62711 PWA

Buster s211548, Marcus s215458, Thomas s215449,
& Valdemar s215483

Saturday 4 March, 2023

1 Introduktion

Denne rapport skitserer design, simulering, implementering og test af en aritmetisk logisk enhed (ALU) og dens tilhørende blokke som en del af projektarbejde A for kurset 62711 - Design af digitale systemer. Formålet med dette delprojekt er at opnå en omfattende forståelse af, hvordan en Datapath fungerer, og hvordan den er bygget op. Datastien i figur PWA-1 på side 3 består af et 16x8-bit Register File (RF) modul, et Function Unit modul med en Arithmetic Logic Unit (ALU) og en shifter og det tilhørende afkodnings- og multiplekseringskredsløb. Hovedfokus i denne rapport er på design og implementering af Register File-modulet, som indeholder 16 registre, der bruges til at lagre lokal information i Datapath. RF-modulet har kombinatorisk logik og processer til registeroverførsel og skal implementeres i VHDL ved hjælp af entiteten RegisterFile. For at strukturere designet skal RF'ens underblokke implementeres separat. Arkitekturen bør derefter instansiere destinationsdekoderen, registrene R0-R15 og multiplekserne. Deres enhed skal være som angivet i den medfølgende tabel. De efterfølgende afsnit af rapporten vil diskutere implementeringen af registerfilen, dens underblokke og de tilhørende dekodning- og multiplekserkredsløb i detaljer.

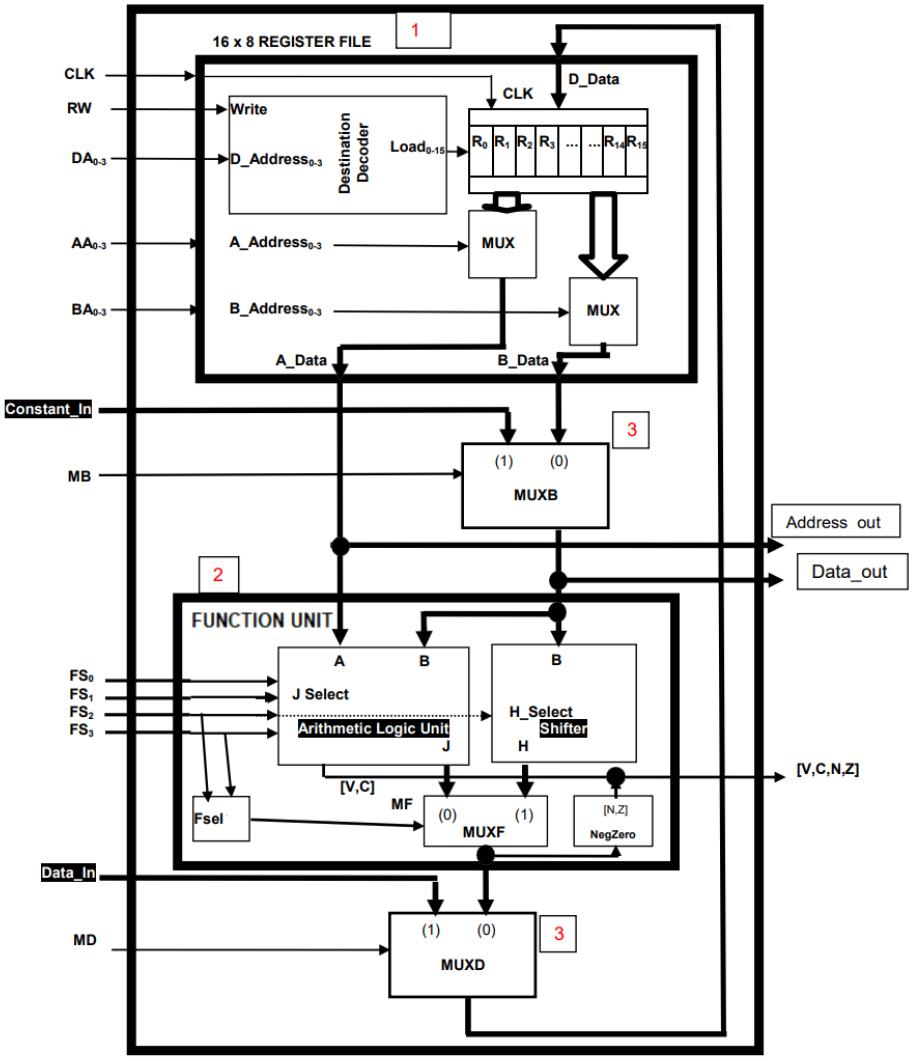
1.1 Specifikation

I kurset design af digitale systemer (62711) arbejdes med FPGA og VHDL programmering. Det langsigtede mål med kurset er at designe en CPU, som lægges på et Nexys DDR4-board.
I dette projekt er fokus på at lave en registerfil, som kan indlæse, gemme og udskrive data samt en funktionsenhed, der kan lave almindelige aritmetiske og logik operationer.

2 Analyse

I denne sektion analyseres datastrømmen i det udlevere topmoduldiagram, samt hvilke forudsætninger de enkelte dele har for at udøve deres funktion.

2.1 Flow i DataPath



Figur 1: PWA DataPath

Top modulet har 2 primære funktioner, at opbevare data, og aritmetiske operationer.
For at opfylde disse 2 funktioner har top modul 2 under moduler, "Register File" modulet som står for at opbevare data i de korrekte registre og "Function Unit" modulet som står for at indhente det opbevarede data udføre aritmetiske operationer på det.
For at kunne gøre dette har vi først og fremmest et Read-Write signal som fortæller Register File hvorvidt den skal gemme det data der er tilgængeligt, skal den gemme det så bliver DA bussen dekodet fra 4 til 16 bit som bestemmer hvilket register det skal gemmes i.
Herefter bruges AA og BA busserne til at afgøre hvilket registers 8-bit data, Function Unitten skal kunnne læse fra.
Dette data kan Function Unitten nu regne på, men før at Function Unitten ved hvilken operation den skal udføre får den en 4 bit bus FS, denne bus bliver dekodet til en 16 bit bus som ALU'en og shifteren bruges til at afgøre hvilken operation vil der vil blive udført.
ALU'en outputter 2 bits til en 4 bit bus, som fortæller hvorvidt operationen giver overflow og hvorvidt vi har en carryout.
Til sidst bruges FS også til at afgøre om det er ALU'ens output eller shifterens output der er relevant gennem en multiplexer.
Det relevante output bliver herved sendt til Register File modulet som kan opbevare værdien hvis nødvendigt.

3 Design

I designafsnittet uddybes designet og schematics for de enkelte delkomponenter af det samlede projekt.

3.1 Modul diagram for top-modul for DATAPATH

Datapath kan indeles i RegisterFile og Functional Unit, de samarbejder ved at RegisterFile gemmer på data som functional unit så kan udføre aritmetiske operationer på, hvorfter den kan gemme nye værdier i RegisterFile, hvorved den rekursivt kan udføre aritmetik.

3.2 Modul diagram for register file top-modul

Register file består af 3 primære komponent, 16 8-bit registre, en dekoder som fortæller registrene hvilket der skal gemmes data i og 2 MUX'er som bruges til at læse data ud af registrene, diagrammet for dette kan ses på appendix 55.
Dekoderen er en 4 til 16 dekoder, som er opbygget af 5 2 til 4 dekodere, hvor vi har en 2 til 4 dekoder hvis output bruges som enable signal til 4 andre 2 til 4 dekodere, schematic for dette kan ses på appendix 50.
2 til 4 dekoderne består udelukkende af kombinatorisk logik som kan ses på appendix 49.
MUX'erne er 16 til 1 MUX'er, som er opbygget af 5 4 til 1 MUX'er, de 4 første MUX'er er styrret af en 2 til 4 dekoder hvis output er de første 2 bits af SelectAddress inputtet, de 2 sidste bits bliver dekodet af en anden 2 til 4 dekoder, som bruges i den sidste 4 til 1 MUX som styre hvilken af de 4 første MUX'ers output er det endelige output, schematic findes i appendix 52.
4 til 1 MUX'en består af 4x8 and gates, hvor 4 8-bit inputs bliver and'et med et toggle bit, denne MUX er opbygget anderledes en de fleste andre MUX'er, eftersom den kræver en toggle input som allerede er dekodet, hvor en MUX ofte har en dekoder indbygget, men eftersom vi allerede havde en 2 til 4 dekoder, valgte vi at genbruge den, schematic kan ses i appendix 51.
Selve registrene(Register Cells) består af 8 D flip-flops, som alle deler clock, reset og enable signal, de får hver 1 bit af 8-bit inputtet, og outputter hver 1 bit ud af 8-bit outputtet, derved udgør de 1 8-bit register, som kan ses i

appendix 53.
Register File modulet indeholder 16 registrer, derved har vi 16 af disse registrer som alle deler clock, reset og input signaler, det de ikke deler er enable signalet, så vi derved kan loade data ind i kun 1 specifikt register, notér at kun dele af schmatic'et kan findes i appendix, eftersom det er for stort, men det fortsætter på præcis samme måde som hvad der er vidst, det fortsætter indtil register nr 15, diagrammet findes i appendix 54.

3.3 Timing diagram

Når Register File modulet skal opbevare data, sættes RW signalet højt, når dette signal er højt og vi har en opadgående clock flanke så vil dataen i D_Data blive loadet ind i det register tilhørende værdien af DA bussen. Hvis AA eller BA bussen så sættes til værdien af DA, vil vi se at dataen i D_Data bliver loadet igennem multiplexeren det øjeblik registerest værdi opdateres, dette kan ses i appendix 31

3.4 Modul diagram for function unit modul

Funktion Unit top modulet består af fem underblokke Arithmetic logic unit (ALU)32, som udfører de aritmetriske funktioner på baggrund af J_Select. J_Select består af FS_0, FS_1, og FS_3. Resultatet fra ALU'en føres via J videre til MUXF modulet. MUXF34 modulet er en multiplexer, der vælger mellem dataen fra J og H. H dataen kommer fra shifter33 modulet, som har til opgave at skifte de 8 bits data fra B til højre eller venstre, alt afhængig af om FS_2 er høj eller lav. MUXF skifter mellem de to signaler ved hjælp af Fsel35 modulet, som går højt hvis både FS_2 og FS_3 er høje. Fra MUXF sendes signalet både ud af 'Funktion Unit', men også til NegZero36 modulet, der har til opgave at tjekke om resultatet er rent nuller og om det er positivt eller negativt.
Hvad disse 5 moduler består af, kan findes i Apendix, sammen med et eksempel på hvordan Funktion Unit'en behandler dataen, i form af skema der er udarbejdet på baggrund at spørgsmål 5.

4 Simulering

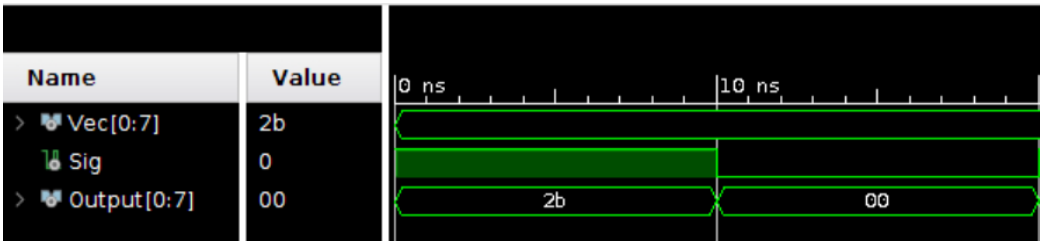
Simulering er et vigtigt trin i designprocessen, da det gør det muligt for designeren at verificere funktionaliteten af kredsløbet og identificere eventuelle potentielle fejl eller fejl.

Dette afsnit af rapporten vil give et omfattende overblik over VHDL-simuleringen udført på det aktuelle digitale kredsløb. Simuleringsprocessen involverede at skabe testbænke til at stimulere kredsløbet og evaluere dets adfærd under forskellige forhold. Resultaterne af simuleringen analyseres og diskuteres i detaljer, hvilket fremhæver eventuelle problemer, der er stødt på, og hvordan de blev løst. Derudover vil dette afsnit også dække de værktøjer og teknikker, der bruges til simulering, sammen med eventuelle udfordringer, der står over for under processen.

Samlet set giver VHDL-simuleringssektionen en dybdegående evaluering af det digitale kredsløbs adfærd og sikrer dets funktionalitet før dets fysiske implementering.

4.1 Simulering af Register file modul

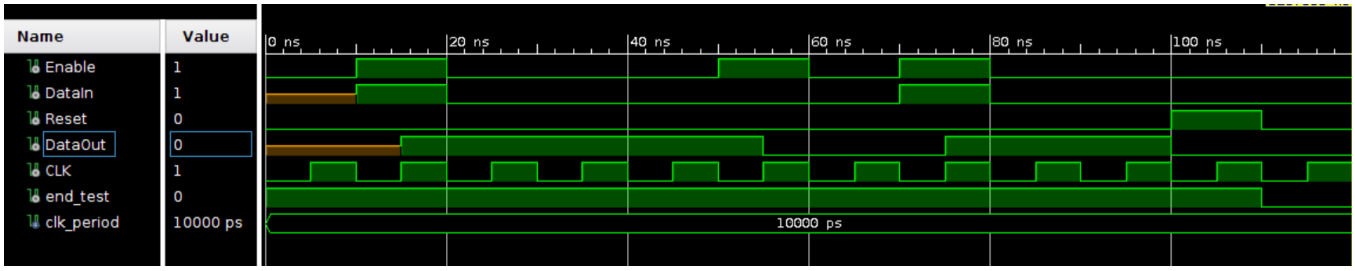
4.1.1 Simulering af AND8-komponent modul



Figur 2: AND-gate med 8-bit input

Som det ses på figur 2 styrer Sig-bittet, hvorvidt inputtet Vec kommer igennem gaten. Når det er højt bliver outputtet lig Vec, og når det er lavt, bliver outputtet lig 0x00. Komponenten virker dermed som designet.

