Machine Architecture
Assignment 2

Casper B. Hansen
University of Copenhagen
Department of Computer Science
fvx507@alumni.ku.dk

Sine Vestergård Jensen
University of Copenhagen
Department of Computer Science
kms698@alumni.ku.dk

Nikolaj Høyer
University of Copenhagen
Department of Computer Science
ctl533@alumni.ku.dk

20. oktober 2013

Resumé

Følgende beskriver løsning af G2 gruppeopgaven til maskinarkitektur på Københavns Universitet, samt forløbet af udarbejdelsen heraf.

Ved at tage udgangspunkt i et simplere kredsløb, nemlig *single-cycle* kredsløbet, forsimplede vi omfanget af opgaven om, at bygge et *pipelined* processor kredsløb, som understøtter en delmængde af MIPS[1] instruktionssættet.

Efter vi havde en fungerende implementation af *single-cycle* kredsløbet, indførte vi *pipes*, som effektivt delte kredsløbet op i eksekveringsstadier, og herefter løste vi de introducerede *hazards* i logisk rækkefølge af deres forekomst.

Under hver udbedring af hvert stadie opfulgte vi med at teste kredsløbet, og reitererede løsningen, om nødvendigt.

Sidst opfulgte vi med omfattende tests af det endelige kredsløb, som sikrede os, at vi fange de sidste fejl.

[1]MIPS - http://en.wikipedia.org/wiki/MIPS_architecture

Indhold

1 Indledende **2**

1.1 Instruktionssæt 2
1.2 Implementering af Single-cycle 2
1.3 Konvertering til Pipeline 2
1.4 Forwarding og Hazard Detection 3
1.5 Done og virker 3
1.6 Fejl og mangler 3

2 Tests **4**

2.1 Aritmetiske og logiske operationer 4
2.2 Branching operations 5
2.2.1 Jumps 5
2.3 Memory operationer 6
2.3.1 Branch Hazards/-Forwarding 7

1 Indledende

Vi startede med at lave en single cycle, som vi lavede om til en pipeline, i nedenstående afsnit har vi forklaret processen fra single cycle til pipeline.

1.1 Instruktionssæt

Vores pipelines instruktionssæt (se Figur 1) er relativt simpelt og indeholder 'kun' 14 instruktioner. På følgende tabel er disse instruktioner blevet beskrevet kort.

Arithmetic-logical

Mnemonic	Code	Type	Description
`addu`	`0x21`	`R`	Add unsigned
`addiu`	`0x09`	`I`	Add imm. unsigned
`slt`	`0x2A`	`R`	Set less than
`slti`	`0x0A`	`I`	Set imm. less than
`subu`	`0x23`	`R`	Subtract unsigned
`and`	`0x24`	`R`	Logical AND
`andi`	`0x0C`	`I`	Logical imm. AND
`or`	`0x25`	`R`	Logical OR
`ori`	`0x0D`	`I`	Logical imm. OR

Memory-reference

Mnemonic	Code	Type	Description
`lw`	`0x23`	`I`	Load word
`sw`	`0x2B`	`I`	Store word

Branching

Mnemonic	Code	Type	Description
`beq`	`0x04`	`I`	Branch on equal
`jal`	`0x03`	`J`	Jump and link
`jr`	`0x08`	`R`	Jump to register

Figur 1: Instruktionssæt

1.2 Implementering af Single-cycle

Da vi mødtes første gang tog vi simpelthen udgangspunkt i et helt nyt kredsløb, og byggede et en fungerende *single-cycle* implementation af den delmængde af MIPS instruktionssættet, som vi skulle understøtte jvf. opgaveteksten i vores løsning.

Vi fik således alle instruktioner til at virke korrekt på første dag, undtagen instruktionerne `jal` og `jr`. Disse blev implementeret umiddelbart dagen efter.

1.3 Konvertering til Pipeline

Da *single-cycle* kredsløbet fungerede korrekt indførte vi *pipes*, som er mellem-registrer, hvori vi midlertidigt gemmer værdier beregnet i, eller overført fra, sidste stadie (eg. `IF/ID`, hvorfra vi viderefører *program counter* (PC) efter at have beregnet den følgende instruktions adresse, den nuværende instruktion, samt returneringsadressen for *jumps* og et *stall*-signal.

