.

به نام خدا

آزمون میانترم درس سیستم‌های دیجیتال کم‌توان

استاد : دکتر اجلالی

مهدی بهرامیان ۴۰۱۱۷۱۵۹۳

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

بهار ۱۴۰۴

سوال ۴ - مقایسه جمع کننده‌های MCC و RCA

برای این سوال ما ابتدا هرکدام از این مدارها را به زبان وریلاگ پیاده سازی میکنیم :

```verilog
module MCCAdder (
    input  [15:0] a,
    input  [15:0] b,
    output [16:0] s
);
    wire [3:-1] C;
    assign C[-1] = 0;
    genvar i;
    generate
        for (i = 0; i < 4; i++) begin : gen_mcc
            MCC4 _mcc (
                .a(a[i*4+:4]),
                .b(b[i*4+:4]),
                .cin(C[i-1]),
                .cout(C[i]),
                .s(s[i*4+:4])
            );
        end
    endgenerate
    assign s[16] = C[3];
endmodule

module MCC4 (
    input [3:0] a,
    input [3:0] b,
    input cin,
    output [3:0] s,
    output cout
);
    wire [3:-1] C;
    assign C[-1] = ~cin;
    genvar i;
    generate
        for (i = 0; i < 4; i += 1) begin : gen_mcc
            MCC1 _1 (
                .a(a[i]),
                .b(b[i]),
                ._cin(C[i-1]),
                ._cout(C[i]),
                .s(s[i])
            );
        end
    endgenerate
```

```
    assign cout = ~C[3];
endmodule

module MCC1 (
    input  a,
    input  b,
    input  _cin,
    output s,
    output _cout
);
    wire P;
    wire G;
    xor (P, a, b);
    and (G, a, b);
    wire _wk1;
    wire _st0;
    assign (pull1, weak0) _wk1 = 1;
    // assign _wk1 = 1;
    assign _st0 = 0;
    pmos _p0 (_cout, _wk1, G);
    wire _inter;
    nmos _n0 (_cout, _inter, G);
    nmos _n1 (_inter, _st0, ~P);
    nmos _n2 (_cout, _cin, P);
    xnor (s, P, _cin);
endmodule
```

نحوه پیاده سازی واحد جمع کننده MCC به این صورت است که ابتدا ماژول ۱ بیتی آن را میسازیم که ورودی های آن a , b و $\overline{cin}$ خروجی های آن s , $\overline{cout}$ است. معماری این MCC بر مبنی مدار بررسی شده در این فیلم است. سپس با استفاده از این ماژول، MCC ۴ بیتی را میسازیم که درواقع ۴ تا از این MCC های یک بیتی را به طور متوالی متصل کرده و زنجیره نقلی را حاتمه میدهیم. قابل توجه است که تعداد ۴ طبقه برای MCC در تحلیل ساده بهینه است. در نهایت با استفاده از ماژول های ۴ بیتی، جمع کننده ۱۶ بیتی خود را میسازیم.

برای مشابهت میان ساختار مدار ها، RCA خود را نیز با ساختار مشابه طراحی میکنیم :

```verilog
module NormalAdder (
    input  [15:0] a,
    input  [15:0] b,
    output [16:0] s
);
    wire [3:-1] C;
    assign C[-1] = 0;
    genvar i;
    generate
        for (i = 0; i < 4; i++) begin : gen_fa
            FA4 _fa (
```

```
                .a(a[i*4+:4]),
                .b(b[i*4+:4]),
                .cin(C[i-1]),
                .cout(C[i]),
                .s(s[i*4+:4])
            );
        end
    endgenerate
    assign s[16] = C[3];
endmodule

module FA4 (
    input [3:0] a,
    input [3:0] b,
    input cin,
    output [3:0] s,
    output cout
);
    wire [2:0] cinter;

    FullAdder _0 (
        .a(a[0]),
        .b(b[0]),
        .c(cin),
        .cout(cinter[0]),
        .s(s[0])
    );

    FullAdder _1 (
        .a(a[1]),
        .b(b[1]),
        .c(cinter[0]),
        .cout(cinter[1]),
        .s(s[1])
    );

    FullAdder _2 (
        .a(a[2]),
        .b(b[2]),
        .c(cinter[1]),
        .cout(cinter[2]),
        .s(s[2])
    );

    FullAdder _3 (
        .a(a[3]),
```

```
58          .b(b[3]),
59          .c(cinter[2]),
60          .cout(cout),
61          .s(s[3])
62      );
63  endmodule
64
65  module FullAdder (
66      input  a,
67      input  b,
68      input  c,
69      output s,
70      output cout
71  );
72      assign {cout, s} = a + b + c;
73  endmodule
```

