Computerarchitectuur 2023
Opdracht 1: Introductie to Assembly Languages

In de eerste opdracht zullen we kennismaken met verschillende assembly languages, ofwel verschillende Instruction Set Architectures (ISAs). Iedere processor definieert een Instruction Set Architecture: een verzameling van instructies die deze processor begrijpt en in feite de interface tot de processor vormen. Software die op deze processor behoort te draaien, moet zijn uitgedrukt in de instructies van de bijbehorende Instruction Set Architecture.

We zullen twee verschillende architecturen gaan bekijken: 64-bit RISC-V en x86_64. We doen dit door assembly codes voor deze verschillende architecturen te bestuderen. Tevens leren we werken met tools om binaries voor deze architecturen te compileren.

Met behulp van deze practicumhandleiding en de gegeven voorbeeldcodes dien je deze stof te bestuderen. Het doel is dat je jezelf het lezen en begrijpen van assembly code eigen maakt. Je zult waarschijnlijk ook buiten de werkcolleges hieraan moeten werken. Er hoeft niets te worden ingeleverd, de toetsing voor deze eerste opdracht zal worden gedaan aan de hand van een *individuele*, online toets welke in de eerste helft van midden oktober zal plaatsvinden (houd BrightSpace in de gaten). In deze toets zul je onder andere worden gevraagd stukjes assembly codes te annoteren, patronen te herkennen, instructietellingen uit te voeren en compileroptimalisaties te identificeren. Hoewel de toets individueel is, mag je tijdens de werkcolleges natuurlijk in koppels werken, dat maakt het bestuderen van de stof makkelijker!

Voorbereiding

Download voordat je verder gaat de materialen van de BrightSpace-pagina van het vak (zie onderdeel Practicumopdrachten). Alle benodigde software is al voor je geïnstalleerd op de NUWD computers van de universiteit. Zorg ervoor dat je met ssh op de universitaire computers kunt inloggen, zodat je vanaf je eigen laptop op een universitaire computer met deze tools kunt werken. Indien je niet bekend bent met ssh en sftp, zie BrightSpace voor een beknopte omschrijving.

Nadat je bent ingelogd op een werkstation moet het volgende commando worden uitgevoerd om de tools te activeren:

```
source /vol/share/groups/liacs/scratch/ca2023/ca2023.bashrc
```

Merk op dat het prompt verandert om aan te duiden dat het environment correct is ingesteld. Let op dat je dit commando opnieuw moet uitvoeren als je opnieuw inlogt of een nieuw terminalvenster opent.

Introductie van de tools

Het compileren van een C-programma bestaat uit een aantal stappen. We zagen dit al tijdens het college Programmeertechnieken. Eerst wordt de preprocessor losgelaten op de C source code. Vervolgens wordt alles geparsed, waarna een assembly code wordt gegenereerd. Deze assembly code is nog in een door de mens leesbare ASCII-vorm. Met behulp van een assembler wordt de assembly code omgezet in een binaire machinecode. Tenslotte worden verschillende binaire object-files gelinkt tot een enkele executable die kan worden uitgevoerd. Deze stappen worden allemaal door een compiler als `gcc` uitgevoerd zonder dat de gebruiker daar erg in heeft.

Laten we aan de hand van een eenvoudig voorbeeld (`hello.c`) en een eenvoudige architectuur (RISC-V) kennis maken met de tools. De compiler die we zullen gebruiken voor de RISC-V architectuur heeft de naam `riscv64-unknown-elf-gcc`. Deze compiler draait op een Intel architectuur (de computer waar je nu achter zit of op bent ingelogd), maar schrijft codes voor de RISC-V architectuur. We noemen dit soort compilers "cross-compilers" om een verschil aan te duiden met reguliere compilers die code genereren voor dezelfde architectuur als waar de compiler op draait (bijvoorbeeld, het normale `gcc` commando is een x86_64 binary welke ook x86_64 codes genereert).

