



تمرین چهارم هم طراحی سخت افزار - نرم افزار

نیلوفر مرادی جم 97243063

کیمیا صدیقی 97243046

مقدمه

در این تمرین قصد داریم الگوریتم محاسبه FFT را به صورت هم طراحی سخت افزار و نرم افزار طراحی کنیم. برنامه ما از چهار قسمت اصلی تشکیل شده است، point1 که برای محاسبه تعداد نقاط می باشد، point2 که به منظور انجام bit reversal می باشد، point3 که برای محاسبه FFT می باشد و در نهایت point4 که وظیفه scale کردن را برای forward transform بر عهده دارد.

توجه شود که برای تقسیم بندی کد به دو قسمت افزار و نرم افزار قسمت های کار با آرایه و قسمت محاسبه sqrt به منظور انعطاف پذیری و راحتی بیشتر در نرم افزار پیاده سازی شده اند و سایر قسمت ها در سخت افزار پیاده سازی شده اند.

ارتباط بین سخت افزار و نرم افزار

تعدادی از متغیرها هستند که باید بین سخت افزار و نرم افزار مبادله شوند. برای این منظور از تعدادی ipblock و به صورت in یا out با توجه به سیگنال مورد نظر در سخت افزار، تعریف می شوند. در نرم افزار نیز با همان اسم و آدرس یک پوینتر به آن تعریف می شود.

نمونه از ipblock های تعریف شده در کد سخت افزار به صورت زیر است:

```
ipblock my_arm {
    iptype "armsystem";
    ipparm "exec=fft";
}

ipblock n_out(out data : ns(64)) {
    iptype "armsystemsource";
    ipparm "core=my_arm";
    ipparm "address=0x80000000";
}

ipblock i_out(out data : ns(64)) {
    iptype "armsystemsource";
    ipparm "core=my_arm";
    ipparm "address=0x80000008";
}

ipblock j_out(out data : ns(64)) {
    iptype "armsystemsource";
    ipparm "core=my_arm";
    ipparm "address=0x80000010";
}
```

پوینترهای مربوطه در نرم افزار نیز مطابق تصویر زیر است:

```
volatile long *n_out = (long *) 0x80000000;  
volatile long *i_out = (long *) 0x80000008;  
volatile long *j_out = (long *) 0x80000010;  
volatile long *l1_out = (long *) 0x80000018;  
volatile double *c1_out = (double *) 0x80000020;  
volatile double *c1_in = (double *) 0x80000028;  
volatile double *c2_out = (double *) 0x80000030;  
volatile double *c2_in = (double *) 0x80000038;  
volatile double *u1_out = (double *) 0x80000040;  
volatile double *u2_out = (double *) 0x80000048;  
volatile int *swap_req = (int *) 0x80000050;  
volatile int *swap_ack = (int *) 0x80000054;  
volatile int *c2_req = (int *) 0x80000058;  
volatile int *c2_ack = (int *) 0x8000005c;  
volatile int *c1_req = (int *) 0x80000060;  
volatile int *c1_ack = (int *) 0x80000064;  
volatile int *in_loop_req = (int *) 0x80000068;  
volatile int *in_loop_ack = (int *) 0x8000006c;  
volatile int *scale_req = (int *) 0x80000070;  
volatile int *scale_ack = (int *) 0x80000074;  
volatile int *in_ready = (int *) 0x80000078;  
volatile int *done_out = (int *) 0x8000007c;
```

