TDT4160

Øving 3

September 28, 2023

Frist: Kl 23:59, Oktober 12, 2023

Contents

1 Introduksjon

1.1 Om Øvingen

Hvis du ønsker å gå rett til oppgaven i øving 3, gå til subsection 2.1.

I denne øvingen skal vi fortsette med koding i assembly, men med litt mer avansert kontrollflyt i programmet ditt. Som i øving 2 inneholder dette dokumentet en introduksjonen i section 1 (det er denne seksjonen), selve oppgaven i section 2, noen hint i section 3, og en liten hjelpeseksjon i section 4. Merk at denne øvingen kommer til å bygge på følgende antakelser:

- Du vet hva assembly er.
- Du vet hva RISC-V er.
- Du har gjort Øving 2.
- Du har fått Ripes til å kjøre.

Siden vi tar disse antakelsene kommer det til å være mye mindre informasjon i denne øvingen, og den kommer som konsekvens til å være mye kortere. Hvis du trenger en oppsummering av assembly, RISC-V, eller Ripes, se Øving 2.

2 Oppgave

2.1 Oppgavebeskrivelse

- **Input-registre:** `a0`
- **Output-registre:** `a0`, `a1`
- **Oppgave**: Registeret `a0` inneholder et positivt heltall – heretter kalt A – som er større enn 1. Programmet ditt skal gjøre to ting:
  1. Finn den største *divisoren/faktoren* som er *mindre enn* A, og putt denne i `a0`.
  2. Sjekk om tallet A er et *kvadrattall*. Putt 1 i `a1` hvis A er et kvadrattall, eller 0 i `a1` hvis det ikke er det.

2.2 Levering

Frist: 12. oktober kl 23:59

Når du er ferdig:

1. Naviger til **"File"** → **"Save File As..."**. Du skal nå få opp et nytt vindu.
2. Kall filen din for: **3**.
3. Huk av for **source code**.
4. "Output" skal nå være satt til: **3.s**.
5. Klikk OK, og lagre filen et lurt sted.
6. Last opp filen **3.s** til innleveringsmappen for øving 3 på **Blackboard**.

2.3 Eksempel

2.3.1 Eksempel 1

Register	Verdi
a0	9

Table 1: Eksempel 1 - Start.

- Gjør klar register `a0` (Table 1).
- Den største divisoren av 9 som ikke er 9 er 3 fordi $9 = 3 * 3$.
- Putter 3 i `a0`.
- Tallet 9 er et kvadrattall fordi $9 = 3 * 3$.
- Putter 1 i `a1`.

Register	Verdi
a0	3
a1	1

Table 2: Eksempel 1 - Slutt.

2.3.2 Eksempel 2

Register	Verdi
a0	39

Table 3: Eksempel 2 - Start.

- Gjør klar register `a0` (Table 3).
- Den største divisoren av 39 som ikke er 39 er 13 fordi $39 = 13 * 3$.
- Putter 13 i `a0`.
- Tallet 39 er ikke et kvadrattall.
- Putter 0 i `a1`.

Register	Verdi
a0	13
a1	0

Table 4: Eksempel 2 - Slutt.

2.3.3 Eksempel 3

Register	Verdi
a0	2401

Table 5: Eksempel 3 - Start.

- Gjør klar register `a0` (Table 5).
- Den største divisoren av 2401 som ikke er 2401 er 343 fordi $2401 = 343 * 7$.
- Putter 343 i `a0`.
- Tallet 2401 er et kvadrattall fordi $2401 = 49 * 49$.
- Putter 1 i `a1`.

Register	Verdi
a0	343
a1	1

Table 6: Eksempel 3 - Slutt.

2.3.4 Eksempel 4

Register	Verdi
a0	2

Table 7: Eksempel 4 - Start.

- Gjør klar register `a0` (Table 7).
- Den største divisoren av 2 som ikke er 2 er 1 fordi $2 = 1 * 2$.
- Putter 1 i `a0`.
- Tallet 2 er ikke et kvadrattall.
- Putter 0 i `a1`.