Voer het volgende commando uit om de C source code om te zetten in een door de mens leesbare assembly code (dit is het resultaat voordat de omzetting naar binaire code zal plaatsvinden):

```
riscv64-unknown-elf-gcc -Wall -S hello.c
```

Het resultaat is te vinden in `hello.s`. Open dit bestand in je favoriete teksteditor. Of kopieer het naar je lokale computer met sftp en open het daar. De commando's die beginnen met een punt, zoals `.align` zijn instructies voor de assembler en geen machine-instructies. Labels zijn helemaal links uitgelijnd en eindigen met een dubbele punt. Deze labels duiden locaties in het programma aan. De meeste andere regels bevatten machine-instructies zoals `addi`.

We komen later op het bestuderen van deze programma's terug. Met het volgende commando kan deze assembly code worden omgezet in een executable, een uitvoerbaar bestand:

```
riscv64-unknown-elf-gcc -o hello hello.s
```

Verifieer het type executable met

```
file ./hello
```

In de uitvoer moet nu blijken dat dit een ELF executable is voor de UCB RISC-V architectuur. Probeer nu het programma uit te voeren:

```
./hello
```

En? Werkt het? Of krijg je de fout "cannot execute binary file"? Hoe zou dat komen?

Het antwoord is eenvoudig: de executable die we hebben gemaakt bevat RISC-V instructies. De CPU in de computer waar we op werken ondersteunt de x86_64 ISA. Deze zijn *niet* compatibel met elkaar! Om ons programma uit te kunnen voeren, moeten we gebruik maken van een *emulator*. Een *emulator* kan instructies van de ene architectuur omzetten in die van een andere architectuur. Er zijn veel verschillende emulatoren en simulatoren. Deze verschillen voornamelijk in de mate van getrouwheid van de emulatie. Zo zijn er emulatoren die simpelweg alleen de instructies correct omzetten van de ene naar de andere ISA en zorgen dat een programma op een equivalente manier worden uitgevoerd. Een stapje verder is een "full-system emulator", die ook alle randapparatuur correct emuleert. Binnen een dergelijke emulator is het mogelijk om een volledig besturingssysteem op te starten. Bijvoorbeeld, de qemu emulator kan een volledige versie van Linux opstarten op een geëmuleerd RISC-V systeem (en overigens voor vele andere architecturen). Qemu houdt geen rekening met gedetailleerde karakteristieken van de processor en voert de instructies zo snel als mogelijk uit. Weer een stapje verder is de "cycle-accurate simulator". Dit zijn zeer precieze CPU simulatoren, die een CPU exact simuleren. Instructies worden uitgevoerd binnen hetzelfde aantal cycles als op de echte CPU en ook de simulatie van bijvoorbeeld caching komt precies overeen met de hardware implementatie.

We zullen nu gebruik maken van een eenvoudige emulator van de eerste categorie: de GDB simulator. GDB is eigenlijk een debugger, maar bevat ook een simpele emulator voor de verschillende architecturen die het ondersteunt. We kunnen onze "RISC-V binary" nu uitvoeren met het volgende commando:

```
riscv64-unknown-elf-run ./hello
```

Als alles nu goed gaat, wordt er "hello" op het scherm gezet. Je kunt de simulator ook alle instructies laten weergeven die het uitvoert (instruction trace):

```
riscv64-unknown-elf-run --trace-insn=on ./hello
```

Gegeven een binary, dan kunnen we ook de assembly code bekijken. Dit noemen we het "disassemblen" van een binary executable. Gegeven een RISC-V binary `hello`, dan kun je het volgende commando gebruiken:

```
riscv64-unknown-elf-objdump -d ./hello
```

Tip: vang de uitvoer op met de toevoeging `| less`. In de uitvoer zie je links voor de dubbele punt het adres van de instructie in het programma, in het midden is de geëncodeerde instructie en rechts is de gedecodeerde instructie. Het `objdump` commando werkt ook voor `.o` bestanden.