توجه شود که برای انجام یک عملیات مشخص در نرم افزار، برای آن یک سیگنال req و ack در نظر گرفته شده است که جزو منابع مشترک سخت افزار و نرم افزار هستند. اگر مقدار req آن عملیات یک باشد، یعنی از سمت سخت افزار درخواست برای اجرای این عملیات در نرم افزار وجود دارد. پس در قسمت نرم افزار یک حلقه while تعریف می کنیم که مقادیر سیگنال req عملیات های متفاوت را به طور مدام چک می کند، اگر این مقدار یک باشد، آن عملیات را اجرا می کند و در پایان اجرا نیز مقدار سیگنال ack آن عملیات را برابر یک قرار می دهد. در قسمت سخت افزار نیز مقدار ack آن چک می شود و در صورتی که این مقدار یک باشد به اجرای ادامه برنامه ادامه خواهد داد، در غیر این صورت منتظر میماند تا ack یک شود و مطمئن شود که اجرای آن عملیات در نرم افزار پایان یافته است. همچنین هنگامی که FSM به state پایانی خود میرسد مقدار یک سیگنال مشخص و مشترک بین سخت افزار و نرم افزار را برابر یک می کنیم، پس در حلقه while چک میکنیم، اگر این مقدار برابر یک باشد از حلقه while خارج شده و اجرای برنامه را تمام می کنیم. قسمتی از حلقه while به صورت زیر است:

```

while(true){

    if (*done_out) { // exit loop when hardware informs us that the computing is done
        break;
    }

    if (*swap_req) { // do the swapping every time the hardware informs us with making swap req 1
        *swap_ack = 0;
        tx = bufr[*i_out];
        ty = bufi[*i_out];
        bufr[*i_out] = bufr[*j_out];
        bufi[*i_out] = bufi[*j_out];
        bufr[*j_out] = tx;
        bufi[*j_out] = ty;
        *swap_ack = 1; // informing the hardware that swapping is done
        *swap_req = 0; // swap is done so no req untill hardware makes it 1 again
    }

    /***** the same logic of swap is used in other if blocks too *****/

    if (*c2_req) {
        *c2_ack = 0;
        *c2_in = sqrt((1.0 - *c1_out) / 2.0);
        *c2_ack = 1;
        *c2_req = 0;
    }
}

```

کد سخت افزاری

پس در قسمت سخت افزار یک FSM تعریف کردیم که برای هر point شامل یک state است و هر point نیز خود می تواند از state های کوچکتری تشکیل شده باشد.

بدین منظور هر بخش از FSM را توضیح میدهم.

در FSM ابتدا با init به point1 رفتیم. init همانطور که در کد اورجینال مشخص شده، بخش مقداردهی های اولیه میباشد.

```

// init
int m=3;
int dir=1;
long n,i,i1,j,k,i2,l,l1,l2;
double c1,c2,tx,ty,t1,t2,u1,u2,z;

```

```

sfg init {
    m = 3;
    dir = 1;

    n = 1;
    i = 0;
}

```

همانطور که مشاهده می شود قسمت های مورد نیاز سخت افزار در قسمت init مقدار گرفته. سپس در point1 طبق برنامه مورد نظر باید تعداد نقطه ها را محاسبه میکنیم که برابر m^2 میباشد. برای طراحی حلقه در fsm یک لوپ ایجاد میکنیم که در هر بار اجرای لوپ وارد cal_points میشود و n را ضربدر 2 کرده و یک واحد به آن که شمارنده حلقه در fsm است اضافه میکند.

```
// point 1
/* Calculate the number of points */
n = 1;
for (i=0;i<m;i++)
    n *= 2;
```

```
// calculate number of points --> 2^m
sfg cal_points {
    n = n * 2;
    i = i + 1;
}
```

پس از پایان حلقه در fsm وارد point2 میشویم. در اینجا ابتدا طبق برنامه اولیه، یک شیفت انجام می دهیم و به دنبال آن وارد حلقه می شویم.

```
// point 2
/* Do the bit reversal */
i2 = n >> 1;
j = 0;
for (i=0;i<n-1;i++) {...
```

```
// shift i2 and init i and j
sfg shift {
    i2 = n >> 1;
    j = 0;
    i = 0;
}
```

حلقه های تو در تو و شرط موجود در برنامه با fsm هندل میشود. مسئله مورد توجه اینجا این است که بخش swap موجود در کد اصلی در بخش نرم افزار و نه سخت افزار هندل شده است و در اینجا سخت افزار تنها برای مطلع کردن نرم افزار برای انجام عملیات، بیت req مربوط به آن را برابر 1 میکند. این مسئله در قسمت ارتباط بین نرم افزار و سخت افزار به صورت کامل تر توضیح داده شده است.

```
// we inform software that it's swap time with making swap reg 1
sfg swap {
    swap_req_sig_ = 1;
}
```

در پایان حلقه موجود در point2 مقدار k را به روز رسانی میکنیم. (طبق کد اصلی)

```
// update j with k and shift k
sfg update_j_k {
    j = j - k;
    k = k >> 1;
}
```

در ادامه وارد point3 میشویم. در ابتدای آن مقدار coefficients مشخص میشود. (طبق کد اصلی) و سپس وارد حلقه for میشویم که در fsm هندل شده است.

```
// point 3
/* Compute the FFT */
c1 = -1.0;
c2 = 0.0;
l2 = 1;
> for (l=0;l<m;l++) { ...
```

```
// init values of c1 and c2 like the main program
sfg init_coeffs {
    c1_outp = -1;
    c2_outp = 0;
    l2 = 1;
    l = 0;
}
```

