Layoutgenerator för en multiplikator i ”overturned stairs” trädstruktur

Examensarbete utfört i Elektroniksystem
vid Linköpings Tekniska Högskola
av
Klas Alner

LiTH-ISY-EX-ET-0214
Linköping 2003

Handledare: Henrik Ohlsson
Examinator: Lars Wanhammar

Avdelning, Institution
Division, Department

Institutionen för Systemteknik
581 83 LINKÖPING

Datum
Date
2003-05-14

Språk
Language
X Svenska/Swedish
Engelska/English

Rapporttyp
Report category
Licentiatavhandling
X Examensarbete
C-uppsats
D-uppsats
Övrig rapport

ISBN

ISRN LITH-ISY-EX-ET-0214-2003

Serietitel och serienummer **ISSN**
Title of series, numbering

URL för elektronisk version
http://www.ep.liu.se/exjobb/isy/2003/214/

Titel
Title
Layoutgenerator för en multiplikator i ”overturned stairs” trädstruktur

Layoutgenerator for a multiplier in ”overturned stairs” treestructure

Författare
Author
Klas Alner

Sammanfattning
Abstract
Multiplikatorer används ofta som ett byggblock vid konstruktion av kretsar som digitala filter, FFT-processorer och aritmetiska enheter. Olika trädstrukturer används i ”höghastighet” applikationer för multiplikatorer. En typ av träd, ”overturned-stairs”(OS) som är ett adderar träd av första ordningen har uppvisat lika optimal prestanda med avseende på hastighet som Wallace träd, vid 18 eller färre ingångar. I moderna integrerade kretsar, ger ledningar och kopplingar upphov till fördröjningar och parasitiska laster. I en jämförelse mellan Wallace träd, och OS1 träd har det sistnämda kortare och mindre komplicerad ledningsdragningar och är därför mer ändamålsenlig för VLSI implementationer.

Nyckelord
Keyword
Multiplikator, Adderarträd, OS-träd, Baugh-Wooley, Wallace, CSA, Heladderare, Trädstrukturer, Skill, Cadence, VLSI

INNEHÅLLSFÖRTECKNING

1. Inledning

1.1. Bakgrund

Multiplikatorer används ofta som ett byggblock vid konstruktion av kretsar som digitala filter, FFT processorer och aritmetiska enheter. Olika trädstrukturer används i ”höghastighet” applikationer för multiplikatorer. En typ av träd, ”overturned-stairs”(OS) som är ett adderar träd av första ordningen har uppvisat lika optimal prestanda med avseende på hastighet som Wallace träd, vid 18 eller färre ingångar. I moderna integrerade kretsar, ger ledningar och kopplingar upphov till fördröjningar och parasitiska laster. I en jämförelse mellan Wallace träd, och OS_1 träd har det sistnämda kortare och mindre komplicerad ledningsdragningar och är därför mer ändamålsenlig för VLSI implementationer.

1.2. Uppgift

Uppgiften bestod i att ta fram en multiplikatorgenerator, utifrån följande förutsättningar

- ST Microelectronics 0.18µm process
- CMOS logik
- Ordlängden på operand W_c och operator W_d skulle kunna varieras från 8 till18 bitar
- OS-träd struktur
- Baugh-Wooley algoritmen
- Generatorn skulle skrivas i programspråket SKILL från Cadence

1.3. Kapitelbeskrivning

Kapitel 2 redogör för teorin om OS-träd och jämförelser mellan olika trädstrukturer bl a Wallace träd. Kapitel 3 behandlar multiplikatordesign med OS-träd, kapitel 4 tar upp heladderarcellen och konstruktionsförloppet av denna. I kapitel 5 redogörs för verktyg som använts under arbetets gång bl a Cadence programpaket med betoning på Virtuoso som använts för layouten samt programspråket SKILL, där några enkla kodexempel ges. Redogörelse för layoutgenereringen av multiplikatorn görs i kapitel 6. Slutligen ser vi på erhållna resultat och slutsatser i kapitel 7. I Appendix A ges hänvisning till skriven kod, Appendix B visar en framtagen 8×8 multiplikator implementerad med generator, samt med tillhörande ”bitslice”på bitnivå.

1.4. Målbeskrivning

Målet för detta examensarbete var att ta fram en fungerande layoutgenerator för en multiplikator i OS-trädstruktur, enligt de villkor givna under *Uppgift* ovan.
Energiförbrukningen skall sedan testas för olika antal ordlängder eg. 8-18 bitar för layouten.

1.5. Målgrupp

Denna rapport vänder sig i första hand till studenter på sista årkursen på elektroingenjörs programmet, samt övriga studenter med elektronikprofil på utbildningen.

1.6. Tack

Ett stort tack till Emil Hjalmarsson på elektroniksystem (ES) som varit behjälplig med SKILL programmering. Tack till Oscar Gustafsson på ES som har varit handledare i början av examensarbetet, och varit behjälplig med teoridelen.

2. OS-träd

2.1. Introduktion

Wallace träd är den teoretiskt snabbaste additionsstrukturen. Dess interna kopplingar är dock komplicerade och därigenom svår att implementera. Därför bör Wallace träd multiplikatorer bara användas för applikationer med långa ordlängder eller där prestanda är kritisk. Wallace beskrev adderar trädet med Carry Save Adders(CSA). På ordnivå reducerar CSA tre n-bitars ord till två utgående ord, summa och carry. Summan av dessa två ord ger i sin tur den totala summan av de tre inmatade orden. På bitnivå utgörs varje enskild CSA av hel- och halvadderare. Flera förslag till trädstrukturer har utvecklats, med målet att erhålla den goda prestanda som Wallace träden har men utan dess komplexitet i de interna ledningsdragningarna.

2.2. OS-träd

För att minska graden av interna kopplingar presenterade Mou och Jutand ”overturned stairs adder trees” (OS- adderarträd) [11]. Namnet kommer av dess utformning som påminner om en ”upp och ned vänd trappa”. OS-trädet har en rekursiv struktur som ger regelbundna interna kopplingar och en kompakt struktur vilket ger en kompakt layout. OS-träden har en beräkningshastighet, för ett visst antal operander N, som överensstämmer med Wallace-trädets kapacitet. En nackdel med OS-träd strukturen är att man får en lång kritisk väg genom strukturen för stora additionsträd , eg. stort antal ingångar. I varianter av OS-träd med högre ordning (OS_n) har man uppnått bättre prestanda, dvs flera ingångar men med en bibehållen trädhöjd. Detta har kunna åstadkommas genom att antalet interna kopplingar i trädstrukturen har ökat.

För att öka antalet operander, finns det två metoder. Antingen ökar man trädets höjd med bibehållen ordning eller så ökas antalet interna kopplingar i trädet vilket ger en högre ordning. Väljer man att öka trädets höjd för att på så sätt kunna erhålla fler ingångar, så försämras beräkningshastigheten. Om man däremot väljer att öka ordningen hos trädet, blir antalet inre kopplingar fler och komplexiteten samt chiparean ökar då för additionsträdet.

2.3. OS-träd av högre ordning samt kombinationsträd

För att öka antal ingångar med bibehållen höjd så kan antal interna ledningar ökas genom att *grenen* i trädstrukturen ersätts med ett träd. *Grenen* se figur 2.1 motsvaras av en kedja av CSA:er, iterativ CSA (ICSA) med låg komplexitet som kan ersättas med ett träd. När *grenen* till ett OS_1-träd ersätts med ett OS_1-träd blir resultatet ett OS-träd av andra ordningen. När *grenen* till ett OS-träd av första ordningen ersätts med ett OS-träd av andra ordningen blir resultatet ett OS-träd av tredje ordningen osv. Det finns andra additionsträd där man har försökt att kombinera OS-trädets goda egenskaper med andra trädstrukturer för att på så sätt finna de fördelar som OS-trädet av första ordningen saknar. Vid kombinationen mellan Zura-McAllister-träd och OS-träd ersätts *grenen* med ZM-trädet. ZM-träd är uppbyggda av enkla 4-2 räknare, figur 2.3 visar ett 14-ingångars ZM-träd.

2.4. Generell beskrivning av OS-träd strukturen

Några grundläggande byggblock kan identifieras i OS-trädet som ges av fig. 2.1, *Kropp*, *Rot*, *Koppling* och *Gren*. Dessa delar kan användas för att formge en generell struktur av OS-trädet. Antag att vi har en trädhöjd, h_{OS}=j+1, kroppens höjd blir då j-1, och grenen får då höjden j-2. Kopplingen tar hand om utgångarna från grenen respektive kroppen i en 5/3 reducering genom två stycken CSA:er. De tre utgångarna från *koppling* reduceras slutligen i roten i en 3/2 reducering genom en CSA till *sum* och *carry*. OS-träd av högre ordning konstrueras så att en gren ersätts med ett OS-träd av motsvarande höjd, detta ger ett bredare träd. Det totala antalet ingångar ökar med bibehållen höjd och beräkningsprestanda som följd. Figur 2.2 visar trädets uppbyggnad på ordnivå med Carry Save Adders för ett 7-9 ingångars OS_1-träd. Vid behov av mindre än 9-ingångar så bytes en heladderare ut mot en halvadderare på bitnivå.