توجه کنید که بعد از مرحله طراحی دیجیتال/سطح سوییچ در وریلاگ این مدارات توسط تست بنچ های جامعی آزموده شدند تا از صحت کارکرد مدار ها آسوده خاطر باشیم و در هنگام شبیهسازی آنالوگ دچار مشکل نشویم.

همینطور برای ارزیابی درست این مدارات به یک تولید کننده ورودی تصادفی نیازمندیم که به مداراتمان ورودی بدهد. برای اینکار مدار زیر طراحی شده است :

verilog
```
1   module RNG (
2       input clk,
3       input rst,
4       input [7:0] seed,
5       output reg [7:0] rnd
6   );
7       always @(posedge clk) begin
8           if (rst) rnd <= seed;
9           else begin
10              repeat (3) begin
11                  {rnd[7:1], rnd[0]} = {rnd[6:0], rnd[3] ^ rnd[4] ^ rnd[5] ^
                    rnd[7]};
12              end
13          end
14      end
15  endmodule
16
17  module INPGEN (
18      input clk,
19      input rst,
20      output [15:0] a,
21      output [15:0] b
22  );
23      RNG _0 (
```

```
        .clk (clk),
        .rst (rst),
        .seed(8'b01010101),
        .rnd (a[7:0])
    );

    RNG _1 (
        .clk (clk),
        .rst (rst),
        .seed(8'b01011010),
        .rnd (b[7:0])
    );

    RNG _2 (
        .clk (clk),
        .rst (rst),
        .seed(8'b01101010),
        .rnd (a[15:8])
    );

    RNG _3 (
        .clk (clk),
        .rst (rst),
        .seed(8'b10101011),
        .rnd (b[15:8])
    );

endmodule
```

حال باید همه این مدارات را تجمیع کنیم و با استفاده از ابزار سنتز yosys آنهارا به نتلیست اسپایس تبدیل کنیم و در نهایت با استفاده از ngspice آنهارا شبیهسازی کنیم و وضعیت انرژی مصرفی را از آن استخراج کنیم.

برای اینکار از اسکریپت yosys زیر استفاده میکنیم :

yosys

```
read_verilog -sv digital/RNG.v
read_verilog -sv digital/MCCAdder.v
read_verilog -sv digital/MCCBench.v
read_verilog -sv digital/NormalAdder.v
read_verilog -sv digital/NormalBench.v


read_verilog -lib yosys/prim_cells.v

proc;; memory;; techmap;;

dfflibmap -liberty yosys/prim_cells.lib
abc -liberty yosys/prim_cells.lib;;
```

```
14
15 write_spice bench.mod
```

این اسکریپت صرفا کد های توضیح داده شده را بارگیری میکند و پس از سنتز به عنوان نتلیست spice با توجه به کتابخانه prim_cells.lib خروجی میدهد.

برای شبیه‌سازی ngspice نیز از تست بنچ زیر استفاده میکنیم :