Werken met x86_64 assembly

We kunnen ook de assembly code bekijken die voor een x86_64 machine wordt gegenereerd. Omdat we al op een dergelijke machine werken, hoeven we geen cross-compiler te gebruiken. Genereer de assembly code als volgt:

```
gcc -Wall -S hello.c
```

Dit resulteert weer in een bestand `hello.s`. Je kunt een executable maken met

```
gcc -o hello hello.s
```

Verifieer weer het type executable met het commando `file`. Probeer het programma uit te voeren. Omdat we werken op een x86_64 machine, hebben we geen emulator nodig om een x86_64 executable uit te voeren. Het te gebruiken objdump-commando is `objdump -d`.

Studie van verschillende ISAs

Je bent nu volledig uitgerust om een studie van verschillende Instruction Set Architectures te beginnen! Je kunt nu werken met de tools voor twee verschillende architecturen: RISC-V en x86_64. Deze architecturen gaan we verkennen door verschillende voorbeeldprogramma's te bestuderen, van makkelijk naar moeilijk. We beginnen met het eenvoudigste voorbeeld en voegen dan steeds elementen toe. Je zult zien dat door het bestuderen van deze programma's je steeds meer patronen gaat herkennen en dat het lezen van assembly code steeds makkelijker wordt.

We raden aan om te beginnen met RISC-V en pas zodra je dit beheerst de sprong te maken naar x86_64.

RISC-V

Laten we beginnen door de assembly te bekijken van het minimale voorbeeldprogramma, dat wanneer geschreven in C alleen een `return 0` uitvoert. Je kunt het programma vinden in de subdirectory `patronen` met naam `minimaal.c`. Genereer voor dit programma de assembly output voor RISC-V door gebruik te maken van de `-S` optie van de (juiste) compiler. Open de assembly output, `minimaal.s` in een teksteditor. Stel je hierbij de volgende vragen:

- Kun je verschillende operanden herkennen? Register, immediate, memory address.
- Kun je klassen van instructies herkennen? De instructies indelen in bepaalde categoriën die iets gemeen hebben?

Zonder verdere informatie is het niet eenvoudig om de assembly code te begrijpen. De namen van de instructies zijn cryptisch. Om leesbaar te zijn voor de mens, moeten we een idee hebben wat deze cryptische afkortingen betekenen en wat deze instructies precies doen. Hiervoor moeten we ons wenden tot de ISA manuals. Elke processor/ISA heeft een handleiding, waarin in

zeer groot detail wordt uitgelegd welke effecten het uitvoeren van elke instructie heeft. De ISA manual voor RISC-V is hier te vinden:

`https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`

Alle instructies worden hierin in redelijk detail uitgelegd. Zoek bijvoorbeeld eens naar `ADDI` (Sectie 2.4, pagina 13) of `SD` (Sectie 4.3, pagina 31).

De RISC-V architectuur heeft 32 general-purpose registers, normaal gesproken genummerd van `x0` tot `x31`. We zien in deze assembly output echter andere registernamen terug. Dit komt doordat er een registerconventie wordt gebruikt, waarin bepaalde registers zijn gereserveerd voor bepaalde taken. Registers krijgen hierdoor een extra naam (alias) welke je vaak terugziet in de assembly codes. De conventie voor RISC-V is als volgt:

Register	**Alias**	**Omschrijving**	**Saver**
x0	zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5-7	t0-2	Temporaries	Caller
x8-9	s0-1	Saved registers	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s3-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

Deze informatie is ook terug te vinden in de handleiding als Tabel 20.1, Hoofdstuk 20, pagina 109. Het is ook mogelijk om assembly output te krijgen waarin niet de aliassen worden gebruikt. Dit kan met objdump, maak dan eerst een object file (of executable) en gebruik dan de objdump optie `-M numeric`:

```
riscv64-unknown-elf-gcc -c minimaal.c
riscv64-unknown-elf-objdump -M numeric -d minimaal.o
```

