.

3.3.2 Intermediate aritmetik - Jesper

Det fungerer ligesom ved direkte aritmetik, borset fra at dekoderen giver signal til MUXAlu, så den giver de sidste 20 bits af instruktionen til ALU'en, i stedet for b fra Register. Det er 20 bit's, da det er mængden af bit's tilbage i instruktionen, når de 12 ud af 32 bits er blevet brugt på at angive opkoden og de 3 registre der skal være storesel, a og b.

3.3.3 Branching - Jesper

Branchingen fungerer ud fra det normale fetching system, hvor at et register med den nuværende adresse holdes, der så inkrementeres hver cycle. Dens værdi bruges så til at finde instruktionen der påkræves fra instruktionshukommelsen. Ved et jump vælges der igennem mux, at registret i stedet skal sættes ligmed den angivne adresse der kommer som input fra instruktionerne.

3.3.4 Memory-operationer Jesper og Andreas

Memoryoperationerne som STORE og LOAD, fungerer igennem MemoryControlleren, der står for håndteringen af bussen og cachen. Der er valgt et centralt komponent, frem for flere separate der håndterer forskellige dele af det, da en del af inputs og outputsene til bussen afhænger af hinanden, især med introduktionen af cachen - mere herom ses i tildelte afsnit. På den måde opnår man et forsimplet diagram og overordnet design, så det er lettere at aflæse og forstå diagrammet, hvormed chancen for at lave fejl bliver minimeret.

Når der laves en memory operation, hvilket vil sige en STORE eller LOAD, vil dekoderen informere memory controlleren om det, samt informere om det er en read eller write operation. Den primære grund til at memory controlleren eksisterer, er så valget imellem bussen og cachen er centraliseret et sted. Memory controlleren håndterer om det er cachen eller bussen der skal bruges, ved at kigge på det target der er valgt, ud fra instruktionssignalet. Hvis target er lig med 4, så vil memory controlleren læse eller skrive til cachen, ellers vil der blive skrevet til bussen.

Ud fra bussens protokol og interface, er der blevet lavet en direkte tilkobling af "a out" fra registret til "M_datain". Samtidigt bliver inputet til "M_cmd" direkte sammensat med komponentet BusCommand, der bare konkatinere "target" delen af signalet fra instruktionerne, R/W kommandoen fra dekoderen, og de første 27 digits i b. Dette kan gøres, da busen ikke aktiverer, ved mindre at M_datainrdy er sat lig 1. a er sat til "M_datain" frem for b, da storeSel så angiver hvilket register der skal gemmes til ved en LOAD operation, imens indikerer addressen der skal hentes fra, og ved en STORE, så indikerer a dataen der skal skrives til placering b.

At læse og skrive til ram er langsomt, hvilket vil sige at det tager mere end en clock cycle at læse information og det samme for at skrive. På grund af dette er det nødvendigt at kunne på cpu'en til at vente med at gå til den næste instruktion indtil at man er færdig med at læse eller skrive. Dette gøres ved at stalle program counteren, hvilket vil sige at man forhindre program counteren i at tælle op. Ved at gøre dette vil den samme instruktion blive kaldet hver clock, indtil at man stopper med at stalle. Det er primært nødvendigt at lave en stalling når en LOAD operation til busen køres, lige før data'en skal bruges, da der skal man eksplicit bruge data'en næsten med det samme. Hvis det ikke var tilfældet, vil det være bedre hvis man kunne køre resten af programmet imens busen behandler inputet - mere herom ses under afsnittet andre optimeringer. Der er ikke implementeret på en måde så at den staller specifikt i 8 cycles, da der i en virkelig situation, godt kunne være variation i dette tal. Når bussen skal tilgås, så stalles der indtil at bussen, via M_dataoutrdy, indikerer at den er færdig med sin operation. Når det kommer til cachen, vides det at det altid vil tage 2 clock cycles at læse og skrive, så memory controlleren sørger for at stalle i en enkelt cycle.

3.3.5 Cachen-som-ikke-rigtigt-er-en-cache™ - Jesper

Hvis ikke at cachen bliver brugt, bruges der i en given kørsel af koden en gang uden hop, 2 LOAD operationer og 1 STORE operation, hvor hver af dem tager 8 cycles at udføre, pga. busens ventetid. Resten af koden som ikke er en del af initialiseringen, består af 7 instruktioner der hver tager 1 cycle at udføre, hvormed man totalt har at analysen af et datapunkt tager $3 \cdot 8 + 7 = 31$ cycles. Heraf kommer 24 af dem fra busen. Altså tager bus-interaktionen langt det meste af tiden for kørslen af programmet. Det er derfor flaskehalsen for programmet, hvorfor det var det der blev set på i forhold til en optimering deraf.