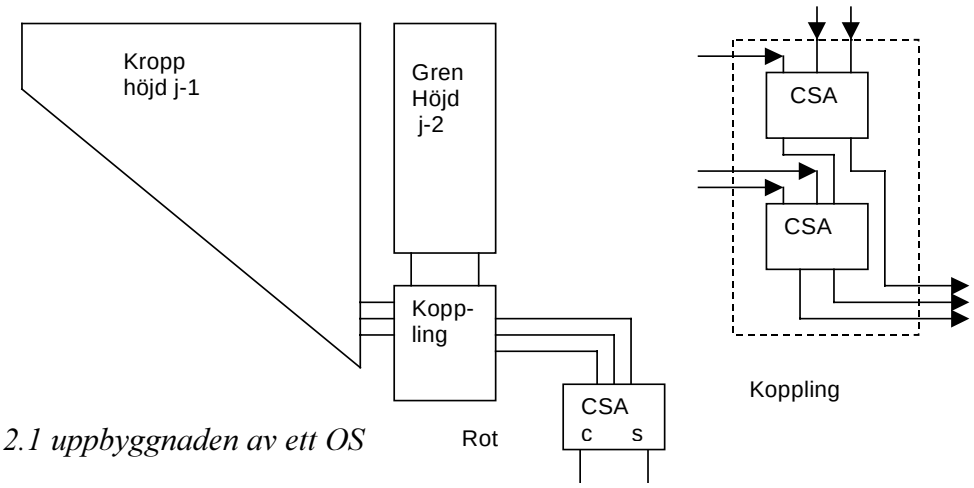


Fig. 2.1 uppbyggnaden av ett OS träd

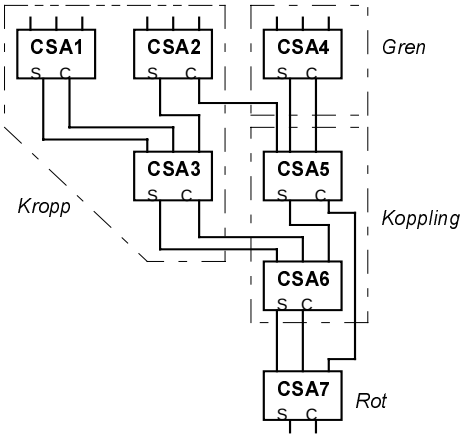


Fig. 2.2 uppbyggnaden av 7-9 ingångars OS_1-träd på ordnivå med CSA

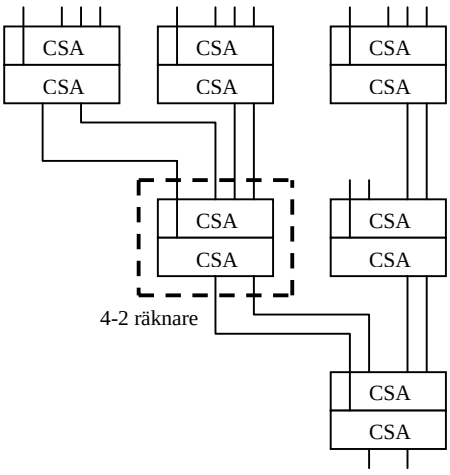


Fig. 2.3 14-ingångars ZM-träd, uppbyggd av 4-2 räknare

2.5. Jämförelser av trädstrukturer

I en högre ordningens OS-träd, kan fler ord adderas med bibehållen höjd av trädet. Strukturen av ett OS-träd bestäms av antalet ord som skall adderas. Som visas i fig.2.1, så ökar bredden och höjden för ett ökat antal operander(ord). Antag att det är N_{OS} antal operander till OS_1-trädet. Trädets höjd h_{OS} ges då av (1)

$$h_{OS} = \lceil (1 + \sqrt{8N_{OS} - 23}) / 2 \rceil \qquad (1)$$

Att jämföras med det samma för Wallace (N_W operandantal, och höjd, h_W), där

$$h_W = \lceil \log(N_W/2)/\log(3/2) \rceil \qquad (2)$$

Från (1) och (2), ges att när antalet operander är lika

$N_{OS}=N_W$

Så är trädhöjden för de båda träden lika, upptill

$N_{OS}=N_W>18$

Med många operander till OS-trädet ökar höjden på trädet, det ger att den kritiska vägen genom trädstrukturen ökar och med denna minskar beräkningshastigheten. En större drivförmåga krävs då samtidigt för att driva de långa interna ledningarna. För att erhålla kortare ledningar så finns det möjlighet att kombinera olika trädstrukturer för att på så sätt erhålla bägges fördelar. Det blir en ”avvägning” mellan komplexiteten på strukturen och höjden e.g. beräkningshastigheten.

Genom att öka antalet inre kopplingar ”feedthroughs”(FT), minskar den kritiska vägen genom strukturen. Organisationen av CSA:er i trädstrukturer leder till komplicerade inre kopplingar. Utgången från en CSA behöver inte alltid vara ingången till en angränsande CSA. I många fall kommer ledningen gå igenom ett antal CSA:er innan den når rätt ingång. Vi använder antalet av dessa vertikala FT:s som ett mått på lednings komplexiteten. OS-träd av första ordningen behövs maximalt tre st FT:s (jämför med ”bitslicen” fig. 2.5). Det ökade antalet av inre kopplingar FT:s ger ett OS-träd av högre ordning. Iterativa CSA-arrayer och Wallace-träd representerar två extremfall av multioperandadderare. ICSA-arrayer har en hög grad av regelbundenhet med korta ledningsdragningar men med låg beräkningshastighet, då ingen parallellism förekommer då bitarna passerar samtliga CSA:er från topp till botten se fig.2.5. Wallace trädet har en hög grad av parallellism och därigenom en hög beräkningshastighet på bekostnad av en högre inre komplexitet i ledningsdragningarna. Spektrumet av olika trädstrukturer ligger mellan dessa båda två ytterligheter. För att kunna göra ett korrekt val vid bedömning av hastighet och chiparea redovisas tre stycken tabeller. Tabell 1 visar det maximala antalet operander de olika trädtyperna klarar att summera då de båda är av samma höjd(h).

Tabell 1

h	ICSA	ZM	OS_1	ZM_2	OS_1-ZM	OS_2	OS_2-ZM	OS_3	Wallace
1	*3*	*3*	*3*						*3*
2	*4*	*4*	*4*						*4*
3	*5*	*6*	*6*						*6*
4	*6*	*8*	*9*						*9*
5	*7*	*11*	*13*	*12*					*13*
6	*8*	*14*	*18*	*16*	*19*	*19*			*19*
7	*9*	*18*	*24*	*23*	*27*	*28*			*28*
8	*10*	*22*	*31*	*30*	*38*	*41*			*42*
9	*11*	*27*	*39*	*41*	*52*	*59*	*60*	*60*	*63*
10	*12*	*32*	*48*	*52*	*70*	*83*	*87*	*88*	*94*
11	*13*	*38*	*58*	*68*	*92*	*114*	*125*	*129*	*141*
12	*14*	*44*	*69*	*84*	*119*	*153*	*177*	*188*	*211*

Tabell 1 antalet ingångar (N), *för olika höjd på trädstrukturer h* [11]

Wallace strukturen jämförs i figur 2.4 med ICSA och OS_1 trädstrukturer. N är antalet ingångar och *h* är höjden på trädstrukturen. För Wallace och OS_1 är höjden på trädstrukturen påfallande lika upp till N=18.

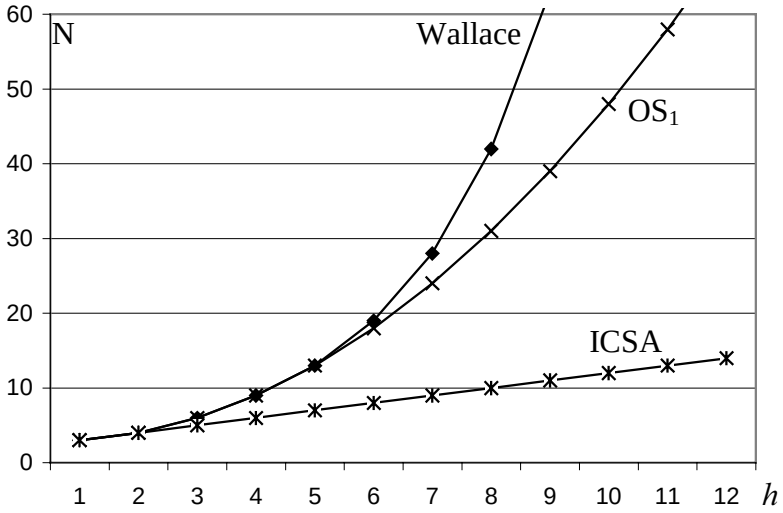


Fig. 2.4 diagram: jämförelse mellan Wallace, OS_1, och ICSA trädstrukturer

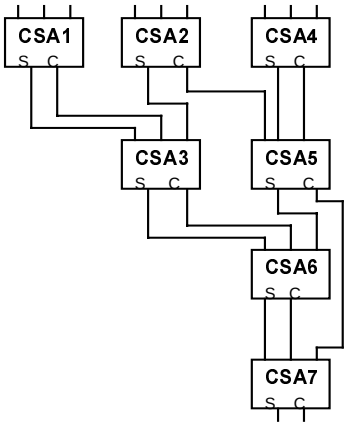
Tabell 2 nedan visar antalet FT:s som erfordras i de olika trädstrukturerna. FT anger antalet nivåer av heladderare i adderar trädet som en ledningar passerar i djupled. Detta ges som ett mått på den inre kopplingskomplexiteten. Ett lågt antal av FT:s betyder korta ledningsdragningar och därigenom en struktur med låg chiparea, samt en enkel struktur att implementera till layout. ICSA har noll stycken FT:s, ledningarna är då dragna till direkt näst följande heladderare. Tabell 2 visar också relationen mellan höjden h, och antalet ingångar till strukturen (N).