محتوای درون حلقه for در نرم افزار هندل میشود و ما در قسمت سخت افزاری تنها با بیت های req و ack مربوطه که توضیح داده شد، از موقعیت نرم افزار مطلع میشویم یا آن را به روز رسانی میکنیم.

```
// evaluate i for the loop
sfg evaluate_i {
    i = j;
}

// inform software to perform c2 update with making c2 req 1
sfg update_c2 {
    c2_req_sig = 1;
}

// inverse value of c2 by multiplying it to -1
sfg inverse_c2 {
    c2_outp = -1 * c2_outp;
}

// inform software to perform c1 update with making c1 req 1
sfg update_c1 {
    c2_outp = c1_inp;
    c1_req_sig = 1;
    l = l+1;
}

sfg update_c1_to_input {
    c1_outp = c1_inp;
}

// inform software to perform in loop (line 57 to 63 of main program)
sfg in_loop {
    in_loop_req_sig = 1;
}
```

در پایان پس از point3 وارد point 4 میشویم که با توجه به dir عمل میکند.
شرط ورود به if در fsm بررسی شده و عملیات مربوطه در نرم افزار انجام میشود و تنها در بخش سخت افزاری
ما با بیت های req و ack مربوطه زمان اجرا را مدیریت میکنیم.

```
//point 4
/* Scaling for forward transform */
if (dir == 1) {
```

```
// init i for the loop
sfg i_0 {
    i = 0;
}

// inform software to do the scaling by making scale req 1
sfg scale {
    scale_req_sig = 1;
    i = i + 1;
}

// update value of z and u according to main program
sfg update_z_u {
    z = u1_outp * c1_outp - u2_outp * c2_outp;
    u2_outp = u1_outp * c2_outp + u2_outp * c1_outp;
    u1_outp = z;
}
```

پس از تمامی این محاسبات با done وارد استیت end_point میشویم. که در done بیت done_sig
یک میشود و به نرم افزار اطلاع میدهد که محاسبات پایان یافته.

```
// set done sig 1 in order to inform software that the program is done
sfg done {
    done_sig = 1;
}
```

همانطور که در بخش ارتباط نرم افزار و سخت افزار گفته شد برای ارتباط این دو ما از ipblock هایی با آدرس
یکسان استفاده کرده ایم. نکته مهم استفاده از این ipblock ها در کد سخت افزار میباشد.
بدین منظور ما سیگنال هایی تعریف کردیم و این سیگنال ها را در ipblock مورد نظر use میکنیم.

```

//signals don't have _ in the end
sig i, j, l1: ns(64);
sig n : ns(64);
sig c1_outp, c1_inp, c2_outp, c2_inp, u1_outp, u2_outp : tc(64);
sig input_ready : ns(1);
sig swap_req_sig, swap_ack_sig, c2_req_sig, c2_ack_sig, c1_req_sig, c1_ack_sig, in_loo
    scale_req_sig, scale_ack_sig : ns(32);
sig done_sig : ns(1);

//use signals (connect to software)
use n_out(n);
use i_out(i);
use j_out(j);
use l1_out(l1);

use c1_out(c1_outp);
use c1_in(c1_inp);
use c2_out(c2_outp);
use c2_in(c2_inp);
use u1_out(u1_outp);
use u2_out(u2_outp);
use in_ready(input_ready);

```

مقدار سیگنال ها در طول برنامه update میشود و در یک بلاک always در رجیستر معادلش ذخیره میشود.
از مقدار این رجیسترها در fsm استفاده میکنیم.

```

//all registers have an _ in the end
reg i_, j_, l1_: ns(64);
reg n_ : ns(64);
reg c1_outp_, c1_inp_, c2_outp_, c2_inp_, u1_outp_, u2_outp_ : tc(64);
reg input_ready_ : ns(1);
reg swap_req_sig_, swap_ack_sig_, c2_req_sig_, c2_ack_sig_, c1_req_sig_, c1_ack_sig_, in_loo
    scale_req_sig_, scale_ack_sig_ : ns(32);
reg done_sig_ : ns(1);

//evaluate registers with signals
always {
    i_ = i;
    j_ = j;
    l1_ = l1;
    c1_outp_ = c1_outp;
    c1_inp_ = c1_inp;
    c2_outp_ = c2_outp;
    c2_inp_ = c2_inp;
    u1_outp_ = u1_outp;
    u2_outp_ = u2_outp;
    input_ready_ = input_ready;
    swap_req_sig_ = swap_req_sig;
    swap_ack_sig_ = swap_ack_sig;