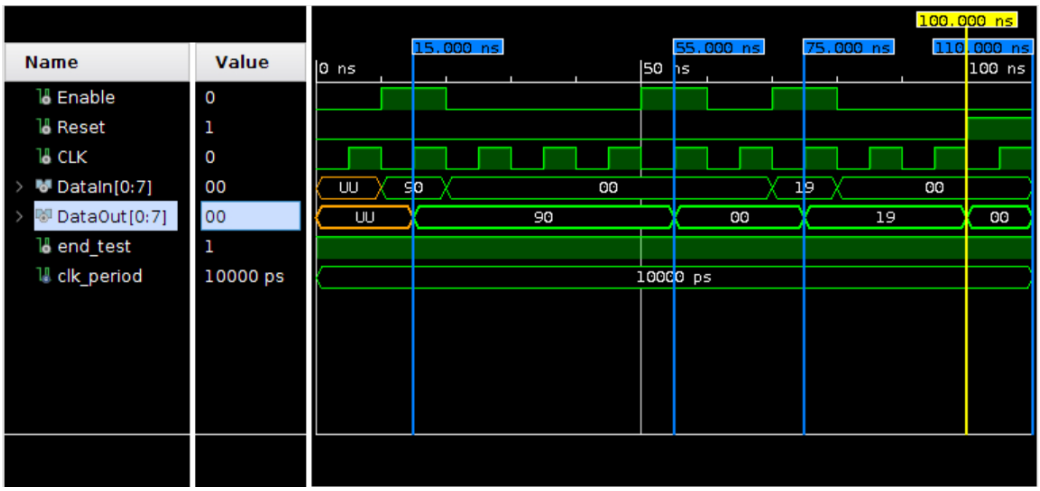
4.1.2 Simulering af Flipflop modul



Figur 3: Flipflop

Som det ses ved 15 ns på figur 3, skifter DataOut til 1 ved den opadgående CLK, da Enable og DataIn begge er høje. DataOut resetter ved 55 ns, hvor enablebittet igen sættes, og DataIn er 0. Ved 100 ns ses også, reset bittet sættes, så DataOut sættes lavt.
Dermed fungerer komponenten som, flipfloppen er tænkt designet.

4.1.3 Simulering af RegisterCell modul



Figur 4: RegisterCell

I figur 53 ses et registers simulering. Ved 15 ns er enable-bittet højt ved opadgående CLK-flanke, og DataIn (her 0x90) læses ind i registret, og den

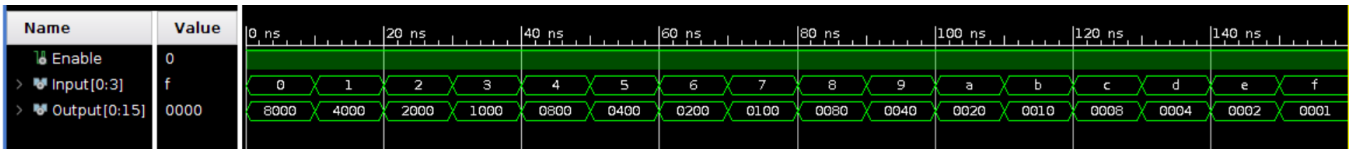
holder DataOut som den indlæste værdi, selvom DataIn skifter værdi. DataIn læses først ind i registret ved 55 ns, hvor enable-bittet igen går højt ved opadgående CLK-flanke. Det ses igen ved 75 ns, men ved 100 ns sker det asynkront. Det er for Reset-bittet sættes, som er en asynkron reset af registret til 0x00.
Alt forløber dermed som det skal.

4.1.4 Simulering af SubDecoder modul

Figur 5: 2-4 dekoder

I figur 5 ses den basale 2-4 dekoder, som er underkomponent til 4-16 dekoderen. Som forventet svarer hver inputværdi til en forskellig output bit, der går højt.

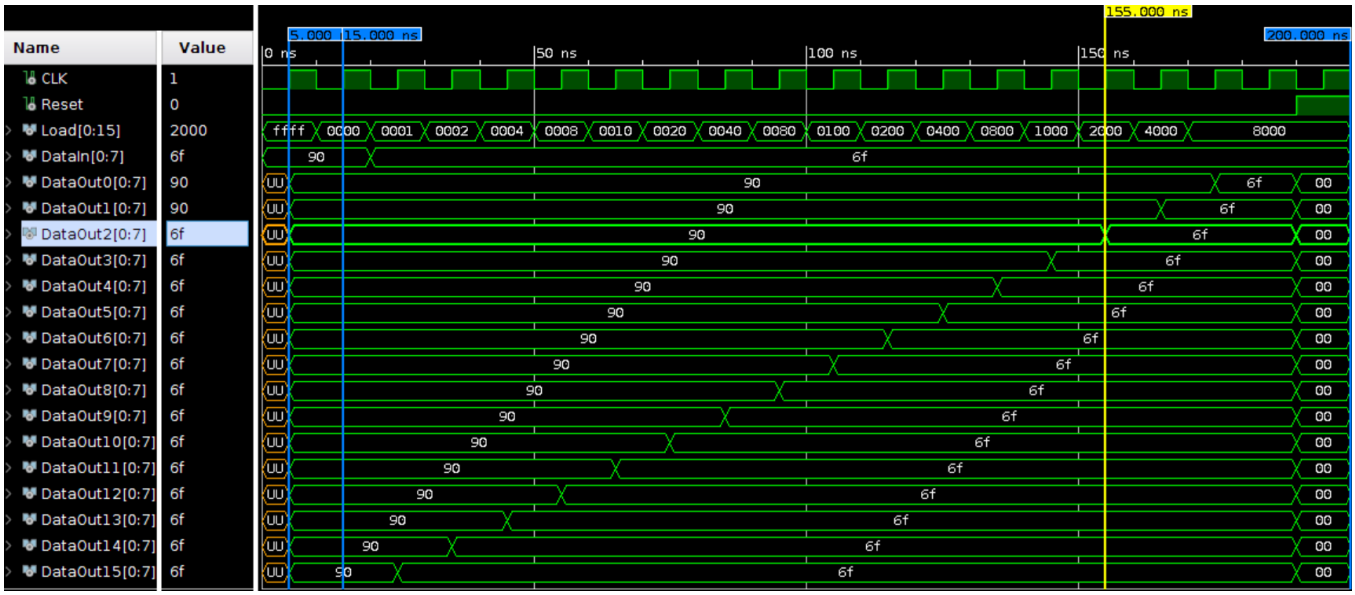
4.1.5 Simulering af 4-16 dekoder modul



Figur 6: 4-16 dekoder

Ligesom i sidste afsnit ses det i figur 6, at hver inputværdi svarer til en forskellig outputbit, præcis som den skal.

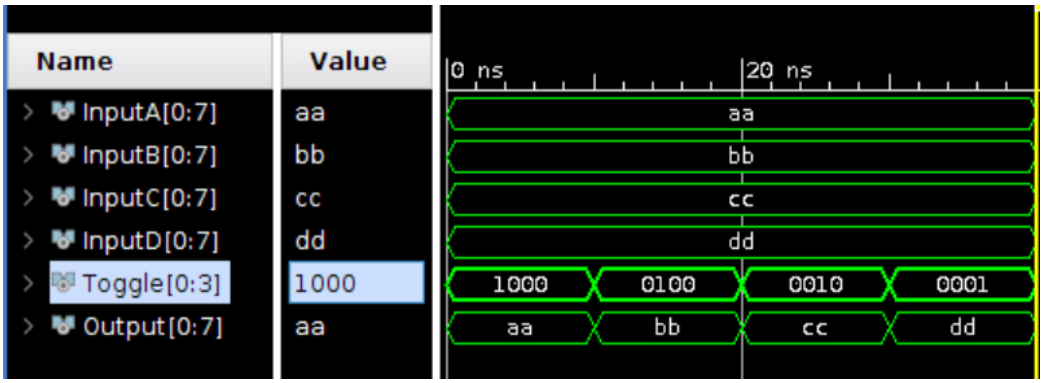
4.1.6 Simulering af RegisterLogic modul



Figur 7: RegisterLogic

Her ses samarbejdet mellem de 16 registerceller og dekoderen. I tredje række cykles Load igennem den 16 muligheder, først går mindst betydende bit højt, så næstmindst og så videre. Men først simuleres alle 16 bit højt på samme tid, som det ses ved 5 ns. Det vil selvfølgelig aldrig kunne lade sig gøre, da dekoderen kun har ét bit højt ad gangen, men for simuleringen "snydes" der, så der fra start kan være loadet en værdi ind i registrene.
Ved 15 ns loades intet ind i nogle registre, hvad der også forventes, da ingen loadbittene er høje. For resten af forløbet opdateres Load mellem hver CLK puls, og ved CLK-pulsen opdateres det næste register i rækken. Det ses for eksempel ved 155 ns, hvor tredjemest betydende bit i Load er sat og tredje register loades med 0x6f. Her der dog en uhensigtsmæssig navngivning, hvor mest betydende loadbit enabler mindst betydende register. Dette ændrer ikke funktionaliteten, hvorfor det i dette projekt ikke ændres, men kan være forvirrende, så til fremtidige projekter ønskes det rettet.

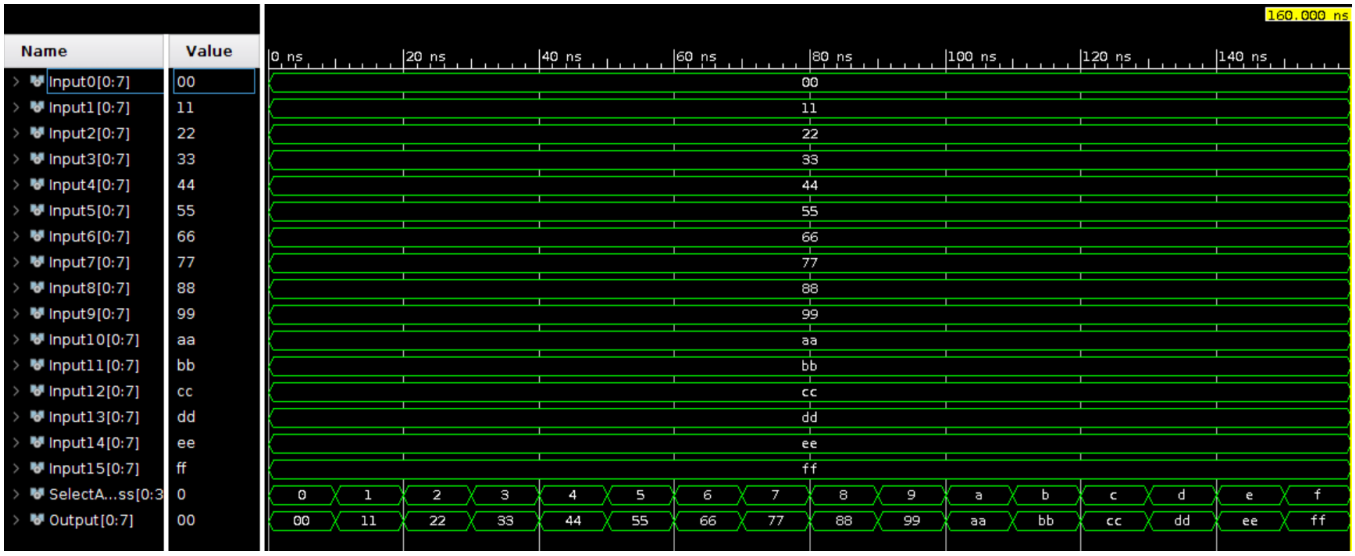
4.1.7 Simulering af 4-1 Multiplexer



Figur 8: 4-1 MUX

I figur 8 ses 4-1 multiplexeren, som anvendes til at bygge den større 16-1 multiplexer. Det ses, at hver Toggle-bit styrer, hvilket af de 4 input, der kommer ud på outputtet.