Register	Verdi
a0	1
a1	0

Table 8: Eksempel 4 - Slutt.

2.4 Testing

Viktig: Lagre koden din jevnlig før du tester koden din. Noen ganger – særlig hvis du "fast-forwarder" simuleringen mens den allerede kjører – kan Ripes kræsje, og du mister da det du gjort i editoren. Hvis dette skjer, last en lagret kopi av koden din gjennom "Load Program" og prøv igjen. Se section 4 hvis du trenger hjelp til dette.

Det er to måter du kan teste koden din på: ved å manuelt sette registre til visse verdier i koden din i Ripes sin editor, eller gjennom Ripes sitt terminal-grensesnitt ("Command Line Interface" = CLI) **på Linux**.

I denne øvingen trenger du bare å sette registeret `a0` til et heltall (større enn 1), men du må i stedet passe på at *både* `a0` og `a1` inneholder riktig data til slutt. For detaljer om hvordan du gjør dette, se øving 2.

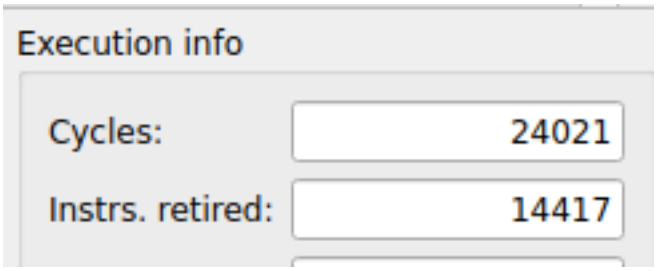


Figure 1: En løsning for A=2401 bruker 24021 sykler før den er ferdig.

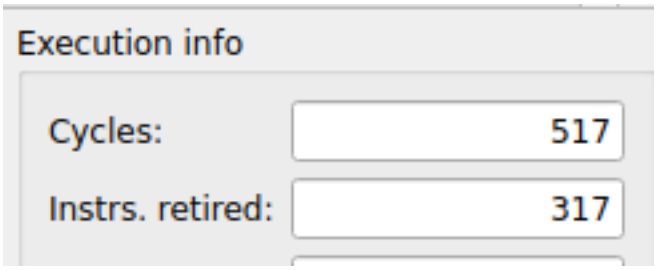


Figure 2: En raskere løsning for A=2401 bruker bare 517 sykler før den er ferdig.

2.5 Bonus: Optimalisering

Det er forskjellige måter du kan komme frem til riktig svar på. For å få godkjent denne øvingen trenger du bare lage et program som gjør det oppgaven sier: finne det største tallet som deler A, og sjekke om A er et kvadrattall. Det finnes likevel både raske og trege måter du kan gjøre dette på. For eksempel kan ytelsen for to forskjellige varianter av programmet sees i Figure 1 og Figure 1. Her prøver begge variantene å løse eksempel 3 med $A = 2401$. Begge variantene gjør nøyaktig det samme og har samme output, men den ene er mer optimalisert. Hvis du har tid til overs og synes assemblyprogrammering virker gøy kan du se om du kan optimalisere programmet ditt for store verdier av A. Sjekk for eksempel hvor raskt programmet ditt greier eksempelet med $A = 2401$.

- **Spørsmål:** Får jeg bonuspoeng for å levere et raskt program på Blackboard?
  - **Svar:** Nei :) Det viktigste er at programmet fungerer for alle verdier av A større enn 1.

3 Hint & Tips

3.1 Relevante Instruksjoner for Assembly

I denne øvingen er det særlig to konsepter som er relevante: **løkker** (eng. "loops"), og **divisjon**[1]:

- **Løkker:** For løkker trenger vi å bruke "labels", "branches", og eller "jumps". Disse kan du lese mer om i øving 2.
- **Divisjon:** Det er *to* instruksjoner som er relevante her:
  - **div**: Den ene er naturligvis **div rd, rs1, rs2**. Denne deler `rs1` på `rs2`, og putter resultatet i `rd`. Merk at divisjon på assembly-nivå alltid er *heltalldivisjon* som betyr at vi alltid får *kvotienten* som svar (og altså aldri et desimaltall). Eksempel: $10/3 = 3$ og $999/1000 = 0$.
  - **???**: Det er en annen instruksjon som også er relevant her, men denne må du finne selv. Denne likner veldig på `div`, men i stedet for å få *kvotienten* gir den deg *resten* etter en divisjon. Sjekk om du finner et instruksjonsnavn i Appendix A som du mistenker kan gjøre noe slik.