Til dette blev der derfor lavet en cache (som ikke rigtigt er en cache, eller i hvert falde ikke ligesom en moderne cache, men mere en hurtig ram blok), som beskrevet ovenfor. Det virker som et lille (32 gange 32 bits plads) men hurtigt lager, hvortil at det circulære array i forhold til mwi bliver placeret. Lagerpladsen er ikke særlig stor, for at være repræsentativ for caches i virkeligheden. Desto større den er, desto langsommere (eller dyrere) bliver den, i virkeligheden.

Da man reducerer antallet af to operationer med 6 cyclesn, sparer man dermed 12 cycles, hvormed at udførelsen i løkken, går fra 31 til 19 cycles, en reducering på ca. 40%. en betydelig optimering.

At det gør implementeringen hurtigere kan selvfølgelig ikke bare direkte ses ud fra antallet af cycles der bliver minimeret. Hvis den kritiske vej f.eks. blev betydeligt større, kunne det være at det ikke kunne betale sig. I forhold til cachen er det dog ikke et særlig stort problem, hvilket ses hvis man laver en analyse af den kritiske vej for CPU'en. Det kan ikke laves særlig præcist, pga. at gezel ikke viser hvor lang tid en cycle tager, og ej heller hvor lang tid det tager i forhold til en cycle, at benytte et komponent eller en operation.

Tiden for benyttelse af normale gates direkte, er rigtig hurtigt, selvom der er en forskel alt efter hvilke komponent der er tale om, (se powerpoint for forelæsning 9). Af samme årsager er simple komponenter som Muxer, der ikke består af særlig mange gates, og små komperatorer (sammenligninger), ligeså meget hurtige. Dette vil selvfølgelig variere det alt efter hvordan gezel i praksis implementerer det, men ved at se på det som hvis man selv vil implementerer det, kan dette estimat opnås. Det meste af CPU'ens komponenter består af kontrolflow, der essentielt set er sammenligninger og få gates. Kun ALU'en, med dets aritmetiske operationer, og InstructionManager, når der inkrementeres på adressen, skiller sig ud. Register gør også til en hvis grad, pga. alle sammenligningerne der sker inde i den.

Tiderne for disse komponenter er derfor en betydelig faktor længere end for de andre komponenter. Hvis man ser på længden af de længste veje der går igennem ALU'en henholdsvis InstructionManager, ser man at den går igennem ALU'en, ved vejen:

Fra instruktionerne hen til Register, enten igennem asel eller bsel, ud fra b, op til MUXALU, over til ALU, ned til MUXreg, og så hen til Register igen.

Dette fordi at selvom InstructionManager venter på input fra stall der selv

får det fra MemoryController, er det på muxen før at den bliver færdig med det den skal i en cycle, altså lige før inputet til PC komponentet. Dermed er delene før den, som er involveret i InstructionManager enten langsommere, hvilket ses ikke at være sandt, da det langsomste vej ellers til den, kun går igennem dekoderen. Resten af vejen er så et plus og en mux. Det skal være den del der er langsommere, fordi at vejen igennem ALU'en før at registreret nås, på samme måde afhænger af outputet fra MemoryController, se MUXReg.

Ved tilføjelsen af cachen, kan der ved en given cycle fås data tilbage fra den, eller også gives der data - ikke begge ting samtidigt. Når der gives data, bliver vejen over MemoryController kun længere hen til cachen (hvilke er komponentet udover Register, der er forbundet til det, og der som er de facto langomest, da det selv får input fra Register). Nu vides det ikke hvordan ipblocken i den fungerer, man da der kun er 32 pladser af 32 bits, antages den som ret hurtig, omend betydeligt langsommere end f.eks. en enkel mux. Da den længste (vægtede) vej hen til MemoryController kun er igennem Register, forøges en kort vej med et betydeligt stykke. Dette betydelige stykke er dog ikke lige så stort som med ALU'en, hvorfor at den totalt er mindre og ikke ændrer på den kritiske vej.

Når data retuneres, skal den fra cachen hen til MemoryController, og så hen til MUXReg, og til sidst Register. Der regnes med at når den giver data tilbage er det hurtigt, ligesom før. Dermed opnås ikke en særlig stor forlængelse på tiden for MemoryController, hvorfor at den kritiske vej ikke ændres og forlænges. Dermed er implementeringen med cachen, hurtigere end uden. Dette er alt sammen estimater, og uden præcise værdier for tiderne over komponenterne, kan der ikke en præcis analyse. Men under antagelsen af at den er præcis, og dermed at den kritiske vej ikke bliver længere, svarer forbedringen til den procentssats opgivet før, ud fra reduceringen af antal af cycles for analysen af et datapunkt.