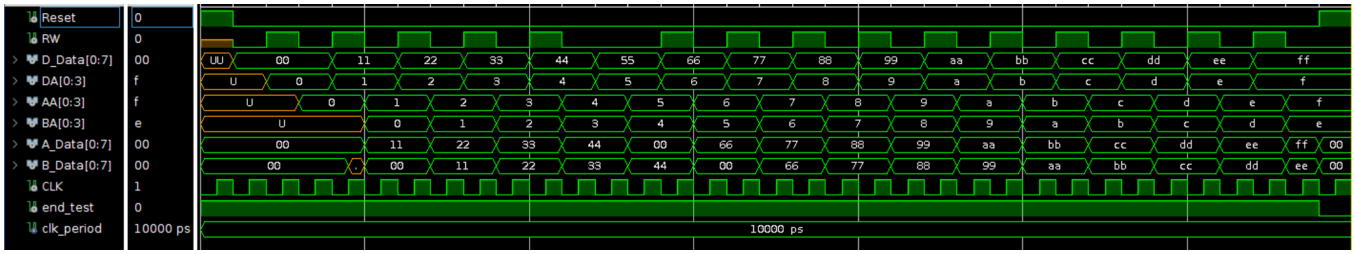
4.1.8 Simulering af 16-1 Multiplexer



Figur 9: 16-1 MUX

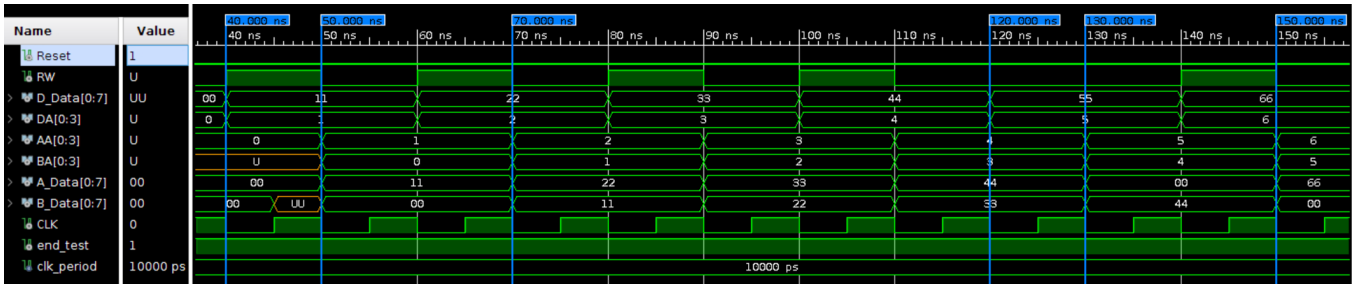
Det ses i figur 9, at der skiftes mellem inputværdierne alt efter Select-vektoren, præcis som det skal.

4.1.9 simulering af RegisterFile modul



Figur 10: RegisterFile modul

På figur 10 ses simuleringen af det fulde RegisterFile modul, hvor forskellige værdier læses ind i de forskellige registre. Selve registerindlæsningen ses dog ikke her, men kan ses i simuleringen af RegisterLogic. I denne situation ses dog, at der kun læses ind i et register, hvis RW også er høj ved indlæsningstid. Der ses også sammenhængingen mellem adressetilskrivningen, og hvilket registerinput på A og B, der kommer ud som output i A_Data og B_Data.



Figur 11: RegisterFile modul med datanedslag

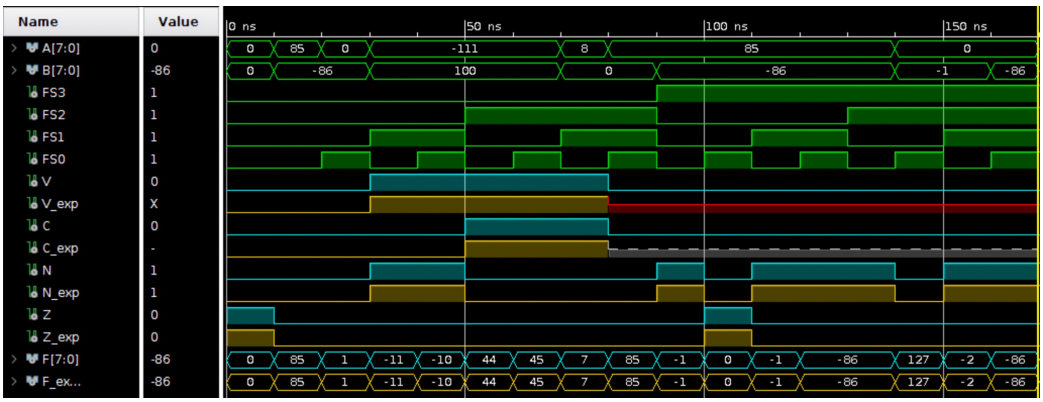
På figur 11 er der blevet zoomet ind på et område af simuleringen fra figur 10 samt lavet nogle datanedslag. Ved 50 ns skiftes AA til 1, som styrer MUXA, og der udskrives indholdet i register 1. Det ses i anden og tredje linje ved 45 ns, at der er blevet addresseret til netop dette register med værdien 0x11, hvor CLK går højt og RW er høj.
Ved 70 ns skiftes AA til 2 og BB til 1. Der læses dermed både på register

2 og register 1, som har henholdsvis loadet den nye værdi 0x22 og holdt den gamle værdi 0x11. Det virker dermed efter tanken om at kunne skrive en adresse til to forskellige MUX og outputte to forskellige værdier fra to forskellige registre.
Ved 120 ns går RW ikke højt, så det forventes ikke, at 0x55 læses ind i register 5, og ved 130 ns ses det netop også, at A_Data outputter 00 i stedet for 0x55. Ved 150 ns ses det igen, at register er opdateret med værdien 0x66, da RW igen gik højt. Og register 5 udskrives fra B_Data og har fortsat værdien 00.

4.2 Simulering af Function unit modul

Til test af dette modul er der benyttet forskellige metoder afhængig af hvad der gav mest mening i forhold til den enkelte komponent. Fælles for dem alle er at inputs er vist med grøn og outputs med blå. For at minimere eventuelle aflæsningsfejl, og for at gøre testen lettere at gennemskue, så er der i nogen diagrammer også vist et forventet output i gul. Dette forventede output bliver enten beregnet ud fra kode i testbenchen og ellers er det blevet regnet på papair og der efter indsat direkte i testbench koden.

4.2.1 simulering af topmodule function unit

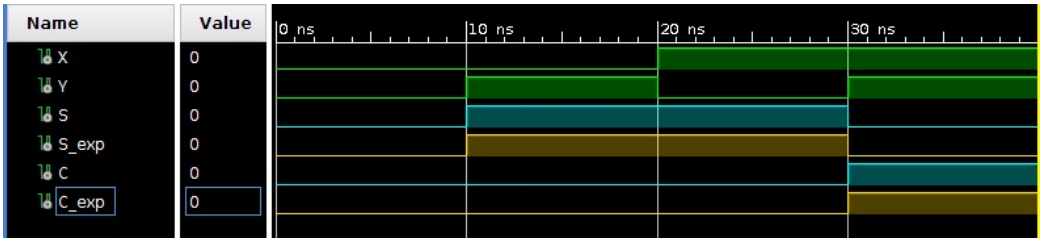


Figur 12: Funktion unit: Grøn: inputs, Blå: outputs, Gul: forventet output, Rød: don't care.

Komponenten kan lave 16 forskellige regneoperationer, og de testes alle sammen. Dette gøres ved at sætte 2 vilkårligt tal på de 2 inputs A og B, og

derefter gennemløbe samtlige input kombinationer på de 4 FS inputs. Alle regneoperationer er først blevet udført på papir og resultaterne er så blevet indsat i testbenchen for let at kunne sammenligne. Ved at analysere diagrammet ses det at komponentens output stemmer med de forventede outputs gennem hele testen.

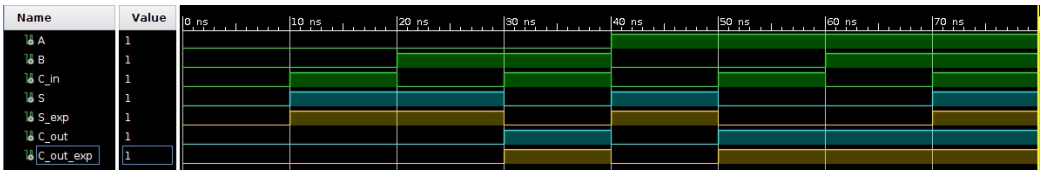
4.2.2 simulering af submodul Half adder



Figur 13: Half adder: Grøn: inputs, Blå: outputs, Gul: forventet output, Rød: don't care.

Her gennemløbet samtlige inputkombinationer og ved at sammenligne de 2 output med de 2 forventede, ses det at komponenten fungerer som den skal.

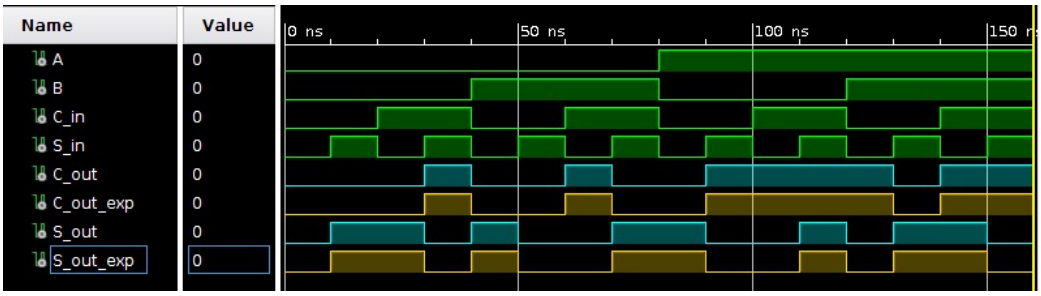
4.2.3 simulering af submodul Full adder



Figur 14: Full adder: Grøn: inputs, Blå: outputs, Gul: forventet output.

I testen gennemløbes samtlige inputs kombinationer, og det forventede out beregnes af testbenchkoden. Det ses at komponentens output stemmer med de forventede outputs gennem hele testen.

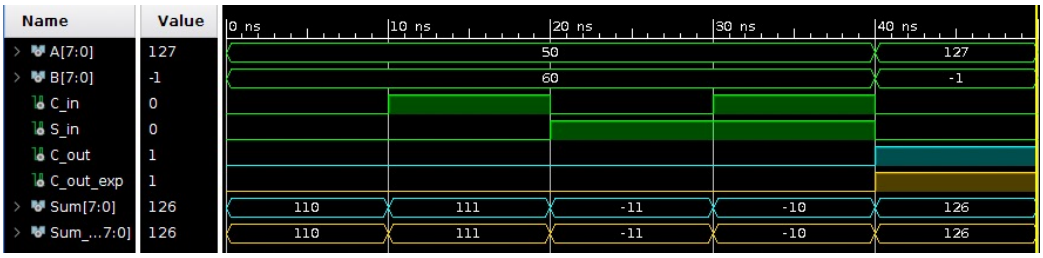
4.2.4 simulering af submodul Adder_Subtractor



Figur 15: Adder Subtractor: Grøn: inputs, Blå: outputs, Gul: forventet output.

I denne test gennemløbes samtlige inputkombinationer, og det forventede output beregnes af softwaren i testbenchkoden. Der er på den måde let at sammenholde komponentens output med det forventede out, og det ses på diagrammet at de to følges ad gennem hele testen.

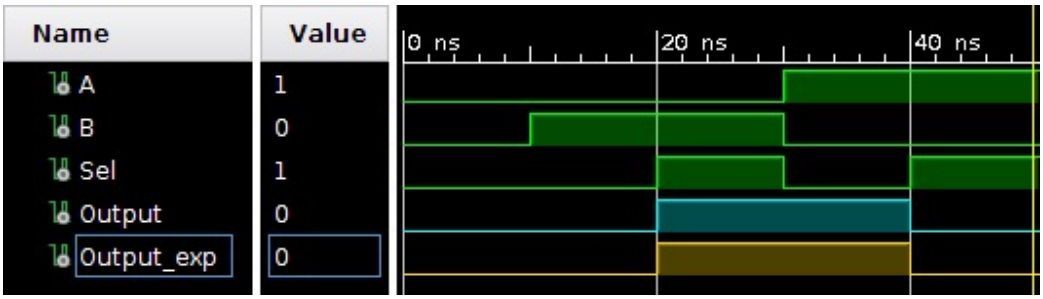
4.2.5 simulering af submodul ADD_SUBB_8bit



Figur 16: ADD SUBB 8bit: Grøn: inputs, Blå: outputs, Gul: forventet output

Værdierne på diagrammet er vist i signed decimal.
0-10 ns: forventet output = A + B = 50 + 60 = 110
10-20 ns: forventet output = A + B + 1 = 50 + 60 + 1 = 111
20-30 ns: forventet output = A – B – 1 = 50 – 60 – 1 = -11
30-40 ns: forventet output = A – B = 50 – 60 = -10
Til sidst testes C_ out funktionen fra 40 ns. Det ses at komponentens output følger det forventede output gennem hele testen.

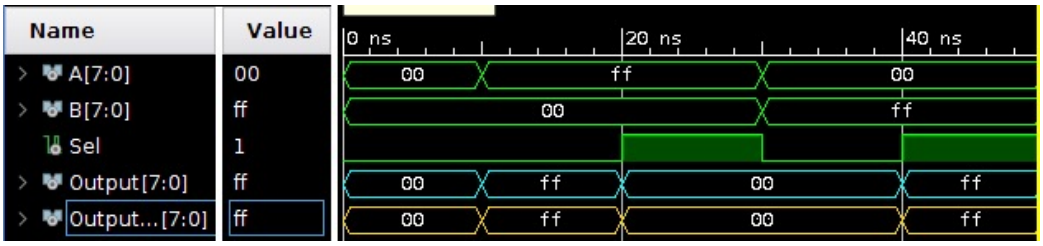
4.2.6 simulering af submodul MUX2x1



Figur 17: MUX2x1: Grøn: inputs, Blå: outputs, Gul: forventet output, Rød: don't care.

Når Sel er lav skal A komme ud på Outputtet, dette ses fra 10-20 ns og igen ved 30-40 ns. Når Sel er høj skal B komme ud på outputtet, og dette ses den resterende tid af testen.

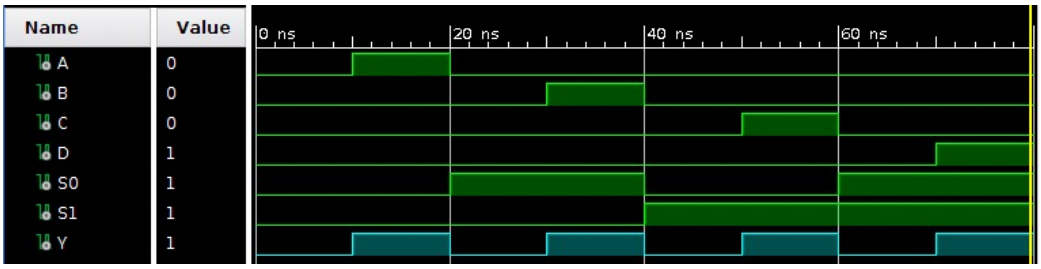
4.2.7 simulering af submodul MUX2x1x8



Figur 18: MUX2x1x8: Grøn: inputs, Blå: outputs, Gul: forventet output, Rød: don't care.