3.2 Løkker

```
a7 = 0
for i in range(10):
  a7 = a7 + 1
```

Listing 1: En "for"-loop i Python.

```
t0 = 0
a7 = 0
while t0 < 10:
  a7 = a7 + 1
  t0 = t0 + 1
```

Listing 2: En "while"-loop i Python.

Løkker brukes i programmering når vi vil gjenta noe logikk flere ganger. De fleste programmeringsspråk har to "forskjellige typer" løkker: "for"-løkker, og "while"-løkker. "For"-løkker repeterer typisk noe logikk N ganger, mens "while"-løkker repeterer noe logikk så lenge et utrykk er "sant". Merk likevel at alle "for"-løkker likevel egentlig bare er en versjon av en "while"-løkke[2]. For eksempel kan "for"-løkken i Listing 1 konverteres til "while"-løkken i Listing 2.

[1] Vi kan benyttes oss av divisjon siden vi bruker **M**-utvidelsen med støtte for multiplisering og deling.

[2] Alle "for"-løkker kan konverteres til en "while"-løkke, men ikke motsatt.

Vi har ikke noen "for"- eller "while"-nøkkelord i assembly, men vi kan lage dem fra bunnen av. Her kommer en oppskrift på hvordan vi kan lage en "for"-løkke:

1. Aller først må vi ha en *teller* som vi øker eller reduserer hver gang vi utfører logikken i en løkke. Vi velger selv hva denne starter på, og senere bestemmer vi om den bør økes eller minkes og med hvor mye.

2. Vi må så ha selve starten på løkken. Dette bør være et sted i programminnet hvor programmet kan gå til for å repetere logikken under.

3. Deretter bør vi ha en sjekk som forteller oss om vi er "ferdige" med løkken eller ikke. For eksempel vil vi bare utføre løkken i Listing 1/Listing 2 10 ganger. Når `t0` er 10 (eller større) vet vi at vi er ferdig, og vi må da komme oss ut av løkken.

4. Etter at vi har sjekket at vi *skal* utføre løkken kan vi gå til den faktiske logikken som skal kjøres.

5. Når vi er ferdig med å utføre logikken må vi oppdatere telleren. Avhengig av hva slags løkke vi har vil vi enten øke eller redusere den. Ofte økes/reduseres denne med 1, men det er også mulig å øke/redusere denne med mer.

6. Vi vil deretter starte en ny iterasjon av løkken. Det første vi bør gjøre i neste iterasjon er å sjekke om vi er "ferdig" med løkken eller om vi skal fortsette.

3.3 Kvotient, rest, og delelighet

Som nevnt utfører bare operasjonen `div rd, rs1, rs2` bare heltallsdivisjon. Det betyr at vi får følgende resultat:

- Dele 12 på 3: $12/3 = 4$. Putter 4 i `rd`.
- Dele 13 på 3: $13/3 = 4$. Putter 4 i `rd`.

I eksempelet over får vi samme resultat om vi deler 12 på 3 og 13 på 3. Med andre ord sier `div` oss ingenting om `rs1` *kan* deles på `rs2`, men i stedet om hva resultatet av en heltallsdivisjon mellom dem er. Dette er også grunnen til at det er *to* relevante instruksjoner her og ikke bare en.

Hvordan kan du bruke resultatet fra den andre instruksjonen til å se om `rs2` deler `rs1`?

3.4 Foreslått fremgangsmåte

```
# Test Start
# Endre denne for å teste andre verdier for A
li a0, 9
# Test End

# Program Start
# GJØR KLAR VERDIER HER
# ...

# a0 = Largest Divisor
# a1 = IsSquare
# a2 = Input Number
# ...

# SKRIV ALGORITMEN DIN HER
# ...

# Program End
```

Listing 3: Du kan dele opp programmet ditt slik hvis du ønsker det.