De volgende stap is om alle instructies in `minimaal.s` te annoteren. Zoek op wat iedere instructie precies doet en welke operanden worden gebruikt. Hierna kun je gaan kijken wat de instructies samen bereiken. In feite ga je op een hoger niveau naar de instructies kijken en op zoek naar patronen. In de eerste drie instructies zien we dat de stack pointer `sp` wordt gebruikt. Hier wordt een stack frame opgebouwd, zie ook de ondersteunende slides voor het practicum (BrightSpace). De laatste 3 instructies bouwen het stack frame af en voeren een return uit. De 2 instructies die in het midden "overblijven" zijn de daadwerkelijke kern van het programma, hier wordt het getal `0` in het return value register `a0` geladen.

Lokale variabelen

Bekijk als volgende stap hoe er wordt omgegaan met lokale variabelen. Genereer hiertoe de assembly voor `lokaal.c`. We zien dat vergeleken met `minimaal.c` het stackframe groter is (32 tegen 16 bytes). Kun je ontdekken op welke geheugenlocaties in het stackframe de verschillende lokale variabelen worden opgeslagen? Annoteer de assembly code en probeer te ontdekken welke regels assembly corresponderen met welke regels van het originele C programma.

Functie-aanroepen

Functie-aanroepen kun je het beste in twee stappen bestuderen. Eerst het aanroepen van functies zonder argumenten, maar die wel een returnwaarde hebben. En vervolgens het aanroepen van functies met argumenten. Met `func.c` kun je het eenvoudige geval bestuderen. Maak zo nodig gebruik van de ondersteunende slides voor het practicum.

`func2.c` bevat een functie met twee integer-argumenten. Hoe geven we nu in assembly code die argumenten door aan de functie? Aan de ene kant kun je zeggen dat je dat zelf kunt bepalen. Aan de andere kant, als iedereen een eigen manier bedenkt voor het doorgeven van functie-argumenten, dan wordt het wel heel moeilijk om functies aan te roepen in modules (bibliotheken, libraries) geschreven door anderen. Om dit op te lossen maken we afspraken hoe we functie-argumenten doorgeven. Wanneer iedereen zich hieraan houdt, kunnen we elkaars software laten samenwerken. Gelukkig worden functie-aanroepen op assemblyniveau normaal gesproken door de compiler gegenereerd, dus de compiler implementeert voor ons deze afspraken. We noemen deze serie van afspraken een *calling convention*. Let op: voor eenzelfde architectuur kunnen er meerdere calling conventions bestaan.

Voor RISC-V kunnen we kijken naar bovenstaande tabel. We vinden daar dat registers `a0` tot en met `a7` worden gebruikt voor functie-argumenten. In de assembly van `func2.c` vinden we dit inderdaad terug, ga dit na! Bekijk ook waar de formele argumenten van de functie worden opgeslagen; zijn dit gewoon lokale variabelen?

Je kunt zelf nog wat nader onderzoek doen naar functie-aanroepen. Schrijf bijvoorbeeld zelf kleine voorbeelden en bekijk de assembly voor gevallen als:

- Hoe worden argumenten van andere types doorgegeven? Bijvoorbeeld pointers?
- Hoe worden arrays doorgegeven als argument?
- Wat gebeurt er als een functie meer dan 8 argumenten heeft?
- En hoe worden structures call-by-value doorgegeven?

Tenslotte nog een laatste opmerkingen over calling conventions. In de tabel met de registerconventie zien we ook een kolom “Saver” staan. Deze geeft aan wie er verantwoordelijk is voor het bewaren van de huidige waarde van het register tijdens de functie-aanroep: de functie die aanroept (caller) of de functie die wordt aangeroepen (callee). Het tijdelijk bewaren van de inhoud van deze registers gebeurt weer middels de stack.