Vi introducerede alle *pipes* på én gang, da det var svært at se, hvordan vi skulle have håndteret en halvt indført pipeline, og tog de fejl som det medførte en efter en, herunder; at rykke *branch*-delen op i *pipelinen* (i `IF`-stadiet), hvilket medbragte den fordel, at vi ikke skulle tænke så meget over *branch-delay slot*, i og med, at den følgende *fetched* instruktion gerne skulle udføres, men PC derefter er blevet ændret til *branch*-adressen.

1.4 Forwarding og Hazard Detection

Da de indledende fejl fra konverteringen var blevet reduceret og vi fik et overskueligt overblik over kredsløbet igen, påbegyndte vi at tage os af *hazards*, hvoraf vi tog os af de nemmeste og mest åbenlyse problemer først, herunder; forwarding for de aritmetiske operationer (eg. `addiu` og `subu`), og derefter *hazard-detection* og *stall* for *memory* operationer (eg. `sw` og `lw`).

Til sidst tog vi os af *jumps* og *branches*, hvilket viste sig at være mere udfordrende end først antaget, og også årsagen til, at vi må erkende, at vores kredsløb har nogle fejl på dette område.

For det første er `jal` implementeret sådan, at `ID`-trinnet håndterer både hvilken værdi der skal skrives i returregistret (sendt direkte fra `IF`-trinnet), samt signalet til rent faktisk at skrive. Dette har den ulempe, at hvis en instruktion senere i pipen (fx. en `addu`) forsøger at skrive til registret, vil dette være 'optaget' og det vil kun være retur-addressen der bliver skrevet. Dette kunne muligvis løses ved *hazard detection*, der kunne stalle hele pipelinen et enkelt trin, så begge værdier kunne nå at blive skrevet. Dette har vi desværre ikke nået at implementere. Bortset fra denne ulempe fungerer `jal` dog som den skal.

Derudover er `jr` også implementeret uden *hazard detection* og *forwarding*, hvilket medfører at en instruktion der forsøger at skrive til register `$31` (`$ra`), ikke vil nå igennem pipen før `jr` udføres. Hvis man her antager, at assembler-programmøren følger MIPS konventionen og kun skriver til `$ra` med `jal`, vil dette ikke blive et problem, eftersom `jal` allerede skriver til registret i `ID`-trinnet.

Til gengæld er har vi indført en seperat hazard detection unit for branches, der sørger for at stalle pipelinen, hvis branch operationen afhænger af endnu ikke produceret output. For eksempel vil pipelinen blive stallet to gange i tilfælde af at branch operationen afhænger af en lw operation i instruktionen netop før branchen. Det har også været nødvendigt med endnu en forwarding unit i ID trinnet, udelukkende til branches, som sørger for at hente fx. adds elle lignende tilbage til branch operationen.

1.5 Done og virker

hvad virker og hvorfor virker det (aka se test)

1.6 Fejl og mangler

Har vi noget der ikke virker og hvordan har vi tænkt os at implimentere det.

2 Tests

Herunder vil vi foretage en gennemgang af de tests vi har udført på kredsløbet iht. de forventede resultater, og sammenligningen deraf.

2.1 Aritmetiske og logiske operationer

Ved at sammenligne de to registrer, som bruges ved et aritmetisk eller logisk operation med de som allerede ligger i *pipen* kan vi forudse, om disse bør anvendes istedet for — altså forwarding.

I det følgende udregnes $((3 + 4) + 7) - 3$.

```
1  addiu   $s0, $zero, 3
2  addiu   $s1, $zero, 4
3  addiu   $s2, $zero, 7
4  addiu   $s3, $zero, 1
5  addu    $t1, $s0, $s1
6  addu    $t1, $t1, $s2
7  subu    $t2, $t1, $s0
8  slt $t3, $t2, $t1
9  slti    $t4, $s0, 4
10 and $t5, $s3, $s3
11 andi    $t6, $t5, 0
12 or  $t7, $s3, $t6
13 ori $t8, $t7, 0
```

Som det forventes producerer ovenstående instruktioner nøjagtig de forventede data i registrerne (se Figur 4).

