Design av MAC algoritme med VHDL

Av Yngve Hafting 2018.

Multiply and accumulate (MAC)

Multiply and accumulate (Multipliser og legg til) er i utgangspunktet en algoritme som multipliserer to tall A og B, og legger til et tredje tall C:

D := A*B + C

MAC algoritmen brukes typisk i matrisemultiplikasjon, som igjen blir brukt mye i maskinlæring, digital signalprosessering, og numeriske algoritmer generelt. Digitale signalprosessorer (DSPer) og grafikkprosessorer (GPUer) har gjerne en eller flere MAC-enheter for å kunne multiplisere matriser hurtig. Moderne prosessorarkitekturer, slik som ARM, amd64 og Intel x86 har alle forskjellige varianter av MAC-instruksjoner[1]. Nyere ARM prosessorer har mulighet til å utføre flere typer MAC-instruksjoner.

I denne oppgaven kommer du til å lage en MAC-algoritme for å multiplisere og legge til eller trekke fra heltall. Ved hjelp av hardware beskrivelsesspråket VHDL «Very High speed Integrated Circuit Hardware Description Language» skal du bygge «innmaten» til denne algoritmen, slik at den kan utføres på en FPGA[2] «Field Programmable Gate Array». I assemblerspråket til ARM heter de tilsvarende kommandoene MLA «Multiply-add» og MLS «Multiply subtract».

Syntax (from the ARM assembler user guide)

```
MLA{S}{cond} Rd, Rn, Rm, Ra
```

```
MLS{cond} Rd, Rn, Rm, Ra
```

[1] I mange sammenhenger brukes begrepet “Fused multiply-add” når man snakker om å bruke MAC algoritmer på flyttall. CISC maskiner (Complex Instruction Set Computers) har gjerne mange FMA instruksjoner, med forskjellig presisjon og om det er addisjon eller subtraksjon som utføres.

[2] En FPGA er en brikke der man kan programmere hvordan logiske byggeblokker samhandler. FPGAen består gjerne av en rekke logiske porter (AND, OR, NOT), oppslagstabeller (LUT/ Look up table), minneelementer (FLIP FLOPer), og noen mer spesialiserte blokker (f.eks addere), samt et nettverk av ledere som kan koble komponentene sammen. Når man programmerer en FPGA, så bestemmer man hvordan disse elementene kobles sammen, slik at de kan gjøre den jobben vi vil. I utgangspunktet vil alt man programmerer en FPGA til være «på» hele tiden og jobbe parallelt, så den er velegnet til å lage ting som foregår samtidig. Ved utvikling av prosessorer og spesialiserte kretser er det vanlig å teste ut arkitekturen på FPGAer før man lager en ny chip. FPGAer brukes også mye i industrien til å gjøre mer eller mindre spesialiserte oppgaver, der vanlige prosessorer ikke er like egnet. For mange studenter er det vanskelig å forholde seg til forskjellen mellom programmering av prosessorer og programmering av FPGAer. Når vi bruker et HDL «Hardware Description Language», så befinner vi oss på et lavere nivå enn når vi programmerer en prosessor med assemblerspråk. Når vi programmerer med assembler, er allerede byggeblokkene til prosessoren fastlagt for oss, mens med et HDL så designer vi hvordan byggeblokkene settes sammen. Vi kan dermed lage egne prosessorer med HDL, og dermed bestemme hvilke muligheter vi skal ha i et assemblerspråk. Det som kanskje er aller vanskeligst å forholde seg til er at vi er vant til at software-programspråk utføres sekvensielt, mens i hardware programspråk vil alt vi bygger virke samtidig, uavhengig av rekkefølgen det står. I tillegg til dette har vi mulighet til å skrive testbenker som brukes i simulering, og disse har gjerne elementer som utføres sekvensielt, fordi testingen foregår med en prosessor.

where:

Cond	is an optional condition code.
S	is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.
Rd	is the destination register.
Rn, Rm	are registers holding the values to be multiplied.
Ra	is a register holding the value to be added

Operation (from the ARM assembler user guide)

The `MLA` instruction multiplies the values from `Rn` and `Rm`, adds the value from `Ra`, and places the least significant 32 bits of the result in `Rd`.

The `MLS` instruction multiplies the values in `Rn` and `Rm`, subtracts the result from the value in `Ra`, and places the least significant 32 bits of the final result in `Rd`.

Oppgave

I denne oppgaven har vi laget ferdig VHDL kode som utfører en MAC operasjon i sin enkleste forstand. DVS koden utfører en MLA operasjon på én klokkesyklus. For enkelhets skyld antar vi at signalene som kommer inn er bufrede og synkrone med klokka. I tillegg er det laget kode til en testbenk som kan brukes sammen med simuleringsverktøyet (Questa-sim), for å generere data for å teste koden med MAC-algoritmen.

