Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Számítógép-alapú rendszerek specializáció

Nagyteljesítményű mikrokontrollerek és interfészek laboratórium

Jegyzőkönyv

1. mérés

Processzorok funkcionális egységeinek vizsgálata

|  |  |
| --- | --- |
| **Mérést végző hallgatók:** | [név] [NEPTUN kód]  [név] [NEPTUN kód] |
| **Mérőcsoport:** | Kurzus: [kurzus kód], Csoport: [csoport száma] |
| **Mérés időpontja:** | [dátum] de/du |
| **Mérés helyszíne:** | BME AUT, labor: QB[teremszám] |
| **Mérésvezető:** | [mérésvezető neve] |

# Mérési feladatok

## A QtRvSim szimulátor áttekintése, az utasításvégrehajtás vizsgálata

Az első feladatban célunk a labormérés legfontosabb eszköze, a **QtRvSim** nevű RISC-V szimulátor és az alap RISC-V utasításkészlet használatának megismerése. Az alábbi részfeladatok többségét a mérésvezető segítségével kell végigcsinálni, az önállóan megoldandó részfeladatokat külön megjelöltük.

* + 1. Nyissa meg a QtRvSim szimulátort. Az új szimuláció felugró ablakában válassza a „No pipeline no cache” opciót, és töltse be az alapértelmezett minta assembly kódot az „Example” gomb megnyomásával (template.S). A mérésvezető segítségével tekintse át a forráskód felépítését (.equ, .org, .text, .data, .word, .asciz direktívák, assembly utasítások).
    2. Fordítsa le a kódot és töltse be a memóriába! (Machine > Compile source) Milyen eltéréseket lát a forráskód és a programmemória nézet „Instruction” oszlopában lévő utasítások között? Futtassa a programot, és a mérésvezető segítségével elemezze a működését!

Tapasztalat:

* + 1. Vesse össze az adatmemória tartalmát („Memory”) a forráskódban látottakkal. Vajon Neumann vagy Harvard jellegű a processzor memóriastruktúrája?

Megoldás:

* + 1. A „Core” nézetben figyelje meg az utasításvégrehajtás folyamatát! A mérésvezető segítségével kövesse végig, hogyan történik az utasítás felhozatala, a programszámláló inkrementálása, az utasítás dekódolása.

A segédanyagok között megtalálható utasításkészlet-összefoglaló alapján vizsgáljuk meg közelebbről a legelső végrehajtott utasításszó (LUI x10, 0xffffc) bináris felépítését, keressük meg benne a célregiszter azonosítóját és az immediate konstanst tartalmazó bitmezőket:

U-type:

| immediate[31:12] | rd | opcode |

|-------------------------------------------------------------|-------------|---------------|

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12 | 11 10 9 8|7 | 6 5 4|3 2 1 0 |

| **F** | **F** | **F** | **F** | **C**  | **5**  |**3**  |**7**  |

| 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 | 0 1 0 1 0 | 0 1 1 0 1 1 1 |

| immediate = 0xFFFFC000 | rd = 0x0A | opcode: LUI |

Figyelje meg, hogy a végrehajtóegység (ALU) bemeneteire milyen forrásból származó adatok kerülhetnek, és hogy az ALU művelet eredménye milyen irányokba terjedhet tovább.

Figyelje meg, hogy adatmemória-írás esetén honnan származik a memóriacím és a beírandó adat, illetve, hogy olvasás esetén hova kerülhet a kiolvasott érték (load/store architektúra). Nézze meg, hogy a regiszter-visszaírásnál a beírandó adat honnan származhat.

* + 1. **Önálló feladat:** Dekódolja a 0x00000210 címen lévő „beq t1, zero, end\_char” utasítást! Mi az rs1, rs2 mezők és az utasításba ágyazott immediate konstans numerikus értéke? (Javasolt a segédanyagok között megtalálható RV32I\_instruction\_formats.txt fájlban lévő minták használata.)
    2. Nézzük meg a példaprogram működését pipeline utasításvégrehajtás esetén! Ehhez indítsunk egy új szimulációt (File > New simlutation…), a felugró ablakban váltsunk a „Core” fülre, és jelöljük be a „Pipelined” opciót (a „Hazard unit”-ot egyelőre ne). Az „Example” gombbal töltsük be a példaprogramot, és futtassuk! Mi történik? Léptetéssel keressük meg a jelenség okát!