3.4 Andre optimeringer - Jesper

Udover de optimeringer der allerede er blevet lavet i forhold til hvordan selve programmet/algoritmen fungerer, og introduktionen af cachen til at reducerer tiden brugt på busen, er der ikke mange optimeringer der umiddelbart ville kunne blive lavet, som vil føre til en betydelig forøgelse i programmets hastighed. Dette er en konsekvens af primært to ting:

- Fordi at der kun kan tilgås 1 datapunkt af gangen, er mwi meget lineært i sin struktur. Det betyder også at programmet ikke kan tage mindre end 8 cycles, ca. halvdelen af det nuværende, da det er tiden for en bus henting. Da der skal ventes på at data'en er kommer tilbage, og fordi man ikke kan lave flere hentinger samtidigt, skal man vente 8 cycles før man kan gå videre til det næste datapunkt. Selv ved en implementering hvor man kunne hente mere data samtidigt, vil problemet stadig være der, da man ikke må hente flere datapunkter samtidigt - det skal nemelig simulerer virkeligheden.

- Tiden brugt på busoperationerne sammenlignet med resten af programmet, udgør en meget betydelig del af det.

Det er dog stadig mulig at lave nogle andre optimeringer, som dog ikke er blevet lavet, som beskrevet nedenfor:

3.4.1 Asynkronisation af loading af data fra busen

Det at aktiverer en bus LOAD eller STORE operation, tager kun en cycle, hvorfor der er en "spildtid" på 7 cycles, hvor der ikke bliver lavet noget, kun stallet. Potentialet for at gøre noget er der dog, så længe at det at busen bliver færdig med kommandoen håndteres. Det er derfor oplagt at implementere en muligheden i CPU'en for at hente eller gemme data til busen asynkront med de andre processor, og når man så skulle bruge den, så kunne man give en kommando om at der skulle ventes til at busen var færdig, hvis den ikke allerede var det. F.eks. kunne man i forhold til den LOAD operation der bruges i den nuværende assemblerkode tilføje flaget ASYNC, så den blev LOAD R5 R4 MEM ASYNC, og når den skulle bruges, så LOAD R5 R4 MEM AWAIT. STALL kunne så tilføjes hvis den skulle bruges umiddelbart.

Hvis der blev gået med den implementering, ville en stor del af programmet skulle omskrives. Pga. programmets størrelse vil det ikke være noget problem. Der er blevet eksperimenteret med en sådan omskrivning, og det er blevet fundet at det ville være muligt at implementere det, hvor der ville være en betydelig gevinst. Det kræver dog at man ville arbejde et datapunkt bagud, ie. at mwi blev forsinket med endnu 1/250 del af et sekund. Forbedringen ser dog ud til at være betydelig nok til at være det værd. Det er værd at bemærke, at hvis man ikke havde cachen, ville der være for mange bus operationer der afhang af hinanden, til at man ville få noget betydeligt ud af at implementerer det. Man vil så næsten altid skulle vente umiddelbart på at busen var færdig med sine operationer. På samme måde, hvis busen skal bruges mere og f.eks. resultatet for databehandlingen skal sendes videre over denne, ville det reducerer den effektive gevinst (procentmæssigt) betydeligt. Det er en konsekvens af at antallet af instruktioner som løkken indeholder, er så få som de er.

Selvom det ville være godt at implementerer, er dog ikke blevet gjort, pga. tidsbegrænsninger. Hvis der havde været mere tid tilgængeligt, vil det være en af de primære fokuser, pga. potentialet for forbedringer herigennem.

3.4.2 Mere avancerede instruktioner

Hvis nogle instruktioner på de samme datapunker bliver udført lige efter hinanden, et betydeligt antal gange, kunne det være at det kunne betale sig at slå operationeren sammen, og på den måde reducerer antallet af instruktioner (og her igennem cycles) brugt på at udføre det samme. Det kræver selvfølgelig at den kritiske vej ikke bliver betydeligt længere, sådan så tiden for en cycle bliver betydeligt længere.

I forhold til dette program, er det kun et fåtal af instruktioner hvorom dette gør sig gældene. Man kan f.eks. slå instruktion 2 og 3 i loopet, "ADDI R1 R1

1" og "AND R1 R1 R2" sammen på den måde, til en instruktion. En AND operation tager ikke meget længere tid sammenlignet med en ADD operation, hvis man ser på dem isolerede, da AND kan laves ud fra rent parallelle AND gates, imens en plus operation kræver komponenter i serie, se powerpoint fra forelæsning 9. At sætte dem sammen vil derfor ikke gøre meget med den kritiske vej. De andre instruktioner egner sig dog ikke på samme måde, da det så ville være f.eks. en SUB og en ADD instruktion i træk, hvilket er en betydelig forværing. Dermed ville det kun være de to instruktioner der kunne kombineres til 1, så effekten af at lave nogle kombinationer ville være lille. I forhold til hvor lille at den er, samt at det ville gøre den kritiske vej længere da det vil side lige efter en plusoperation for ALU'en, blev det derfor besluttet ikke at gøres.

Hvis man skulle lave noget lignende, skulle man havde et separat stykke hardware der stod for nogle af beregningerne, som så asynkront kunne arbejde med CPU'en.