Tabell 2

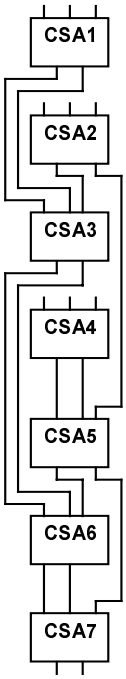
Träd	FT	N(h)
ICSA	0	$h+2$
ZM	2	$h^2/4+h/2+[17-(-1)^h]/8$
OS_1	3	$h^2/2-h/2+3$
OS_1-ZM	5	$h^3/12-3h^2+(29h)/12-[1-(-1)^h]/16$
OS_2	6	$h^3/6-(3h^2)/2+(22h)/3-7$
OS_2-ZM	8	$h^4/48-h^3/3+(35h^2)/12-(29h)/3+17+([1-(-1)^h]/32)$
OS_3	9	$h^4/24-(11h^3)/12+(215h^2)/24-(445h)/12+63$
Wallace	≥O(log(N))	$2(3/2)^{h-1}+1\leq N\leq 2(3/2)^h$

Tabell 2 Antal "feedthroughs" för olika trädstrukturer [11]

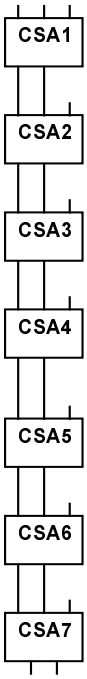
Figur 2.4 nedan visar en 9 ingångars ICSA "bitslice", och ett 9-ingångars OS_1 träd med höjden *h*=4 och dess"bitslice". Den längsta vägen genom OS_1 trädet är 4T, där T är tiden genom en CSA på ordnivå, eller tiden genom en heladderare på bitnivå. För motsvarande ICSA är den längsta vägen 7T.



OS_1 i "trappform"



OS_1-bitslice på ordnivå



ICSA

Fig. 2.5 9-ingångars OS_1 och ICSA

Tabell 3 nedan visar på det optimala valet mellan trädstrukturer för olika antal operander, jämfört med Wallace trädet för N ingångar(N=3-60). När de olika träden tangerar Wallace gränsen dvs d=0, eller har samma höjd för ett givet antal ingångar (N), faller valet på det träd som har minst antal FT:s.

Tabell 3

N	Träd	d	N	Träd	d
3-4		0	25-27	OS_1-ZM	0
5-8	ZM	0	28	OS_2	0
9	OS_1	0	29-31	OS_1	0
10-11	ZM	0	32-38	OS_1-ZM	0
12-13	OS_1	0	39-41	OS_2	0
14	ZM	0	42	OS_2	1
15-18	OS_1	0	43-52	OS_1-ZM	0
19	OS_1-ZM	0	53-59	OS_2	0
20-24	OS_1	0	60	OS_2-ZM	0

Tabell 3 jämförelser för olika trädstrukturer till Wallacegränsen [11]

I detta fall skall en multiplikatorgenerator från 8 till 18 bitar konstrueras. Med hänsyn taget till chiparea och beräkningshastighet föll i detta fall valet på OS_1-träd strukturen.

3. Multiplikator design med OS-träd

3.1. Baugh-Wooley 2-komplement multiplikator

Baugh-Wooleys algoritm är snabb, hanterar teckenbitarna effektivt, och behöver liten chiparea vid multiplikation av 2-komplementstal [10]. Nedan presenteras en kortfattad genomgång av Baugh-Wooleys algoritm. Två tal A och B, skall multipliceras. Antag 2-komplements tal och en ordlängd av N bitar för båda talen. A och B kan då skrivas som

$$A = -a_5 + \sum_{i=1}^{N-1} a_i 2^{-i} \qquad (1)$$

$$B = -b_5 + \sum_{i=1}^{N-1} b_i 2^{-i} \qquad (2)$$

Där produkten P av dessa kan skrivas enligt följande:

$$P = A \cdot B = \left(-a_5 + \sum_{i=1}^{N-1} a_i 2^{-i} \right) \left(-b_5 + \sum_{i=1}^{N-1} b_i 2^{-i} \right) \qquad (3)$$

Detta ger följande fyra termer:

$$P = a_5 b_5 - \sum_{i=1}^{N-1} a_5 b_i 2^{-i} - \sum_{i=1}^{N-1} b_5 a_i 2^{-i} + \sum_{i=1}^{N-1} a_i 2^{-i} \cdot \sum_{i=1}^{N-1} b_i 2^{-i} \qquad (4)$$

För 2-komplementstal gäller följande förhållande

$$-(X_0, X_1, \ldots, X_{N-1}) = (\overline{X}_0, \overline{X}_1, \ldots, \overline{X}_{N-1}) + (0,0,\ldots,0,1) \qquad (5)$$

Omskrivning av (4) med 2-komplementsförhållandet (5)

$$P = a_5 b_5 + \sum_{i=1}^{N-1} \overline{a_5 b_i} 2^{-i} + \sum_{i=1}^{N-1} \overline{b_5 a_i} 2^{-i} + 2 \cdot 2^{-(N-1)} + \sum_{i=1}^{N-1} a_i 2^{-i} \cdot \sum_{i=1}^{N-1} b_i 2^{-i} \qquad (6)$$

Den första termen representerar teckenbiten för produkten. Den andra termen representerar vänsterkanten av delsummorna, den tredje termen motsvarar underkanten av delsummorna (6). Fig. 3.2 visar 6×6 multiplikation med Baugh-Wooleys 2-komplements algoritm.

3.2. En 6-ingångars multiplikator av OS-trädstruktur

I figur 3.1 nedan visas strukturen för ett 6-ingångars OS-träd. Höjden på trädet, här $h_{OS}=3$, representerar den längsta vägen genom trädet. Maximalt antal adderare som passeras från ingång till utgångarna, *sum* och *carry* är 3 stycken adderare. Vänster kolumn flyttas över till höger tills det skapas en kolumn, här läggs CSA1 över CSA2, en ”bitslice” på ordnivå skapas. ”bitslicen” är ett viktigt byggblock för att kunna skapa chiplayout för multiplikatorn.

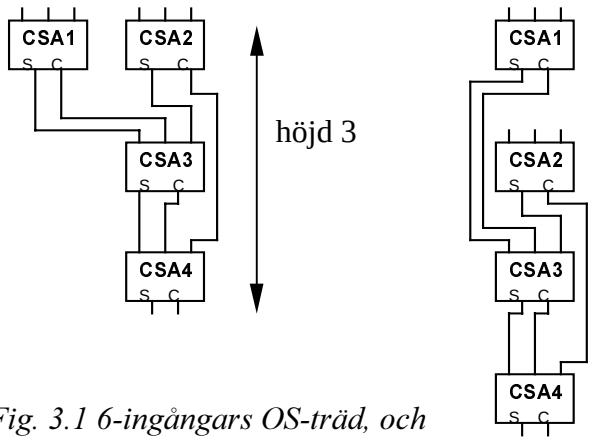


Fig. 3.1 6-ingångars OS-träd, och motsvarande ”bitslice” på ordnivå

3.3. Arbetsgången för multiplikatordesign

Har redovisas kort den generella arbetsgången för design av en multiplikator med OS-träd struktur samt Baugh-Wooleys 2-komplements algoritm [6].

- Välj ordlängd för multiplikatorn, N antal bitar, här väljs en 6×6 multiplikation
- Välj multiplikatoralgoritm, här väljs Baugh-Wooley
- Bestäm trädstruktur, här väljs OS_1
- Beräkna höjden på trädstrukturen, här blir $h_{OS} = 3$ enligt

$$h_{OS} = \left\lceil (1 + \sqrt{8N_{OS} - 23}) / 2 \right\rceil$$

- Skapa en ”bitslice” från OS-trädet genom att flytta varje kolumn över kolumnen till höger. Se fig 3.1.
- Bestämning av partialprodukterna enligt Baugh-Wooley’s algoritm. Här skiftas partialprodukterna in på bitnivå in i adderarträdet. Multiplikatorns naturliga form blir då ett parallellogram. I hörnet uppe till höger har biten lägst signifikans(LSB), och biten längst ner till vänster har högst signifikans(MSB). Partialprodukterna genereras av en AND-grind, förutom partialprodukterna i diagonala vänsterkanten och underkanten som genereras av en NAND-grind då dessa ska inverteras, i fig. 3.2 markerat med skuggade boxar.