$zero	00000000	$s0	00000003
$at	00000000	$s1	00000004
$v0	00000000	$s2	00000007
$v1	00000000	$s3	00000001
$a0	00000000	$s4	00000000
$a1	00000000	$s5	00000000
$a2	00000000	$s6	00000000
$a3	00000000	$s7	00000000
$t0	00000000	$t8	00000000
$t1	0000000*e*	$t9	00000000
$t2	0000000*b*	$k0	00000000
$t3	00000001	$k1	00000000
$t4	00000001	$gp	00000000
$t5	00000001	$sp	00000000
$t6	00000000	$fp	00000000
$t7	00000001	$ra	00000000

Figur 2: Forventet resultat

2.2 Branching operations

...

2.2.1 Jumps

I det følgende testes branching, `jr` og `jal`.

```
1  # Her testes for branching, jr og jal.
2  # Status: PASS
3  # ----------------------------------------- #
4  #       Instruktion      |    Forventet output
   #
5  # ----------------------------------------- #
6  addiu   $s0, $zero, 4    # s0: 4
7  addiu   $s1, $zero, 2    # s1: 2
8  jal label                #
9      nop                  # 1. og 2. gang jumpes
10     nop                  #   - ra: 16
11     nop                  #
12 label:                   # jump hertil
13 beq $s0, $s1, next   # tages trejde gang
14 addiu   $s1, $s1, 1      # Taelles op til 5
15 jr  $ra                  #
16 next:                    # branch hertil
17     nop                     #
18 addiu   $s2, $zero, 3    # s2: 3 - men bliver aldrig skrevet! fail
19     nop
20     nop
21 jal end
22 nop
23 end:
24     nop
```

Som det fremgår af register `$s2` kan vi se fejlen der opstår (se Figur 3).

$zero	00000000	$s0	00000004
$at	00000000	$s1	00000002
$v0	00000000	$s2	00000000
$v1	00000000	$s3	00000000
$a0	00000000	$s4	00000000
$a1	00000000	$s5	00000000
$a2	00000000	$s6	00000000
$a3	00000000	$s7	00000000
$t0	00000000	$t8	00000000
$t1	00000000	$t9	00000000
$t2	00000000	$k0	00000000
$t3	00000000	$k1	00000000
$t4	00000000	$gp	00000000
$t5	00000000	$sp	00000000
$t6	00000000	$fp	00000000
$t7	00000000	$ra	00000016

Figur 3: Forventede branch resultater

2.3 Memory operationer

```
1 addiu $s0, $zero, 4
2 sw $s0, 0($0)
3 lw $s1, 0($0)
4 sw $s2, 4($0)
```

Som det forventes producerer ovenstående instruktioner nøjagtig de forventede data i registrerne. Der bliver også skrevet til hukommelsaddressen 000000 en værdi på 4. Den anden sw operation bliver dog ikke skrevet, så addresse 000004 forbliver 0. Dette skyldes at lw ikke når at loade før sw skal bruge værdien. Der er således tale om en uafklaret hazard, som vi desværre ikke har fået løst. (se Figur 4).

$zero	00000000	$s0	00000004
$at	00000000	$s1	00000004
$v0	00000000	$s2	00000000
$v1	00000000	$s3	00000000
$a0	00000000	$s4	00000000
$a1	00000000	$s5	00000000
$a2	00000000	$s6	00000000
$a3	00000000	$s7	00000000
$t0	00000000	$t8	00000000
$t1	00000000	$t9	00000000
$t2	00000000	$k0	00000000
$t3	00000000	$k1	00000000
$t4	00000000	$gp	00000000
$t5	00000000	$sp	00000000
$t6	00000000	$fp	00000000
$t7	00000000	$ra	00000000

Figur 4: Forventet resultat

2.3.1 Branch Hazards/Forwarding

...

Litteratur

[1] David A. Patterson, John L. Hennessy, *Computer Organization and Design*. Morgan Kaufmann, Revised 4th Edition, 2009.