3.5 Gezel implementeringen - Andreas

For at kunne holde styr på gezel koden, er det vigtigt at kunne holde ting separate. Det er dog ikke muligt i gezel som standard, da gezel i sig selv ikke understøtter brugen af flere forskellige filer. Det er dog muligt at bruge c++ preprocessor til at include andre filer. På den måde har det været muligt at sørge for at der er separate filer for alle de forskellige komponenter (med undtagelsen af nogle af komponenterne til InstructionHandler, der ligger i samme fil). Følgende kommando er brugt til at simulere 10 cycles af cpu'en:

```
cpp -P CPU/src/Platform.fdl | fdlsim 10
```

Et bash program for dette lille program er lavet, så man kan kalde:

```
bash run-CPU.sh CLOCKS
```

Hvor CLOCKS erstattes med antallet af clocks man gerne vil simulere.

På grund af denne separering af filer, har det været muligt at splitte testene fra implementerings filerne. Disse test bruges til at verificere at de forskellige komponenter giver de rigtige outputs. Verifikation af outputtet gøres manuelt, da et automatisk system til dette ikke blev set som nødvendigt.

Test er kun i mindregrad brugt til at debugge, da $display statements i de fleste tilfælde har været i stand til at pinpointe hvad problemet var, hurtigt og effektivt. Nogle af dem er derfor forældet. Af samme grund er trace ikke blevet brugt, da det ikke blev anset som nødvendigt at bruge.

Koden vil heller ikke indeholde nogen finite state machines (hvilket vil sige at den er strukturalt opbygget). Det er der primært to årsager til: Først og fremmest blev CPU'en anset som at være tilpas simpelt til, at det ikke var nødvendigt at bruge final state machinenes struktur, der som bekendt er lettere

at få et overblik over. Dette i hvert falde hvis man har et tilhørende diagram. Derudover ønskede os der har lavet rapporten og CPU'en, at få mere erfaring med hardware og hvordan det i virkeligheden fungerer. Da den strukturelle skrivemåde er tættere på dette end den opførselsbaserede/behaviorale skrivemåde, var den derfor oplagt at vælge. I tilfældet af at der skulle laves en større CPU med et mere kompliceret instruktionssæt, havde den anden metode dog nok blevet overvejet noget mere.

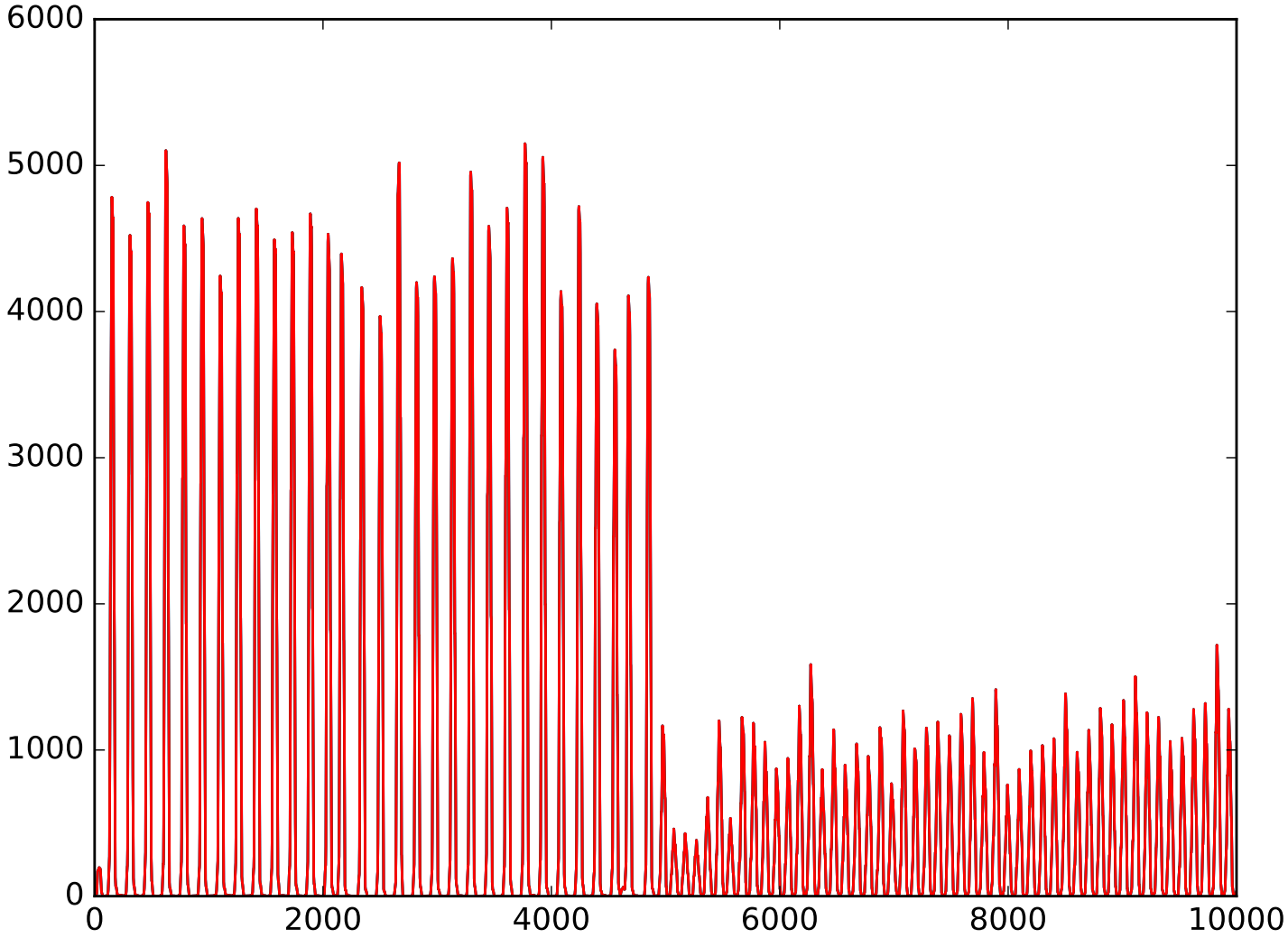
En specielt ting ved implementeringen i gezel, er, at der er blevet brugt lookuptables til Register. Det giver både en mere elegant kode, og man undgår samtidigt potentielt at lave en masse operationer, da de kan pre computeres og indsættes som konstanter i et lookuptable. Det kommer selvfølgelig an på implementeringen af lookuptablesne i virkeligheden.