Megoldás:

* + 1. Indítsunk új szimulációt, de most a „Core” fülön kapcsoljuk be a „Hazard unit”-ot is, és válasszuk a „Stall when hazard is detected” lehetőséget. Ismét töltsük be a példaprogramot a memóriába, és futtassuk! Hová szúr be az utasítás-csővezeték stall fázisokat és miért? (Ezeket a szimulátor „nop”-ként mutatja a megfelelő fázisoknál.) Összesen hány órajelciklus alatt hajtja így végre a processzor a programot, és ebből mennyi a stall fázis?

Megoldás:

* + 1. Ismét indítsunk új szimulációt, és a „Core” fülön a „Hazard unit” működésénél válasszuk a „Stall or forward when hazard is detected” lehetőséget. Töltsük be a példaprogramot a memóriába, és futtassuk! Figyeljük meg, hogy mi változott a processzormag blokkvázlatán! Mit tapasztalunk a függőséget okozó utasításpárok végrehajtásánál? Összesen hány órajelciklus alatt hajtja így végre a processzor a programot, és ebből mennyi a stall fázis? Mekkora sebességnövekedés érhető el az előrecsatolás segítségével?

Megoldás:

* + 1. **Önálló feladat:** Töltse be a „template\_mod.s” módosított mintaprogramot a szimulátorba és futtassa! *(File > Open source)* Miben tér el a működése az eredetihez képest? Vizsgálja meg a forráskódot és keresse meg a jelenség okát! Módosítsa úgy a kódot, hogy „\*” helyett „?” karakterek jelenjenek meg a terminálon!

Megoldás:

# Módosított kódrészlet:

# ----------------------

## RISC-V utasításkészlet és assembler vizsgálata

Indítson egy szimulációt pipeline végrehajtás nélkül, és nyissa meg a segédanyagok közül a „bcd2bin.s” assembly programot. Ez egy főprogramot és egy szubrutint (függvényt) tartalmaz. A szubrutin (bcd2bin) egy cím szerint átadott 32 bites BCD számot (amely bájtonként 1 decimális számjegyet tartalmaz) konvertál egyszerű bináris formátumra. A főprogram (main) feladata a bemeneti paraméter előkészítése és a függvényhívás.

* + 1. Fordítsa le a kódot és töltse be a memóriába! (Machine > Compile source) Milyen eltéréseket lát a forráskód és a programmemória nézet „Instruction” oszlopában lévő utasítások között? Sorolja fel a forráskódban megtalálható pszeudoutasításokat, és a nekik megfelelő valódi gépi utasításokat!

Megoldás:

* + 1. Futtassa a programot, és figyelje meg a működését! Mennyi lesz a „bcd2bin” szubrutin visszatérésekor az a0 regiszter értéke?

Megoldás:

* + 1. Eljut a programvégrehajtás az „ebreak” szoftveres töréspontig? Miért nem? Hogyan oldaná meg ezt a problémát? Módosítsa a kódot!

Válasz:

# Módosított kódrészlet:

# ----------------------

* + 1. A bcd2bin szubrutinon belül a *li* regisztertöltő pszeudoutasítások fölöslegesen 2–2 db gépi utasításra fordulnak. Egyszerűsítse a kódot, hogy erre ne legyen szükség! Végül mentse el a kijavított forráskódot „bcd2bin\_fix.s” néven!

# Módosított kódrészlet:

# ----------------------

## Pipeline végrehajtás vizsgálata, manuális kódoptimalizálás

* + 1. Futtassa az előző feladatban kijavított assembly programot „single cycle” üzemmódban (azaz kikapcsolt pipeline mellett). Hány órajelciklus alatt hajtja végre a processzor?

Megoldás:

* + 1. Mi történik, ha a processzor „pipeline” üzemmódban működik, de kikapcsolt hazard unit mellett?

Megoldás:

* + 1. Oldja fel a felmerülő függőségeket szofveresen, NOP-ok beszúrásával, hogy a működés újra helyes legyen! Hány plusz NOP utasításra van szükség ehhez? Hány órajelciklus alatt hajtja végre a processzor a NOP‑okkal kiegészített programot? Tegyük fel, hogy az ötfázisú pipeline ötszörös órajelfrekvenciát tesz lehetővé a single cycle üzemmódhoz képest. Mekkora sebességnövekedést sikerült elérni a pipeline végrehajtással így, hogy NOP-ok beszúrására is szükség volt?