Når Sel er lav skal A komme ud på Outputtet, dette ses fra 10-20 ns og igen ved 30-40 ns. Når Sel er høj skal B komme ud på outputtet, og dette ses den resterende tid af testen.

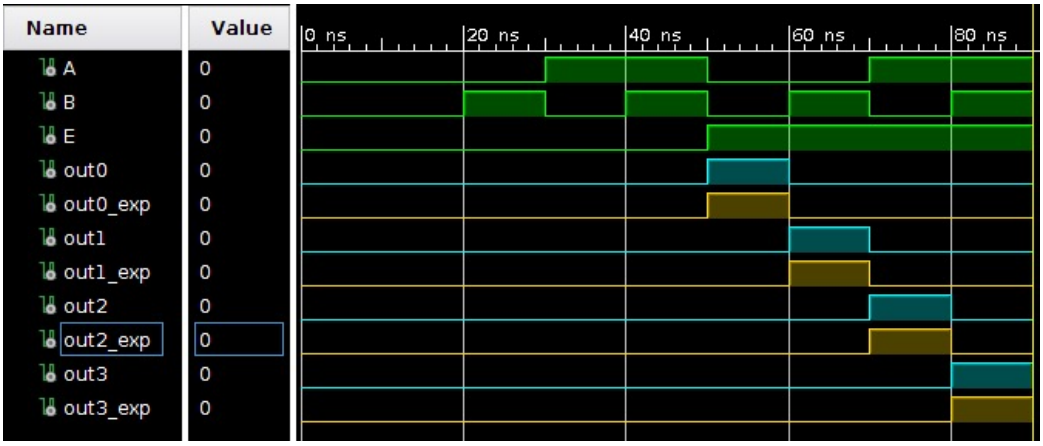
4.2.8 simulering af submodul MUX4x1



Figur 19: MUX4x1: Grøn: inputs, Blå: outputs, Gul: forventet output, Rød: don't care.

De to inputsignaler S0 og S1 gennemløber alle 4 inputkombinationer. Fra 0-20 ns forventes A at komme ud på Y, Fra 20-40 ns forventes B at komme ud på Y, fra 40-50 ns forventes C at komme ud på Y og fra 60-80 ns forventes D at komme ud på Y. Dette er det samme vi ser når vi kigger på diagrammet.

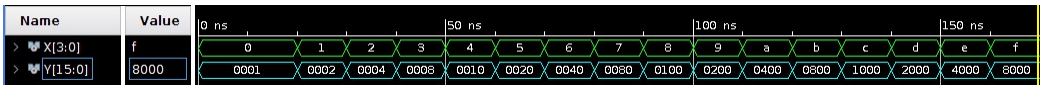
4.2.9 simulering af submodul dekoder2x4



Figur 20: decoder2x4: Grøn: inputs, Blå: outputs, Gul: forventet output.

Komponenten har kun 3 std_ logic input, og der udføres derfor en test af samtlige input kombinationer. Det ses at komponentens out stemmer med de forventede output.

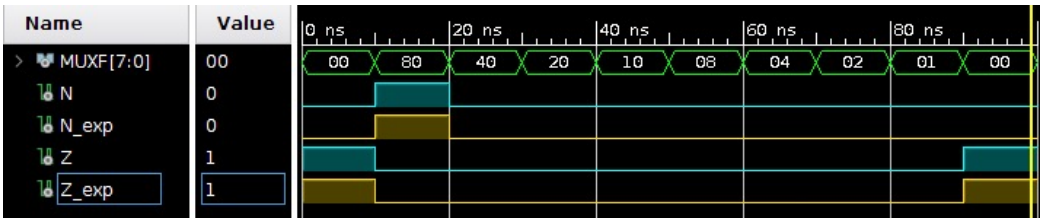
4.2.10 simulering af submodul dekoder4x16



Figur 21: decoder4x16: Grøn: inputs, Blå: outputs, Gul: forventet output.

Denne test udføres ved at gennemløbe samtlige inputkombinationer og derefter analysere outputtet. Dataen på diagrammet er vist i HEX og det ses dermed at outputtet, Y, bliver dobbelt så stort hver gang inputtet, X, tælles op. Det samme ses på diagrammet.

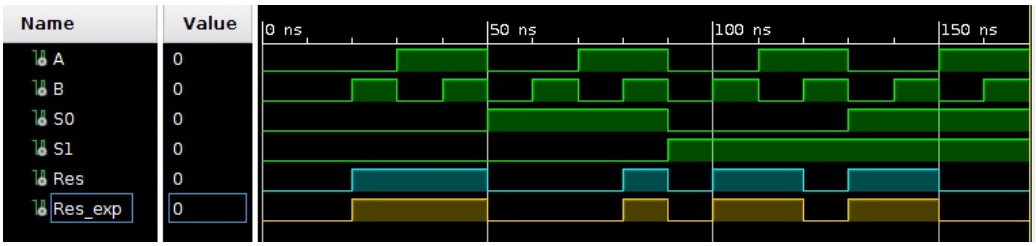
4.2.11 simulering af submodul NegZero



Figur 22: NegZero: Grøn: inputs, Blå: outputs, Gul: forventet output, Rød: don't care.

Inputtet MUXF er vist i hex. Komponenten skal sættes N høj når den mest betydende bit i MUXF er høj. Det vil på diagrammet være når den mest betydende af de to 2 hexadecimal tal er større end 8. Dette sker mellem 10-20 ns og det ses her at N går høj, og at N er lav for alle andre input værdier. Komponenten skal sætte Z høj når samtlige inputs er lave. Dette er tilfældet fra 0-10 ns og igen fra 90-100 ns, og det ses her at Z går høj som forventet og er lav for alle andre inputkombinationer.

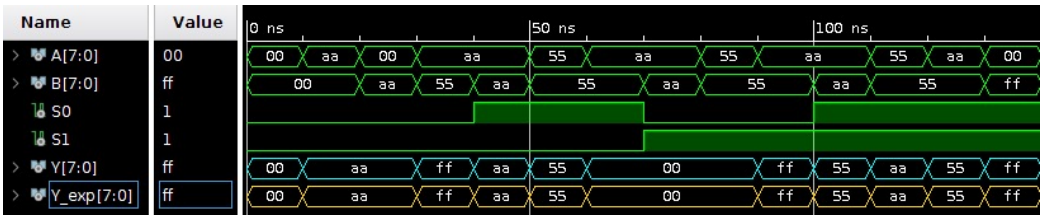
4.2.12 simulering af submodul logic_unit



Figur 23: logic unit: Grøn: inputs, Blå: outputs, Gul: forventet output, Rød: don't care.

Her gennemløbes samtlige inputkombinationer og det ses at outputtet Res stemmer med det forventede output Res_ exp.

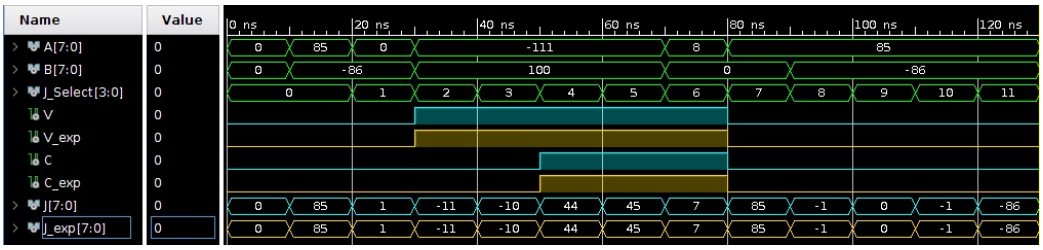
4.2.13 simulering af submodul logic_unit8bit



Figur 24: logic unit 8bit: Grøn: inputs, Blå: outputs, Gul: forventet output, Rød: don't care.

De to select inputs sættes til alle fire mulige kombinationer, og outputtet Y sammelignes med det forventede output Y_ exp. Det er ses at Y stemmer med Y_ exp gennem hele testen, og komponeneten fungerer derfor som den skal.

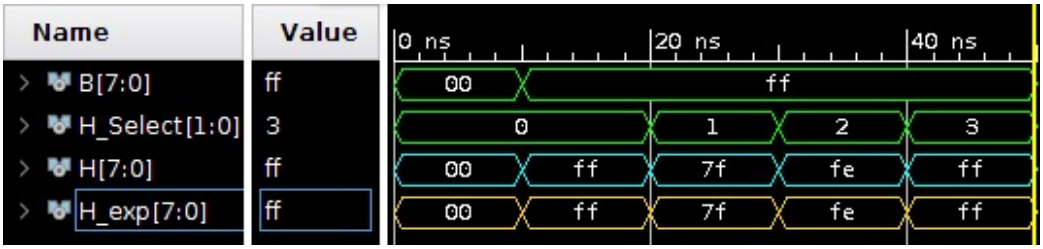
4.2.14 simulering af ALU



Figur 25: ALU: Grøn: inputs, Blå: outputs, Gul: forventet output.

I løbet af denne test sættes J_ Select skiftevis til alle de 12 kombinationer som har en tilhørende operation i ALU'en. Det ses at alle 12 operationer giver et output (blå) som stemmer med det forventede output (gul).

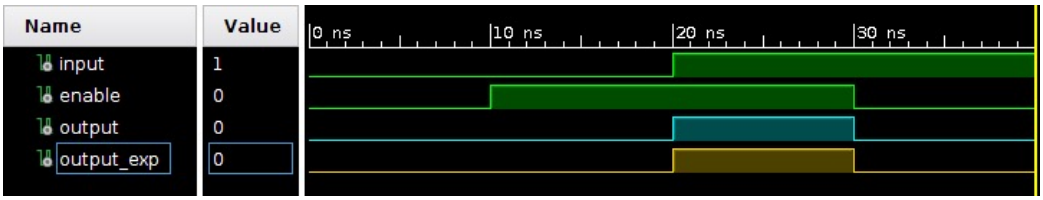
4.2.15 simulering af Shifter



Figur 26: Shifter: Grøn: inputs, Blå: outputs, Gul: forventet output, Rød: don't care.

B, H og H_ exp er vist i HEX. H_ select er vist i unsigned integer.
Komponentens 4 Select muligheder testes. Fra 0-20 ns forventes det at inputtet B kommer ud på H. Fra 20-30 ns skal komponenten bitshifte B til højre. Fra 30-40 ns bitshiftes der til venstre, efter 40 ns skal B igen komme ud på H som den er. Det ses at outputtet stemmer med det forventede output gennem hele testen.

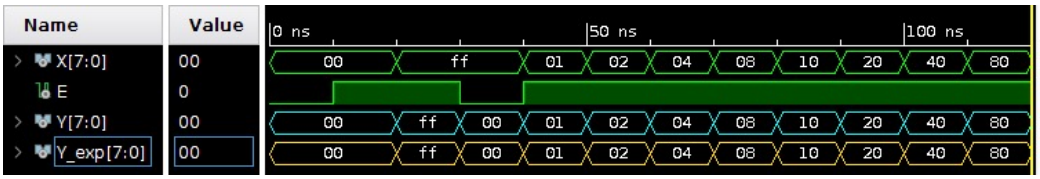
4.2.16 simulering af Enabler



Figur 27: Enabler: Grøn: inputs, Blå: outputs, Gul: forventet output.

Det forventes at inputtet kommer ud på outputtet når enable er høj, og ellers skal outputtet være 0.
0 ns
Enable er lav og outputtet aflæses til lavt som forventet.
10 ns
Enable er nu høj og det ses at inputtet kommer ud på outputtet frem til at enable går lav igen ved 30 ns.

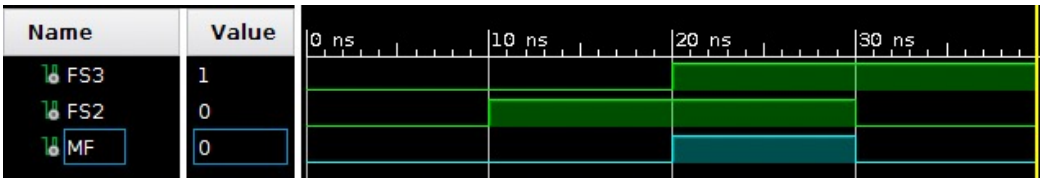
4.2.17 simulering af Enabler 8bit



Figur 28: Enabler 8bit: Grøn: inputs, Blå: outputs, Gul: forventet output.

Det forventes at inputtet, X, kommer ud på outputtet, Y, når enable signalet, E, går høj. Når E ikke er høj skal Y være 00. Fra 0 ns til 10 ns og fra 30 ns til 40 ns er E lav og Y aflæses til 00 som det forventes. I resten af testen er E høj og det aflæses at Y er lig X her som det forventes.