4 Resultater

I denne sektion vil der blive set på de faktiske resultater, og der vil blive lavet en performance analyse af selve implementeringen.

4.1 Faktiske resultater - Andreas

Som en endelig test, for at verificere at programmet giver de rigtige resultater, er programmet testet på 10000 datapunkter. For at kunne teste dette, skulle mængden af ram på bussen øges og adressen til rammen skulle øges fra 12 bits til 14 bits. Herunder kan en graf ses for outputtet af programmet der bruger cachen, programmet der bruger hukommelsen fra bussen, og en graf for det korrekte output som vi får fra c kode programmet. Som det kan ses, så ligger de tre grafer oven i hinanden, hvilket passer overens med at summen af differensen imellem de tre grafer er lig 0.


6000
5000
4000
3000
2000
1000
0
0
2000
4000
6000
8000
10000

Hvis ikke andet, virker vores implementering som det var meningen at den skulle. Resultatfilerne med og uden cachen ses vedhæftet til afleveringen. De er begge to for 10000 datapunkter, og heder henholdvis "result_cache_10000.txt" og "result_mem_10000.txt". De ligger begge i hovedmappen for projektet. Resultatfilen for c koden ligger i filen "dersqrmwi_results.txt" samme sted. c koden der generede dette, findes under lokationen "c-program-mwi-filter.c".

4.2 Performance

4.2.1 Tid - Jesper og Andreas

Da det vides hvor mange cycles det tager at eksekvere hver instruktion, er det muligt at kigge på assembler koden for at finde ud af hvor mange cycles det tager for programmet at gå igennem 250 datapunkter.

Mwi filter initialisering

```
//initialize values

//R1 start index
ADDI R2 R0 31 //ALLOWED_BITS
//R3 sum
ADDI R4 R0 1   //load index
//R5 loaded value
//R6 former mwi value
//R7 new mwi result
```

Mwi filter

```
#LOOP
//load new sensor value from memory
LOAD R5 R4 MEM

// do mwi filter on sensor value
ADDI R1 R1 1
AND R1 R1 R2
LOAD R6 R1 CACHE
SUB R6 R5 R6
ADD R3 R3 R6
STORE R5 R1 CACHE
SRI R7 R3 5

//increment sensor load index and restart
ADDI R4 R4 1
JMP R0 LOOP
```

Ovenfor er programmet delt op i to dele. En initialisering som skal køre en gang inden filtret kan køre, og selve mwi filter loopet. Initialiseingen indeholder to instruktioner som hver tager 1 cycle at køre, hvilket betyder at initialiseringen tager 2 cycles at køre. At køre igennem loopet en gang er mere kompliceret, da det ikke er alle instruktioner der tager 1 cycle hver:

```
LOAD R5 R4 MEM
LOAD R6 R1 CACHE
STORE R5 R1 CACHE
```

Hver LOAD og STORE til MEM tager 8 cycles og hver LOAD og STORE til CACHE tager 2 cycles. Ud fra dette kan det ses at mwi loopet tager $8+1+1+2+1+1+2+1+1+1 = 19$ cycles at køre igennem en gang. Det er så tiden for analysen af et datapunkt i cycles. Da programmet skal køre igennem 250 datapunkter vil hele programmet tage:

$$2 + (8+1+1+2+1+1+2+1+1+1) \cdot 250 = 4752 \text{ cycles}$$

Hvis programmet så også bliver manuelt gået igennem, for at se hvor mange cycles det tager at køre igennem, finder man også frem til 4752 cycles.