Válasz:

# Módosított kódrészlet:

# ----------------------

* + 1. Hány órajelciklus alatt hajtja végre a processzor az eredeti, NOP-ok nélküli kódot (bcd2bin\_fix.s) „pipeline” üzemmódban, ha be van kapcsolva a hazard unit, és stall fázisok beszúrásával oldja fel a függőségeket? Hogyan viszonyul ez a NOP-ok beszúrásával történő szoftveres megoldáshoz?

Megoldás:

* + 1. Mekkora sebességnövekedés érhető el ehhez képest az utasítások sorrendjének megváltoztatásával? Rendezze át a „bcd2bin” szubrutinban lévő utasításokat úgy, hogy az eredmény ugyanaz maradjon, de a lehető legkevesebb stall ciklusra legyen szükség! Számítsa ki az elért sebességnövekedést! Mentse el az átrendezett forráskódot „bcd2bin\_reordered.s” néven!

Válasz:

# Módosított kódrészlet:

# ----------------------

* + 1. Mekkora sebességnövekedés érhető el az eredeti (bcd2bin\_fix.s) és az átrendezett (bcd2bin\_reordered.s) kód esetén, ha bekapcsoljuk az operandus-előrecsatolást az utasítás-csővezetékben? Mennyi az előrecsatolás által okozott sebességnövekedés a stall fázisok beszúrásához képest?

Megoldás:

* + 1. Mennyi a pipeline végrehajtás, az operandus-előrecsatolás és az utasítások átrendezése segítségével összesen elért sebességnövekedés az eredeti (bcd2bin\_fix.s) kódhoz és a „single cycle” üzemmódhoz képest?

Megoldás:

## Optimalizáló fordító hatásának vizsgálata

Ebben a feladatban egy egyszerű C programot fogunk RISC-V assembly-re, illetve futtatható bináris formátumra (ELF – Executable and Linkable Format) fordítani. Ennek során közelebbről is megismerkedünk a fordítók által alkalmazott standard regiszterkezelési, függvényhívási, stack-kezelési konvenciókkal (Application Binary Interface – ABI), amelyeket RISC-V architektúra esetén a „RISC-V ABIs Specification” [3] dokumentum ír elő. Ezen kívül arra is kíváncsiak leszünk, hogy milyen hatással vannak az eredményül kapott assembly kód hatékonyságára a fordító optimalizálási beállításai.

A fordításhoz a GNU RISC-V Embedded GCC fordítót fogjuk parancssorból meghívni. A C forrásfájlok és a generált assembly kódok áttekinthető kezeléséhez és vizsgálatához a Visual Studio Code IDE-t használjuk.

* + 1. Indítsa el a VSCode programot, és nyissa meg benne a mérési segédanyag „source” mappáját (File > Open Folder…)! Nyissa meg a „gcd.c” forráskódot, és tekintse át a működését! Ezután jelenítse meg a VSCode beépített terminálablakát (View > Terminal), és az alábbi parancs segítségével fordítsa le RISC-V assembly-re:

riscv-none-elf-gcc -O0 -march=rv32im -mabi=ilp32 gcd.c -S -fverbose-asm

A létrejött „gcd.s” assembly forrásfájlt nevezze át „gcd\_O0.s”-re, ezzel jelezve, hogy ez optimalizálás nélküli fordítással készült. Nyissa meg úgy a VSCode-ban, hogy a C forrás mellett látszódjon az assembly (View > Editor Layout > Split Right). Tanulmányozza az assembly kódot, keresse meg az összetartozó részeket! (Megjegyzés: A fordítási parancsban az „-fverbose-asm” kapcsoló hatására kommentként megjelennek az assembly fájlban a C forrás megfelelő sorai, ami segíti a tájékozódást.)

Érdekességként megemlítjük, hogy a különböző fordítók működésének tanulmányozására, illetve a forráskód és a generált assembly kód közötti összefüggések vizsgálatára remekül használható a Compiler Explorer webalkalmazás is (<https://godbolt.org/>). Bemásolva a C forráskódot, és kiválasztva az általunk használt fordítót, ugyanazt az assembly kódot kell kapjuk. Az általunk használt GCC fordító verziószámát az alábbi paranccsal tudjuk lekérni:

riscv-none-elf-gcc -v

Az assembly kód vizsgálata alapján válaszolja meg az alábbi kérdéseket:

1. A „main” függvény hol tárolja az a, b, c lokális változókat?

Megoldás:

1. A „main” függvény példáján keresztül mutassa be a stack-kezelés legfontosabb tulajdonságait! Mire szolgálnak az alábbi utasítások a függvény elején? Írja melléjük kommentben!

main:  
 addi sp,sp,-32 #   
 sw ra,28(sp) #   
 sw s0,24(sp) #   
 addi s0,sp,32 #

1. Hogyan adja át a main a paramétereket a gcd függvénynek? Stack-en vagy regiszterekben? Hol található a gcd visszatérési értéke?

