LABORATION

DATORKONSTRUKTION D

Pipelining

Version: 1.7
2020 (OS,OVA,AN)

blank sida

Innehåll

1 Inledning **5**
1.1 Syfte . . . 5
1.2 Förberedelser . . . 5

2 Teori **6**
2.1 CPU . . . 6
2.2 Instruktionsformat . . . 6
2.3 Pipelinen . . . 6
2.4 Hopp . . . 8
2.5 Simuleringsprogrammet `lpia` . . . 8
2.6 Ett testprogram . . . 11
2.7 Uppgift . . . 11
2.8 Pipelinediagram . . . 11
2.9 Uppgift . . . 12

3 Naivt pipelinad dator, NPD **13**
3.1 Uppgift . . . 13
3.2 Uppgift . . . 13
3.3 Uppgift . . . 13
3.4 Uppgift . . . 13

4 Riktigt pipelinad dator, RPD **14**
4.1 Uppgift . . . 14
4.2 Uppgift . . . 14
4.3 Uppgift . . . 14
4.4 Uppgift . . . 14
4.5 Uppgift . . . 14

A Instruktionsrepertoar **15**

B Filen test.pia **17**

C Pipelinediagram **19**

blank sida

1 Inledning

1.1 Syfte

I denna laboration får du lära dig hur en pipelinad processor är uppbyggd. Mycket förenklat kan man likna databehandlingen i en sådan processor med tillverkningen av bilar på ett löpande band. Men det finns avgörande skillnader mellan biltillverkning och instruktionsexekvering. I bilfabriken är alla bilar lika (i stort sett), det finns inga beroenden mellan bilarna, vi behöver inte stoppa pipelinen ibland eller att vissa stationer ibland inte har något att göra.
Denna laboration koncentrerar sig på ovan nämnda besvärligheter och varför de uppstår och hur man med extra hårdvara får en effektivare pipeline. Resultatet blir en realistisk processor, som bra nära klarar en instruktion per klockcykel. Önskvärt är också att dölja pipelinen för assemblerprogrammeraren. Så när som på fördröjda hopp blir pipelinen osynlig.
Två varianter av pipelinen kommer att visas i laborationen:

- en förenklad, kallad naiv pipeline, se figur 1
- en realistisk, kallad riktig pipeline, se figur 2

1.2 Förberedelser

Innan du kommer till laborationen måste du vara väl förberedd. Alla uppgifter som ska redovisas är utmärkta med ett pekfinger. Många av dem kan med fördel utföras hemma som förberedelseuppgifter. Gå alltså igenom uppgifterna och lös så många av dem som möjligt hemma.

2 Teori

2.1 CPU

Laborationen är baserad på en delmängd av en riktig processor OpenRisc OR1200, [1, 2, 3]. Denna processor finns inte att köpa som ett chip utan existerar bara som Verilog-kod (en konkurrent till VHDL), som dessutom är gratis att ladda ner. Det finns också en gcc anpassad till denna arkitektur, men i den här laborationen är det dock assembler som gäller.
Några enkla data för vår dator:

- 5-stegs pipeline
- 15 instruktioner implementerade, se bilaga A. Alla instruktioner är 32 bitar breda.
- LOAD/STORE-arkitektur. Endast dessa instruktioner accessar minnet. Aritmetiska/Logiska instruktioner använder endast register.
- register: 32 à 32 bitar
- programminne: 64 instruktioner à 32 bitar
- dataminne: 64 data à 32 bitar. Detta är också den enda datatypen.

2.2 Instruktionsformat

Alla instruktioner är 32 bitar långa och är uppbyggda på i stort sett samma sätt, se bilaga A. Här visas formatet för `ADD RD,RA,RB` :
Operation: $rD = rA + rB, SR[CY], SR[OV]$
Opkod:

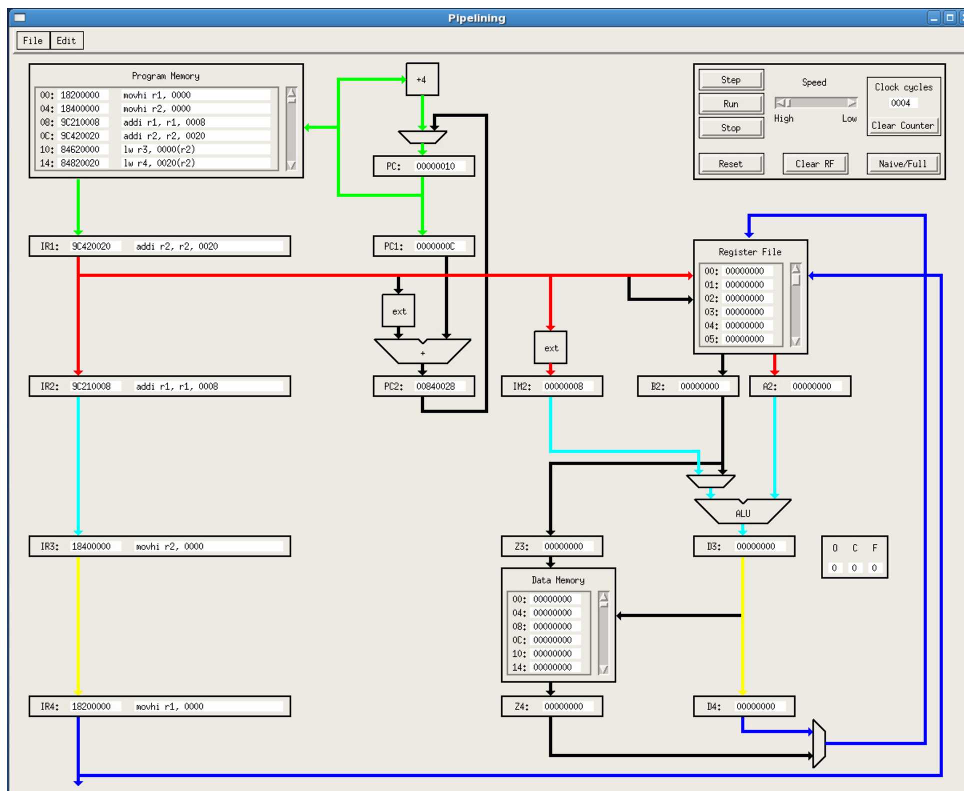
6	5	5	5	11
0x38	D	A	B	-

Alltså först 6 bitar för opkoden och därefter 3 fält om vardera 5 bitar. Dessa fält pekar ut de register, som ska skrivas respektive läsas. Det sista fältet innehåller visserligen några extra bitars information, men de spelar ingen roll i denna laboration. Instruktion påverkar flaggorna för carry och overflow i statusregistret (SR).

2.3 Pipelinen

I figuren 1 visas ett blockschema för vår dator. Den fungerar på följande sätt:

1. *IF = instruction fetch*. Pipelinen börjar med programräknaren PC. I detta steg sker 2 saker parallellt: den utpekade instruktionen laddas i instruktionsregistret IR1 och PC får ett nytt värde. Det nya värdet är antingen ett inkrement med 4 eller en hoppadress.



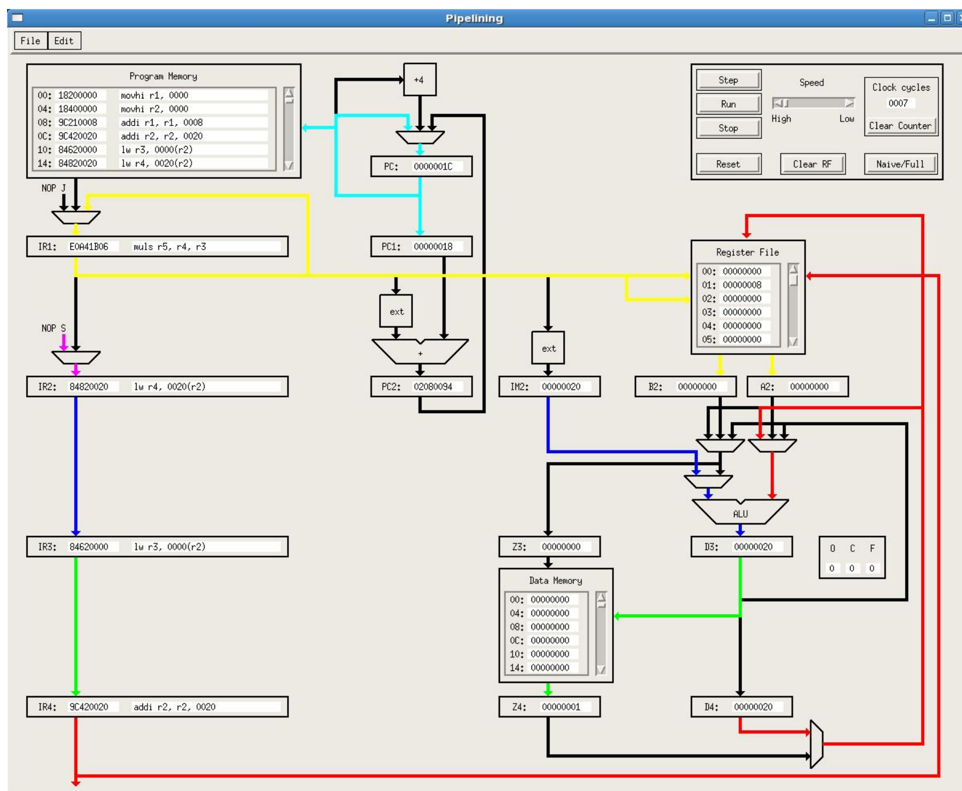
Figur 1: Naivt pipelinad dator.