I Gezel kan man ikke se hvad tiden for en cycle er, så man kan ikke ud fra ovenstående finde ud af hvor meget tid der faktisk vil blive brugt på at eksekverer programmet på de 250 datapunkter. Det samme gælder for hvor lang tid det tager at analyserer et datapunkt. Antallet af cycles kan dog bruges til at lave en relativ sammenligning imellem to forskellige implementeringer, hvis og kun hvis at den kritiske vej for de to implementeringer er nogen lunde lige lang, eftersom det er den der fastsætter tiden for en cycle. I dette tilfælde haves en ret kort kritisk vej (se afsnit om cachen for en analyse), der ikke indeholder meget mere end en ALU, og de essentielle komponenter og kontrollogik, til at udføre en instruktion, af størrelsenordnen en addition og bitshifting. Dette er i forhold til, at den ikke indeholder pipelining. Det kan derfor tages som om at tiden for en cycle er ret lille, sammenlignet med andre CPU'er uden pipeligning. Da der samtidigt bruges ret få cycles på et datapunkt, kan det konkluderes at implementeringen af programmet på CPU'en er hurtig.

4.2.2 Energi forbrug - Andreas

For at kunne beregne energiforbruget af mwi filter programmet som kører på cpu'en, vil nedenstående to linjer blive tilføjet til programmet. De to linjer holder styr på hvor mange gange programmet bruger en bestemt operation(f.eks add_op) og holder også styr på hvor mange toggles fra 0 til 1 og fra 1 til 0 den operations skaber.

```
$option "profile_toggle_alledge"
$option "profile_display_operations"
```

Ved at kigge på antallet af toggles i programmet burde det være muligt at kunne få et billede af energi forbruget i programmet, da det er en af de primære faktor for energiforbruget af en CPU. Der er dog en del problemer med denne metode til at aflæse energi forbruget, men der er ikke andre måder at gøre det på med gezel.

Et af de største problemer er at det ikke er alle typer operationer som har det samme energiforbrug. Dette kan lede til at en add_op operation som skaber 0 toggles, kan bruge mere energi end en and_op operation som skaber 32 toggles. Da det dog er den eneste måde at gøre det på vil denne metode stadig blive brugt. Det betyder så bare at resultatet skal tages med et bjerg salt og det faktiske energiforbrug kan godt afvige med en stor del.

Herunder er det gennemsnitlige antal evalueringer og toggles af forskellige operationer for analysen af et datapunkt, fundet over de 250 datpunkter.

Type	Evals	Toggles
dpoutput	1672	664
reg	827	214
sig	1047	447
assign_op	2366	756
ior_op	114	16
and_op	1320	48
shl_op	1	3
add_op	44	75
sub_op	1	3
concat_op	38	63
cast_op	4	3
neg_op	0	0
not_op	76	10
sel_op	561	430
eq_op	1943	98
ne_op	149	8
lu_op	0	0
ipoutput	414	33
Total		2863

Resultatet er ikke helt optimalt, da der stadig er en klar, relativ nem, måde at reducerer antallet af toggles, og pr. korrelation, dermed også energiforbruget. Dette er navnligt ved at reducerer bitbredden fra de generelle signaler og komponenter fra 32, til f.eks. 25. Der skal dog være plads til de forskellige værdier der kan opnås, og adresserne i hukommelsen der skal tilgås. Som et hurtigt estimat burde man i ovenstående tilfælde så reducerer antallet af toggles med ca. 20 procent, da der er ca. 20 procent færre bits at toggle - en betydelig besparelse, taget i betragtning af at de sidste par bits ikke bliver brugt til noget. Det vil dog for det meste værre mindre, da de fleste værdier der arbejdes med, langt

fra er oppe i de værdier der kræver brugen af 32 bits, og dermed ikke toggler dem. Det vil dog reducerer antallet af togles når der bruges not operationer eller lignende på signaler, da det altid toggler alle bits i signalet.

Dermed kan det siges at det vides ikke helt hvor godt energiforbruget er, men taget i betragtning af de simple instruktioner, og de relativt få cycles der benyttes pr. datapunkt, er den nok ikke meget høj i forhold til systemet der er tale om. Men da energiforbruget for et produkt som dette er vigtigt at holde neden, og der relativt let kunne laves en god optimering, er det dog ikke optimalt.

4.2.3 Område - Andreas

For et kredsløb som ikke må fylde meget, hvilket er en kategori det tænkte produkt for projektet hører under, er det vigtigt at pladsen som hardwaren tager, er så lille som muligt (dog uden at gå for meget i kompromis med funktionaliteten). Da et kredsløb kun består af komponenter, som kan udføre forskellige operationer, og signaler, som forbinder disse komponenter sammen, vil der blive kigger på disse to ting for at bedømme størrelsen af kredsløbet.