Megoldás:

1. A gcd függvény hol tárolja és hogyan használja az x és y lokális változókat?

Megoldás:

* + 1. Fordítsa le újra assembly-re a „gcd.c” programot, de most engedélyezze az 1-es szintű (O1) optimalizálást:

riscv-none-elf-gcc -O1 -march=rv32im -mabi=ilp32 gcd.c -S -fverbose-asm

A létrejött „gcd.s” assembly forrásfájlt nevezze át „gcd\_O1.s”-re, és nyissa meg a VSCode-ban a C forrás és az optimalizálás nélküli assembly kód mellett! Tanulmányozza az új assembly kódot, és válaszolja meg az alábbi kérdéseket!

1. Hol tárolja a „main” függvény az a, b, c lokális változókat?

Megoldás:

1. Változott-e valamiben a „main” függvény stack-kezelése?

Megoldás:

1. Hol tárolja a „gcd” függvény az x, y lokális változókat?

Megoldás:

1. Változott-e valamiben a „gcd” függvény stack-kezelése?

Megoldás:

* + 1. Hasonlítsa össze a kétféle generált kód futásidejét! Ehhez a GCC segítségével futtatható tárgykódot kell készíteni, amit aztán be lehet tölteni a szimulátorba. A kétféle tárgykód létrehozásához adja ki a terminálban a kövezkező két parancsot:

riscv-none-elf-gcc -O0 -march=rv32im -mabi=ilp32 -nostdlib -o gcd\_O0.elf gcd.c crt0local.s -lgcc  
  
 riscv-none-elf-gcc -O1 -march=rv32im -mabi=ilp32 -nostdlib -o gcd\_O1.elf gcd.c crt0local.s -lgcc

(Vegyük észre, hogy a „gcd.c” mellett szerepel a parancsban egy „crt0local.s” is a bemeneti fájlok között. Ez nem más, mint egy minimális C startup kód, ami pl. beállítja a stack pointer kezdőértékét, meghívja a main függvényt, és kezeli azt az esetet, amikor a main visszatér.)

Töltse be a létrejött .elf fájlokat egymás után a QtRvSim szimulátorba, és futtassa! Ehhez a File > New simulation… felugró ablakában adja meg az .elf fájl elérési útját (Elf executable > Browse), majd nyomja meg a „Load machine” gombot. A szimulált processzorban legyen bekapcsolva a pipeline, a hazard unit és az operandus-előrecsatolás. A futtatásnál érdemes az „Unlimited” vagy a „Max” szimulációs sebességet választani. Jegyezze fel mindkét lefordított programváltozat esetén az órajelciklusok és a stall fázisok számát! Mekkora az optimalizálás által okozott sebességnövekedés?

|  |  |  |  |
| --- | --- | --- | --- |
| **Optimalizálási szint** | **Cycles** | **Stalls** | **Sebességnövekedés** |
| -O0 |  |  | – |
| -O1 |  |  |  |

## Optimalizáló fordító hatásának vizsgálata #2 (kiegészítő feladat)

* + 1. Nyissa meg a „sumArray.c” forráskódot, és tekintse át a működését! Vizsgálja meg, milyen assembly kódot generál belőle a fordító „O1” optimalizálási szint mellett:

riscv-none-elf-gcc -O1 -march=rv32im -mabi=ilp32 sumArray.c -S -fverbose-asm

A létrejött „sumArray.s” assembly forrásfájlt nevezze át „sumArray\_O1.s”-re, hogy egy későbbi fordítás ne írja majd felül. Válaszolja meg az alábbi kérdéseket az assemby kódban látottak alapján!

1. Hol tárolja a „main” függvény az alábbi lokális változóit?
   * int a[64]
   * long long s
   * int i

Megoldás:

1. Hogyan adja át a *main* a paramétereket a *sumArray* függvénynek? Stack-en vagy regiszterekben? Hol található a *sumArray* visszatérési értéke?

