G-Opgave 1

Michael Thulin, Philip Munksgaard

2. oktober 2010

Implementering af 4-bit ALU

Opbygning

Vi har fulgt den i opgaveformulering anbefalede fremgangsmåde og er startet med at implementere en 1-bit full adder, dernæst en 1-bit ALU og til sidst en 4-bit ALU.
Vi valgte for enkelthedens skyld at lave en traditionel ripple carry adder. Vi kunne have valgt at lave en mere kompliceret carry lookahead adder, og hvis vi havde haft mere tid kunne det sikkert have været en givtig øvelse.

1-bit full adder

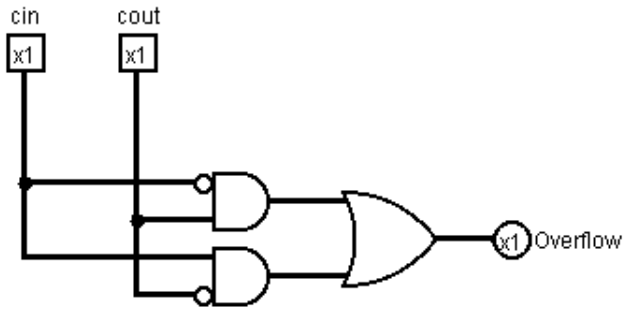
1-bit full adderent består af en carry-komponent og en sum-komponent. Vi startede derfor med at implementere hver af disse komponenter som delkredsløb i logisim, for derefter at sætte dem sammen i en 1-bit full adder. Da vi havde valgt ikke at lave en carry lookahead adder var dette ganske simpelt.

1-bit ALU

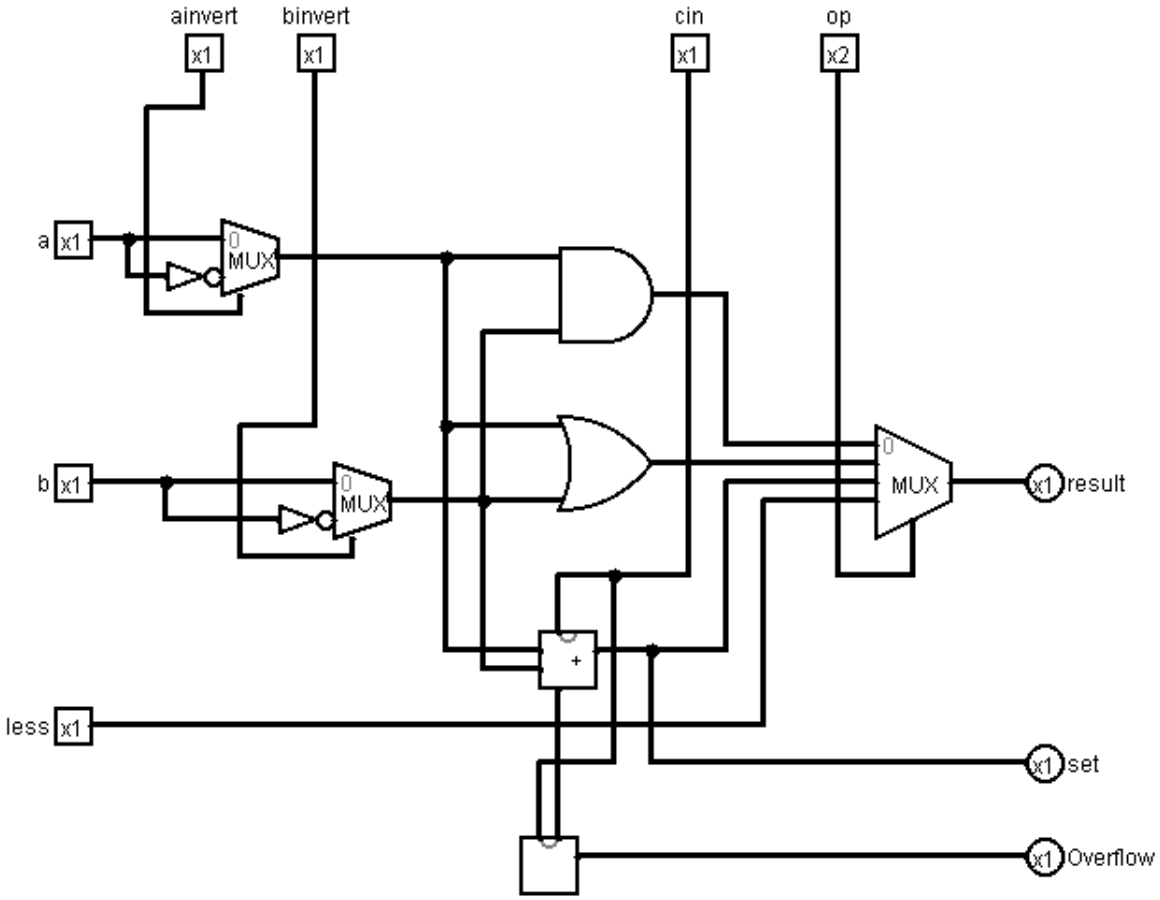
Igen valgte vi for nemhedens skyld at følge beskrivelsen i Appendix C, hvorfor 1-bit ALUen var relativt simpel at implementere som delkredsløb.