Signalerne som forbinder komponenterne sammen, kan have forskellige bit bredder og forskellige længde, som begge påvirker størrelsen af kredsløbet en del. Da det ikke er muligt at kunne bedømme et signals længde ud fra gezel kode, vil et signals bitbredde kun blive betragtet. Da denne simplificering laves, kan overvejelsen af signaler helt fjernes, da der for hvert input på en komponent vil være et tilsvarende signal med samme bit bredde. Signaler kan derfor fjernes hvis de ses som en del af størrelsen af komponenten. Dette giver mening da simple komponenters og signalers størrelse vokser lineært med den påkrævende bitbredde.

For at kunne bedømme et komponents størrelse, bliver man nød til at kende dens størrelse per input bit, eller lignende. Denne størrelse er forskellig for alle komponenter og er også ukendt, da gezel ikke opgiver denne information for hvordan den implementerer det, i forhold til koden. Der kunne laves et opslag for arealet af alle basiskomponenterne (eller rettere, en udvalgt implementering deraf) der indegår i kredsløbet, og for kredsløbene for alle funktionerne i gezel, som f.eks. et adderings kredsløb. Dette vil dog tage alt for lang tid, og analysen vil alligevel ikke være præcis, da man ikke kender ledning/signallayoutet. Størrelsen af kredsløbet vil i stedet blive set som summen af alle komponenter multipliceret deres størrelse og påkrævet bitbredde:

$$kredslb_{strrelse} = \sum_{x} (komponent_x \cdot bitbredde_x \cdot strrelse_x)$$

Dette skulle gerne give en approksimation af det relative pladsoptag for kredsløbet. For at kunne lave ovenstående beregning skal alle komponenter, deres bitbredde og antal findes. Gezel cpu'en er gået igennem og nedenfor kan en liste af de forskellige komponenter, antal og bitbredde findes:

Operator/Komponent navn	Bitbredde	Antal
equal	2	3
equal	3	21
equal	4	1
add	32	2
sub	32	1
sri	32	1
and	1	6
and	3	7
and	32	1
not	1	4
or	1	2
reg	1	1
reg	32	9
ternary (mux)	32	30
concat	32	1
ram	32	96
lookup	7	8
convert	32	2

Bemærk at alle muxene tager 1 selektor bit input, og 2 32 bits input. Den er lineært i forhold til begge, så der skrives en bitbredde på 32*1=32 for den.

Dette inkluderer instruktionshukkomelsen og cachen, men ikke arealet af bussen og dets hukommelse. Ud fra tabellen er det muligt opstille en ligning for hvor meget plads hver komponent tager og så derefter også hvor meget plads hele kredsløbet vil tage. Størrelsen af hver komponent er dog ikke kendt, så den vil blive repræsenteret som en ubestemt variabel.

$$
\begin{aligned}
equal_s = (2 \cdot 3 + 3 \cdot 21 + 4 \cdot 1) \cdot a &= 73 \cdot a \\
add_s = 32 \cdot 2 \cdot b &= 64 \cdot b \\
sub_s = 32 \cdot 1 \cdot c &= 32 \cdot c \\
sri_s = 32 \cdot 1 \cdot d &= 32 \cdot d \\
and_s = (1 \cdot 6 + 3 \cdot 7 + 32 \cdot 1) \cdot e &= 59 \cdot e \\
not_s = 1 \cdot 4 \cdot f &= 4 \cdot f \\
or_s = 1 \cdot 2 \cdot g &= 2 \cdot g \\
reg_s = (1 \cdot 1 + 32 \cdot 9) \cdot h &= 289 \cdot h \\
ternary_s(mux) = 32 \cdot 30 \cdot i &= 960 \cdot i \\
concat_s = 32 \cdot 1 \cdot j &= 32 \cdot j \\
ram_s = 32 \cdot 96 \cdot k &= 3072 \cdot k \\
lookup_s = 7 \cdot 8 \cdot l &= 56 \cdot l \\
convert_s = 32 \cdot 2 \cdot m &= 64 \cdot m
\end{aligned}
$$

$$
\begin{aligned}
total_{areal} &= equal_s + add_s + sub_s + sri_s + and_s + not_s \\
&\quad + or_s + reg_s + ternary_s + concat_s + ram_s \\
&\quad + lookup_s + convert_s \\
&= 73 \cdot a + 64 \cdot b + 32 \cdot c + 32 \cdot d + 59 \cdot e + 4 \cdot f \\
&\quad + 2 \cdot g + 289 \cdot h + 960 \cdot i + 32 \cdot j + 3072 \cdot k \\
&\quad + 56 \cdot l + 64 \cdot m
\end{aligned}
$$

For at få arealet skal man finde ud af hvad a,b. . . ,m skal være. Det er dog en opgave som ligger uden for denne rapports fokus og vil derfor ikke blive gjort. Som en simpel løsning, kan de ubestemt variabler alle sættes til 1. Hermed fås så at $total_{areal} = 4739$, og uden at inkluderer instruktionshukommelsen, at $total_{areal} = 2691$.