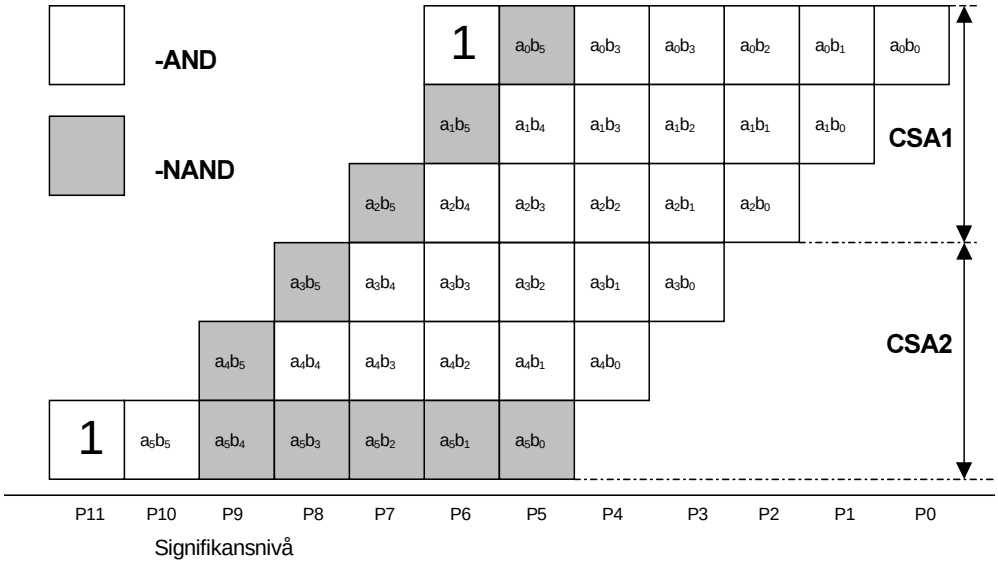


Fig. 3.2 partialprodukt formation för en 6×6 multiplikator

- I varje CSA görs vertikala grupperingar på varje signifikansnivå om tre partialprodukter. Varje CSA (ordnivå) byts ut mot N+1 heladderare på bitnivå. För 6-ingångars trädet i fig.3.1 ger detta CSA1 ersätts med 7 stycken heladderare, en för varje kolumn. På samma sätt ersätts CSA2 med heladderare. CSA3 och CSA4 ersätts här med 7 stycken heladderare vardera, dessa har som funktion att koppla ihop trädet och har inga ingångar för partialprodukterna. Jämför med ”bitslicens” koppling för OS trädet i fig. 3.1. Strukturen av heladderare har nu formen av ett parallellogram med 4×7 heladderare.

- Summorna kopplas vertikalt nedåt, carryn kopplas rakt nedåt till kolumnen till vänster, detta ger en högre signifikans nivå för carryn jämfört med summorna.

- Förenklingar av layouten kan göras. I de tre kolumnerna längst till höger kan förenklingar göras. Första heladderaren uppe i högra hörnet har endast en partialprodukt att ta hand om, den ersätts med en ledare direkt till utgången, heladderare byts mot halvadderare där det är möjligt osv.

- Slutligen rätas parallellogramet av heladderare upp och bildar en rektangel. Detta för att få en regelbunden och kompakt layout

Nedan följer ett exempel på processen för multiplikaturstrukturen beskriven i texten ovan, inklusive de förenklingar som kan göras i de tre sista kolumnerna.

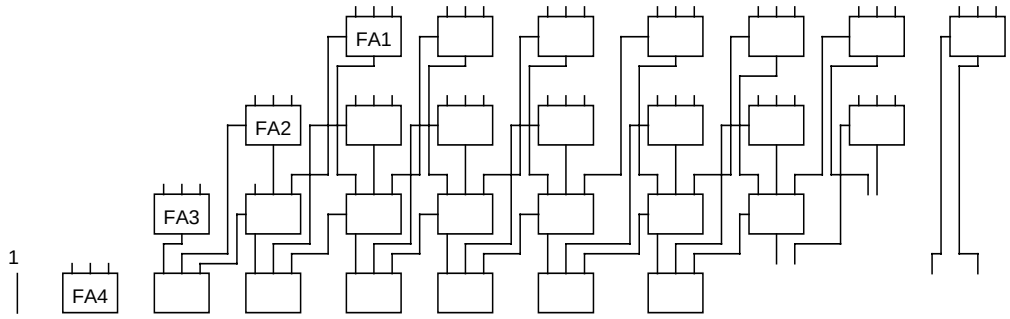


Fig. 3.3 Heladderare placeras ut enligt Baugh-Wooleys algoritm, de interna kopplingarna dras enligt ”bitslicen” jämför fig.3.1

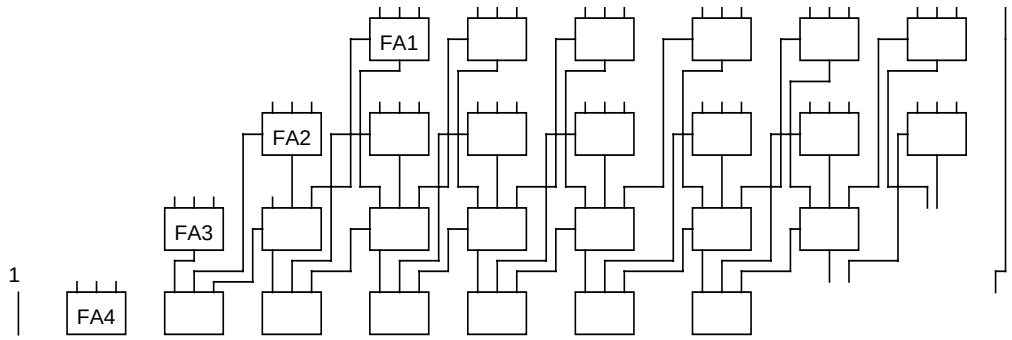


Fig. 3.4 då det enbart är en partialprodukt längs till höger bytes heladderaren ut mot en ledning

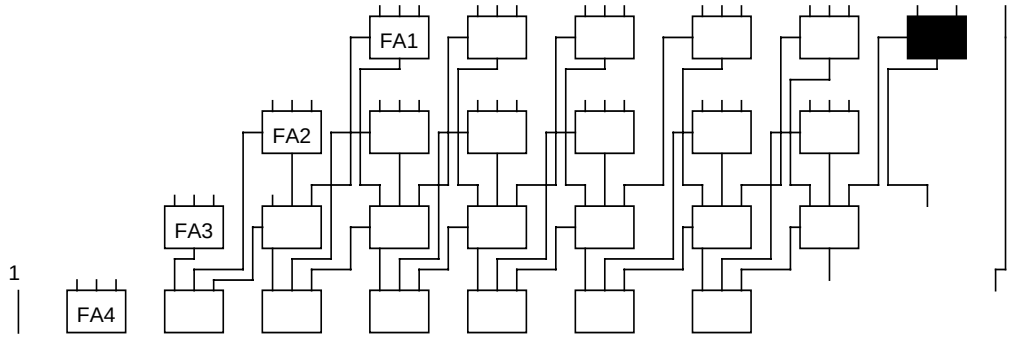


Fig 3.5 positionen näst längst till höger har två partialprodukter att beakta heladderaren bytes mot en halvadderare

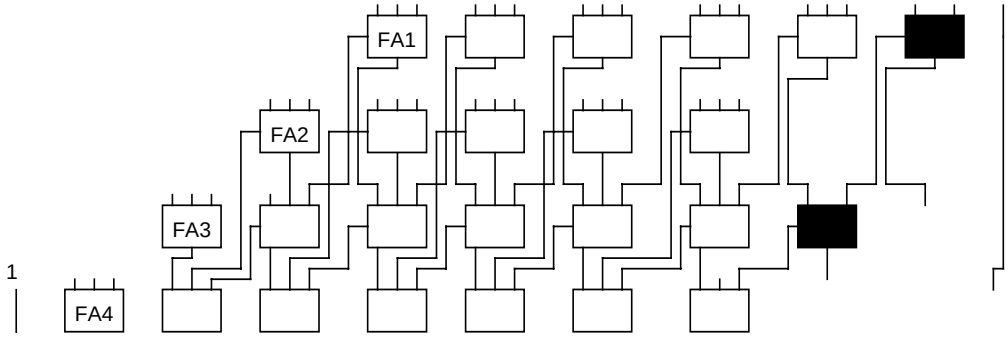


Fig. 3.6 I tredje kolumnen från höger beaktas tre partialprodukter, en heladderare kan avlägsnas, samt en heladderare bytes mot en halvadderare