4-bit ALU

For at kunne lave en 4-bit ALU, var vi nødt til at lave en komponent som kunne detektere overflow. Vi valgte at bruge resultatet fra opgave C.25, så vores overflow detector kontrollerer bare, at cin og cout i ALU for den mest betydende bit ikke er det samme.



Figur 1: Vores overflow-komponent.



Figur 2: Vores implementering af en 1-bit ALU med overflow detection.

Enkeltcyklusarkitektur

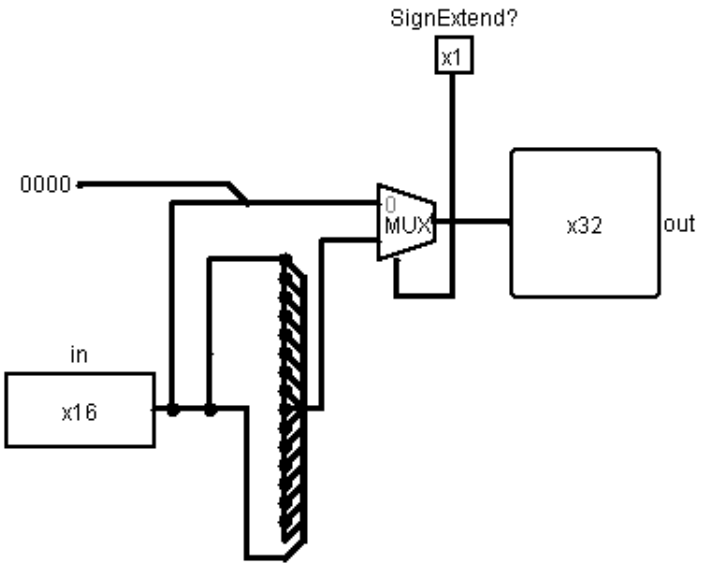
Vi valgte at tage udgangspunkt i figur 4.17 i COD. Vi har desuden valgt, ikke at bruge vores egen ALU, men den ALU der fulgte med logisim.

16-32 bit extender

Vi startede derfor med at implementere vores 16-32 sign extender. Dette gjorde vi ved at tage den mest betydende bit i input-værdien og kopiere den ud på de 16 mest betydende bits i output-værdien.
Udover en sign extender havde vi også brug for en zero extender, så vi valgte at bygge den ind i vores extender-komponent. Hvis SignExtend? er sat sign extender vi, og ellers bliver tallet zero extendet.

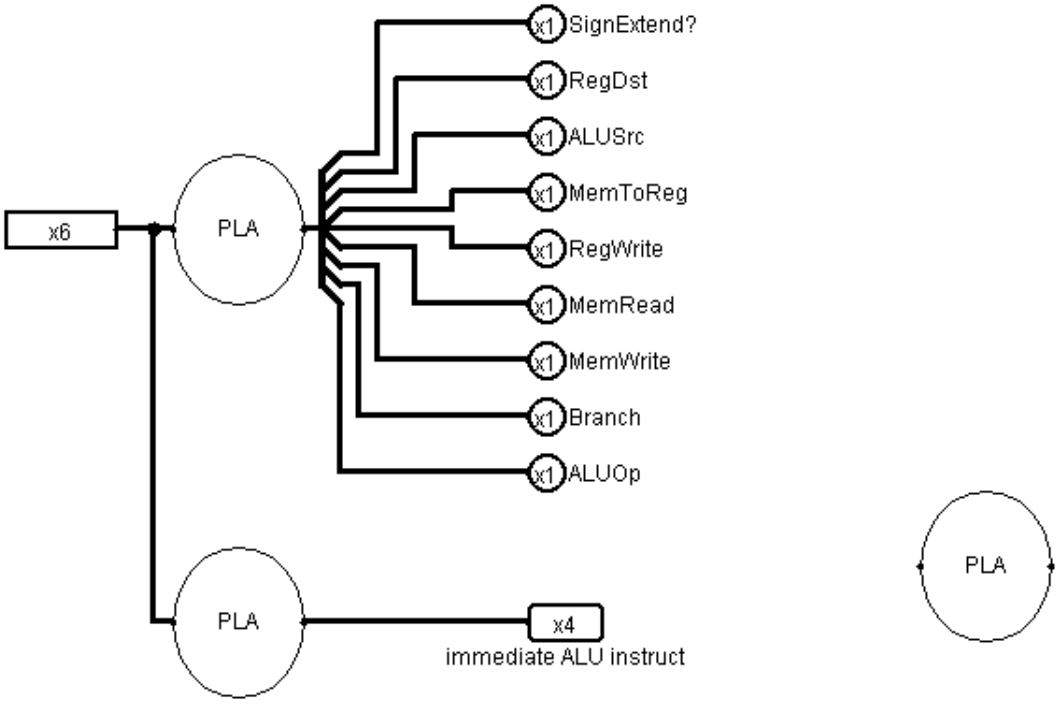
Control

Vi har valgt at bruge PLAer til at understøtte vores sandhedstabeller både i Control og i ALUControl. Dette gjorde det til en simpel opgave at mappe OP-codes til Control-lines.
For at kunne understøtte immediate instruktioner valgte vi at lægge en ekstra PLA ind i vores Control delkomponent, som kunne mappe OPcodes til ALU operationer.
Af denne grund kunne vi simplificere vores ALUOp controllines, således at alle R-instruktioner fik ALUOp 1 og resten af instruktionerne fik ALUOp 0. Det vil altså sige, at vi har lavet en lille



Figur 3: Implementering af 16-32 sign extender.