I den første delen av oppgaven skal du bli litt kjent med verktøyene for simulering implementering av FPGA kretser, og du skal rette opp én feil i koden til MAC algoritmen, slik at den lar seg kompilere og kjøre. Videre skal du kjøre simulering og dokumentere at den nye koden virker.

Til den første delen følger det med en «kokebokoppskrift» på hvordan oppgaven kan utføres.

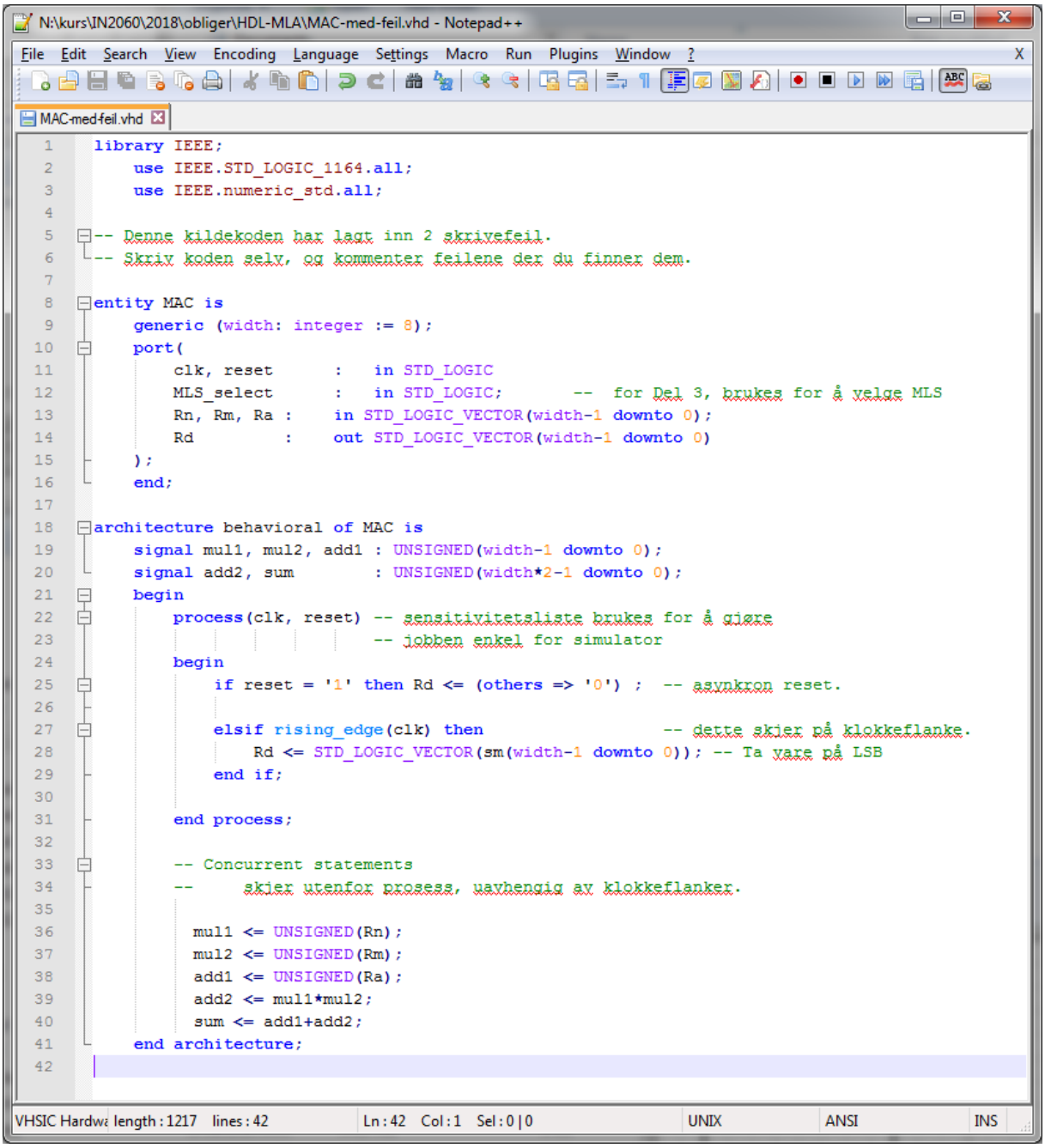
I den andre delen av oppgaven skal du modifisere MAC algoritmen for å lage en pipe-line, simulere og dokumentere at algoritmen gir forventet resultat og til slutt syntetisere den og vurdere hvilken innvirkning pipelining gir.