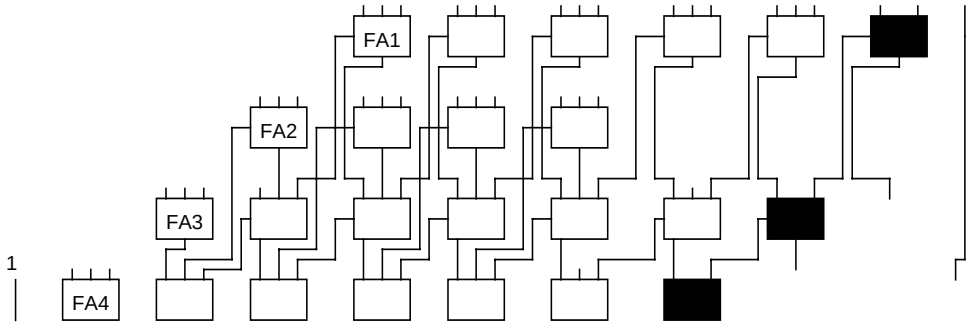


Fig. 3.7 I fjärde kolumnen från höger skall fyra partialprodukter beaktas, en heladderare avlägsnas samt en heladderare ersätts med en halvadderare

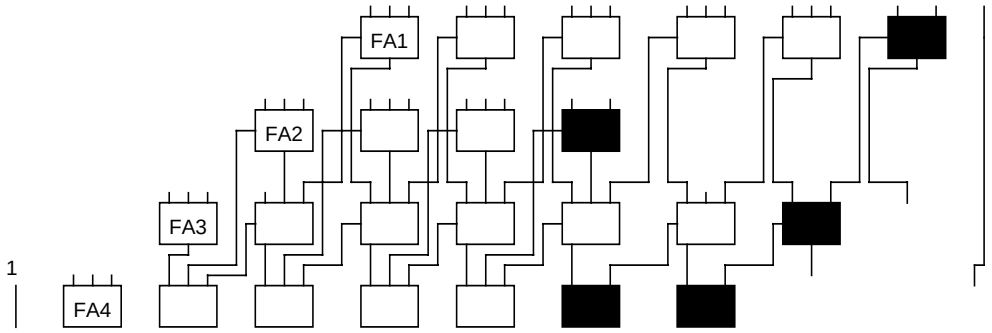


Fig. 3.8 I femte kolumnen från höger skall fem partialprodukter beaktas, två heladderare ersätts med två stycken halvadderare

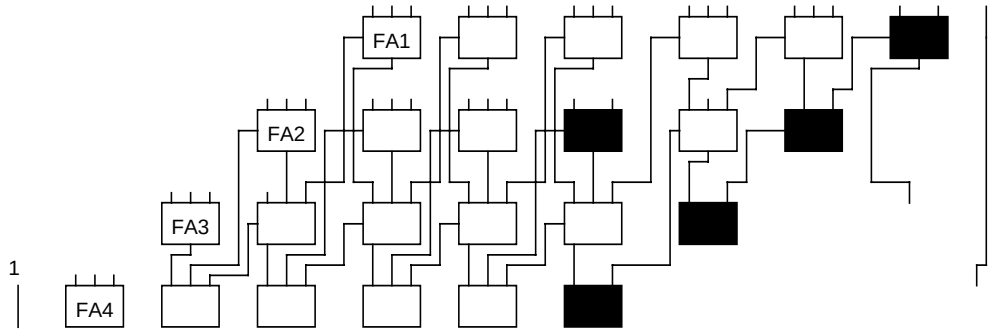


Fig. 3.9 Inga förenklingar kan göras i sjätte kolumnen, sex partialprodukter skall beaktas, sex ingångar finns. Heladderare och halvadderare justeras till, så strukturen får formen av ett parallellogram

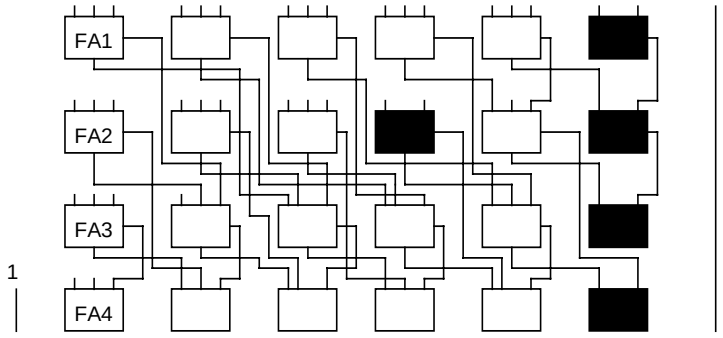


Fig. 3.10 Slutligen rätas parallellogramet upp, till en rektangulär array för att få en regelbunden form att implementera

Utsignalerna från varje adderare är delresultat från varje signifikansnivå. Dessa kopplas samman i en carryaccelerator, då *carry* annars måste transporteras hela vägen från LSB till MSB. *Sum*man och *carry* från varje adderare beaktas för att generera ett slutresultat så som visas i fig. 3.11

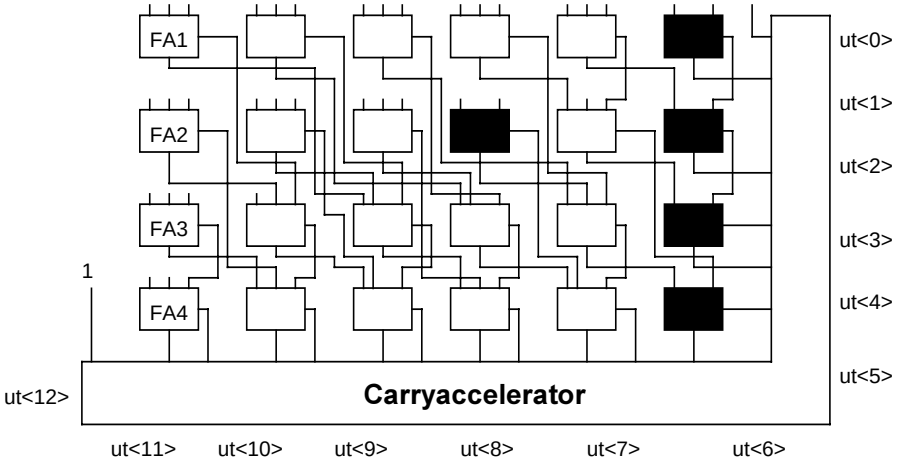


Fig. 3.11 en carryaccelerator används för att generera slutresultatet

3.3.1. Carryaccelerator

Med en carryaccelerator kan sista *carryn* för en grupp av heladderare propageras betydligt snabbare än i en vanliga adderare. Principen för en carryaccelerator visas i fig.3.12. Lämplig carryacceleratorstruktur är t ex Carry Look-ahead Adder (CLA), eller Brent-Kung Adder. Dessa har varierande prestanda och fysisk storlek. De olika strukturerna tas inte vidare upp i detta arbete.

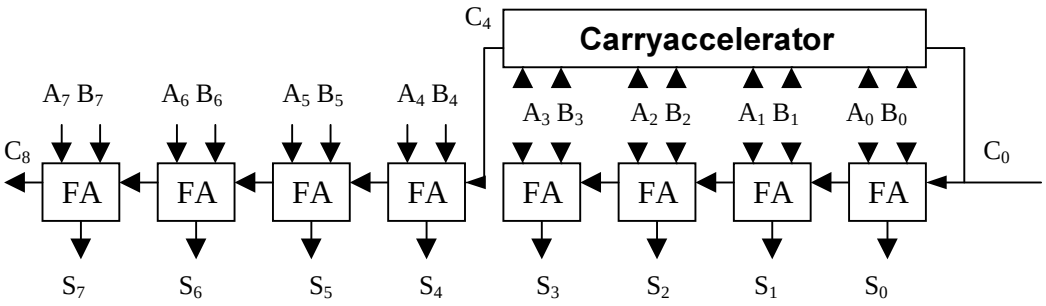


Fig.3.12 Principen för en carryaccelerator

4. Logik och layout

4.1. Heladderar cellen

För att konstruera heladderar cellen i layout användes Cadence layout verktyg Virtuoso, CMOS processen som användes var ST Microelectronics 0.18µm process. Logikstilen som har används, är CMOS. Bredden på nmos transistorn var 0.28µm, pmos transistorn var 0.84µm bredd, förhållandet mellan bredden pmos/nmos var 3/1. Heladderaren reducerar 3 ingångar *a*, *b*, och *cin'* till 2 utgångar, *sum* och *carry*. Modellen för heladderaren var en heladderare som används i kursen *Digitala Kretsar,* tillhandahållen av prof. Mark Vesterbacka på Linköpings universitet [8]. Den består av 13 stycken nmos transistorer, och lika många pmos transistorer, sammanlagt 26 transistorer. Se figur 4.1 för uppbyggnaden av heladderaren på transistornivå.

