دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر

عنوان:

تمرین شبیه‌سازی سری ۳

نگارش

رضا آدینه پور - ۴۰۲۱۳۱۰۵۵

استاد راهنما

جناب آقای دکتر فربه

۲۳ آذر ۱۴۰۲

• هدف

هدف این تمرین آشنایی با زبان Gem۵ برای توصیف مجموعه دستورات است. برای این کار فایل‌های ISA موجود در مسیر src/arch/x86/isa مورد بررسی قرار می‌گیرد. شما ابتدا مطابق با توضیحات یک دستور از معماری X۸۷ با نام FSUBR را برای X۸۶ پیاده سازی می‌کنید. سپس برای تست دستور پیاده سازی شده، یک برنامه می‌نویسید که از این دستور خاص با استفاده از ویژگی Inline assembly استفاده کند. سپس برنامه را با استفاده از Gem۵ شبیه‌سازی کنید.

• توضیح مراحل

در دستورات X۸۶ معمولا هر دستور به عنوان ترکیبی از بخش‌های کوچکتر پیاده‌سازی می‌شود. به‌طور کلی هر دستور به عنوان macro-op و بخش‌های کوچک‌تر به عنوان micro-op شناخته می‌شود. برای پیاده ساری یک دستورالعمل در Gem۵ ابتدا اطلاعات مربوط به macro-op به دیکودر ISA ارائه می‌شود. سپس macro-op به صورت تعدادی micro-op پیاده‌سازی می‌شود و در نهایت micro-op هایی که قبلا پیاده‌سازی نشده اند، پیاده‌سازی می‌شوند. این مراحل برای پیاده‌سازی دستور FSUBR مورد استفاده قرار می‌گیرد و مشابه پیاده‌سازی FSUB خواهد بود که از قبل در Gem۵ موجود است.

در ISA X۸۶ دستورات به چندین روش مختلف کدگذاری می‌شوند. که در این تمرین تمرکز بر روی زیر مجموعه X۸۷ است. (برای مصالعه بیشتر، درمورد کدگذاری دستورات می‌توانید به فایل راهنمای ارائه شده توسط AMD مراجعه کنید.)

ابتدا با مراجعه به فایل src/arch/x86/isa/decoder/one_byte_opcodes.isa می‌توان بررسی کرد که در Gem۵ دیکود دستورات ISA X۸۶ به چه نحو انجام می‌شود. در این فایل دستورات با ساختار مشخصی تعریف شده اند و محتوای آن درنهایت به یک Switch Case در زبان C++ تبدیل می‌شود. برای این کار ابتدا ۵ بیت اول بایت مربوط به opcode دیکود می‌شود که امکان ایجاد ۳۲ حالت مختلف را دارد که به ترتیب مشخص شده است. تمام دستورات X۸۷ با یک بایت opcode در محدوده 0xD8 تا 0xDF شروع می‌شوند. بنابر این ۵ بیت اول دستورات X۸۷ همیشه 0x1B هستند. همانطور که قابل مشاهده است برای این حالت، فایل دیگری با آدرس src/arch/x86/isa/decoder/x87.isa به فایل حاضر include شده است. که با مراجعه به آن شروع به دیکود سه بیت باقی مانده از بایت مربوط به opcode می‌کنیم. می‌توانید به جدول A-۱۵ (صفحه ۴۴۳) فایل راهنما برای بررسی دستورات مشخص شده توسط مقادیر مختلف سه بیت مذکور مراجعه کنید. به عنوان مثال دستورات FSUB و FSUBR با رمزهای 0xD0 و 0xDC مشخص می‌شوند. برای تشخیص تفاوت عملکرد opcode های ارائه شده، برای یک دستور مشابه، شما باید مفهوم فیلد ModRM مربوط به دستورات را بدانید. (برای این کار می‌توانید به فایل راهنما) مراجعه کنید. در فایل x87.isa می‌توانید بررسی کنید که ما دستور FSUB را برای مقادیر 0x0 و 0x4 داریم. همچنین مشاهده می‌شود که پیادهسازی دستور FSUBR حذف شده است.