spice
```
1  .title "MCC Adder CIRC"
2
3  * supply voltages
4  .global Vss Vdd
5  Vgnd Vgnd 0 DC 0
6  Vpwr Vpwr 0 DC 3
7  * Vss Vss Vpwr DC 0
8  * Vdd Vdd 0 DC 3
9  Rgnd Vgnd Vss 0.001
10 Rpwr Vpwr Vmeasin 0.001
11 Vmeasin Vmeasin Vdd DC 0
12
13 * simple transistor model
14 .MODEL cmosn NMOS LEVEL=1 VT0=0.7 KP=110U GAMMA=0.4 LAMBDA=0.04 PHI=0.7
15 .MODEL cmosp PMOS LEVEL=1 VT0=-0.7 KP=50U GAMMA=0.57 LAMBDA=0.05 PHI=0.8
16
17 * load design and library
18 .include yosys/prim_cells_cmos.mod
19 .include bench.mod
20
21 * input signals
22 Vclk clk 0 PULSE(0 3 1 0.1 0.1 0.8 2)
23 Vrst rst 0 PULSE(0 3 0.5 0.1 0.1 2.9 40)
24
25 Xadder clk rst x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16
   MCCBench
26
27 .tran 0.1 40
28 .save i(Vmeasin) v(Vmeasin)
29 * .save v(Vmeasin)
30 * .probe p(counter)
31 * .save counter:power
32 .control
33 set filetype=ascii
34 * run
35 * plot v(clk) v(x0)/4+10 v(x1)/4+11 v(x2)/4+12 v(x3)/4+13 v(x4)/4+14 v(x5)/4+15
   v(x6)/4+16 v(x7)/4+17 i(Vmeasin)+20
36 * save i(Vmeasin)
```

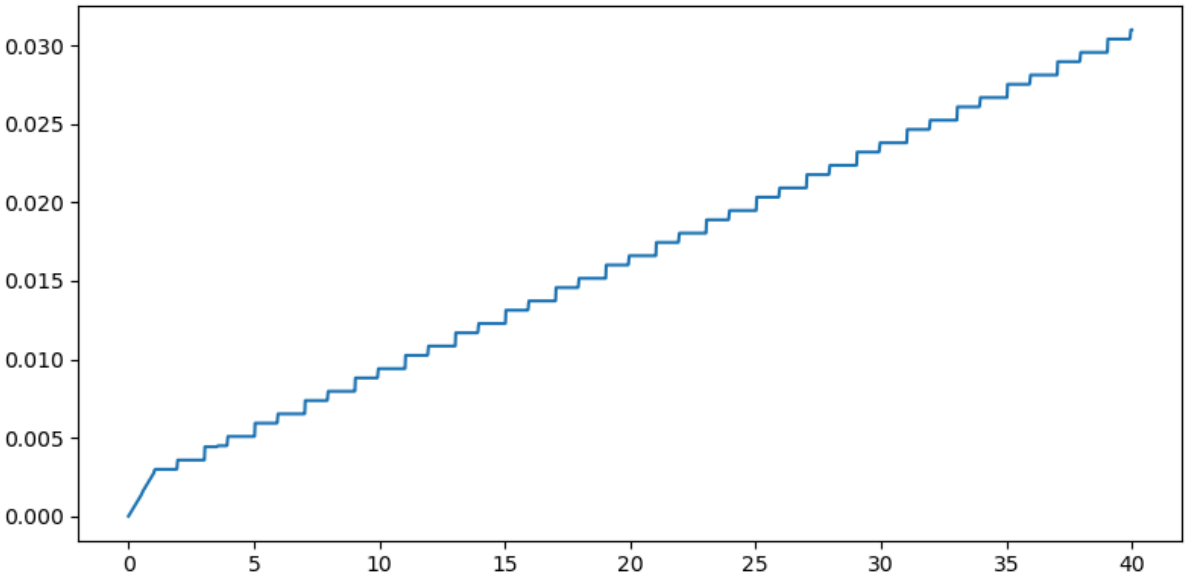
```
* plot v(clk) v(rst)+5 v(en)+10 v(out0)+20 v(out1)+25 v(out2)+30
Vmeasin#branch+40
.endc

.end
```

این تست بنچ مسئول تولید سیگنال clk و rst و همینطور مطرح کردن probe های لازم برای اندازه گیری توان مصرفیست.