I den tredje delen skal du utvide koden til å implementere MLS algoritmen. For å gjøre det må du benytte signalet MLS_select fra testbenken. Simuler funksjonen til kretsen, og dokumenter hvordan den virker.

I den fjerde delen skal du innføre enda ett nivå med pipelining, tegne skjema selv, og vurdere bruken av flipflopper.

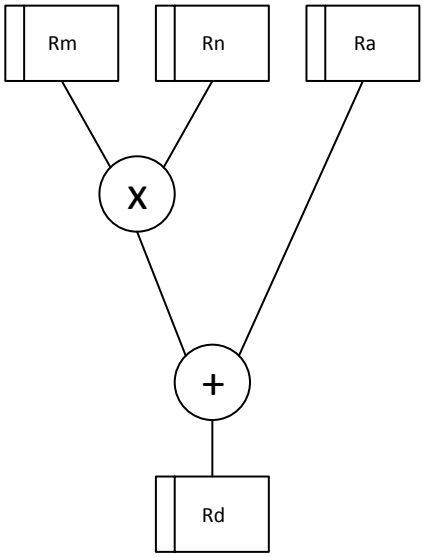
Oppgaven i sin helhet kan leveres som én pdf-fil. Innholdet spesifiseres i hver del.

NB: Vi oppfordrer alle til å gjøre denne oppgaven i sin helhet selv! Hvis du skal ha fullt utbytte av en slik oppgave har du ingenting å vinne på å kopiere fra andre selv om det er lett. Erfaringsmessig er det alltid noen som prøver å flyte på andres arbeid, noe som typisk straffer seg til eksamen.



Figur 1: Kildekode til del1, MAC med MLA

Del 1:



Figur 2:MLA algoritmen

Lag et nytt prosjekt i Questa, der du lager en fil for MAC algoritmen, og tar med testbenken.

(*File-> new Project…* , velg folder, og gi prosjektet et navn. *Trykk «ok»* , Velg «Add existing files», og legg til den vedlagte kildekoden til testbenken. (Senere kan du høyreklikke i prosjektvinduet og velge «Add to project» for å legge til eller opprette filer).

Kompiler filene (*«Compile all»*) og rett opp den første feilen i MAC algoritmen (testbenken skal være uforandret). Kommenter endringen din i kildekoden. Ved å dobbeltklikke på filen med feil (markert med rød 'X'), eller den røde teksten i «transcript» vinduet får du frem feillisten.

Etter å ha rettet den første feilen kan du kompilere igjen for å finne den andre feilen. Gjør de nødvendige endringene i koden og kommenter rettingen i kildekoden.

Når koden kompilerer skal du simulere: *Velg Simulate-> Start simulation. I vinduet som kommer opp velger du «work->mac_testbenk» før du trykker OK.*

Legg til signalene (clk, reset, Rm, Rn og Ra) fra testbenken inn i waveform vinduet. *(Velg signalene fra det mørkeblå vinduet, og høyreklikk med musen og velg «add wave»)*

Velg å vise Rm, Rn, Ra og Rd som desimaltall (*velg signalene i «Wave» vinduet og høyreklikk og velg radix->decimal*)

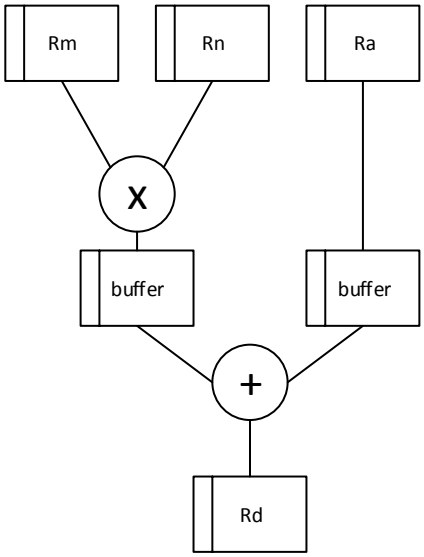
Simuler testbenken i 4 mikrosekunder (*skriv «run 4 us» i transcript vinduet*). Zoom inn slik at du ser hele waveformen i bildet *(Høyreklikk i vinduet med waveforms og velg «zoom full»)*.

Ta en skjermdump av waveform (Alt-printscreen på windows), og bruk paint eller et annet tegneprogram til å lagre filen. Bildet settes inn i rapportdokumentet.

Innlevering:

- **Skjermdump av simuleringsresultatet**
- **Kildekode til den korrigerte MAC algoritmen**

Del 2:



Figur 3:MLA med pipeline.

Modifiser MAC algoritmen slik at du får en pipeline der det multipliserte resultatet bufres før det legges til Ra. MAC algoritmen skal kunne gis nye data for hver klokkesyklus, så Ra må også bufres. Kommenter endringene du gjør i kildekoden.