1. Skriv ned kommentarer hvor du holder rede på hvilke "variabler" du kommer til å bruke, og hvilke registre disse variablene "representer". Du kan så klart bruke registre om igjen men pass på å ikke overskriv et register som inneholder noe data du vil bruke senere. For eksempel kan du la `a0` inneholde det største tallet som deler A til enhver tid. Det være lurt å la A få et eget register for seg selv. Du kan følge malen i Listing 3 hvis du ønsker det.
2. Lag algoritmen hvor du finner det største tallet som deler A. Vent med kvadrattall-sjekkingen til senere for enkelhetens skyld.
   - Hvordan finner du ut om et tall deler A?
   - Hvordan finner du *det største* tallet som deler A?
3. Når du har en algoritme du tror fungerer, test med å putte forskjellige tall i `a0` og se at du får riktig resultat.
4. Endre deretter på algoritmen sånn at den også sjekker om tallet er et kvadrattall.
   - Når er et tall et kvadrattall?

- Merk: Kvadratroten av et kvadrattall vil alltid dele tallet (hvis det har en heltall-rot), men det er ikke nødvendigvis det *største* tallet som deler det[3].
- Tips: Det er ofte enklere å sjekke om noe *er* et kvadrattall enn om det *ikke* er det.

5. Test programmet ditt med forskjellige tall og se at du får riktig svar.

3.5 Fast-forwarding

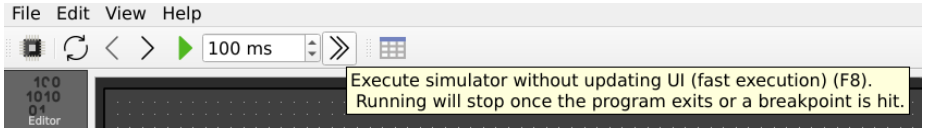


Figure 3: Trykk på de to pilene for å skippe til slutten av programmet.

Viktig: Lagre filen din før du fast-forwarder gjennom simuleringen. Ripes kan kræsje når du gjør dette, og da mister du alt du har skrevet i editoren

Du kan velge at Ripes skal "fast-forwarde" gjennom simuleringen uten å oppdatere brukergrensesnittet hvis du vil se resultatene fra simuleringen med en gang. For å gjøre dette, trykk på de to pilene som vises i Figure 3. Dette er spesielt nyttig hvis du har lange simuleringer – for eksempel av at du teller opp til et stort tall. Merk at dette kan veldig enkelt kræsje eller fryse - særlig ved lange simuleringer. Hvis det ikke skjer noe i grensesnittet etter du har trykket på pilene er det en stor sannsynlighet for at programmet ditt er "stuck" i en evig løkke som den aldri avbryter.

[3]Det er det største hvis tallet et produkt av to primtall, men ikke nødvendigvis ellers.

4 Hjelp

4.1 Hvis du trenger hjelp

Hvis du sitter fast eller trenger hjelp med koding eller simulatoren anbefales det å møte opp i en av øvingstimene og få hjelp av en studass. Timeplanen for øvingstimene kan finnes her. Du kan naturligvis også diskutere med andre medstudenter, men plagiat av andre kommer til å behandles som juks.

4.2 Spørsmål og Svar

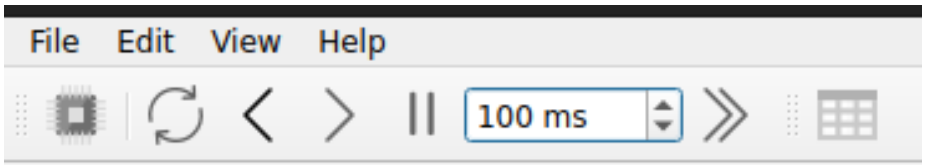


Figure 4: En bug(?) i Ripes hvor simuleringsbaren ikke oppdateres etter at en simulering er ferdig. Merk at flere av knappene er "grået" ut.