2. *RR* = *register read*. Nu finns en instruktion i IR1. Med hjälp av informationen i IR1 sker nu 3 saker parallellt: beräkna en hoppadress till PC2, lägg en konstant (immediate) i IM2 och läs 2 register tll B2,A2. Om det, exempelvis, inte var en hoppinstruktion så blir det skräp i PC2. Men det gör väl inget, eller hur?

3. *EXE* = *execute*. Observera att instruktionen nu finns i IR2. Aritmetisk/Logisk operation mellan register och register/immediate.

4. *MEM* = *access data memory*. Instruktionen finns i IR3. En av 2 saker händer: läs i minnet eller skriv i minnet. D3 kopieras alltid till D4.

5. *WB* = *write back new register value*. Instruktionen finns i IR4. Om instruktionen i IR4 ska uppdatera ett register, så skrivs antingen Z4 eller D4 till registerfilen.



Figur 2: Riktigt pipelinad dator.

2.4 Hopp

Vår dator har bara två hoppinstruktioner: ett ovillkorlig hopp `J N` och ett villkorligt hopp `BF N`. Båda dessa hopp är fördröjda, dvs instruktionen efter hoppet exekveras alltid. Lägg också märke till att det villkorliga hoppet beror på flaggan F, som påverkas av instruktionerna `SFEQ`, `SFNE`, `SFEQI` och `SFNEI`. Lägg noga märke till var i pipelinen F finns.

2.5 Simuleringsprogrammet `lpia`

Starta simuleringsprogrammet med `lpia`. Ladda in en uppsättning med **Arkiv->Ladda nytt** och välj filen `test.pia`. Tryck också på knappen **Naive/Full**, så att datorn hamnar i sitt naiva läge. Skärmen bör nu se ut som figur 1.
Filen `test.pia` är en text-fil, titta gärna i den. Den innehåller datorns tillstånd, nämligen:

- innehåll i programminnet
- innehåll i dataminnet

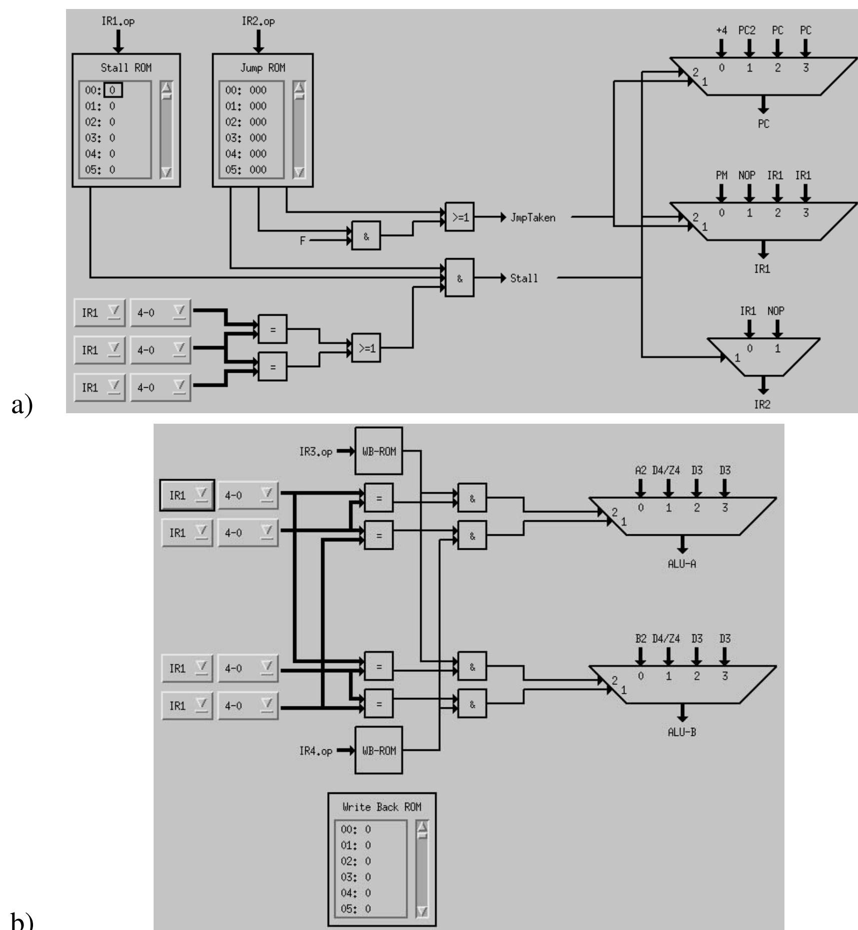
- innehåll i registerfilen
- innehåll i övriga pipelineregister

Du kan nu singlesteppa (**Step**) eller exekvera (**Run**) programmet i fullt tempo (hastighetsreglering (**Speed**) kan göras) och observera resultatet. Programkörning stoppas med **Stop**. Antal cykler för ett prgram kan mätas med räknaren **Clock cycles**, som kan nollställas med **Clear Counter**. För att cykelräkningen ska bli rätt måste programmet avslutas med instruktionen `TRAP 0`, som stoppar processorn. **Reset** nollställer PC och **Clear RF** nollställer R0 till R31. Du behöver alltså inte göra någon sorts mikroprogrammering för att datorn ska fungera i sitt naiva läge.
Tryck på knappen **Naiv/Full** så att datorn hamnar i sitt riktiga eller fulla läge. Skärmen bör nu se ut som figur 2.
Nu har fyra multiplexrar och diverse ledningar tillkommit. Muxarnas uppgifter är (vi börjar uppe till vänster):

- *hoppmux*. Vid ett taget hopp ska en NOP-instruktion laddas i instruktionsregistret IR1.
- *stallmux*. Vid vissa tillfällen måste den övre delen av pipelinen stå still. Detta sker genom att PC och IR1 behåller sina värden och IR2 laddas med en NOP. Resten av pipelinen fungerar som vanligt.
- *forwardingmuxar*. Rätt värden för ALU:n kan finnas i registerfile, register D3 eller register Z4/D4. Det är forwardingmuxarnas uppgift att se till att ALU:n får rätt värden.

Dessa muxar måste styras och det blir din uppgift. Gör **Redigera->Definiera jump and stall logic** och fönstret i figur 3a dyker upp. Fönstrets högra halva visar de muxar som ska styras. Programmeringen sker genom att fylla i de två ROM:en och de sex valknapparna till vänster. Med valknapparna väljer du register samt bitfält i registret.
Gör **Redigera->Definiera dataforwarding logic** och fönstret i figur 3b dyker upp. Fönstrets högra halva visar de muxar som ska styras. Programmeringen sker genom att fylla i Write Back ROM och de åtta valknapparna till vänster.

a)



b)

Figur 3: a) Logik för jump and stall b) Logik för data forwarding.

I figurerna ovan är `IR1.op`, `IR2.op`, `IR3.op` och `IR4.op` operationskoder för de instruktioner som ligger i `IR1`, `IR2`, `IR3` respektive `IR4`. Dessa operationskoder kommer att utgöra adresser till olika minnen.
`Stall ROM`, `Jump ROM` och `Write Back ROM` (`WB-ROM`) är minnen där varje adress kan ges ett innehåll. Varje adress i respektive minne motsvarar alltså en operationskod, så att man för en viss operationskod kan få ut ett visst minnesinnehåll, som utgör signaler till den övriga logiken. `Write Back ROM` (`WB-ROM`) används av både `IR3.op` och `IR4.op`.
`Stall ROM` påverkar alltså när det är dags att utföra stall i pipelinen.
`Jump ROM` påverkar också stall, men även villkorliga och ovillkorliga hopp.
`Write back ROM` styr logiken för data forwarding.