در گام اول، تفاوت بین دو پیاده سازی دستور FSUBR را بررسی کنید: یکی با بایت opcode برابر با D۸h و دیگری برابر با DCh (توضیحات مربوط به دستور FSUBR را در فایل راهنما دستورات X۸۷ مطالعه کنید.) سپس در سه مکان مختلف در فایل x87.isa جملات مرتبط با دستور FSUBR را جست و جو کنید کنیپ و با جملاتی مشابه جملات مشخص شده برای دستور FSUB جایگذین کنید. با استفاده از عبارتی مانند Inst::FSUB درخواست می‌کنید که از این دستور به جای دستور پیش‌فرض که فقط یک هشدار عدم پیاده‌سازی چاپ می‌کند، استفاده شود.

در گام دوم باید پیاده‌سازی macro-op برای FSUBR به صورت micro-op انجام شود. مجددا می‌توان از پیادهسازی دستور FSUB الگوبردازی کرد. به دایرکتوری src/arch/x86/isa/insts/x87/arithmetic/ مراجعه کنید. این دایرکتوری شامل تعریف دستورات مختلف حسابی x87 به صورت micro-op ها است. نگاهی به نحوه پیاده سازی دستور FSUB با استفاده از micro-op ها بندازید، FSUB1 و FSUB2 به دو opcode مختلف اشاره دارند. برای هر نوع باید سه پیاده سازی مختلف فراهم شود. یکی که فقط از ثبات ها استفاده می‌کند و دیگری یکی از عملوندها را با استفاده از آدرس موجود در دستور از حافظه می‌خواند و آخرین مورد که از آدرس اشاره‌گر دستور برای خواندن عملوند استفاده می‌کند. micro-op های مورد استفاده برای این سه پیاده‌سازی باید به سادگی قابل درک باشد. روشی که پارسر دستور در Gem۵ کار می‌کند، باعث الزام تعریف هر سه پیاده سازی برای دستور FSUBR می‌شود. در کل، شما باید شش بلوک کد جداگانه برای FSUBR داشته باشید. مانند آنچه برای FSUB مشخص شده است.

سر انجام، باید یک پیاده سازی برای micro-op subfp ارائه شود. می‌توانید بررسی کنید که این پیاده‌سازی درحال حاضر،

در فایل `src/arch/x86/isa/micro-ops/fpop.isa` موجود است. بنابراین، برای این مرحله نیاز به انجام کاری نیست.

Gem۵ را برای معماری x86 کامپایل کنید تا اطمینان حاصل شود که در پیاده‌سازی هیچ اشتباهی رخ نداده است.

درنهایت لازم است پیاده‌سازی دستور FSUB تست شود. برای این کار، یک برنامه C می‌نویسید که یک فایل با دو عدد اعشاری می‌خواند، آنها را از یکدیگر کم می‌کند و خروجی را چاپ می‌کند. به منظور اطمینان از استفاده از دستور FSUBR برای تفریق، از ویژگی `Inline assembly` کامپایلر GCC استفاده کرده و به طور مشخص از دستور FSUBR در کد استفاده می‌کنید.

مواردی که لازم است در فایل پاسخ تمرین موجود باشد:

۱. فایل C استفاده شده به منظور تست دستور FSUBR

۲. فایل Patch شامل تغییرات اعمال شده به `src/arch/x86/isa/insts/x87/arithmetic/` و `src/arch/x86/isa/decoder/x87.isa`

۳. فایل Patch را می‌توان با استفاده از دستور زیر ایجاد کرد:

```
hg diff src/arch/x86/isa>/tmp/changes.path
```

۴. گزارشی از مراحل انجام کار و نتایج حاصل از شبیه‌سازی.

۱ پاسخ

در این قسمت به بررسی نتایج بدست آمده می‌پردازیم. همچنین شما می‌توانید کد نوشته شده در این بخش را در انتهای گزارش در «پیوست آ» مشاهده کنید.