Arrays

Een array is een reeks van elementen van hetzelfde type die achterelkaar worden opgeslagen. In de meeste gevallen slaan we een pointer op naar het eerste array element. Middels deze pointer, de array-index en de grootte van een array-element kunnen we dan het adres van ieder element in de array berekenen. We zien dit in de assembly codes terug. Bekijk en annoteer de assembly van `array.c`.

Wat hier opvalt is het gebruik van `%hi` en `%lo`. Constante waarden die in instructies worden geëncodeerd (immediates) kunnen in RISC-V maximaal 12-bit groot zijn. Dat is niet genoeg voor de meeste geheugenadressen. We zien daarom een speciale constructie waarbij eerst de hoge bits van een adres (bepaald met `%hi`) worden geladen (met de instructie `lui` – ga na wat deze precies doet) en daarna worden de lage bits (`%lo`) erbij opgeteld.

For-Loops

Arrays worden interessanter wanneer deze in combinatie met een loop-structuur worden gebruikt. Zodra we loops gaan gebruiken wordt de assembly code ook ingewikkelder. Er moet een stuk code worden herhaald, een conditie getest en gestopt met herhalen wanneer nodig. Bekijk en annoteer de assembly van `loop.c`. Denk in het bijzonder aan de volgende vragen:

- Op welke regels wordt de loop-variabele geïnitialiseerd?
- Op welke regels wordt de loop-variabele opgehoogd?

- Kun je de loop-body aanwijzen (regelnummers)?
- Hoe wordt de loop-variabele gebruikt in de loop body? En hoe wordt deze gebruikt in de adresberekeningen voor de array-elementen?
- Waar wordt de loop-conditie getest?

Wanneer we delen van een programma gaan herhalen, is het logisch dat het aantal instructies dat is opgenomen in het programma gaat afwijken van het aantal instructies dat daadwerkelijk door het programma wordt uitgevoerd. Voer daarom ook met de hand een aantal instructietellingen uit:

- het aantal statische instructies (het aantal instructies vastgelegd in de assembly code),
- het aantal dynamische instructies (het aantal instructies dat daadwerkelijk wordt uitgevoerd),
- het aantal loads,
- het aantal stores.

Het aantal dynamische instructies kun je controleren door gebruik te maken van de emulator met de instruction trace optie. Let op: er worden meer functies uitgevoerd dan de functie die is gedefinieerd in ons `.s`-bestand. Met `objdump` kun je nagaan op welke geheugenadressen de functie staat waarin je bent geïnteresseerd. Tel vervolgens in de instruction trace alleen die instructies op deze adressen (of schrijf zo nodig een klein Python-script om te filteren).

If-statements

Het bestand `if.c` bevat een if-elseif-else statement. Genereer en annoteer de assembly code. Ga na welke regels assembly corresponderen met welke regels C-code. Bepaal ook een statische en dynamische instructietelling, de uitvoer van dit bepaalde programma ligt vast. Maak zo nodig gebruik van het if-statement zoals besproken in de ondersteunende slides.

Andere patronen

Je kunt ook nog een aantal andere patronen bestuderen door hier zelf kleine programma's voor te schrijven:

- De combinatie van een loop en een if-statement in de loop body.
- Het werken met 2-d arrays.
- While-loops.
- Enzovoort.

Compileroptimalisaties

De gcc compiler gebruikt standaard de optie `-O0` en dat betekent dat de compiler geen optimalisaties toepast op de code. In sommige van de programma's die je tot nu toe hebt bekeken heb je wellicht al gezien dat deze verschillende onnodige instructies en herhalingen bevatten. Bijvoorbeeld een waarde die naar het geheugen wordt geschreven met een store-instructie en met de instructie daarna direct weer uit het geheugen wordt gehaald.

De compiler is uitgerust met optimalisaties waarmee het de code die het genereert significant kan verbeteren. Er zijn verschillende niveaus van optimalisatie, aan te zetten met één van de opties `-O1`, `-O2` en `-O3`. Hoe hoger het nummer, hoe meer en agressievere optimalisaties zullen worden toegepast.