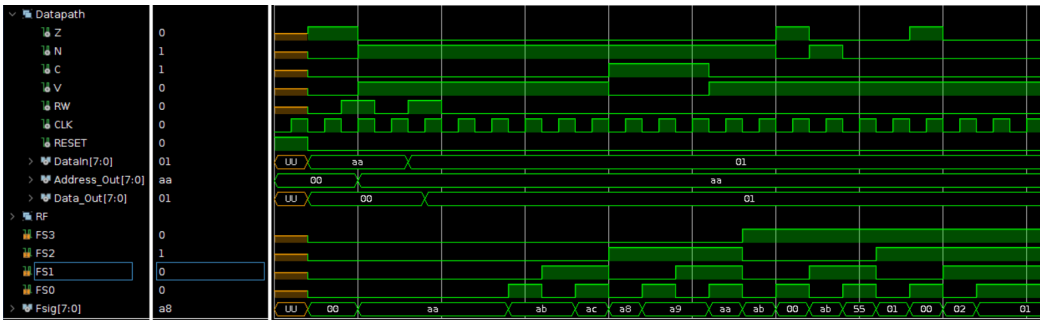
4.2.18 simulering af Function Select



Figur 29: FunctionSelect: Grøn: inputs, Blå: outputs

Outputtet MF forventes at gå høj når begge inputs FS3 og FS2 er høje. Dette ser vi fra 20-30 ns. Det at MF går høj som forventet og er lav i resten af testen.

4.3 Simulation PWA toplevel



Figur 30: Datapath sim

På simulationen kan vi i bunden se Fsig, som er FU'ens output, Z, N, C og V er flag som FU'en outputter, når RW er højt kan der skrives til registre, Address_out er dataen loadet ud fra MUXA og Data_Out er dataen loadet ud fra MUXB.
Vi kan se at når der bliver skrevet til registrene er dataen også tilgængelig på henholdsvis MUXA og MUXB, som herefter bearbejdes i henhold til funktionstabellen givet for FS0-3, vi kan se at ALU kan udføre matematiske operationer på registrenes indhold korrekt, yderligere kan vi se at når FS0-3 bliver stort nok kan den også shifte dataen korrekt, såvel som at Z, N, C og V flagene er høje når de skal være det, ergo fungere datapath modulet.

5 Implementation

Tekst

5.1 Test i FPGA board nexys DDR4

6 Konklusion

Hovedformålet med dette delprojekt var at forstå, hvordan en Datapath fungerer, og hvordan den er bygget. Datastien inkluderer et 16x8-bit Register File (RF) modul, et funktionsenhedsmodul med en aritmetisk logisk enhed (ALU) og en skifter og det tilhørende afkodnings- og multiplekser kredsløb. Registerfilen (RF) indeholder 16 registre og er implementert i VHDL ved hjælp af entiteten RegisterFile. Komponenten FunctionUnit (FU) udfører mikrooperationer på A- og B-operanderne, og 4-bit FunctionSelect-input (FS) bestemmer disse operationer. Implementeringen af RF- og FU-underblokkene blev udført separat for at strukturere designet. Designet blev testet og verificeret for at sikre korrekt funktionalitet. Samlet set gav dette projekt en praktisk forståelse af design af digitale systemer og implementering af en Datapath.

Et af læringsmålene i denn opgave har til forskel fra tidligere opgaver inden for FPGA programering, været at opbygge hele projektet udelukkende ved brug af simpel logik frem for at lave process statements. Dette har medført at gruppen selv har betemt, hvordan componenerne ser ud helt ned på logi-gate niveau. PWA består derfor at mange lag af komponenter som bygger oven på hinanden, og rapporten indeholder derfor mange komponenter og testbenches.

Modulet FunktionUnit er designet, kodet, testet og dokumenteret af Buster Thomas. Modulet RegisterFile er designet, kodet, testet og dokumenteret af Marcus og Valdemar. Arbejdet er fra projektets start blevet indelt således, og de to undergrupper har undervejs sparet med hindanden og hjulpet, hvor der har været brug for det.

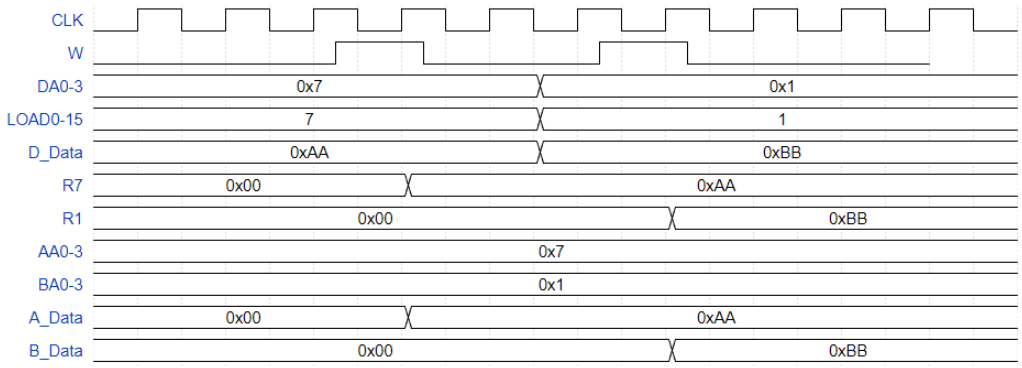
7 Appendix

7.1 Spørgsmål 5

A1=10010001, B1=01100100
A2=10010001, B2=10100011
A3=00001000, B3=11111000
A4=00011000, B4=00100111
A5=00011000, B5=00011000
A6=10011000, B6=10100111
A7=01010011, B7=01110110
A8=01010010, B8=01010101

	Function Unit Output	J select	H select	MF	V	C	N	Z
F=A1+B1	11110101	0010	XX	0	0	0	1	0
F=A2+B2	00110100	0010	XX	0	1	1	0	0
F=A3+B3	00000000	0010	XX	0	1	1	0	1
F=A4-B4	11110001	0101	XX	0	1	0	1	0
F=A5-B5	00000000	0101	XX	0	0	1	0	1
F=A6-B6	11110001	0101	XX	0	0	0	1	0
F=A7 $\oplus$ B7	00100101	1X10	XX	0	0	0	0	0
F=lsB8	10101010	XXXX	10	1	0	0	1	0

7.2 PWA Timing

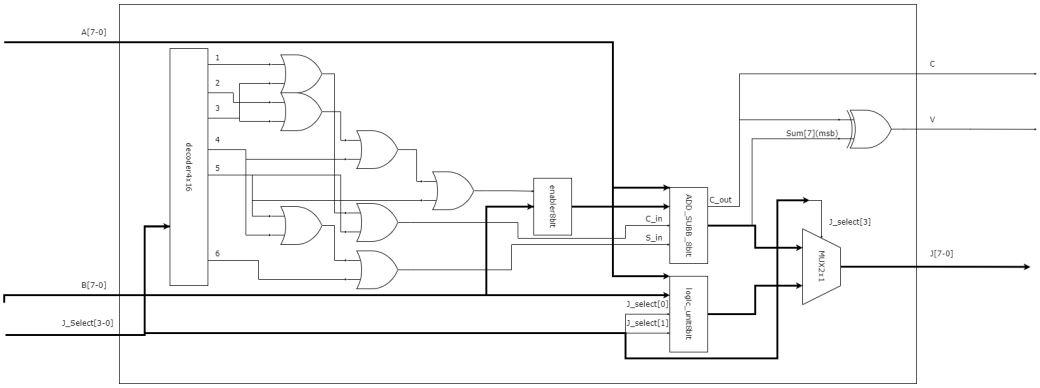


Figur 31: PWA Timing

7.3 Funktion Unit schematic

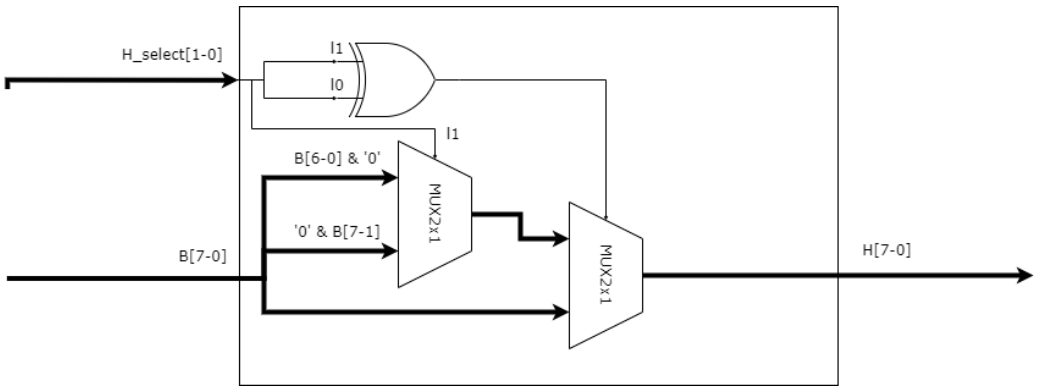
Her er en oversigt af alle de blokke og under blokke som Funktion Unit'en består af:

7.3.1 ALU



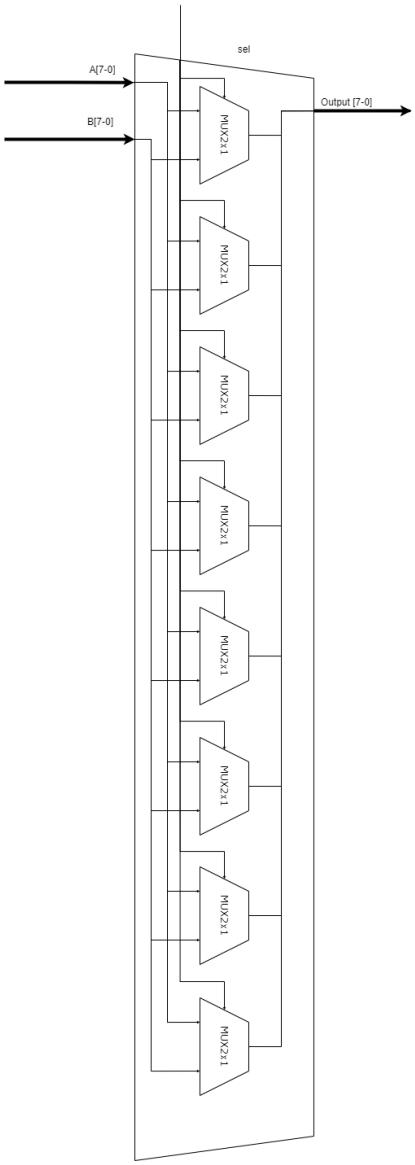
Figur 32: ALU

7.3.2 Shifter



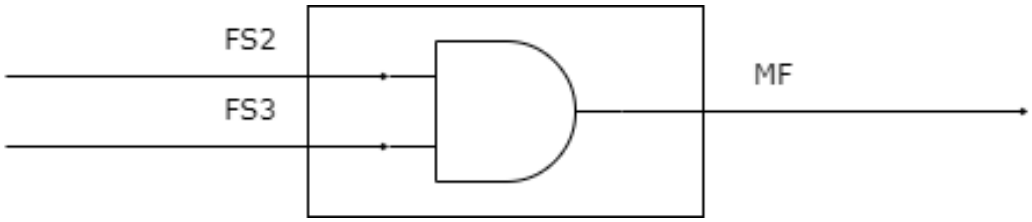
Figur 33: Shifter

7.3.3 MUXF/MUX2x1x8



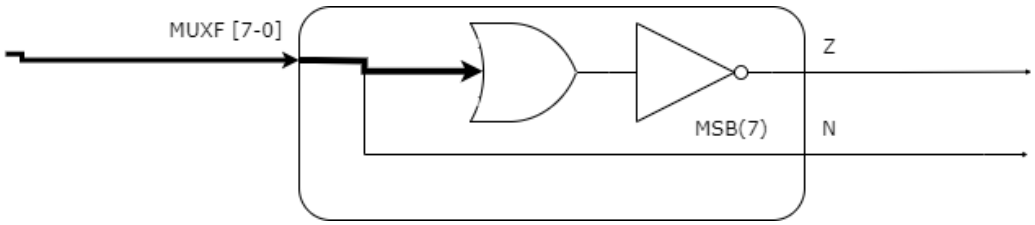
Figur 34: MUX2x1x8

7.3.4 Fsel



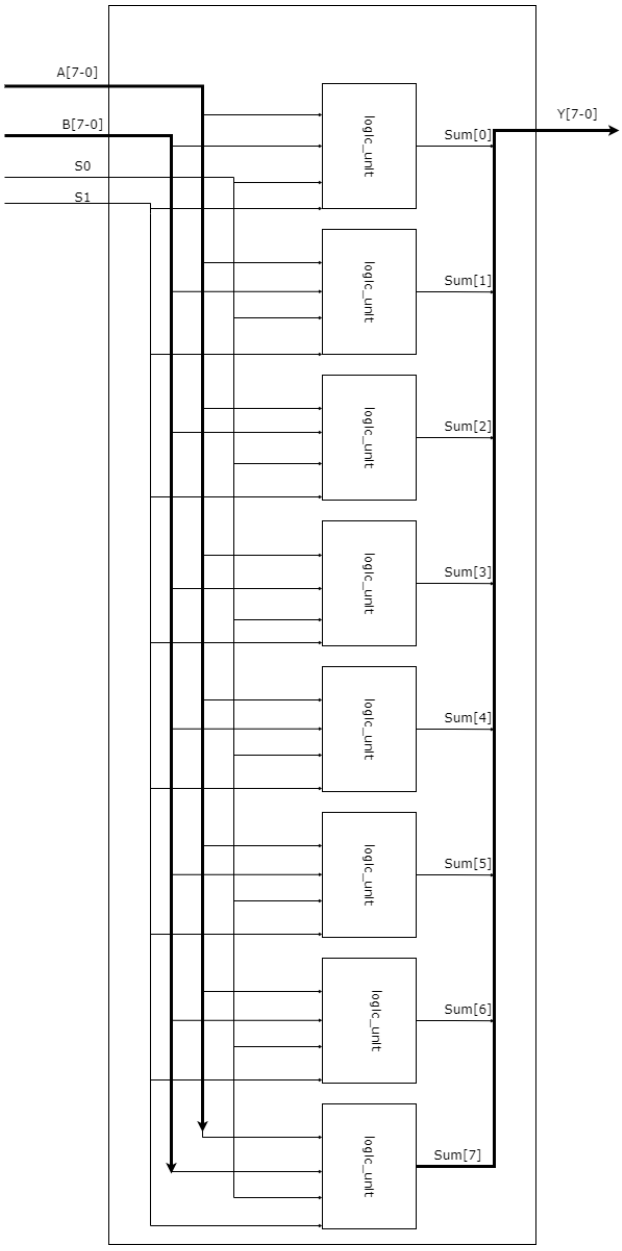
Figur 35: Fsel

7.3.5 NegZero



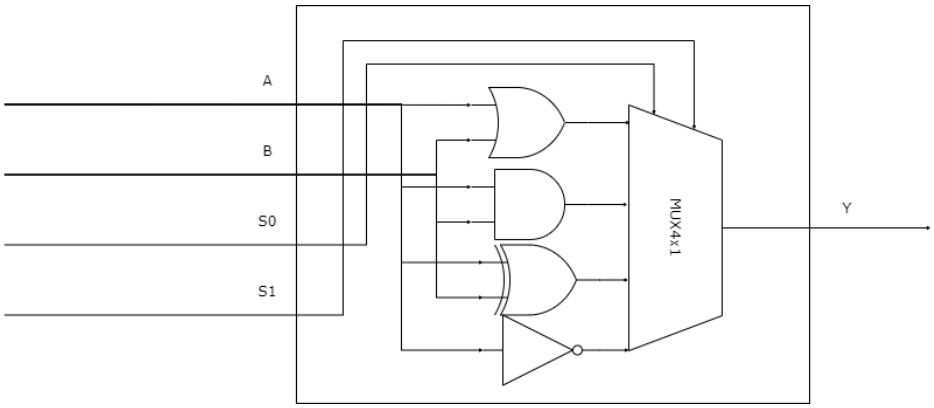
Figur 36: NegZero

7.3.6 logic unit8bit



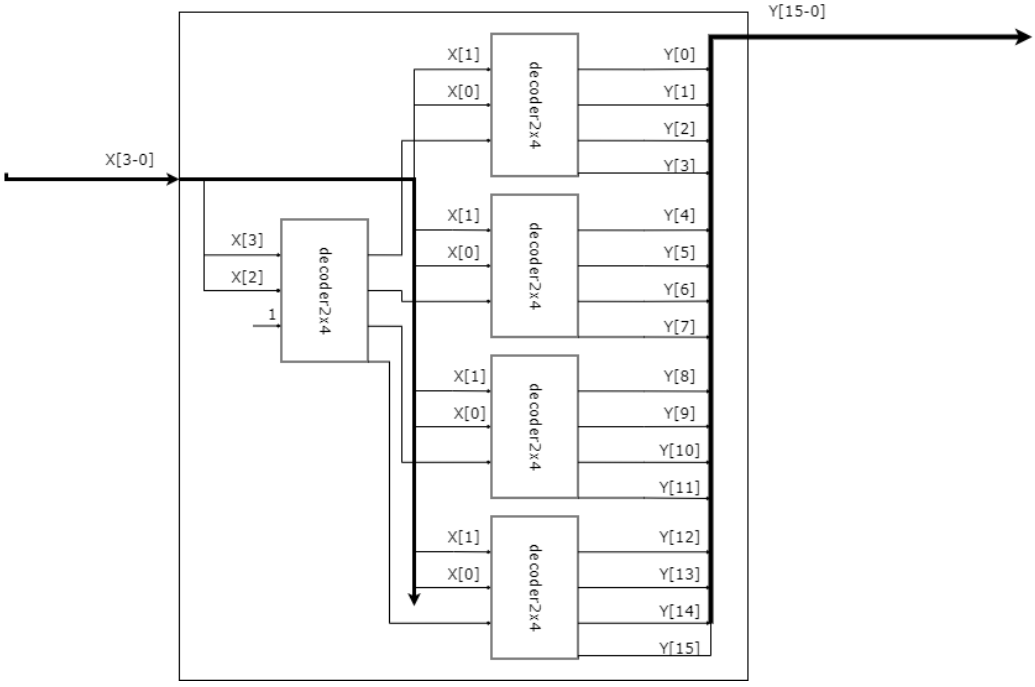
Figur 37: logic unit8bit

7.3.7 logic unit



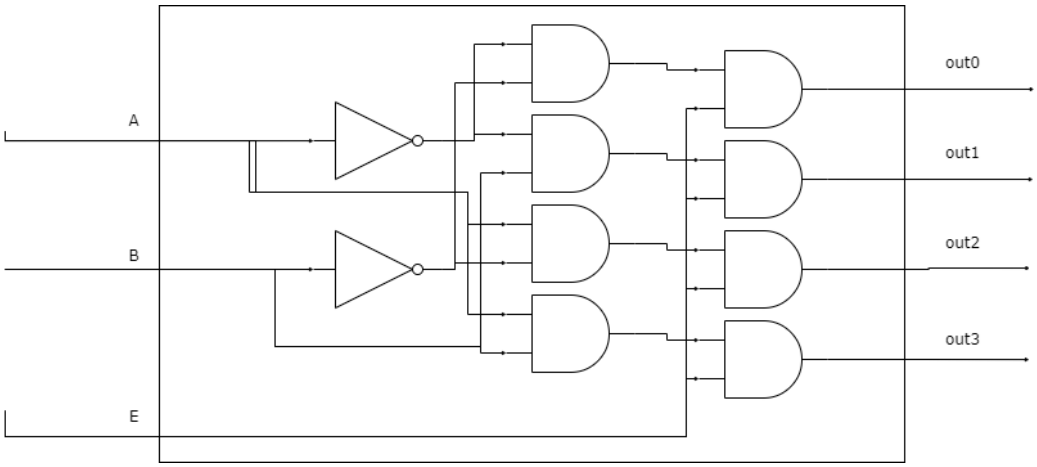
Figur 38: logic unit

7.3.8 decoder4x16



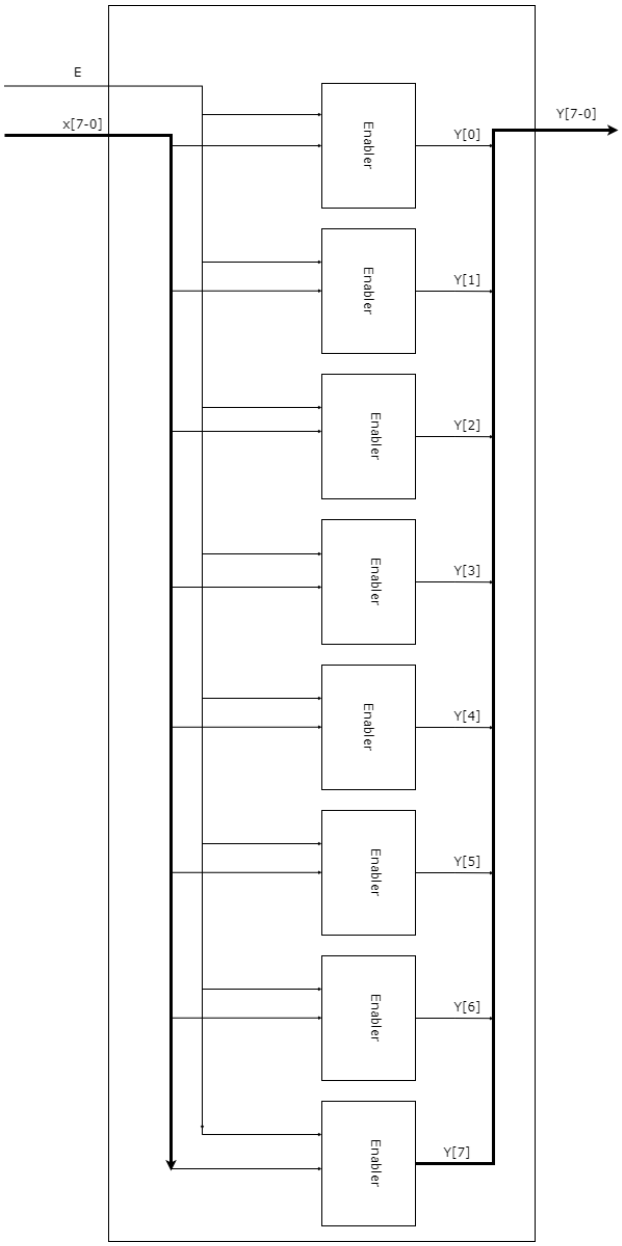
Figur 39: decoder4x16