```


مشاهده fsm در حالت کلی:

```
fsm
@s0 if (input_ready_) then (init) -> point1;
    else (skip) -> s0;

@point1 if (i_ < n) then (cal_points) -> point1;
    else (skip) -> point2_1;

@point2_1 (shift) -> point2_2;

@point2_2 if (i_ < n-1) then (skip) -> point2_3;
    else (skip) -> point3_1;

@point2_3 if (i_ < j_) then (swap) -> point2_4;
    else (skip) -> point2_4;

@point2_4 if (swap_ack_sig_) then (update_k) -> point2_5;
    else (skip) -> point2_4;

@point2_5 if (k <= j_) then (update_j_k) -> point2_5;
    else (update_j) -> point2_2;

@point3_1 (init_coeffs) -> point3_2;

@point3_2 if (l < m) then (update_l_u) -> point3_3;
    else (skip) -> point4_1;

@point3_3 if (j_ < l1_) then (evaluate_i) -> point3_4;
    else (update_c2) -> point3_5;

@point3_5 if (dir & c2_ack_sig_) then (inverse_c2) -> point3_6;
    else if (c2_ack_sig_) then (update_c1) -> point3_2;
    else (skip) -> point3_5;

@point3_6 (update_c1) -> wait_c1_ack;

@wait_c1_ack if (c1_ack_sig_) then (skip) -> update_c1_to_c1_inp;
    else (skip) -> wait_c1_ack;

@update_c1_to_c1_inp (update_c1_to_input) -> point3_2;

@point3_4 if (i_ < n_) then (in_loop) -> wait_loop_ack;
    else (update_z_u) -> point3_3;

@wait_loop_ack if (in_loop_ack_sig_) then (skip) -> point3_4;
    else (skip) -> wait_loop_ack;

@point4_1 if (dir == 1) then (i_0) -> point4_2;
    else (skip) -> end_point;

@point4_2 if (i_ < n_) then (scale) -> wait_scale_ack;
    else (skip) -> end_point;

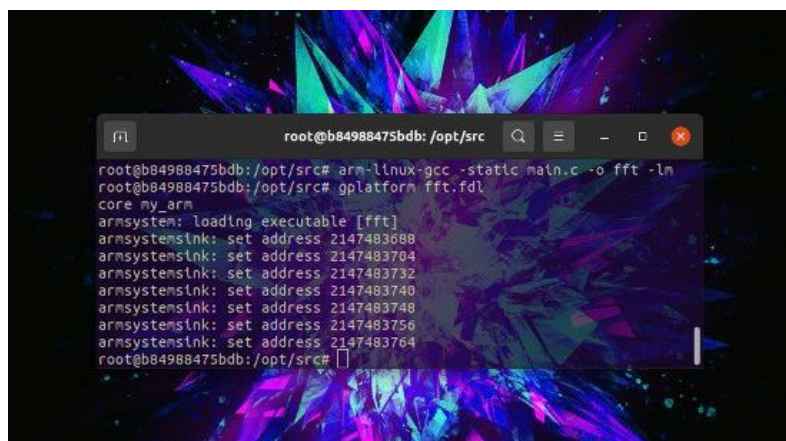
@wait_scale_ack if (scale_ack_sig_) then (skip) -> point4_2;
    else (skip) -> wait_scale_ack;

@end_point (done) -> end_point;
```

کد نرم افزاری

همانطور که توضیح داده شد، در این بخش تنها مقاردهی دو آرایه ورودی و چند عملیات خاص مثل جا به جایی خانه های آرایه و یا ریشه دوم گرفتن یک مقدار و `print` کردن خروجی ها انجام می شود. عملکرد حلقه `while` آن نیز در قسمت ارتباط بین سخت افزار و نرم افزار گفته شد.

خروجی:



```
root@b84988475bdb: /opt/src
root@b84988475bdb: /opt/src# arm-linux-gcc -static main.c -o fft -ln
root@b84988475bdb: /opt/src# gplatform fft.fdl
core my_arm
armsystem: loading executable [fft]
armsystemsink: set address 2147483688
armsystemsink: set address 2147483704
armsystemsink: set address 2147483732
armsystemsink: set address 2147483740
armsystemsink: set address 2147483748
armsystemsink: set address 2147483756
armsystemsink: set address 2147483764
root@b84988475bdb: /opt/src#
```