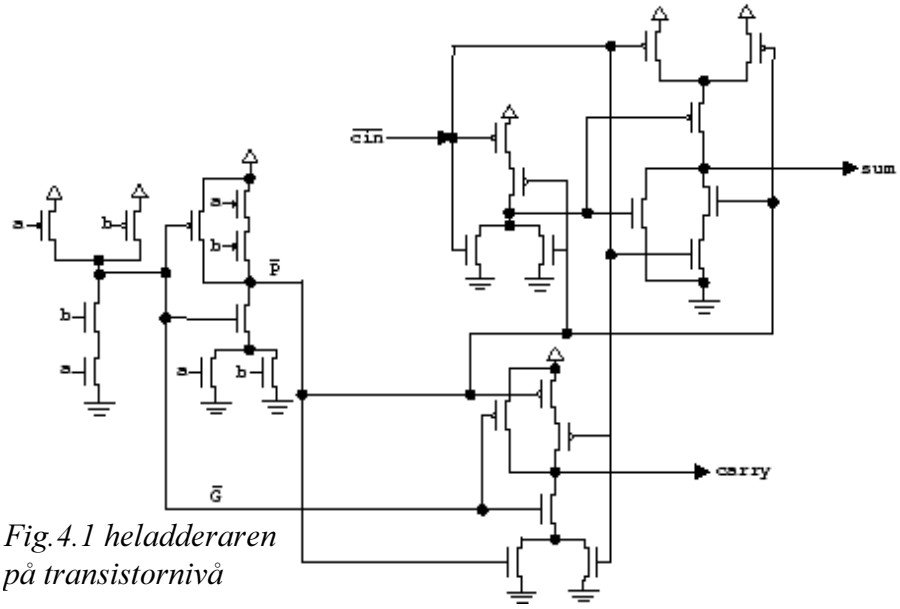


Fig.4.1 heladderaren på transistornivå

Generate(G') är en NAND funktion, *Propagate*(P') är en XOR funktion. Om *Generate* är hög genereras *carry*, när *Propagate* och *cin*' samtidigt är höga genereras också *carry*. Summakretsen(S) är en XOR funktion och carrykretsen(C') är en inverterad XOR funktion.

4.2. Heladderaren som layout

Heladderaren byggdes ”full-custom” i Virtuoso med början i enstaka transistorer för att erhålla en så kompakt design som möjligt. Detta är av stor betydelse då OS_1-trädet enbart består av heladderare, om man bortser från några få förenklingar som kan göras med

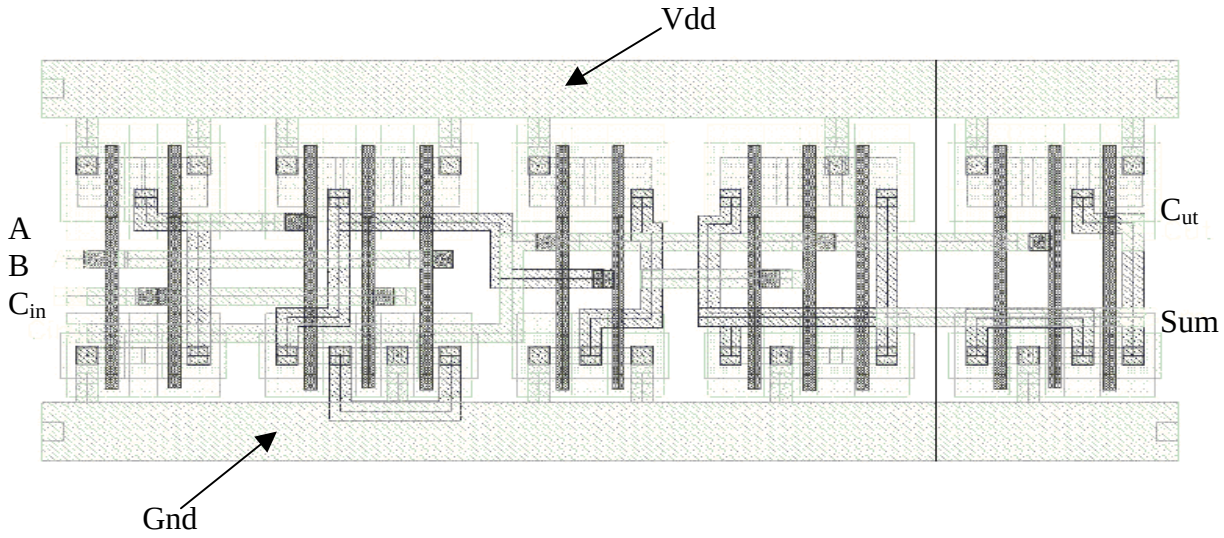


Fig. 4.2 Layout av heladderar cellen

halvadderare. Så liten area som möjligt vill uppnås för heladderaren då generatorn kommer att skapa ett stort antal av heladderare i trädstrukturen, och påverkar därigenom den slutliga chiparean. Vid konstruktion av heladderarcellen användes lagren *poly*, *metal1*, och *metal2*. Polylagret kopplar ihop gate mellan nmos och pmos transistorerna, polylagret som är något smalare än metallagren 0.18µm jämfört med 0.32µm. Polylagret bör dras så kort väg som möjligt p g a att stora kapacitiva och resistiva laster kan erhållas annars. *Metal1* används för att koppla ihop drain och source mellan pmos och nmos transistorerna, detta lagret drogs vertikalt. *Metal2* användes för att koppla ihop de olika grindarna i heladderarcellen, och drogs horisontellt. *Metal2* användes även för matningsspänning och jord. Figur 4.2 ovan visar den färdiga layouten av heladderaren.

4.3. Areaberäkning av layout

Antal adderare (hel eller halvadderare) är (N-2)(N-1) för en N×N multiplikator. Ett OS_1-träd behöver tre vertikala ”feedthroughs”, se tabell 2. Horisontellt behövs utrymme för de tre ledningarna. På grund av parallellogram-till-rektangel transformation, behövs lika mycket utrymme vertikalt. Antag att heladderaren har dimension C×D och att varje ledning har bredd *w*. Areabehovet för en N×N OS-träd multiplikator kommer då att vara (C+3 *w*)(D+3*w*)(N-2)(N-1). Detta är en ”värsta fallet” uppskattning, då vi antar att de tre FT:s inte överlappar varandra eller med heladderaren.

5. Cadence Virtuoso och SKILL

5.1. Virtuoso

Cadence Virtuoso är ett designverktyg för att skapa layout, som stödjer olika chiptillverkares processer. I detta projekt har ST Microelectronics 0.18µm process används. Med Virtuoso skapas layout med hjälp av transistorer, kopplingar och kontakter i olika metallager. Det finns speciella designregler rörande avstånd, storlek och metallager mellan kontakter och ledningar. Dessa designkrav kan kontrolleras med funktionen DRC, Design Rule Check. DRC kontrollerar på så sätt att gällande designregler för den aktuella tillverkningsprocessen efterlevs.

5.2. Texteditor

Texteditorn *NEdit* erbjuder SKILL-mode, den är lättanvänd och markerar funktioner och i SKILL reserverade uttryck med olika färger som gör koden lättläst. Dock är *NEdit* mindre bra på paranteshantering, vilket kan ge en del extra jobb.

5.3. SKILL

Alla objekt som skapas i Virtuoso layout-miljö ges ett specifikt ID som lagras i en databas som heter ”design database”. I Cadence programpaket så används programspråket SKILL som ett kommandospråk för att kommunicera med databasen. SKILL är en dialekt av språket Lisp, ett skriptspråk som inte behöver kompileras innan det exekveras. SKILL-program kan användas för att t ex koppla ihop färdiga byggblock i layout, generera flera celler från layout till en större krets i layout etc. Med SKILL matas kommandon direkt till Cadence-layoutverktyg genom kommandofönstret CIW (Command Interpret Window), alternativt skapas en programfil i en texteditor som har SKILL-mode t ex *NEdit* sedan kan denna kod laddas via kommandon i CIW-fönstret. Då detta görs exekverar programmet och layouten skapas.

5.3.1. Syntaxen i SKILL

Syntaxen för SKILL liknar till stora delar C språkets syntax med t ex for-, if-, while- och printf-satser. De logiska funktionerna såsom OCH samt ELLER och taljämförelser är detsamma som i C. Till detta finns färdiga specialkommandon för att skapa layout. Tillsammans ger detta ett kraftfullt programspråk anpassat för enbart hårdvarukonstruktion i Cadence Virtuoso. Tabell 4 ger några exempel på dessa specialkommandon.

Tabell 4

Kommando	Funktion
dbCreateInst	Skapar instans (cell)
dbCreateCellViewByType	Skapar en ny cellvy
rodCreatePath	Skapar ledare mellan två rod-objekt
rodAlign	Placerar två rod-celler bredvid varandra
leCreateContact	Skapar en kontakt
prog	Skapar lokala variabler i en funktion

Tabell 4. Förklaring av några vanliga kommandon i SKILL

5.3.2. Rod objekt och db

Rod står för Relative Object Design och syftet med rod-kommandon är att relatera olika layout objekt till varandra och för att förenkla hierarkisk åtkomst av layout celler mellan olika hierarkiska nivåer. Förkortningen db står för design database. I denna databas samlas all information om de olika cellerna som existerar i Cadence systemet. Dessa ges då ett databas ID-nummer, med information om objektets fysiska mått, vilka metallager som används, den geometriska orienteringen i layouten (x och y koordinater).

