گزارشِ تکلیفِ پنجمِ هم‌طرّاحی

روزبه صیّادی - امیرعلی منجر

۱ مقدمه

هدف از این پروژه آشنایی و پیاده‌سازی با یک معماری micro-programmed است. به این منظور ماشین پیاده‌سازی‌شده با استفاده از زبان GEZEL در بخش ۶,۵ از کتاب مرجع را مورد بررسی قرار داده‌ایم. ماشین مورد نظر دارای قسمت‌های مجزای controller و datapath است که در این تمرین تغییرات مختلفی را در آن ایجاد کرده‌ایم که در ادامه آن‌ها را توضیح داده و راه‌حل‌های پیشنهادی را ارائه خواهیم کرد.

۲ شرح تغییرات

۲,۱ اضافه‌کردن دستور به **ALU**

به ماژولِ ALU باید دو دستور را اضافه می‌کردیم.

- دستور ADDI مقدار موجود در بیت‌های ۲۷-۳۰ را با ACC جمع می‌کند.
- دستور SUBI مقدار موجود در بیت‌های ۲۷-۳۰ را از ACC کم می‌کند.

برای اضافه کردن این دستورها ابتدا در قسمت encodingهای مربوط به ALU دو مقدارِ جدید را برای دو دستورِ جدید اضافه کردیم.

```
#define ALU_ADDI    10          /* ALU <- ACC + [27:30] */
#define ALU_SUBI    11          /* ALU <- ACC - [27:30] */
```

در مرحله‌ی بعدی باید این دو دستور را پیاده‌سازی می‌کردیم. برای این کار بیت‌های ۲۷-۳۰ متعلق به instruction را به عنوان یک ورودی جدید به ALU اضافه کردیم و سپس از آن برای دستورهای جدید استفاده کردیم.

```
(ctl_alu == ALU_ADDI) ? acc_out + ctl_sbus :
(ctl_alu == ALU_SUBI) ? acc_out - ctl_sbus :
```

این دو خط به خط‌های قبلیِ این ماژول اضافه شده‌اند.

بدیهتاً خطّ مربوط به تعریف ALU در ماژولِ hmm را نیز تصحیح کردیم.

```
use alu(ctl_dest, ctl_alu, ctl_sbus,  sbus,   alu_in,  alu_out);
```

۲,۲ استخراج **ACC** از داخل **ALU**

واحدِ ACC تنها یک رجیستر ساده است. برای ساختن یک ماژول رجیستر از کد زیر استفاده کردیم؛

```
dp acc    (in input    :    ns(WLEN);
           out output    :    ns(WLEN)) {
  reg temp : ns(WLEN);
  always {
          output = temp;
          temp = input;
```

```
  }
}
```

این کد در هر کلاک مقدار داخلیش را روی خروجی گذاشته و مقدار ورودیش را ذخیره می‌کند.

بعد از تعریف این ماژول، استفاده از آن راحت بود. کافی بود دو سیم در ALU تعریف کنیم و آن‌ها را به ورودی و خروجی این ماژول متصل کنیم.

```
sig acc_in: ns(WLEN);
sig acc_out: ns(WLEN);
use acc(acc_in, acc_out);
```

سپس در هر کلاک مقدار ورودی ACC مشابه با روش قبلی تعیین می‌شود.

```
acc_in = (ctl_dest == DST_ACC) ? shift : acc_out;
```

۲,۳ استخراج مالتی‌پلکسر ورودی

با استخراج مالتی‌پلکسر، این ماژول از حالت asynchronous خارج می‌شود و با کلاک کار می‌کند. ابتدا ماژول را تعریف می‌کنیم؛

```
dp mux1x2(   in sel   :   ns(4);
             in in1   :   ns(WLEN);
             in in2   :   ns(WLEN);
             out q    :   ns(WLEN) ) {
  always {
      q = (sel == SBUS_IN) ? in1 : in2;
  }
}
```

سپس برای استفاده از این ماژول به hmm می‌رویم و این خط را به کدش اضافه می‌کنیم؛

```
use mux1x2(ctl_sbus, din, rf_out, sbus);
```

بدیهتاً باید از داخل بلاک always خطّ مربوط به مقداردهی به sbus را حذف کنیم.

۲,۴ پیاده‌سازی ضرب

برای قسمت آخر پروژه، پیاده‌سازی ضرب با کمک instructionهای موجود و به کمک عملیات جمع و شیفت انجام شده است.