- **Spørsmål:** Ripes "henger seg opp" etter en simulering og jeg får ikke trykket på "pause" eller "resett" – tilsvarende Figure 4.
  - **Svar:** Ja, dette er sannsynligvis en bug. Hvis du trykker på "tilbake" en gang bør du få muligheten til å resette simuleringen. Da vil simuleringen skje på nytt, men kjøre automatisk. Mens simuleringen kjører kan du prøve å trykke på "pause". Hvis simuleringen ikke stopper nå, prøv å trykk en gang til. Etter dette *bør* simuleringen pauses, og du kan resette den igjen uten at den kjøres automatisk.
- **Spørsmål:** Editoren klager på "unknown opcode mul" eller "unknown opcode div".
  - **Svar:** Du prøver sannsynligvis å kjøre en multiplikasjon eller divisjon på "RV32I" uten utvidelsen "M". Trykk på prosessorvelgeren øverst i venstre hjørne, og huk av for "M" i "ISA Exts."
- **Spørsmål:** Editoren klager på at jeg har et "unknown symbol X" når jeg prøver å hoppe til labelen X i en branch, selv om labelen eksisterer.
  - **Svar:** Dette skjer sannsynligvis fordi du ikke har noen instruksjoner etter label X. Derfor går det heller ikke an å "hoppe" til den adressen i programminnet siden programmet "stopper" etter den siste instruksjonen. Prøv å legge til en instruksjon etter label X.
- **Spørsmål:** Hvorfor oppdateres ikke registrene mine når koden min kjører?

- **Svar:** De gjør sannsynligvis det, men ikke før i "Write-Back"-steget (WB). Husk at Write-Back er det siste steget i en "5 stage RISC-V"-prosessor, og ligger to sykler back Execute (EX). Prosessoren er likevel smart og greier å bruke verdien av et register før det har blitt skrevet til WB hvis den trenger det[4]. Hvis du synes dette gjør det vanskelig å debugge eller følge med på hva som skjer kan du putte fem `nop`-instruksjoner ("No operation") mellom hver av de andre instruksjonene.

- **Spørsmål:** Jeg får ikke lov til å lagre filen min.
  - **Svar:** Ripes prøver sannsynligvis å lagre filen til et sted på disken som er "read-only". Trykk på "Save File As..." og deretter de tre dottene for å få opp en oversikt over filsystemet ditt. Velg deretter for eksempel "Downloads"/"Nedlastinger", "Desktop"/"Skrivebord", eller "M:/" hvis du er på en NTNU-datamaskin. Deretter bør du kunne lagre filen som vanlig.

- **Spørsmål:** Jeg vil laste enn assemblyfil.
  - **Svar:** Dette kan du gjøre ved "File → Load program", velge "Source file", og deretter finne assemblyfilen du vil laste inn.

- **Spørsmål:** Jeg har et annet spørsmål eller trenger hjelp med assembly.
  - **Svar:** Møt opp i en øvingstime og få hjelp av en studass. Øvingstimene i TDT4160 kan finnes her.

4.3 Problemer med å kjøre Ripes

Hvis du fortsatt har problemer med Ripes er det enkleste å bruke Web-versjonen på enten Ripes sin egne Web-versjon eller NTNU sin eksterne desktop. Ripes sin versjon er veldig rask og enkel å bruke, men fortsatt i det eksperimentelle stadiet. NTNU sin versjon skal være stabil og gjøre at du får tilgang på hjemmeområdet og liknende, men her må du først logge deg inn med Feide.

[4]For spesielt interesserte er dette fordi denne prosessoren har noe som kalles "Hazard Detection og en "Forwarding Unit". Hvis du vil kan du teste ut en "5-stage processor w/o forwarding or hazard detection" ved å bytte prosessortype. Se hva som skjer med resultatet av koden din når du bruker denne. Husk i så fall å bytte tilbake!