Utöver att definiera rätt innehåll i respektive minne så behöver rätt kombinationer av IR-register och grupper av bitar i instruktionsordet väljas i logiken, för både jump and stall samt data forwarding.

2.6 Ett testprogram

Nedanstående program beräknar skalärprodukten mellan två vektorer.

```
       MOVHI R1,0          ;
       MOVHI R2,0          ;
       ADDI  R1,R1,8       ; loopräknare = 8
       ADDI  R2,R2,20      ; pekare = 20
LOOP:  LW    R3,0(R2)      ; hämta det ena talet
       LW    R4,20(R2)     ; hämta det andra talet
       MUL   R5,R4,R3      ; multiplicera dem
       ADD   R6,R6,R5      ; och ackumulera till resultat
       ADDI  R2,R2,4       ; pekare++
       ADDI  R1,R1,-1      ; loopräknare--
       SFNE  R0,R1         ; sätt flagga=1 om loopräknare != 0
       BF    LOOP          ; fortsätta?
       SW    0(R0),R6      ; spara resultatet i minnet
       TRAP  0             ; stanna processorn
```

Vektorerna är 8 ord långa och befinner sig i dataminnet på adress 0x20 respektive 0x40. Resultatet skrivs till adress 0. Programmet och även vektorerna laddas på rätt plats med hjälp av filen `test.pia`.

2.7 Uppgift

Vad blir skalärprodukten om båda vektorerna är $< 1, 2, 3, 4, 5, 6, 7, 8 >$?

2.8 Pipelinediagram

Betrakta programmet (och antag `R0=0`)

```
       ADDI  R1,R0,1       ; R1=1
       ADD   R2,R1,R1      ; R2=2
```

Programmeraren har naturligtvis tänkt sig att `R2=2`. Detta är ett exempel på databeroende, i det här fallet kring registret `R1`.

cykel	PC	IR1	IR2	IR3	IR4
1	0				
2	4	ADDI			
3	8	**ADD**	ADDI		
4	C		ADD	ADDI	
5	10			ADD	**ADDI**
6	14				ADD
7	18				
8	1C				

cykel	PC	IR1	IR2	IR3	IR4
1	0				
2	4	ADDI			
3	8		ADDI		
4	C			ADDI	
5	10	**ADD**			**ADDI**
6	14		ADD		
7	18			ADD	
8	1C				ADD

Pipelinediagram är en enkel metod för att reda ut hur en pipeline fungerar. PC och pipelineregistren IR_1 - IR_4 fylls i för varje klockcykel. Klockningen sker på de horisontella strecken. I det första diagrammet hanteras databeroendet på ett felaktigt sätt. Instruktionen `ADD` läser registret `R1`, i klockcykel 3, innan `ADDI` har skrivit till `R1`, i klockcykel 5. I det andra diagrammet visas att avståndet mellan de två beroende instruktionerna måste vara minst 2. I klockcykel 5 skrivs det nya värdet till `R1`. Eftersom en läsning av samma register sker vid samma klockning skrivs det nya värdet också till `A2` och `B2`. Registerfilen är alltså implementerad som write-before-read.

2.9 Uppgift

Om registerfilen är synkron, dvs read-before-write, hur kan man då komplettera hårdvaran för att det ska fungera?
Allså, hur ska man göra för att få tag i det nya datat, samtidigt som det skrivs till registerfilen? (Dvs, så att registerfilen blir write-before-read).

3 Naivt pipelinad dator, NPD

Ställ datorn i det naiva läget. Tyvärr kommer inte exekvering av testprogrammet att ge rätt resultat.

3.1 Uppgift

Ändra testprogrammet, genom att peta in NOP-instruktioner, så att programmet ger rätt resultat. Gör också ett pipelinediagram, se avsnitt 2.8.

3.2 Uppgift

Optimera testprogrammet från föregående uppgift, genom att ändra ordningsföljden mellan instruktionerna. Programmet måste naturligtvis fortfarande ge rätt resultat.

3.3 Uppgift

Kontrollera att ditt program ger rätt resultat genom att köra det.

3.4 Uppgift

Hur många klockcykler tar nu skalärproduktsprogrammet att exekvera? Använd klockcykelräknaren.