7.3.9 decoder2x4



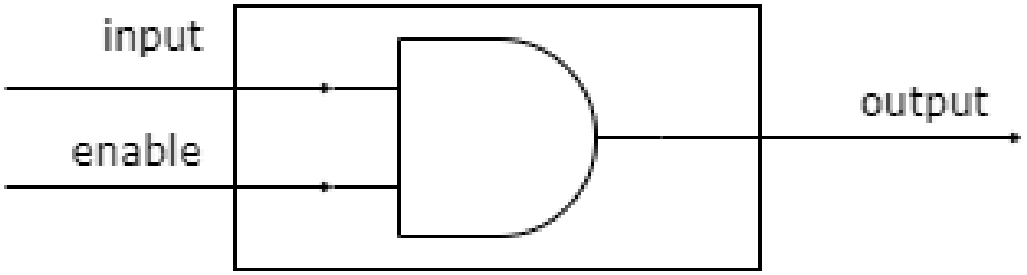
Figur 40: decoder2x4

7.3.10 enabler8bit



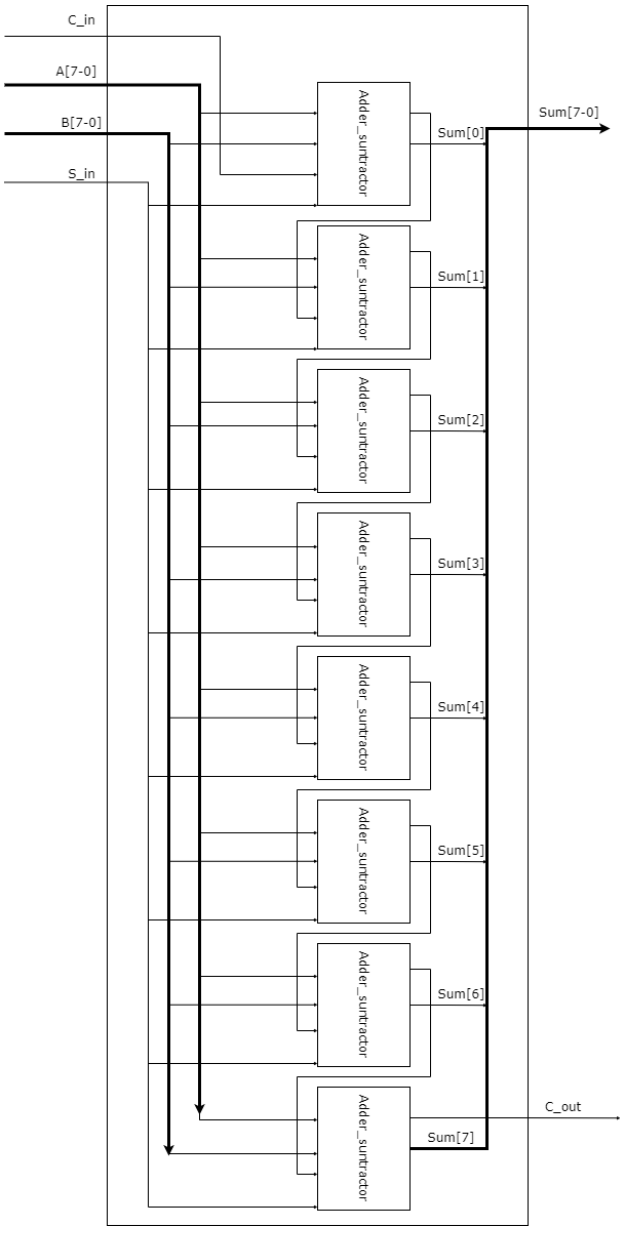
Figur 41: enabler8bit

7.3.11 enabler



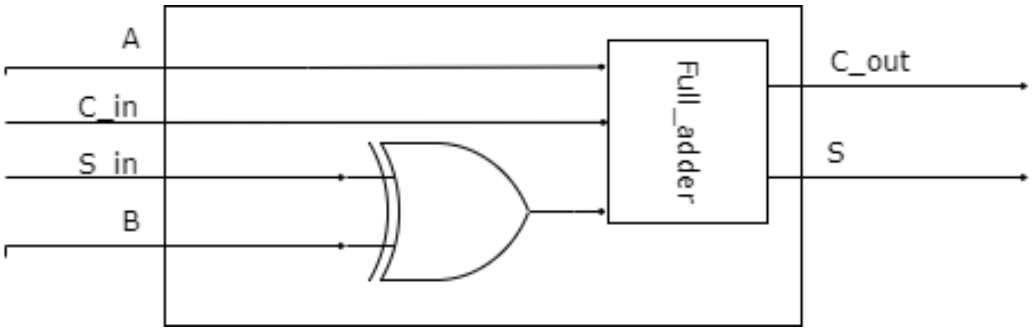
Figur 42: enabler

7.3.12 ADD SUB 8bit



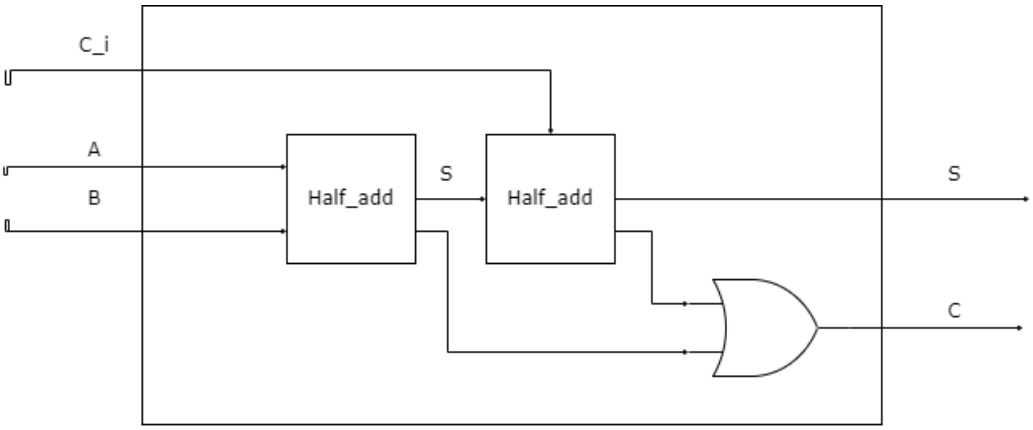
Figur 43: ADD SUB 8bit

7.3.13 Adder Subtractor



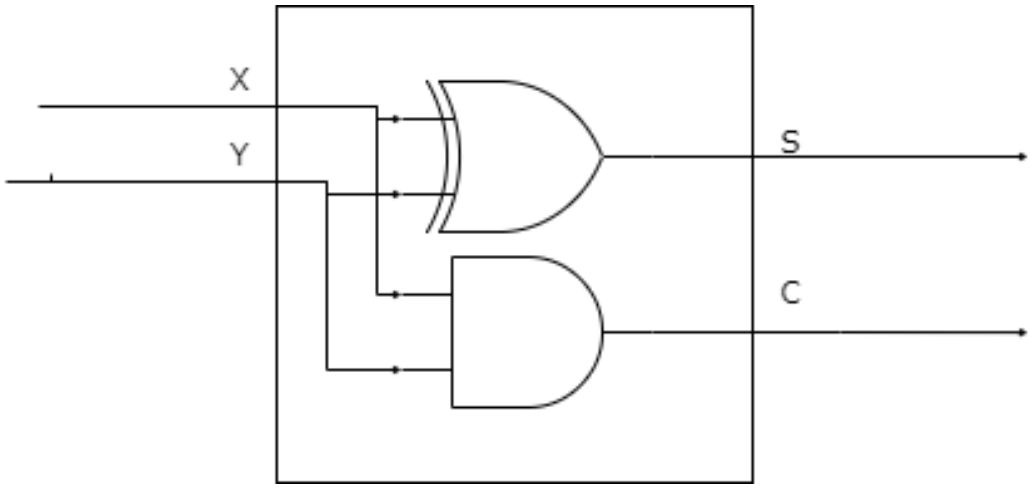
Figur 44: Adder Subtractor

7.3.14 Full adder



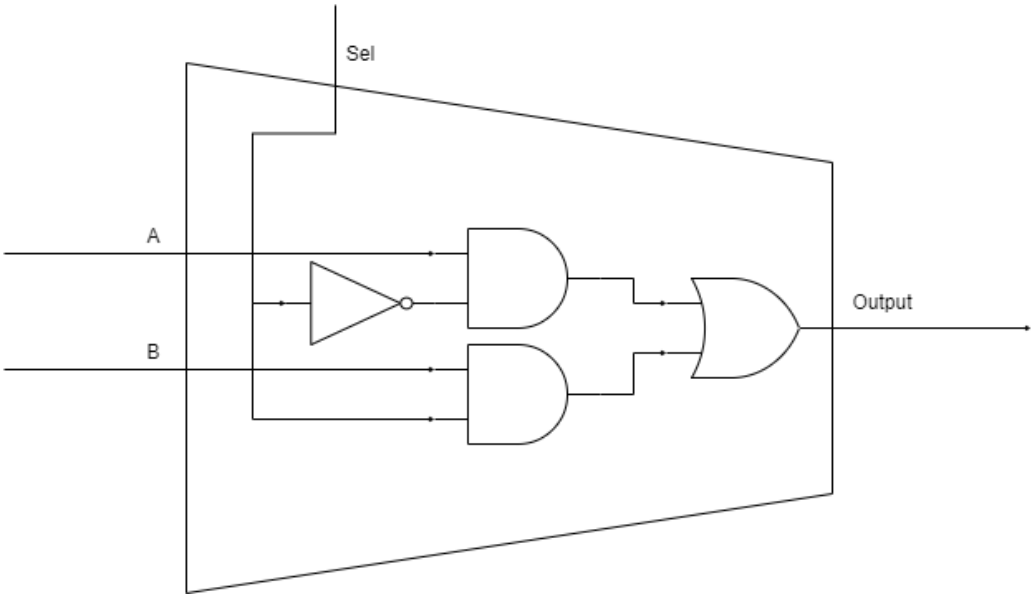
Figur 45: Full adder

7.3.15 Half adder



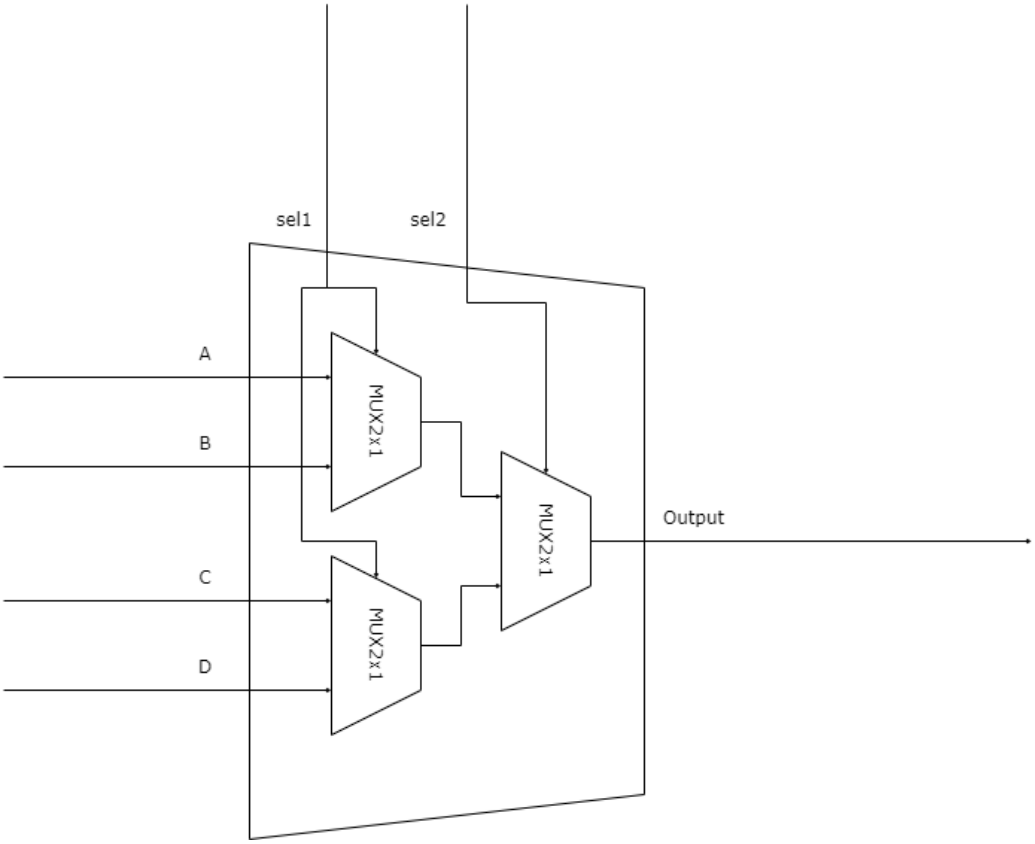
Figur 46: Half adder

7.3.16 MUX2x1



Figur 47: MUX2x1