Het is interessant en leerzaam om te bekijken wat voor optimalisaties de compiler nu precies toepast. Om dit te doen compileer je één keer het bestand met `-S` en de standaardopties en een keer met de optie `-O2` erbij. Vang de uitvoer op in verschillende bestanden! Gebruik de optie `-o`. Vervolgens annoteer je beide bestanden en daarna kun je ze vergelijken.

Bekijk om te beginnen eens de `-O0` en `-O2` versies van `minimaal.c`. Wat is het verschil in aantal instructies tussen beide versies? Worden onnodige operaties weggehaald? Andere interessante gevallen om te bekijken zijn bijvoorbeeld:

- `lokaal.c`: kun je verklaren wat hier gebeurt? En hoe kan een compiler dat bepalen?
- `loop.c`: we zien ook hier de code veel compacter worden. Wat voor veranderingen zijn er gemaakt?
- `if.c`: wat gebeurt hier? Kijk goed naar de main-functie!

Voorbereiding voor de toets

Het bestuderen van de verschillende patronen is een belangrijke stap om assembly te doorgronden. Om nog verder te oefenen voor de toets is er nog wat extra oefenmateriaal in de directories `prog` en `oefen-riscv`. De programma's in `oefen-riscv` zijn programma's die in het verleden in toetsen zijn gebruikt. Je kunt werkende executables maken van deze programma's en deze uitvoeren met de emulator. Let hier bij op: voor `sumdemo.c` en `fibo.c` moet je ook het bestand `roman.c` meecompileren en meelinken.

Dit academisch jaar zal de toets online worden afgenomen. We zullen op BrightSpace een voorbeeldtoets met uitwerking publiceren. Wat voor vragen kun je verwachten?:

- Geef een regelnummer voor ieder van de volgende typen instructies: I-type, R-type, S-type. Of gegeven een instructie, van welk type is deze?
- Geef regelnummer van instructies die een immediate-operand bevatten.
- Welke regels maken deel uit van een functie-aanroep? Of gegeven een instructie, bevat deze een immediate?
- Gegeven een assembly code, wat wordt er precies in het stackframe opgeslagen bij een aanroep naar een bepaalde functie?
- Geef regelnummers van de loop-initialisatie, loop-body en loop conditie check.
- Gegeven een C en assembly code, op welke geheugenlocatie(s) wordt variabele `a` opgeslagen?
- Gegeven een C en assembly code, in welke registers wordt variabele `b` tijdelijk opgeslagen? En bij welke regelnummers is dat het geval?
- Gegeven een C en assembly code, geef voor elke regel assembly aan met welke regel van het C programma deze correspondeert.
- Leg uit op welke regels assembly en hoe het geheugenadres wordt berekend voor de array-access `A[31]`.
- Gegeven een assembly code, hoeveel statische en dynamische instructies zullen worden uitgevoerd?
- Gegeven een assembly code gecompileerd met `-O0` en `-O2`. Leg uit wat voor compileroptimalisaties hier zijn toegepast.
- Gegeven een assembly code, maak een aanpassing zodanig dat de loop 20 iteraties uitvoert in plaats van 10.

Wanneer er een (stukje) programma moet worden geannoteerd, vinden wij het belangrijk dat niet de letterlijke werking van de instructie wordt gegeven ("laad 4 bytes van adres 0x1234 in register 13"), maar dat de daadwerkelijke werking van het programma wordt uitgelegd en wat er met de instructies wordt bereikt.

x86_64

Naast RISC-V, bekijken we ook x86_64 assembly. Wel gaan we iets minder diep op deze assembly in. Binnen x86_64 zijn er 16 general-purpose integer registers[1]:

Register	**Omschrijving**
rax	Return value
rbx	Callee saved
rcx	Function argument 4
rdx	Function argument 3
rsi	Function argument 2
rdi	Function argument 1
rbp	Base pointer / frame pointer (callee saved)
rsp	Stack pointer
r8	Function argument 5
r9	Function argument 6
r10	Caller saved
r11	Caller saved
r12	Callee saved
r13	Callee saved
r14	Callee saved
r15	Callee saved

Wanneer er `eax` wordt geschreven in plaats van `rax`, dan wordt er gewerkt met de laagste 32-bits uit het `rax` register. Hetzelfde geldt voor de andere eerste 8 registers, bijvoorbeeld `rdi` wordt `edi`.