---

4 Riktigt pipelinad dator, RPD

I detta avsnitt ska vi, så långt det är möjligt, göra pipelinen osynlig för assembler-programmeraren. Börja med att ställa datorn i det riktiga/fulla läget.

4.1 Uppgift

Gör ett pipelinediagram för programmet i avsnitt 2.6. Observera att NOP:ar från jump- och stallogiken ska synas i diagrammet.

4.2 Uppgift

Programmera jumpandstall- och dataforwardinglogiken så att testprogrammet i avsnitt 2.6 går att exekvera i sin ursprungliga form.

4.3 Uppgift

Hur många klockcykler tar det nu?

4.4 Uppgift

Går det att optimera bort stall, genom att ändra i programmet?

4.5 Uppgift

Går det att optimera bort jmp-NOP:en, genom att ändra i prgrammet?

Referenser

[1] *OpenRISC 1000 Architecture Manual*, www.opencores.org

[2] Lampret,D: *OpenRISC 1200 IP Core Specification*, www.opencores.org

[3] *OpenRISC 1200 RISC/DSP Core*, www.opencores.org

A Instruktionsrepertoar

- exts(X) betecknar sign extension, utvidgning till 32-bitars tal genom kopiering av teckenbiten.
- - betecknar fält som inte används i denna laboration
- SR[CY], SR[OV] och SR[F] betyder att instruktionen påverkar carry-, overflow- respektive F-flaggan i statusregistret.
- Vissa instruktioner, t ex ADD och MUL, har samma Opkod (0x38), men de kommer att åtskiljas genom att de 11 minst signifikanta bitarna i hela instruktionsordet innehåller olika bitmönster.

ADD rD, rA, rB

Operation: $rD = rA + rB, SR[CY], SR[OV]$
Opkod:

6	5	5	5	11
0x38	D	A	B	-

ADDI rD, rA, I

Operation: $rD = rA + exts(I), SR[CY], SR[OV]$
Opkod:

6	5	5	16
0x27	D	A	I

BF N

Operation: $PC = (SR[F])?PC + exts(N << 2) : PC = PC + 4$
Opkod:

6	26
0x4	N

J N

Operation: $PC = PC + exts(N << 2)$
Opkod:

6	26
0x0	N

LW rD, I(rA)

Operation: $rD = M(rA + exts(I))$
Opkod:

6	5	5	16
0x21	D	A	I

MOVHI rD, K

Operation: $rD[31:16] = K$, $rD[15:0] = 0$
Opkod:

6	5	5	16
0x6	D	-	I

MUL rD, rA, rB

Operation: $rD = rA * rB, SR[CY], SR[OV]$
Opkod:

6	5	5	5	11
0x38	D	A	B	-

NOP K

Operation:
Opkod:

6	10	16
0x15	-	K

SFEQ rA, rB

Operation: $SR[F] = (rA == rB)?1 : 0$
Opkod:

6	5	5	5	11
0x39	-	A	B	-

SFNE rA, rB

Operation: $SR[F] = (rA == rB)?0 : 1$
Opkod:

6	5	5	5	11
0x39	-	A	B	-

SFEQI rA, I

Operation: $SR[F] = (rA == exts(I))?1 : 0$
Opkod:

6	5	5	16
0x2f	-	A	I

SFNEI rA, I

Operation: $SR[F] = (rA == exts(I))?0 : 1$
Opkod:

6	5	5	16
0x2f	-	A	I

SW I(rA), rB

Operation: $M(rA + exts(I)) = rB$
Opkod:

6	5	5	5	11
0x35	I	A	B	I

TRAP K

Operation: Processorn stannar
Opkod:

16	16
0x2100	K

B Filen test.pia

```
PM:
00: 18200000   ; movhi r1,0000
04: 18400000   ; movhi r2,0000
...

DM:
00: 00000000
04: 00000000
...
fc: 00000000

RF:
00: 00000000
01: 00000000
...
```

```
1f: 00000000

WB:
00: 0
01: 0
...
3f: 0

JP:
00: 0
01: 0
...
3f: 0

ST:
00: 0
01: 0
...
3f: 0

IR1:
54000000

IR2:
54000000

...

D4:
00000000

O_flag:

C_flag:

F_flag:
Reg_field:
00
...
00
```

C Pipelinediagram

PC	Register IR1	Register IR2	Register IR3	Register IR4