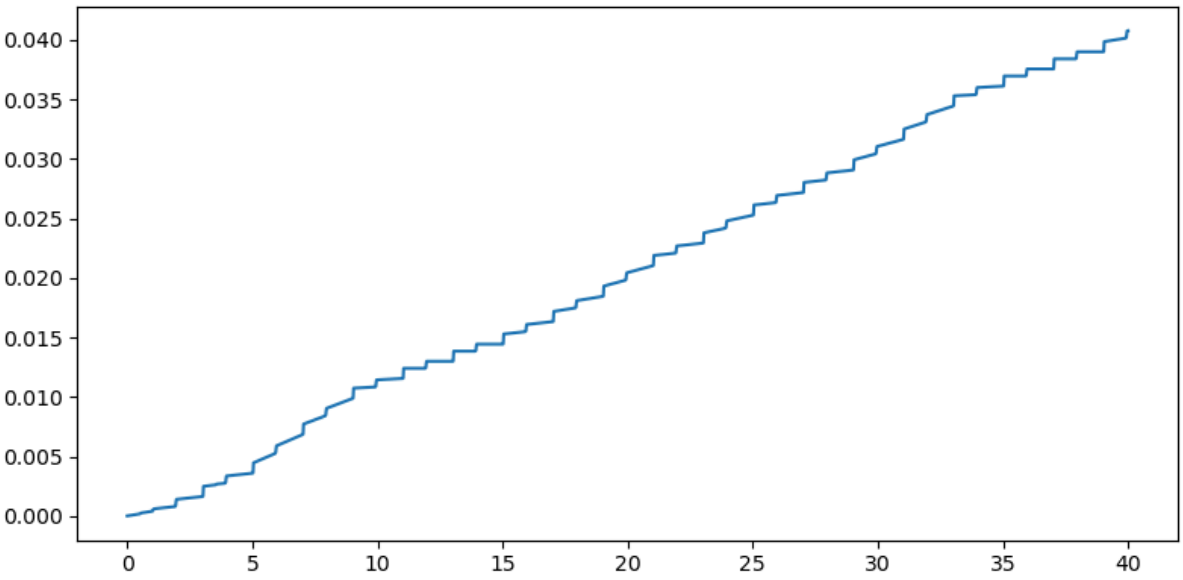
نحوه استفاده از ngspice نیز به این صورت است که با استفاده از مود batch و خروجی raw، جدول کامل ولتاژ و جریان دو سر تغذیه مدار را اندازه میگیریم و با استفاده از یک اسکریپت پایتون ساده آنرا تحلیل میکنیم.

نتیجه آزمون به این صورت است :

- RCA



- MCC



همانطور که مشاهده میکنید، به طور میانگین، انرژی مصرفی MCC $\frac{4}{3}$ انرژی مصرفی RCA است و با توجه به این که PDP تقریبا برابر با میزان انرژی بر محاسبه است و در اینجا تعداد دفعات حساب کردن جمع برابر است (به علت یکسان بودن نرخ کلاک) پس شاخص PDP نیز رفتار مشابهی دارد. بخش عمده ای از این تفاوت نیز به علت wired logic استفاده شده برای انتقال بیت نقلی است.

سوال ۵ - مقایسه الگوریتم‌های quick-sort و merge-sort با استفاده از MEET

برای این سوال کافیست از سایت آزمایشگاه، MEET را دریافت کنیم و و merge-sort و حالت بیکار را پیاده سازی کنیم. توجه کنید که کد quick-sort به عنوان مثال خود MEET قرار دارد. همچنین کد merge-sort ما نیز بر مبنی کد quick-sort مخزن است.

```cpp
void mergeSort(int arr[], int res[], int tmp[], int len) {
    if (len <= 1) {
        res[0] = arr[0];
        return;
    }
    int llen = len / 2;
    int rlen = len - llen;
    mergeSort(arr, tmp, res, llen);
    mergeSort(arr + llen, tmp + llen, res + llen, rlen);
    int *lpt = tmp;
    int *rpt = tmp + llen;
    int *respt = res;
    int l = 0;
    int r = 0;
    int rr = 0;
    while (rr < len) {
        if (r >= rlen || (l < llen && lpt[l] <= rpt[r]))
            respt[rr++] = lpt[l++];
        else
            respt[rr++] = rpt[r++];
    }
}

int main() {
  ...
    // فراخوانی تابع
    int res[LENGTH];
    int tmp[LENGTH];
    int i;
    mergeSort(array, res, tmp, LENGTH);
  ...
}
```

برای اینکه در حالت idle نیز بتوانیم توان مصرفی را اندازه بگیریم، کافیست یک حلقه داشته باشیم که درونش پر از دستورات nop اسمبلی آرم باشد. هدف از اینکه این حلقه این تعداد nop دارد و صرفا یک nop نیست نیز این است که نسبت تعداد nop به سایر دستورات مانند پرش تا جای امکان بالا باشد در نتیجه تا جای امکان پردازنده هیچ کار اضافه نکند. هرچند که اینکار باعث میشود مقداری توان مصرفی باس آدرس میکرو کنترلر بالا برود.

```cpp
int main() {
    register int i = 237064 / 128;
    while (i--) {
        asm("nop");
        asm("nop");
        ...
        asm("nop");
        asm("nop");
    }
}
```

این کد به گونه ای تنظیم شده است که دقیقا به همان تعدادی دستور اجرا کند که quick sort دستور اجرا میکند تا مقایسه بهتری داشته باشیم.