۲,۴,۱ توضیح الگوریتم

این الگوریتم به این شکل است: فرض کنید می‌خواهیم دو عدد n و m را در هم ضرب کنیم. دو متغیّر کمکی ans و count را نیز داریم که در ابتدای کار مقدارشان صفر است. در یک حلقه چک می‌کنیم که آیا مقدار m برابر با صفر هست یا نه. اگر بود یعنی جواب در ans موجود است. در غیر این صورت وارد حلقه می‌شویم. در آن‌جا چک می‌کنیم که آیا m فرد هست یا نه. اگر فرد بود، مقدار n را به اندازه‌ی count به چپ شیفت می‌دهیم و مقدار حاصل را با ans جمع می‌کنیم. سپس در انتهای حلقه و خارج از شرط مقدار count را یکی زیاد کرده و m را تقسیم بر دو می‌کنیم. سپس به ابتدای حلقه برمی‌گردیم. کدِ این الگوریتم به زبانِ C++ به شرحِ زیر است:

```
int multiply(int n, int m)
{
```

```
    int ans = 0, count = 0;
    while (m)
    {
        // check for set bit and left
        // shift n, count times
        if (m % 2 == 1)
            ans += n << count;

        // increment of place value (count)
        count++;
        m /= 2;
    }
    return ans;
}
```

۲,۴,۲ پیاده‌سازی در **GEZEL**

الگوریتم، الگوریتمِ ساده‌ای بود و توانستیم به راحتی آن را به تعدادی micro-instruction بشکنیم و کدِ آن را بزنیم.
کد به همراهِ توضیحات:

```
// 0 get n and store it in r0
MI(O_NIL,   SBUS_IN,   ALU_PASS,   SHFT_NIL,   DST_R0,        NXT_NXT,   0),
// 1 get m and store it in r1
MI(O_NIL,   SBUS_IN,   ALU_PASS,   SHFT_NIL,   DST_R1,        NXT_NXT,   0),
// answer is in r2
// 2 check if m is zero
MI(O_NIL,   SBUS_R1,   ALU_PASS,   SHFT_NIL,   DST_X,         NXT_JZ,    9),
// 3 we store 1 in ACC for checking if a number is even or not
MI(O_NIL,   SBUS_X,          ALU_SET,   SHFT_NIL,   DST_ACC,   NXT_NXT,   0),
// 4 check if m is even
MI(O_NIL,   SBUS_R1,   ALU_AND,   SHFT_NIL,   DST_NIL,   NXT_JZ,          8),
// 5 Store n in ACC
MI(O_NIL,   SBUS_R0,   ALU_PASS,   SHFT_NIL,   DST_ACC,   NXT_NXT,   0),
// 6 ans += n
MI(O_NIL,   SBUS_R2,   ALU_ADD,   SHFT_NIL,   DST_R2,        NXT_NXT,   0),
// 7 shift n to left
MI(O_NIL,   SBUS_R0,   ALU_PASS,   SHFT_SHL,   DST_R0,       NXT_NXT,   0),
// 8 m = m / 2
MI(O_NIL,   SBUS_R1,   ALU_PASS,   SHFT_SRA,   DST_R1,       NXT_JMP,   3),
// 9 done!
MI(O_WR,   SBUS_R2,   ALU_X,      SHFT_X,      DST_X,        NXT_JMP,   0)
```

۳ تست

برنامه با موفّقیّت اجرا می‌شود. اشکالی که دارد این است که هیچ‌گاه از ورودیِ اوّل عبور نمی‌کند (در واقع مقدار din_strb هیچ‌گاه یک نمی‌شود). بنابراین نتوانستیم صحّتِ خروجیِ برنامه را تست کنیم. با این حال برنامه بدونِ هیچ اشکالی کامپایل می‌شود.

۴ منابع

1. https://www.geeksforgeeks.org/multiplication-two-numbers-shift-operator/

۴ نتیجه‌گیری

ما در این پروژه با نحوه‌ی توصیف یک سخت‌افزار ساده با وسیله‌ی زبان GEZEL آشنا شدیم و توانستیم یک برنامه‌ی ضرب را با کمکِ این سخت‌افزار و instructionهای تعریف شده برای آن پیاده‌سازی کنیم.