به طور کلی، دستورالعمل FSUBR دو مقدار بالا را از استک x۸۷ می گیرد، مقدار دوم را از مقدار اول کم می کند و نتیجه را در رجیستر اول ذخیره می کند. تفاوت اصلی بین دستور FSUBR و همتای آن FSUB ترتیب تفریق است. FSUB مقدار دوم را از مقدار اول کم می کند و نتیجه را در رجیستر دوم ذخیره می کند، در حالی که FSUBR مقدار دوم را از مقدار اول کم می کند و نتیجه را در رجیستر اول ذخیره می کند.
برای تشخیص تفاوت عملکرد opcode های ارائه شده برای یک دستور مشابه به بررسی مفهوم ModRM می‌پردازیم. ModRM که مخفف Mode-Register/Operand است، یک انکودینگ بایتی است که در معماری in-strucrion های x۸۶ برای تعیین حالت آدرس دهی دستورالعمل‌های خاص استفاده می‌شود. بایت ModRM معمولا دومین بایت از یک instruction است و از بایت opcode پیروی می‌کند.
بایت ModRM به سه قسمت Mod و Reg/Opcode و R/M تقسیم می‌شود.

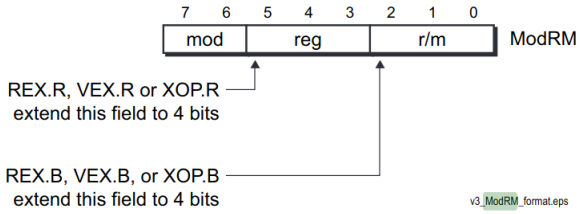
۱. **Mod ۲بیت:** فیلد Mod حالت آدرس دهی مورد استفاده توسط instruction را مشخص می کند.
تعیین می کند که چگونه آدرس موثر عملوند محاسبه می شود. Mod می تواند مقادیر زیر را بگیرد:
- ۰۰: حالت آدرس دهی غیر مستقیم (بدون جابجایی).
- ۰۱: حالت آدرس دهی غیر مستقیم با جابجایی ۸ بیتی.
- ۱۰: حالت آدرس دهی غیر مستقیم با جابجایی ۳۲ بیتی.
- ۱۱: حالت آدرس دهی بدون جابجایی.

۲. **Reg/Opcode ۳بیت:** فیلد Reg/Opcode اطلاعات بیشتری در مورد دستورالعمل ارائه می دهد. تفسیر آن به دستورالعمل خاص بستگی دارد و می تواند اهداف مختلفی را دنبال کند:
- در برخی دستورالعمل ها، یک عملوند رجیستر یا یک عملیات مبتنی بر رجیستر را مشخص می کند.
- در دستورالعمل‌های دیگر، پسوندهای op-code دستورالعمل‌های اضافی را در همان

فضای opcode رمزگذاری می‌کند.

۳. **R/M ۳بیت:** فیلد R/M ثبت عملوند یا عملوند حافظه مورد استفاده توسط دستورالعمل را مشخص می کند. تفسیر آن بسته به حالت آدرس دهی مشخص شده توسط فیلد Mod می تواند متفاوت باشد:

- در حالت آدرس دهی رجیستر، یعنی زمانی که Mod=۱۱ باشد، R/M عملوند رجیستر را رمزگزاری می‌کند.
- در حالت آدرس‌دهی حافظه یعنی زمانی که Mod=۰۰،۰۱،۱۰ باشد، R/M رجیستر پایه یا عملگر حافظه مورد استفاده برای حافظه آدرس را رمزگزاری می‌کند.



شکل ۱: فرمت بایت ModRM

حال پس از آشنایی با تفاوت عملکرد opcode ها، D۸h و DCh مبدا و مقصد عمگرهای تفریق و محل ذخیره را تعیین می‌کنند.
به مسیر `src/arch/x86/isa/decoder/x87.isa` می‌رویم و در فایل `x87.isa` جملات FSUBR را به `Inst::FSUBR1(Ed)` و `Inst::FSUBR2(Eq)` و `Inst::FSUBR2(Mq)` تغییر می‌دهیم.
سپس به مسیر `src/arch/x86/isa/insts/x87/arithmetic` می‌رویم و در فایل `subtraction.py` پیاده‌سازی macro-op ها را انجام می‌دهیم. این پیاده سازی ها باید متناسب و به تعداد توابع FSUB به کار رفته در این فایل انجام شود.
Macro-op های نوشته شده برای عملیات FSUBR به صورت زیر است:

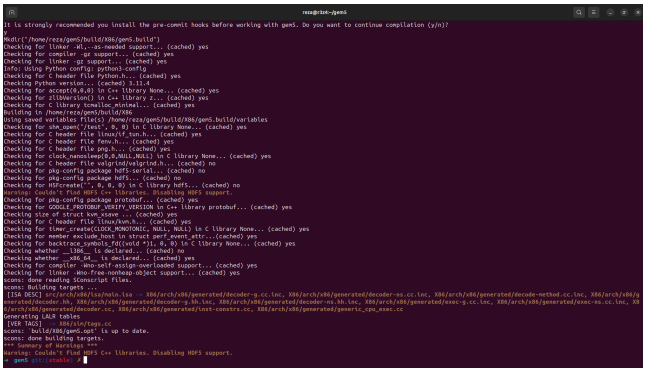
```
def macroop FSUBR1_R
{
    subfp st(0), sti, st(0)
};


def macroop FSUBR1_M
{
    ldfp ufp1, seg, sib, disp
    subfp st(0), ufp1, st(0)
};

def macroop FSUBR1_P
{
    rdip t7
    ldfp ufp1, seg, riprel, disp
    subfp st(0), ufp1, st(0)
};

def macroop FSUBR2_R
{
    subfp sti, st(0), sti
};

def macroop FSUBR2_M
{
    ldfp ufp1, seg, sib, disp
    subfp st(0), ufp1, st(0)
};

def macroop FSUBR2_P
{
    rdip t7
    ldfp ufp1, seg, riprel, disp
    subfp st(0), ufp1, st(0)
};
```

در macro-op های بالا در خط subfp عملگر اول رجیستر محل ذخیره سازی خروجی است و دو عملگر بعدی رجیستر هایی هستند که قرار است با هم تفریق شوند.

برای پیاده‌سازی micro-op subfp نیازی نیست کاری انجام دهیم چرا که این پیاده سازی در فایل `fpop.isa` در مسیر `src/arch/x86/isa/micro-ops/` وجود دارد.

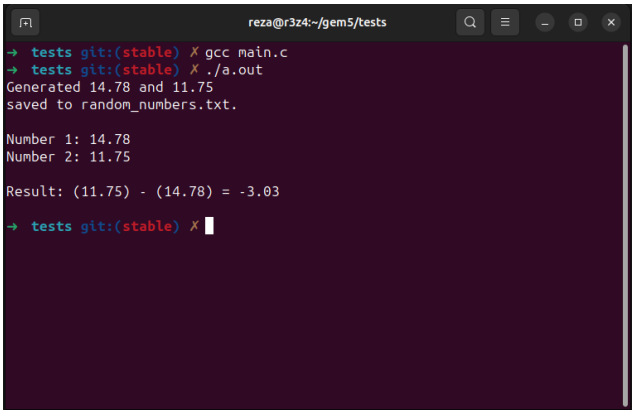
سپس با دستور `scons build/X86/gem5.opt -j 9` جم۵ را برای معماری x86 کامپایل می‌کنیم تا مطمئن شویم در پیاده سازی اشتباهی رخ نداده است.

با اجرای دستور گفته شده خروجی زیر حاصل می‌شود که نشان‌دهنده آن است در مراحل پیاده‌سازی دچار مشکل نشده ایم و پیاده‌سازی با موفقیت انجام شده است.



شکل ۲: کامپایل مجدد Gem۵

و در مرحله اخر برای تست دستور FSUBR برنامه ای به زبان C نوشتیم که از فایل `random_numbers.txt` دو عدد را بخواند و عملیات FSUBR را انجام دهد. دو عدد خوانده شده از فایل به صورت رندوم تولید می‌شود. کد نوشته شده در «**پیوست آ**» آورده شده است.
با کامپایلر gcc کد نوشته شده را build و اجرا می‌کنیم می‌کنیم:

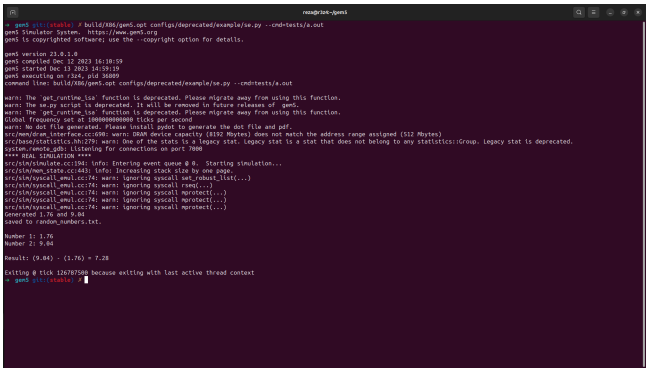


شکل ۳: کامپایل برنامه نوشته شده

سپس خروجی `a.out` تولید شده را با دستور زیر در محیط جم۵ اجرا می‌کنیم:

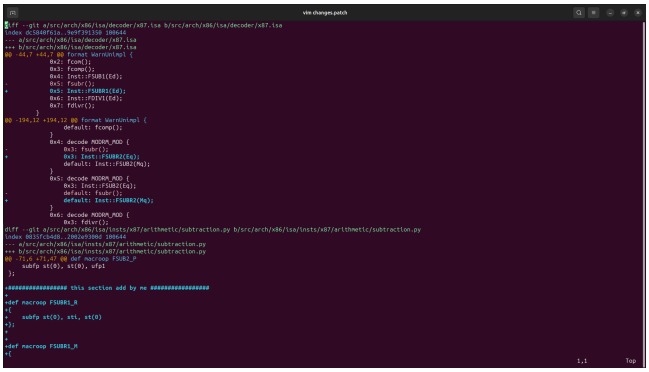
```
build/X86/gem5.opt configs/deprecated/
example/se.py --cmd=tests/a.out
```

خروجی شبیه سازی به درستی اجرا شده است و ورودی دوم را منهای ورودی اول کرده است:



شکل ۴: خروجی شبیه‌سازی

در نهایت با دستور `src/arch/x86/isa diff git > /tmp/changes.patch` از تغییراتی که در فایل ها انجام دادیم، یک log می‌گیریم.



شکل ۵: هیستوری تغییرات

پیوست آ

- کد C نوشته شده برای تست :FSUBR

Listing 1: C++ code for part1

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


typedef struct
{
    float num1;
    float num2;
} RandomNumbers;

// FSUBR function
float subtract(float in1, float in2)
{
    float ret = 0.0;
    asm ("fsubr %2, %0" : "=&t" (ret) : "%0" (in1), "u" (in2));
    return ret;
}

// Generate two random number
void random_gen(const char* filename)
{
    srand(time(NULL));

    // Generate two random floating-point numbers between -20 and 20
    float num1 = ((float)rand() / (float)RAND_MAX) * 40 - 20;
    float num2 = ((float)rand() / (float)RAND_MAX) * 40 - 20;

    FILE *file = fopen(filename, "w");

    if (file == NULL)
    {
        printf("Error opening the file.\n");
        return;
    }

    fprintf(file, "%.2f\n%.2f", num1, num2);

    fclose(file);

    printf("Generated %.2f and %.2f\n", num1, num2);
    printf("saved to %s.\n", filename);
}


// Read Number From File
RandomNumbers read_random_num(const char* filename)
```

```
{
    FILE *file = fopen(filename, "r");

    if (file == NULL)
    {
        printf("Error opening the file.\n");
        RandomNumbers emptyNumbers = {0.0, 0.0};
        return emptyNumbers;
    }

    RandomNumbers numbers;

    fscanf(file, "%f\n%f", &(numbers.num1), &(numbers.num2));

    fclose(file);

    return numbers;
}


int main()
{
    random_gen("random_numbers.txt");
    printf("\n");
    RandomNumbers result = read_random_num("random_numbers.txt");

    printf("Number 1: %.2f\n", result.num1);
    printf("Number 2: %.2f\n", result.num2);

    printf("\n");


    float fsubr_result = subtract(result.num1, result.num2);

    printf("Result: (%.2f) - (%.2f) = %.2f\n\n", result.num2, result.num1,
    fsubr_result);


    return 0;
}
```