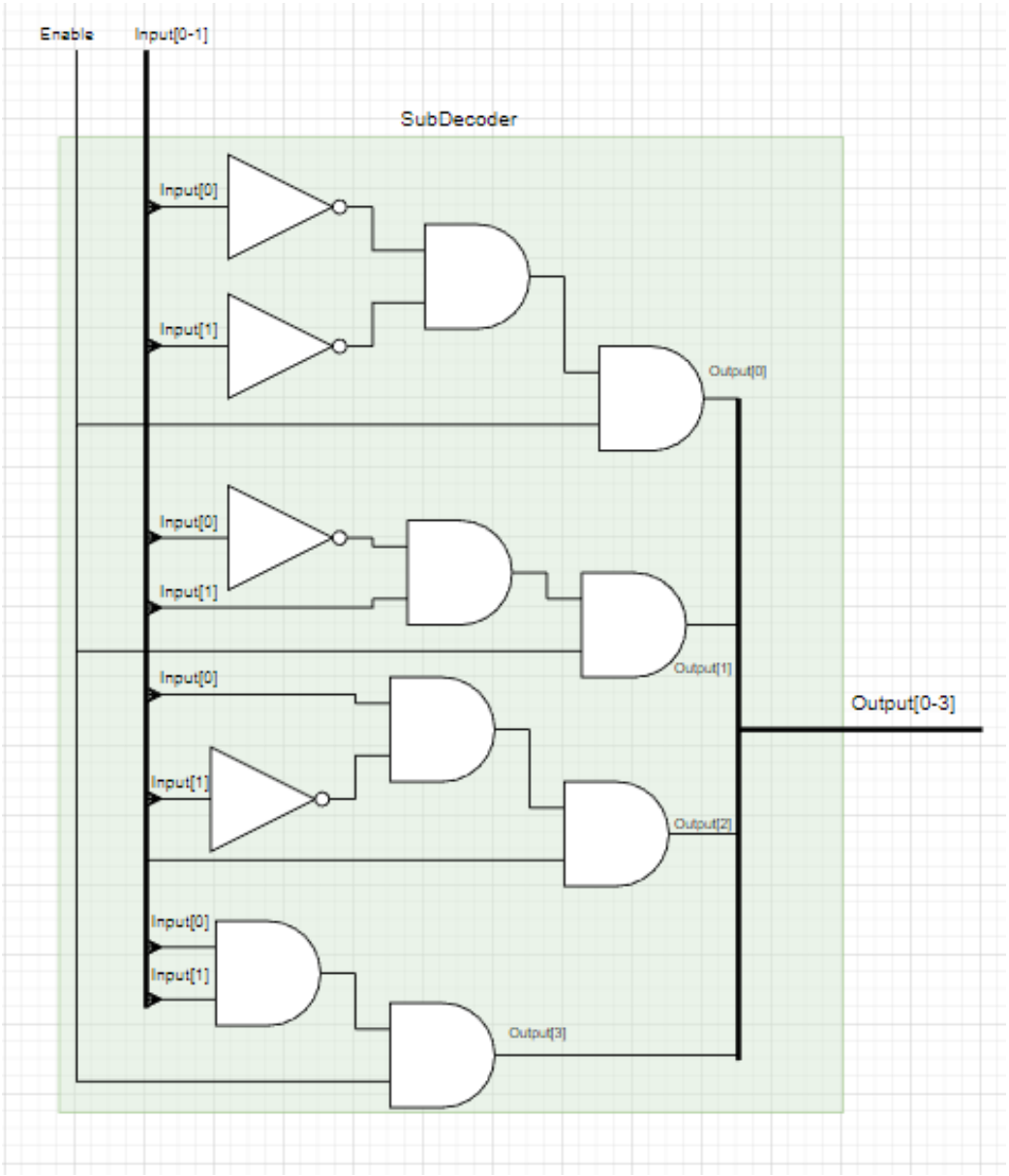
7.3.17 MUX4x1



Figur 48: MUX4x1

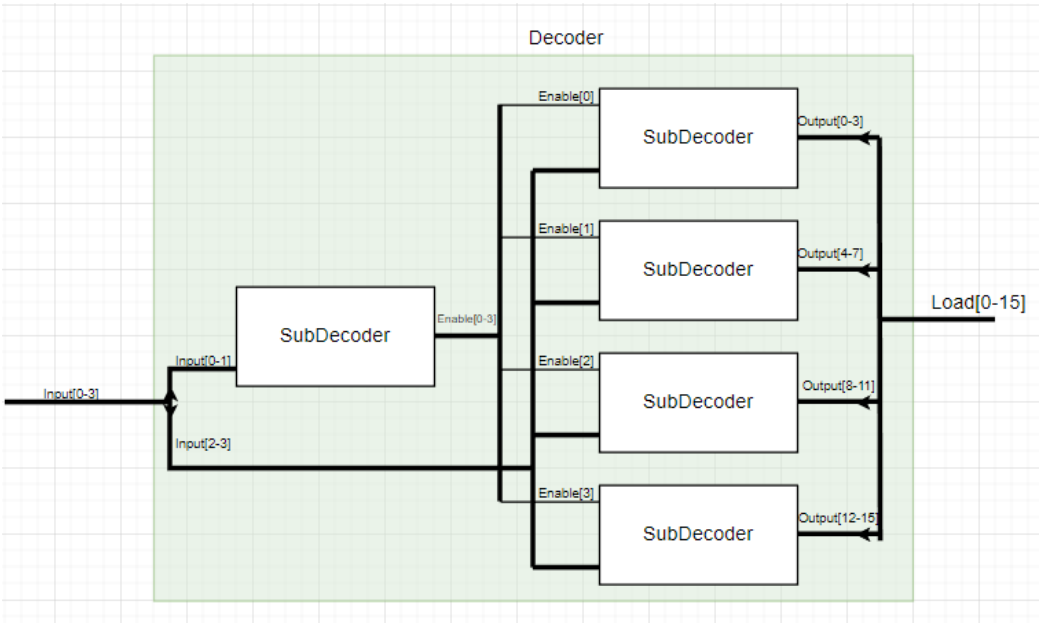
7.4 Register File Schematics

7.4.1 SubDecoder



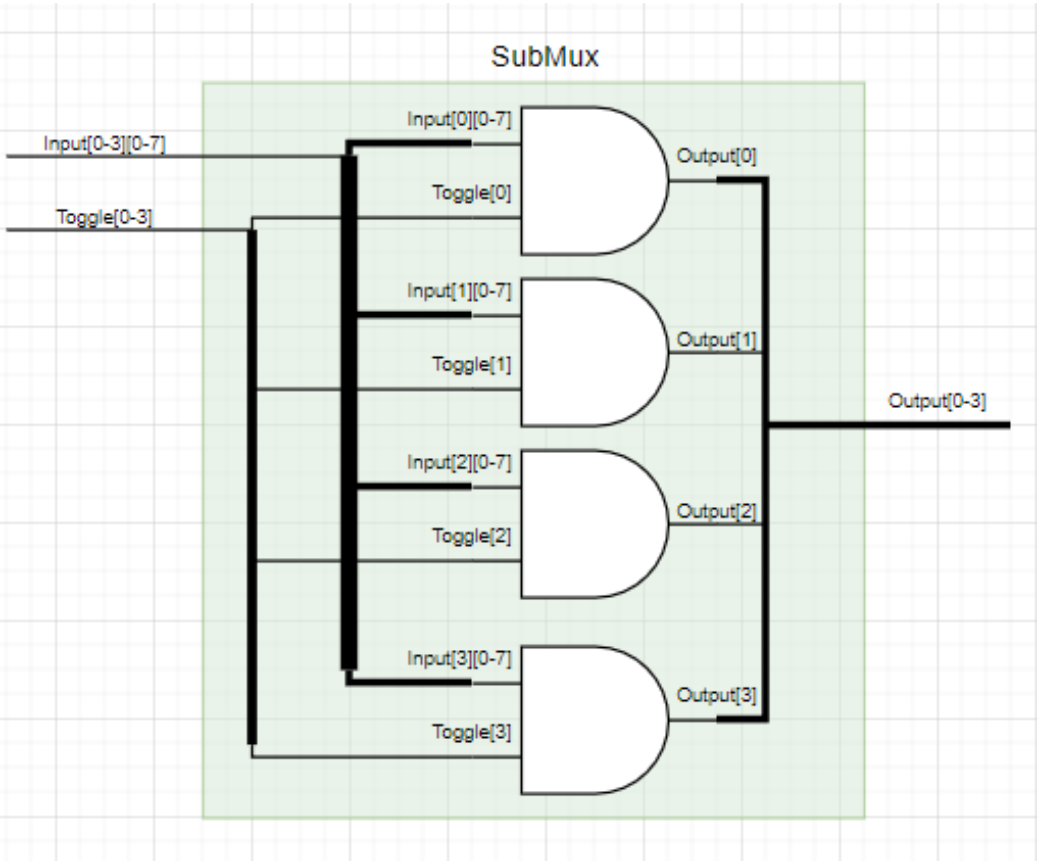
Figur 49: SubDecoder

7.4.2 Decoder



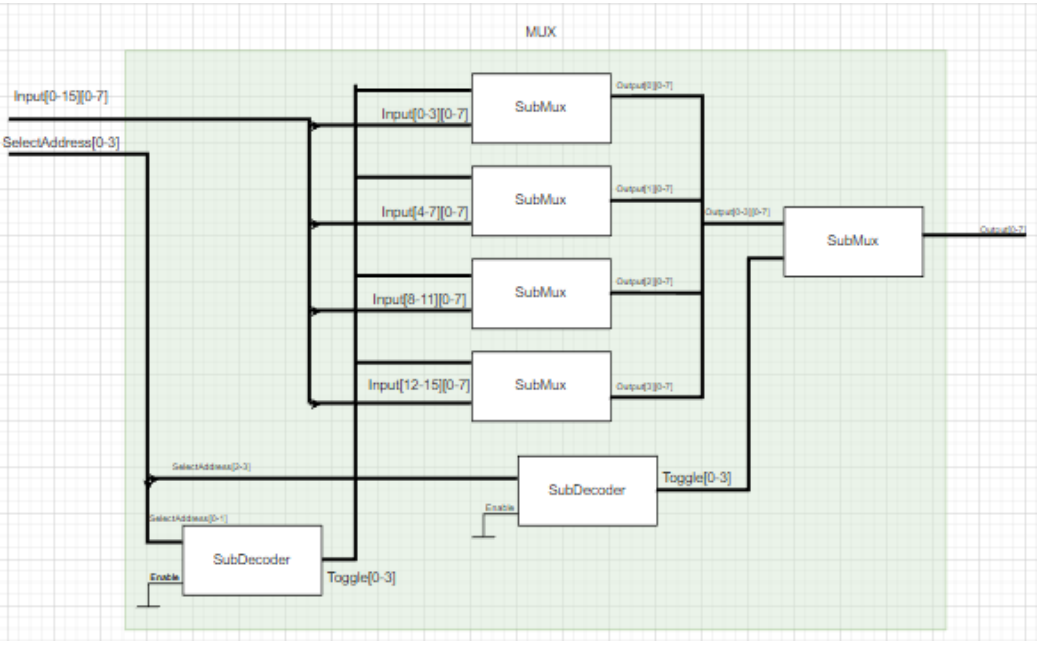
Figur 50: Decoder

7.4.3 SubMux



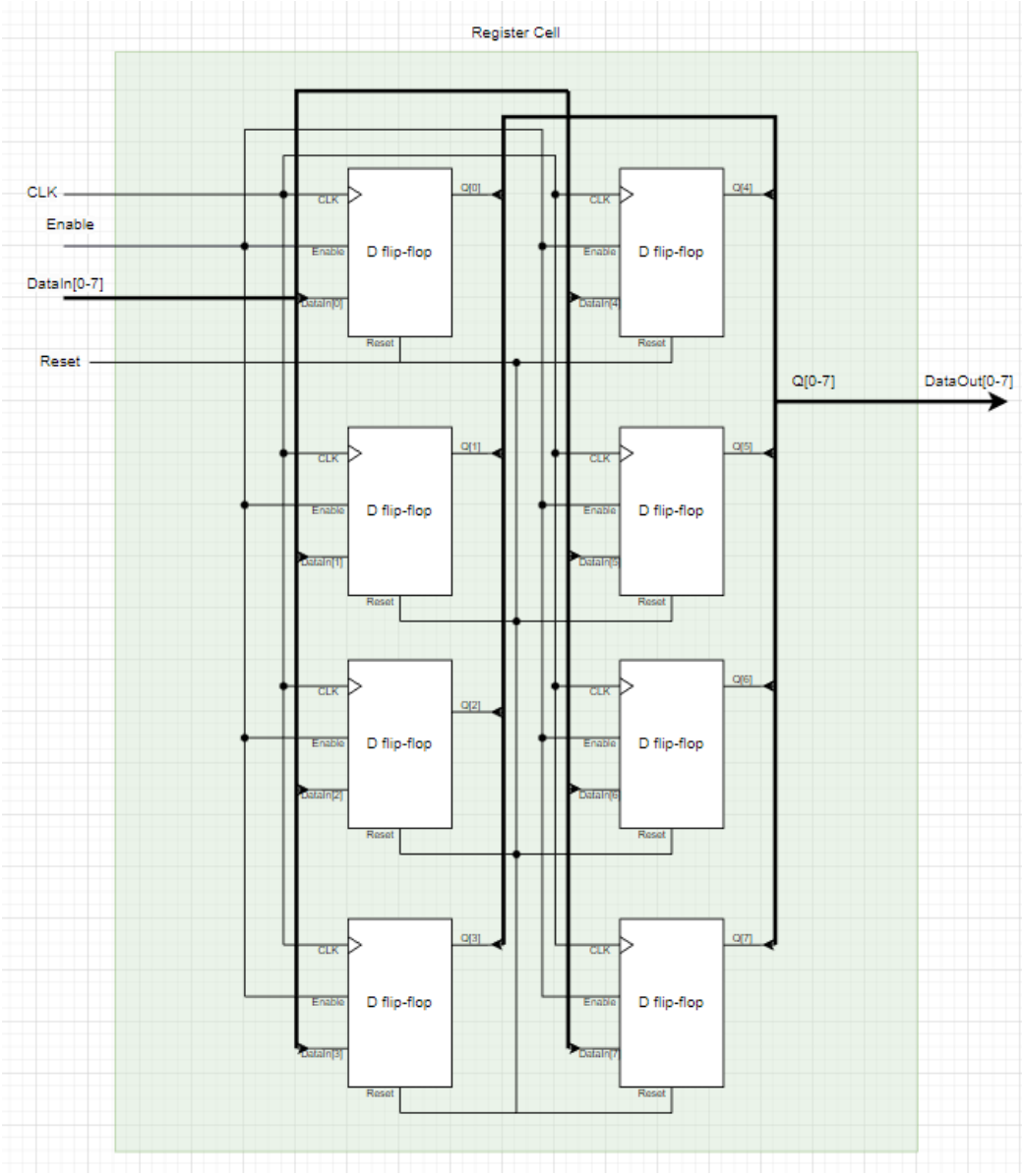
Figur 51: SubMux

7.4.4 Mux



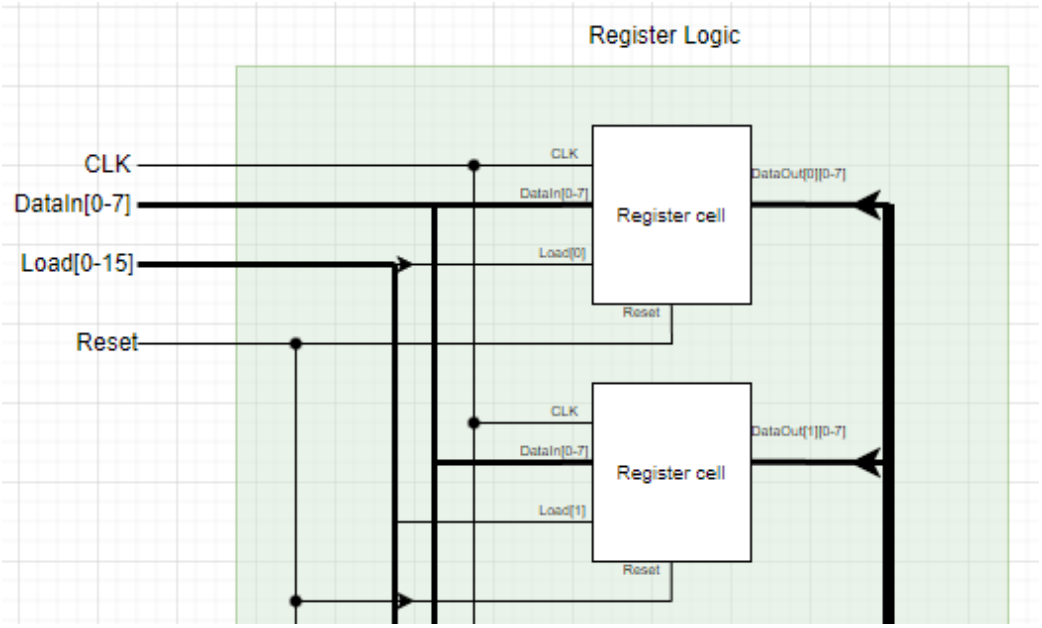
Figur 52: Mux

7.4.5 RegisterCell



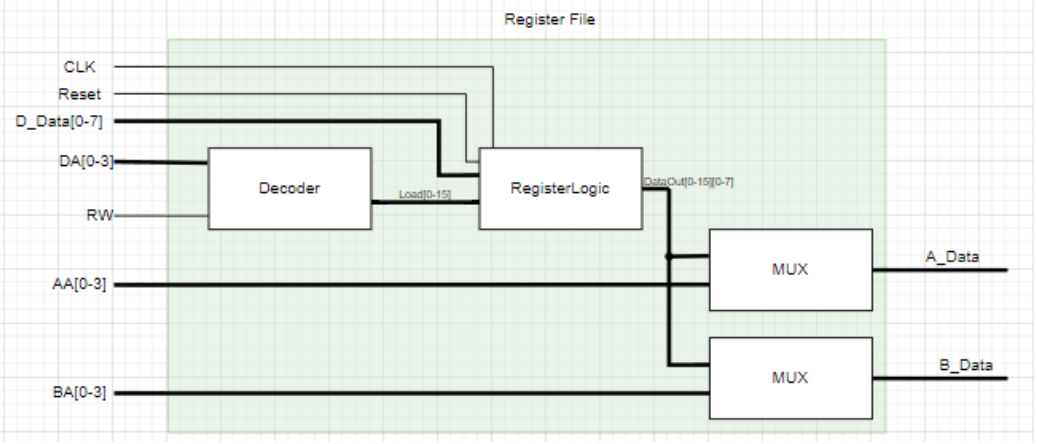
Figur 53: RegisterCell

7.4.6 RegisterLogic



Figur 54: RegisterLogic

7.4.7 RegisterFile



Figur 55: RegisterFile