Det kan fra regnestykket se, at det at inkluderer cachen i designet, har en stor betydning på pladsforbruget (ud fra ovenstående model), af CPU'en, da den indeholder 32 pladser af 32 bits, hvilket produktet er lig 1024. Dette svarer til ca. en femtedel af alt pladsen, og dette med instruktions hukommelsen inkluderet. Selv med dette forøgede pladsforbrug, bedømmes det dog stadig som det værd at have cachen med, i forhold til forøgelsen af hastigheden i behandlingen af et datapunkt. Dette hjælper bl.a. med at spare mere energi, og for et projekt som dette, hvor der er tale om en størrelsesordensforøgelse med ca. 1/4, i forhold til den benævnte gevinst, tages det at spare pladsen som mindre vigtigt. Pladsforbruget bliver dermed ikke optimalt.

På samme måde som med energien, vil der også kunne spares en del plads ved at reducere antallet af bits for et normalt signal fra 32 til f.eks. 25. Ved udregning der ikke vil blive vist her, men som består i at finde frem til formlen ovenfor ud fra at komponenter/operatorer der bruger 32 bits bliver erstatte med 25 bit, kan man se at man kan spare ca. 21 procent, da man så får $total_{areal} = 3738$. Dette er en betydelig forbedring, og til forskel fra med energiforbruget hvor der ikke var et direkte lineært sammenhængende mellem antallet af bits og energiforbruget, er det sandt ud fra modelen her. Dermed vil det være den rigtige effektive besparing.

Selvom pladsforbruget er større end det kunne havde været, hvilket er den endelige bedømmelse deraf, anses det ikke som noget større problem. Dette fordi at selv hvis man kan reducerer det med en faktor 4, vil det i et system som med et ECG apperatur, ikke værre selve CPU'en der vil stå for det meste plads, da den kan gøres meget lille. Det er derimod apperaturet til måling og enten visningen af resultaterne, eller kabler til overførslen deraf. Dette kommer dog an på hvor meget teknologi der er med der.

5 Konklusion

Alt i alt ses det at en CPU og det dertilhørende program, som opfylder de opgivede krav, er blevet lavet. Ved kørslen af programmet på CPU'en, giver

den det samme resultat som c koden den er baseret på.

Det ses at med optimeringerne der er lavet til selve programmet, er der opnået et program der kan eksekverer hurtigt, også når det er konverteret om til assemblerkode. Koden kræver kun 8 forskellige simple typer instruktioner, og fordi at det ikke bruger division, er tiden for en cycle i praksis reduceret betydeligt i forhold til hvis der blev brugt det, så den en cycle burde være hurtig. Tiden for en cycle kan dog ikke ses, på grund af gezels begrænsninger.

Ud fra de udvalgte instruktioner, ses det også at der er opnået et simpelt letforståeligt CPU design. Det viste sig også at ideen med at tilføje et lille hurtigt stykke hukommelse som en slags cache var en god ide, da den reducerede antallet af cycles brugt på et datapunkt med næsten 40%, uden at forlænge den kritiske vej. Det siger mest af alt noget om busen som en begrænsende faktor. Som bemærket i løbet af rapporten er det ikke ensbetydende med at der er opnået et fuldkommen optimeret design. Specifikt ville der være en god mulighed for at udvikle en asynkron hentning og gemning af data til busen, som ville kunne reducerer antallet af cycles brugt pr. datapunkt endnu mere.

Som sagt, er der opnået de forventede resultater. Ud fra en analyse af tiden, energien og pladsforbruget for programmet, baseret på eksekveringen af de opgivede datafiler, er det kommet frem til at

- Implementeringen er hurtig, da den kun bruger 19 (relativt korte) cycles, til at behandle et datapunkt. Dette uden pipelining. Det kunne derudfra også ses at den maksimalt (som en øvre grænse) kunne gøres 2 til 2,5 gange hurtigere, pga. tiden det tager at hente den nye data fra bussen, hvilket skal ske en gang pr. datapunk behandlet.

- Energiforbruget var svært at bedømme, men at det relativt let kunne blive optimeret ved at reducere bitbredden af signallet fra 32 til et tal som 25.

- Pladsforbruget var større end det kunne havde været, pga. brugen af cachen, men at det stadig blev fundet som det værd pga. forøgelsen i hastigheden der følger med den, og dermed et formindsket energiforbrug. Det blev også fundet at det relativt let kunne blive optimeret, ved at reducere bitbredden af signalet fra 32 til et tal som 25, og dermed spare ca. 20 procents plads.

Som afrundingen kan det dermed siges, at der er blevet lavet en god implementering, med plads til forbedringer til tidsforbruget, energiforbruget og pladsforbruget.

References

[1] A real time QRS detection algorithm, by Jiapu Pan, and Willis J. Tompkins.