5.3.3. Lokala och globala variabler

En stor skillnad mellan C och SKILL ligger i hanterandet av variabler. I SKILL är alla variabler globala och behåller sitt värde även när programmet lämnar en funktion. Detta kan ställa till en del problem om samma variabel namn används i flera olika funktioner. Då användes lämpligen kommandot *prog* som gör variabel lokal till den funktion eller ett program avsnitt som variabel användes i eller om så önskas gör variabel lokal i en bestämd del av funktionen. På nästa sida ges ett kort programexempel för layout generering i SKILL.

5.3.4. Programmerings exempel i SKILL

```
procedure (run(N)

1.  cv2=dbOpenCellViewByType(”Full_adder”  ”fa_matris”  layout”  ”maskLayout” ”w”)

2.  bitslice=dbCreateInst(cv2 cv  ”bitslice”  list(0 0)  ”R0”  1)

3.  for( j 1 N
4.              sprintf ( bitslice_name ”bitslice.%n”  1  j )

5.  firstkolumn=dbCreateInst(cv2  cv  bitslice_name  list(0 0)  ”R0”  1)

6.  rodAlign(
?alignObj       rodGetObj(strcat( firstkolumn~>”name”  ”/Heladd_small/gndl”)cv)
?alignHandle  ”cC”
?refObj         rodGetObj(strcat(oldfirstkolumn~>”name”  ”/Heladd_small/gndr”) cv)
?refHandle     ”cC”
?maintain       nil
)
)
oldfirstkolumn=firstkolumn
)
```

Kort förklaring till kodexemplet

1. Öppnar upp en cellvy (cv2) för skrivning (w), Fulladder är här namnet på bibloteket som layout ska ligga i, och fa_matris är cellnamnet

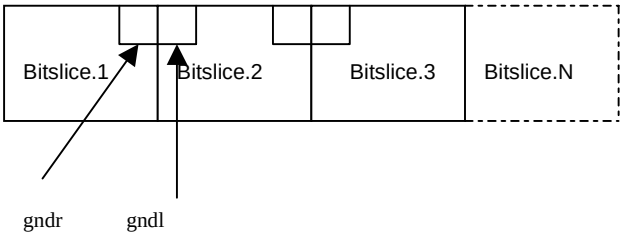
2. Hämtar instans med namnet bitslice från cellvy (cv), för att lägga ut cellvy2 (cv2) enligt punkt 1 här står ”R0” för vinkeln som den ska läggas ut i (0°) kan varieras i jämna 90° intervall, list(0 0) är x och y koordinaterna och 1 är antalet instanser.

3. For-sats där antalet utlagda celler kan varieras från 1 till N.

4. Ger löpande namn till cellerna, bitslice.1 till bitslice.N, mha %-operatorn.

5. Första cellen namnges till firstkolumn, efter utplacering sparas pekaren till den första cellen oldfirstkolumn, nästa cell blir då firstkolmn osv till N antal.

6. Med rodAlign läggs cellerna bredvid varandra enligt följande princip, referensobjekt är oldfirstkolumn objektet som skall flyttas är firstkolumn, kontakterna namngivna gndr(right) och gndl(left) läggs bredvid varandra för varje ny cell.



6. Uppbyggnaden av SKILL programmet

6.1. Designflöde

SKILL koden till generatorn skrevs i 2 olika filer, *bitslice2.il* och *rout2.il*, alla SKILL-filer ges prefixet .il. Figur 6.1 nedan visar designflödet från en enkel heladderare konstruerad i Virtuoso, till färdig multiplikator genererad med SKILL kod. Den färdiga heladderarcellen *Heladd_small*, som med filen *bitslice2.il* bildar en bitslice av heladderarcellen. Filen *rout2.il* lägger ”bitslicarna” kant till kant till en array av ”bitslicar”som bildar den färdiga multiplikatorstrukturen.

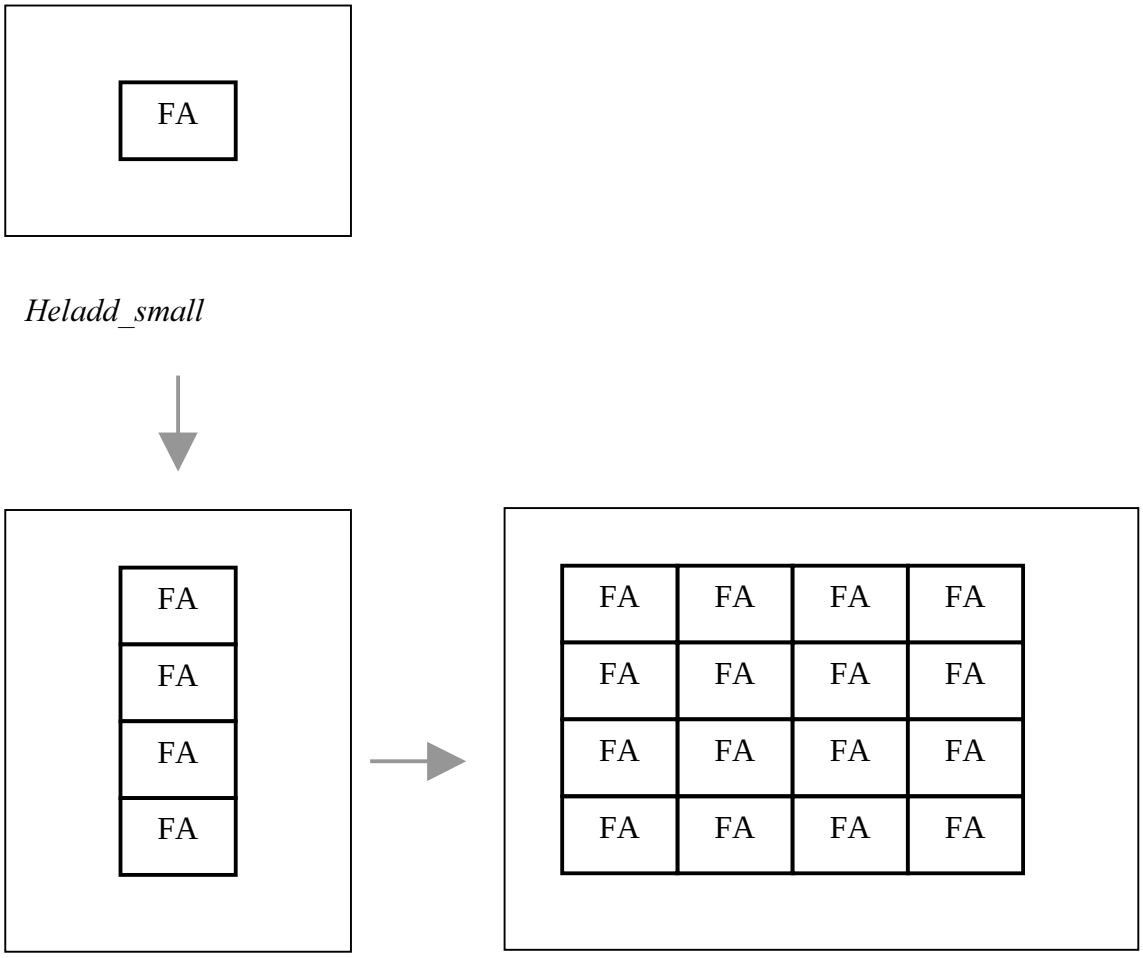


Fig. 6.1

6.2. Bitslice filen

Filen *bitslice2.il* tar den i Virtuoso(layout) konstruerade heladderar cellen, mastercellen och kopierar denna önskat antal gångar och genererar en kolumn av heladderare, en ”bitslice” med höjden 7 till 16 stycken heladderare. Antalet heladderare i kolumnen är beroende på antalet ingångar som varierar mellan 8 och 18 ingångar (N). Detta representerar 7 olika fall med 7, 10, 11, 13, 14, 15 och 16 stycken heladderare i ”bitslicen” beroende på antal ingångar som önskas. Lösningen har varit att se på de olika utfallen som varit möjlig och satt upp villkor för dessa med hjälp av if- och cond-satser. Koden skrevs så att varannan heladderar cell lades upp och ned för att på så sätt kunna utnyttja att matningsspänningen(Vdd) och jord(Gnd) mellan cellerna kunde kopplas samman och ge en kompakt struktur. För att orientera och lägga heladderarna kant mot kant med varandra användes kommandot rodAlign.