Megoldás:

1. Mutassa be, hogyan készíti elő a sumArray függvény a saját lokális változóit és a for ciklusban elvégzendő tömbműveletet! Mire szolgálnak a következő utasítások a függvény elején? Milyen szerepe van az a0, a1, a4 és a6 regisztereknek a kódrészletet követően? (Mi a bennük lévő adatok gyakorlati jelentése az algoritmus szempontjából?) Írja melléjük kommentben!

sumArray:  
 ble a1,zero,.L4 #   
 mv a4,a0 #   
 slli a1,a1,2 #   
 add a6,a0,a1 #   
 li a0,0 #   
 li a1,0 #

1. Hogyan végzi el a 64 bites összeadást (long long = long long + int) a 32 bites processzor? Mi a szerepe az SRAI és az SLTU utasításoknak a kódban?

Megoldás:

# Vonatkozó kódrészlet:

# ---------------------

* + 1. Az eddig látottakon túl egy további lehetséges optimalizálási módszer a ciklusok kiterítése (loop unrolling, ld. [1], 2.3.4.2. fejezet). Ezt a funkciót alapértelmezetten nem tartalmazza a GCC fordító „O1” optimalizálási szintje, de manuálisan engedélyezhető a „-funroll-loops” kapcsolóval. Fordítsa le most így a „sumArray.c” forráskódot:

riscv-none-elf-gcc -O1 -funroll-loops -march=rv32im -mabi=ilp32 sumArray.c -S -fverbose-asm

A létrejött „sumArray.s” assembly forrásfájlt nevezze át „sumArray\_O1\_unroll.s”-re. Mi változott a „sumArray\_O1.s” változathoz képest? Válaszolja meg az alábbi kérdéseket!

1. Történt-e lényeges változás a sumArray függvény előkészítő részében, vagyis az 5.1.c) pontban idézett kódrészlet megfelelőjében? Ha igen, mi a változás lényege?

Megoldás:

1. Történt-e lényeges változás a sumArray függvény for ciklusának törzsében? Ha igen, mi a változás lényege?

Megoldás:

* + 1. Hasonlítsa össze az optimalizálás nélküli (O0), optimalizált (O1) és a kiterített ciklusokat alkalmazó optimalizált változat futásidejét! Ehhez generálja le a megfelelő tárgykódokat az alábbi parancsok segítségével, majd töltse be a szimulátorba és futtassa őket!

riscv-none-elf-gcc -O0 -march=rv32im -mabi=ilp32 -nostdlib -o sumArray\_O0.elf sumArray.c crt0local.s -lgcc  
  
riscv-none-elf-gcc -O1 -march=rv32im -mabi=ilp32 -nostdlib -o sumArray\_O1.elf sumArray.c crt0local.s -lgcc  
  
riscv-none-elf-gcc -O1 -funroll-loops -march=rv32im -mabi=ilp32 -nostdlib -o sumArray\_O1\_unroll.elf sumArray.c crt0local.s -lgcc

A szimulált processzorban legyen bekapcsolva a pipeline, a hazard unit és az operandus-előrecsatolás. A futtatásnál érdemes a „Max” szimulációs sebességet választani. Jegyezze fel mindhárom esetben az órajelciklusok és a stall fázisok számát! Mekkora az különböző optimalizálási beállítások által okozott sebességnövekedés?

|  |  |  |  |
| --- | --- | --- | --- |
| **Optimalizálási szint** | **Cycles** | **Stalls** | **Sebességnövekedés** |
| -O0 |  |  | – |
| -O1 |  |  |  |
| -O1 -funroll-loops |  |  |  |

# Irodalomjegyzék

1. Gál Tibor, Tevesz Gábor, Kiss Domokos: Nagyteljesítményű mikrokontrollerek és interfészek (Elektronikus jegyzet), I. rész – Nagyteljesítményű mikrokontrollerek  
   <https://www.aut.bme.hu/Course/VIAUMA18>
2. RISC-V ISA Specifications Volume 1, Unprivileged Specification version 20191213, <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>
3. RISC-V ABIs Specification, Version 1.0: Ratified,   
   <https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/tag/v1.0>
4. RISC-V Instruction Set Specifications (Összefoglaló),  
   <https://msyksphinz-self.github.io/riscv-isadoc/html/index.html>
5. QtRvSim: RISC-V CPU simulator for education, <https://github.com/cvut/qtrvsim>
6. QtRvSim online, <https://comparch.edu.cvut.cz/qtrvsim/app/>
7. RISC-V Assembler Reference, <https://michaeljclark.github.io/asm.html>
8. Matt Godbolt: Compiler Explorer, <https://godbolt.org/>