5 Versjoner

- Høst 2023 – Anders Gaustad

A Vedlegg

I denne seksjonen kan du finne følgende vedlegg:

1. Fullt referansekort med instruksjoner for RISC-V assembly - inkludert utvidelser.

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	`LB rd,rs1,imm`	
	Load Halfword	I	`LH rd,rs1,imm`	
	Load Word	I	`LW rd,rs1,imm`	`L{D|Q} rd,rs1,imm`
	Load Byte Unsigned	I	`LBU rd,rs1,imm`	
	Load Half Unsigned	I	`LHU rd,rs1,imm`	`L{W|D}U rd,rs1,imm`
Stores	Store Byte	S	`SB rs1,rs2,imm`	
	Store Halfword	S	`SH rs1,rs2,imm`	
	Store Word	S	`SW rs1,rs2,imm`	`S{D|Q} rs1,rs2,imm`
Shifts	Shift Left	R	`SLL rd,rs1,rs2`	`SLL{W|D} rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI rd,rs1,shamt`	`SLLI{W|D} rd,rs1,shamt`
	Shift Right	R	`SRL rd,rs1,rs2`	`SRL{W|D} rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI rd,rs1,shamt`	`SRLI{W|D} rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA rd,rs1,rs2`	`SRA{W|D} rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI rd,rs1,shamt`	`SRAI{W|D} rd,rs1,shamt`
Arithmetic	ADD	R	`ADD rd,rs1,rs2`	`ADD{W|D} rd,rs1,rs2`
	ADD Immediate	I	`ADDI rd,rs1,imm`	`ADDI{W|D} rd,rs1,imm`
	SUBtract	R	`SUB rd,rs1,rs2`	`SUB{W|D} rd,rs1,rs2`
	Load Upper Imm	U	`LUI rd,imm`	
	Add Upper Imm to PC	U	`AUIPC rd,imm`	
Logical	XOR	R	`XOR rd,rs1,rs2`	
	XOR Immediate	I	`XORI rd,rs1,imm`	
	OR	R	`OR rd,rs1,rs2`	
	OR Immediate	I	`ORI rd,rs1,imm`	
	AND	R	`AND rd,rs1,rs2`	
	AND Immediate	I	`ANDI rd,rs1,imm`	
Compare	Set <	R	`SLT rd,rs1,rs2`	
	Set < Immediate	I	`SLTI rd,rs1,imm`	
	Set < Unsigned	R	`SLTU rd,rs1,rs2`	
	Set < Imm Unsigned	I	`SLTIU rd,rs1,imm`	
Branches	Branch =	SB	`BEQ rs1,rs2,imm`	
	Branch ≠	SB	`BNE rs1,rs2,imm`	
	Branch <	SB	`BLT rs1,rs2,imm`	
	Branch ≥	SB	`BGE rs1,rs2,imm`	
	Branch < Unsigned	SB	`BLTU rs1,rs2,imm`	
	Branch ≥ Unsigned	SB	`BGEU rs1,rs2,imm`	
Jump & Link	J&L	UJ	`JAL rd,imm`	
	Jump & Link Register	UJ	`JALR rd,rs1,imm`	
Synch	Synch thread	I	`FENCE`	
	Synch Instr & Data	I	`FENCE.I`	
System	System CALL	I	`SCALL`	
	System BREAK	I	`SBREAK`	
Counters	ReaD CYCLE	I	`RDCYCLE rd`	
	ReaD CYCLE upper Half	I	`RDCYCLEH rd`	
	ReaD TIME	I	`RDTIME rd`	
	ReaD TIME upper Half	I	`RDTIMEH rd`	
	ReaD INSTR RETired	I	`RDINSTRET rd`	
	ReaD INSTR upper Half	I	`RDINSTRETH rd`	