Simuler med testbenken. Ta en skjermdump av waveform

Vurdér resultatet av simuleringen, blir resultatet slik du ventet deg? Begrunn svaret.

Den simuleringen du har foretatt nå tar ikke hensyn til propageringsforsinkelser (propagation delay) i portene som brukes dersom dette hadde vært implementert i en FPGA. Dersom vi tar i bruk et syntese- og implementeringsverktøy, kan vi få generert simuleringsfiler som tar med slike tidsforsinkelser.

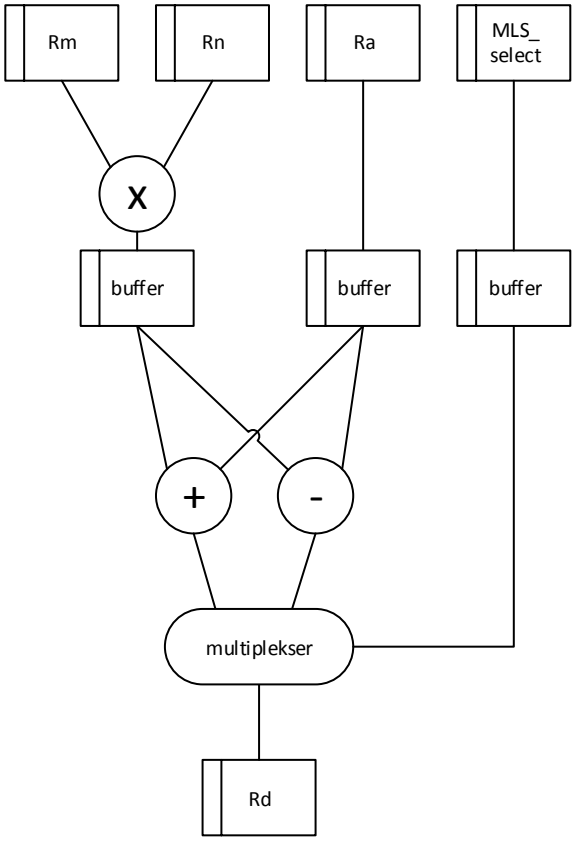
Vurdér kvalitativt (dvs uten tall og utregninger) hvilke endringer du kan forvente i simuleringsresultatene, uten og med pipeline, dersom vi tok hensyn til propageringsforsinkelser?

Vurderingene kan skrives i et vilkårlig verktøy, men lagres med en godt lesbar font i pdf format.

Innlevering:

- **Skjermdump av simuleringsresultatet**
- **Kildekode til MAC algoritmen**
- **Vurderingene til oppgaven.**

Del 3:



Figur 4: MAC algoritme med MLA og MLS og ett nivå pipeline

I den tredje delen skal du utvide koden til å implementere MLS algoritmen. Signalet MLS_select skal benyttes til å velge om det er addisjon eller subtraksjon som skal kjøres. Når MLS_select er høy skal MLS kjøres, ellers- når det er lavt ('0') – kjøres MLA.Husk at det er multiplikasjonsproduktet som skal trekkes fra Ra, ikke omvendt. Simuler funksjonen til kretsen, og dokumenter hvordan den virker. Kommenter endringer i kildekoden.

Innlevering:

- **Skjermdump av simuleringsresultatet**
- **Kildekode til MAC algoritmen med kommentarer**

Del 4:

Ut i fra skjemaet i del 3, kan man innføre enda et nivå med pipelining, uten å dele opp de matematiske operasjonene i seg selv. Tegn ditt eget skjema slik pipelining, og gjør endringer i koden slik at vi får med det siste nivået med pipelining. Beregn hvor mange flipflop'er koden din vil generere. Hvor mange klokkesykler trenger algoritmen fra input registrene er satt til Rd kan leses av?

Innlevering:

- **Skjermdump av simuleringsresultatet**
- **Kildekode til MAC algoritmen med kommentarer**
- **Skjema og besvarelse av oppgaven.**

MERK: Alle skjermdumpene og vurderingene til alle delene samles i ett pdf-dokument ved innlevering. Kildekode leveres i separate filer for hver del.

Kilder i tilfeldig rekkefølge:

https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation

https://devblogs.nvidia.com/inside-volta/

https://graphics.stanford.edu/papers/gpumatrixmult/gpumatrixmult.pdf

https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm

https://www.felixcloutier.com/x86/

http://vision.gel.ulaval.ca/~jflalonde/cours/1001/h17/docs/arm-instructionset.pdf

http://www.keil.com/support/man/docs/armasm/armasm_dom1361289878654.htm