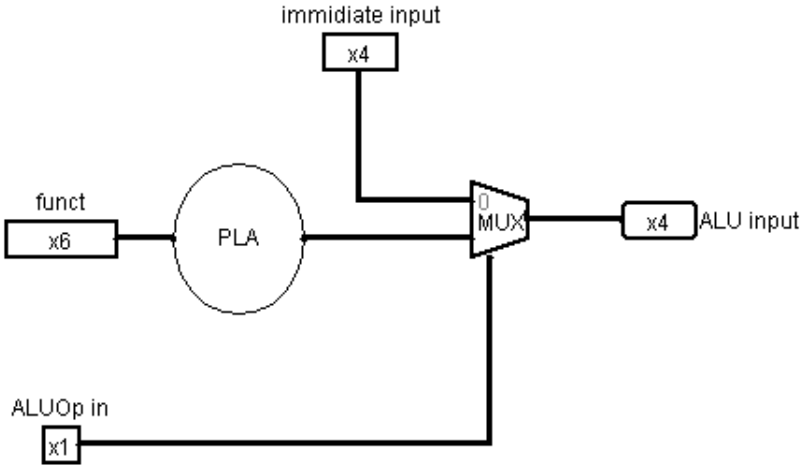
smule om i forhold til implementationen i bogen.
Desuden skulle vi afgøre, for hver instruktion, om der skal sign extendes eller zero extendes. Vi har fulgt det der står i MIPS instruktionsarket forrest i COD. Eneste instruktion der var lidt tvivl om var beq, men da beq fungerer ved at springe relativt i forhold til det aktuelle instruction count, er det nødvendigt at den kan springe baglæns, hvorfor BranchAddr skal kunne være negativ. Derfor skulle BranchAddr sign extendes.



Figur 4: Vores Control unit.

ALU Control

Her har vi brugt en multiplexer, som vælger om ALU instruktionen skal komme fra funct feltet i en R-instruktion, eller fra den ALU instruktion vi genererer i Control-komponenten. Hvis ALUOp er 1 bruger vi funct og hvis ikke bruger vi instruktionen fra Control.
Også her har vi valgt at bruge en PLA til at mappe fra funct til ALU instruktion.



Figur 5: Vores ALUControl komponent.

Test af MIPS-fortolker

Vi har lavet 2-3 tests af hver MIPS-instruktion. Programmet test.asm kan ses i bilag A. Alle instruktioner fungerer efter hensigten og testen fejler ikke på noget tidspunkt.

Bilag A: Test af MIPS fortolker

```
# test af MIPS instruktioner.

addiu $1, $0, 0x1               # $1 = 0 + 1 = 0x1
addiu $2, $0, 0xc               # $2 = 0 + 12 = 0xc
addiu $3, $0, 0xffff            # $3 = 0 + 0xffff = 0xffffffff (pga sign extend)
addu  $4, $1, $2                # $4 = 1 + 12 = 0xd
subu  $5, $2, $1                # $5 = 12 - 1 = 0xb
subu  $6, $1, $2                # $6 = 00000001 - 12 = 0xfffffff5
and   $7, $4, $5                # $7 = 1011 and 1101 = 0x9
and   $8, $0, $5                # $8 = 0000 and 1011 = 0x0
or    $9, $4, $5                # $9 = 1101 or 1011 = 0xf
or    $10, $5, $0               # $10 = 1011 or 0000 =0xb
slt   $11, $2, $5               # $11 = 12 < 11 = 0
slt   $12, $5, $2               # $12 = 11 < 12 = 1
andi  $13, $2, 0x6              # $13 = 1100 and 0110 = 0x4
andi  $14, $0, 0xffff           # $14 = 0 and 0xffff = 0x0
sw    $7, 4($1)                 # store the contents of $7 (0x9) in memory 0x4
lw    $15, 4($1)                # read memory 0x4 into register 15 (9)
slti  $16, $2, 0xa              # $16 = 12 < 10 = 0
slti  $17, $2, 0xff             # $17 = 12 < 0xff = 1
ori   $20, $10, 0x2             # $20 = 0xb or 0x2 = 0xf
ori   $21, $13, 0x8             # $21 = 0x4 or 0x8 = 0xc

addu $18, $1, $0                # load contents of $1 into $18, $18 = 1
Loop:   addiu $18, $18, 0x1     # add one to $18
        slti $19, $18, 0x5      # check if 18 is still less than 5
        beq $19, $zero, Exit    # if it isn't we're done, go out of the loop
        beq $zero, $zero, Loop  # if it is, loop again.

Exit:
```