Verder is het belangrijk om te vermelden dat er voor x86_64 twee verschillende assembly language syntaxen bestaan: Intel syntax en AT&T syntax. Op UNIX systemen is AT&T syntax gangbaar en daarnaast is de output van de GCC compiler ook in AT&T syntax. We richten ons daarom op de AT&T syntax, zodat we de output van GCC kunnen begrijpen en daarmee kunnen werken. Maar je zult in documentatie en handleidingen ook de alternatieve Intel syntax tegenkomen.

Kenmerkend voor de AT&T syntax is het plaatsen van `%` voor de namen van registers (in Intel syntax gebeurt dat niet) en het gebruik van ronde haken om *indirection*, of *memory access*, aan te duiden (Intel syntax gebruikt hiervoor vierkante haken). Daarnaast is de volgorde van operanden tegenovergesteld. Voorbeelden van x86_64 instructies in AT&T syntax zijn:

```
mov    %rax,%rdx
mov    %rdx,0x58(%rsp)
lea    (%rdx,%rbp,8),%rax
```

Een paar andere tips wat betreft syntax:

- De argumenten van Intel assembly in AT&T syntax moeten de andere kant op worden gelezen vergeleken met RISC-V, dus van links (source) naar rechts (destination).
- `$0` is een immediate met de waarde `0` en geen register.
- `cltq` heet in de Intel documentatie `CDQE`.

[1] Er zijn ook nog aparte floating-point, MMX en SSE registers. We bekijken deze op dit moment nog niet.

De ISA manual voor x86_64 kan hier worden gevonden:
`https://community.intel.com/legacyfs/online/drupal_files/managed/a4/60/325383-sdm-vol-2abcd.pdf`

Op de volgende website is het misschien prettiger zoeken naar instructies: `http://www.felixcloutier.com/x86/`.

Ook bestaan er verschillende websites waar je meer kunt leren over x86_64, bijvoorbeeld:

- Intel's introductie tot x86_64 assembly (wel in Intel syntax!): `https://www.intel.com/content/dam/develop/external/us/en/documents/introduction-to-x64-assembly-181178.pdf`.
- Duidelijke uitleg over het uitvoeren van functieaanroepen binnen x86_64: `http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64`.
- x86_64 cheatsheet van Brown University (in AT&T syntax): `https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf`.

Patronen

Net als bij RISC-V raden we je aan om te beginnen met het bestuderen van de verschillende patronen met behulp van eenvoudige testprogramma's. Je kunt hiervoor dezelfde programma gebruiken. Gebruik `gcc` om assembly-output te genereren.

Compileroptimalisaties

Voor x86_64 kijken we *niet* naar compileroptimalisaties.

Voorbereiding voor de toets

Voor de toets beperken we ons tot kleine programmafragmenten, in AT&T syntax, waarover we vragen stellen als:

- Bekijk het volgende assembly programma en leg de werking uit (wat doet het programma precies?)
- Gegeven een assembly code, hoeveel dynamische instructies worden er uitgevoerd? Maak dit afhankelijk van een variabele.
- Geef regelnummers van instructies waarbij geheugentoegang plaatsvindt.
- Geef regelnummers van instructies die gebruik maken van een immediate.
- Geef regelnummers van instructies waarin een effectief geheugenadres wordt berekend.
- Geef regelnummers van de loop body en loop conditie check.

Als oefenmateriaal vind je in de directory `oefen-intel` twee programma's. Richt je in het bijzonder op de functie waarin de berekening wordt uitgevoerd en waar je een loop en array-accesses terugvindt.