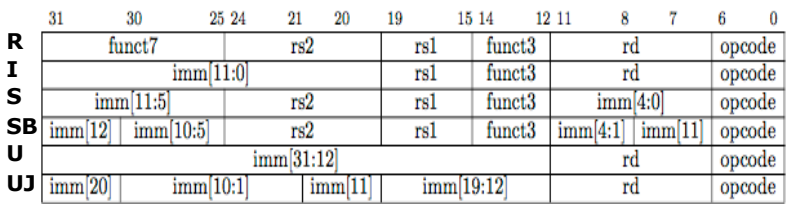
RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	`CSRRW rd,csr,rs1`
	Atomic Read & Set Bit	`CSRRS rd,csr,rs1`
	Atomic Read & Clear Bit	`CSRRC rd,csr,rs1`
	Atomic R/W Imm	`CSRRWI rd,csr,imm`
	Atomic Read & Set Bit Imm	`CSRRSI rd,csr,imm`
	Atomic Read & Clear Bit Imm	`CSRRCI rd,csr,imm`
Change Level	Env. Call	`ECALL`
	Environment Breakpoint	`EBREAK`
	Environment Return	`ERET`
Trap Redirect	to Supervisor	`MRTS`
	Redirect Trap to Hypervisor	`MRTH`
	Hypervisor Trap to Supervisor	`HRTS`
Interrupt	Wait for Interrupt	`WFI`
MMU	Supervisor FENCE	`SFENCE.VM rs1`

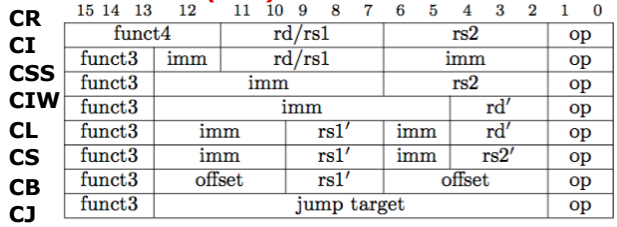
Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	`C.LW rd',rs1',imm`	`LW rd',rs1',imm*4`
	Load Word SP	CI	`C.LWSP rd,imm`	`LW rd,sp,imm*4`
	Load Double	CL	`C.LD rd',rs1',imm`	`LD rd',rs1',imm*8`
	Load Double SP	CI	`C.LDSP rd,imm`	`LD rd,sp,imm*8`
	Load Quad	CL	`C.LQ rd',rs1',imm`	`LQ rd',rs1',imm*16`
	Load Quad SP	CI	`C.LQSP rd,imm`	`LQ rd,sp,imm*16`
Stores	Store Word	CS	`C.SW rs1',rs2',imm`	`SW rs1',rs2',imm*4`
	Store Word SP	CSS	`C.SWSP rs2,imm`	`SW rs2,sp,imm*4`
	Store Double	CS	`C.SD rs1',rs2',imm`	`SD rs1',rs2',imm*8`
	Store Double SP	CSS	`C.SDSP rs2,imm`	`SD rs2,sp,imm*8`
	Store Quad	CS	`C.SQ rs1',rs2',imm`	`SQ rs1',rs2',imm*16`
	Store Quad SP	CSS	`C.SQSP rs2,imm`	`SQ rs2,sp,imm*16`
Arithmetic	ADD	CR	`C.ADD rd,rs1`	`ADD rd,rd,rs1`
	ADD Word	CR	`C.ADDW rd,rs1`	`ADDW rd,rd,imm`
	ADD Immediate	CI	`C.ADDI rd,imm`	`ADDI rd,rd,imm`
	ADD Word Imm	CI	`C.ADDIW rd,imm`	`ADDIW rd,rd,imm`
	ADD SP Imm * 16	CI	`C.ADDI16SP x0,imm`	`ADDI sp,sp,imm*16`
	ADD SP Imm * 4	CIW	`C.ADDI4SPN rd',imm`	`ADDI rd',sp,imm*4`
	Load Immediate	CI	`C.LI rd,imm`	`ADDI rd,x0,imm`
	Load Upper Imm	CI	`C.LUI rd,imm`	`LUI rd,imm`
	MoVe	CR	`C.MV rd,rs1`	`ADD rd,rs1,x0`
	SUB	CR	`C.SUB rd,rs1`	`SUB rd,rd,rs1`
Shifts	Shift Left Imm	CI	`C.SLLI rd,imm`	`SLLI rd,rd,imm`
Branches	Branch=0	CB	`C.BEQZ rs1',imm`	`BEQ rs1',x0,imm`
	Branch≠0	CB	`C.BNEZ rs1',imm`	`BNE rs1',x0,imm`
Jump	Jump	CJ	`C.J imm`	`JAL x0,imm`
	Jump Register	CR	`C.JR rd,rs1`	`JALR x0,rs1,0`
Jump & Link	J&L	CJ	`C.JAL imm`	`JAL ra,imm`
	Jump & Link Register	CR	`C.JALR rs1`	`JALR ra,rs1,0`
System	Env. BREAK	CI	`C.EBREAK`	`EBREAK`