6.3. Routing och array filen

Filen *rout2.il* placerar ut genererad ”bitslice” från filen *bitslice2.il* och placerar dessa kant i kant med hjälp av rodAlign kommandot. På detta sätt bildas en array av ”bitslicar” som bildar den fysiska strukturen till multiplikatorn. Antalet ”bitslicar” i arrayen är N+1 stycken, där N är antalet bitar på operanden. Filen *rout2.il* innehåller också den kod som behövs för att generera de interna ledningsdragningarna mellan ”bitslicar” och heladderarna. De interna ledningsdragningarna varierar med antalet ingångar till multiplikatorn, in/utgångar kopplas olika för olika fall, beroende på ”bitslicens” höjd. Algoritmen för de interna ledningarna är de som har tagit mest anspråk, i form av tid och antalet rader kod. Här har varje möjligt utfall för storlek på ”bitslicen” beaktas samt kopplingarna mellan in- och utgångarna. Med uppställda villkor för de enskilda fallen, i koden motsvarar det här if- och cond-satser. Detta har gjort det möjligt att återanvända kod med mindre modifieringar. Motivet för detta har varit att programmeringen har blivit enklare, på bekostnad av exekveringstiden för programmet. Om exekveringstiden för generatorn t ex tar 3 sek istället för 6 sek får här anses vara av mindre betydelse. De interna ledningarna mellan heladderare och ”bitslicar” i multiplikatorstrukturen läggs ut i andra metallager än de som används för heladdararen, här *metal3* och *metal4*. De ligger ovanpå de genererade heladderarna och påverkar därför inte de fysiska måtten på multiplikatorstrukturen. *Metal3* användes för utgången *cut* , *metal4* användes för utgången *sum*. För ledningsdragning mellan in- och utgångar användes kommandot *rodCreatepath*.

7. Resultat

Den i SKILL implementerade generatorn genererar en multiplikator för två 2-komplementstal från 8 till 18 bitar. OS-trädstrukturen har kombinerats med Baugh-Wooley's algoritm. Det fysiska måttet på heladderaren är 6.94×16.72 [μm]. Den största genererande multiplikatorn, för 18×18 bitar har de fysiska måtten 95.72×318.02 [μm]. Då har totalt 7072 transistorer används. Generatorn i SKILL innehåller ca 2.000 rader kod. Mer uppgifter om koden som skrivits, kan ses i Appendix A. I Appendix B visas layout för en implementerad 8×8 bitars multiplikator med tillhörande "bitslice".

7.1. Problem

Mycket av tiden gick åt för att lära sig Cadence programpaket det är avancerat och svårtillgängligt till en början, men har sedan en mängd funktioner som är uppskattade och gör att det går snabbt att jobba med. Algoritmen i SKILL för att dra ledningarna mellan de olika heladderarna har tagit mycket tid, då det inte är regelbundna ledningsdragningar utan ändras från fall till fall med antalet ingångar. Det skulle ha varit intressant att testa den slutliga designen ur energiförbrukningssynpunkt men detta har inte hunnits med.

7.2. Förbättringar

Förbättringar som skulle kunna göras är; en generator som kan göra förenklingar i de tre sista kolumnerna, för att på så sätt få ner chiparean; hänsyn till olika behov av drivförmåga hos heladderarna beroende på längden på ledningarna i multiplikatorn; samt en bättre optimerad kod är alla önskemål på förbättringar.

7.3. Slutsats

En multiplikator generator för ett godtyckligt antal bitar är svår att konstruera speciellt om hänsyn till de förenklingarna i de tre sista kolumnerna skall göras. I denna lösning har inga hänsyn tagits till de förenklingar med neddragningar av ingång direkt till utgång, eller att heladderare bytes mot halvadderare, när detta varit möjligt. Att skriva en algoritm i SKILL som beaktar detta för olika antal operander kräver mer än goda kunskaper och vana i programmering och får ses som överkurs och ligger utanför detta arbete. Inga hänsyn har tagits till olika drivförmåga hos heladderarna beroende på om det har varit kort eller lång ledning att mata informationen över. I denna lösning av en multiplikatorgenerator har det "slösats" med heladderare. Vid förenklingar av de tre sista kolumnerna kan enligt exempel på sidan 12, kan besparing på en kolumn göras. För en 6×6 multiplikation blir besparingen 4 heladderare av totalt 28 stycken heladderare. Att konstruera en generator som tar hänsyn till förenklingar, olika behov av drivförmåga hos heladderarna samt att strukturen ändrar sig beroende på antalet operander för anses som många variabler att ta hänsyn till och mycket tid bör gå åt för att ta fram en fullt fungerande generator. Det kan då vara mer tidsbesparande att konstruera en multiplikator för "hand" för varje enskilt behov.

Referenser

Litteratur

[1] Hemert, Hugo, *Digitala Kretsar*, Studentlitteratur, Lund, ISBN 91-44-00099-5, 1996

[2] Wanhammar, Lars, *DSP integrated circuits*, Academic Press, San Diego, ISBN 0-12-734530-2, 1998

[3] Kang, Sung-Mo och Leblebici, Yusuf, *CMOS Digital integrated circuits*, Mc Graw Hill, Boston, ISBN 0-07-292507-8, 1999

Rapporter och dokument

[4] Amaya, José och Lindberg, Anders, *Layoutgenerering av bitparallell multiplikator*, Linköpings tekniska högskola, ISY, LiTH-ISY-EX-ET-0135, 1997

[5] Nyqvist, Björn, *Utredning av multiplikatorgenerator*, Linköpings tekniska högskola, ISY, LiTH-ISY-EX-ET-0156, 1999

[6] Gustafsson, Oscar et al., *Understanding Multiplier Design using "Overturned-Stairs" Adder Trees*, Linköpings tekniska högskola, ISY, LiTH-ISY-R-2016, 1998

[7] Hjalmarson, Emil, *Introduction to Skill*, Linköping, 2001

[8] Vesterbacka, Mark, *Digitala kretsar proj. kurs*, Linköpings tekniska högskola, 1999

[9] Andreani, Pietro et al., *CADENCE 4.4 En sammanfattning*, Lunds tekniska högskola, 1999

[10] Baugh, C.H och Wooley, B, *A two's complement parallel array multiplication algorithm*, IEEE Transactions on Computers, vol. C-22 sid 1045-1047, 1973

[11] Mou, Zhi-Jian och Jutand, Francis, *Overtuned-stairs, adder trees and multiplier design*, IEEE Transactions on Computers, vol. 41, sid 940-948, 1992

[12] Karlsson, Magnus, *A generalized carry-save adder array for digital signal processing*, OKG AB, SE-572 83 Oskarshamn, 2000

Appendix A

Då skriven kod motsvarar ca 30 sidor redovisas detta inte i tryckt form. Koden till multiplikatorgeneratorn ligger i filerna *rout2.il* och *bitslice2.il.* Dessa ligger under katalogen /home/tde/klasa. Programmet startas i CIW fönstret med kommandot *run2* (*N), N* är här antalet bitar som önskas på multiplikatorn (8-18).

Appendix B

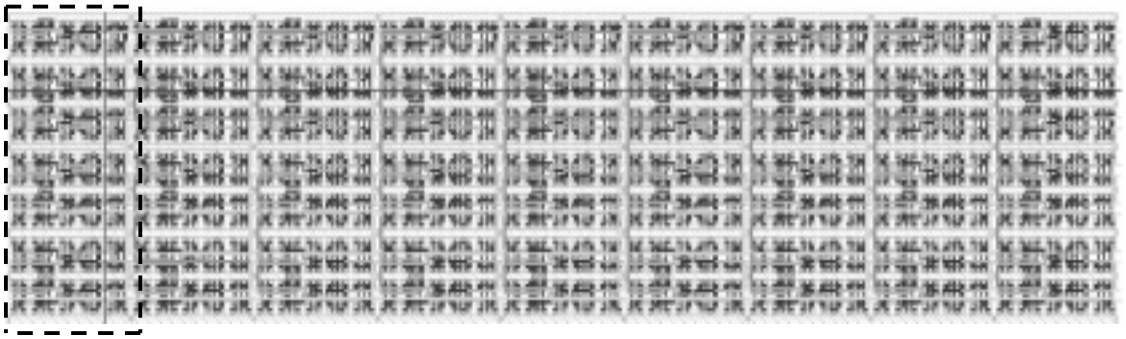


Fig.B.1 Layout för en 8×8 bitars multiplikator

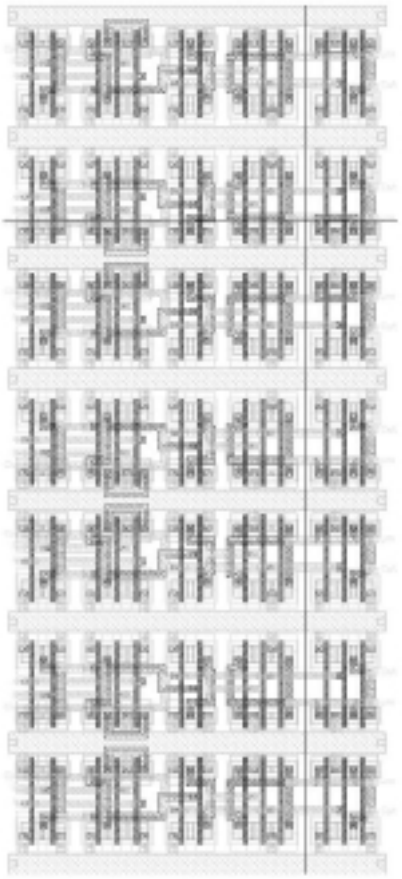


Fig.B.2 Bitslice till multiplikatorn, 7 st heladderare