32-bit Instruction Formats



16-bit (RVC) Instruction Formats



RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Free & Open RISC-V Reference Card (riscv.org) ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV{64,128}	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MUL{W|D}	rd,rs1,rs2
	MULtiply upper Half	R	MULH	rd,rs1,rs2		
	MULtiply Half Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply upper Half Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIV{W|D}	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REM{W|D}	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMU{W|D}	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV{64,128}	
Load	Load Reserved	R	LR.W	rd,rs1	LR.{D|Q}	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.{D|Q}	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.{D|Q}	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.{D|Q}	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.{D|Q}	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.{D|Q}	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.{D|Q}	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.{D|Q}	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.{D|Q}	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.{D|Q}	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.{D|Q}	rd,rs1,rs2

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ

Category	Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)		+RV{64,128}	
Move	Move from Integer	R	FMV.{H|S}.X	rd,rs1	FMV.{D|Q}.X	rd,rs1
	Move to Integer	R	FMV.X.{H|S}	rd,rs1	FMV.X.{D|Q}	rd,rs1
Convert	Convert from Int	R	FCVT.{H|S|D|Q}.W	rd,rs1	FCVT.{H|S|D|Q}.{L|T}	rd,rs1
	Convert from Int Unsigned	R	FCVT.{H|S|D|Q}.WU	rd,rs1	FCVT.{H|S|D|Q}.{L|T}U	rd,rs1
	Convert to Int	R	FCVT.W.{H|S|D|Q}	rd,rs1	FCVT.{L|T}.{H|S|D|Q}	rd,rs1
	Convert to Int Unsigned	R	FCVT.WU.{H|S|D|Q}	rd,rs1	FCVT.{L|T}U.{H|S|D|Q}	rd,rs1
Load	Load	I	FL{W,D,Q}	rd,rs1,imm		
Store	Store	S	FS{W,D,Q}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{S|D|Q}	rd,rs1,rs2		
	SUBtract	R	FSUB.{S|D|Q}	rd,rs1,rs2		
	MULtiply	R	FMUL.{S|D|Q}	rd,rs1,rs2		
	DIVide	R	FDIV.{S|D|Q}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{S|D|Q}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D|Q}	rd,rs1,rs2,rs3		
	Multiply-SUBtract	R	FMSUB.{S|D|Q}	rd,rs1,rs2,rs3		
	Negative Multiply-SUBtract	R	FNMSUB.{S|D|Q}	rd,rs1,rs2,rs3		
	Negative Multiply-ADD	R	FNMADD.{S|D|Q}	rd,rs1,rs2,rs3		
Sign Inject	SiGN source	R	FSGNJ.{S|D|Q}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D|Q}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D|Q}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{S|D|Q}	rd,rs1,rs2		
	MAXimum	R	FMAX.{S|D|Q}	rd,rs1,rs2		
Compare	Compare Float =	R	FEQ.{S|D|Q}	rd,rs1,rs2		
	Compare Float <	R	FLT.{S|D|Q}	rd,rs1,rs2		
	Compare Float ≤	R	FLE.{S|D|Q}	rd,rs1,rs2		
Categorization	Classify Type	R	FCLASS.{S|D|Q}	rd,rs1		
Configuration	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

RISC-V Calling Convention

Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries
f0-7	ft0-7	Caller	FP temporaries
f8-9	fs0-1	Callee	FP saved registers
f10-11	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, { } means set, so `L{D|Q}` *is both* `LD` *and* `LQ`*. See risc.org. (8/21/15 revision)*