نتیجه آزمون این سه کد به شکل زیر است :

- infloop

```result
sim_num_insn                237058 # total number of instructions executed after analysis started
sim_num_refs                     1 # total number of loads and stores executed
sim_num_loads                    1 # total number of read memory accesses
sim_num_flash_loads              1 # total number of Flash read memory accesses
sim_num_sram_loads               0 # total number of SRAM read memory accesses
sim_num_stores                   0 # total number of write memory accesses
sim_total_energy        286937.2188 # total energy consumption (nJ)
sim_elapsed_time                 1 # total simulation time in seconds
sim_inst_rate           237058.0000 # simulation speed (in insts/sec)
```

- qsort

```result
sim_num_insn                237064 # total number of instructions executed after analysis started
sim_num_refs                 47195 # total number of loads and stores executed
sim_num_loads                35372 # total number of read memory accesses
sim_num_flash_loads           2002 # total number of Flash read memory accesses
sim_num_sram_loads           33370 # total number of SRAM read memory accesses
sim_num_stores               21227 # total number of write memory accesses
sim_total_energy        628401.5625 # total energy consumption (nJ)
sim_elapsed_time                 1 # total simulation time in seconds
sim_inst_rate           237064.0000 # simulation speed (in insts/sec)
```

- msort

```result
sim_num_insn                380755 # total number of instructions executed after analysis started
sim_num_refs                 75367 # total number of loads and stores executed
sim_num_loads                61411 # total number of read memory accesses
sim_num_flash_loads           2001 # total number of Flash read memory accesses
```

```
5  sim_num_sram_loads            59410 # total number of SRAM read memory accesses
6  sim_num_stores                41950 # total number of write memory accesses
7  sim_total_energy        1076878.7500 # total energy consumption (nJ)
8  sim_elapsed_time                   1 # total simulation time in seconds
9  sim_inst_rate           380755.0000 # simulation speed (in insts/sec)
```

توجه کنید که با توجه به نحوه توسعه برنامه ها، merge-sort و quick-sort وظیفه مشابهی در مرتب کردن یک آرایه یکسان دارند. با این حال quick-sort با توجه به locality بسیار بالا تر، هم از دستورات کمتری استفاده نموده و هم تقریبا به اندازه نصف merge-sort انرژی مصرف نموده. از نظر توان مصرفی نیز اوضاع شکل زیر است :

برنامه اجرا شده	تعداد دستورات اجرا شده	انرژی مصرف شده(nJ)	توان مصرف شده(nJ/Inst)
پردازنده بیکار	۲۳۷۰۵۸ (۱x)	۲۸۶۹۳۷ (۱x)	۱.۲۱
quick-sort	۲۳۷۰۶۴ (۱x)	۶۲۸۴۰۱ (۲.۱۹x)	۲.۶۵
merge-sort	۳۸۰۷۵۵ (۱.۶۱x)	۱۰۷۶۸۷۸ (۳.۷۵x)	۲.۸۳

همانطور که در جدول نیز مشاهده میکنید، مرج سورت هم از نظر زمانی و هم از نظر مصرف توان و انرژی از quick-sort بد تر است. در نتیجه در بیشتر اوقات بهتر است از quick-sort استفاده کنیم.

سوال ۶ - الگوریتم DVFS

حل این مسئله را با توجه به این نکته شروع میکنیم که اگر یک وظیفه به اندازه زمان T طول میکشد، در این صورت، برای انجام آن حداقل $\frac{T}{f_{\max}}$ چرخه ساعت نیاز است. همینطور میدانیم که ρ_0 قطعا طوری تعیین میشود که وظیفه مان در بد ترین حالت، دقیقا در زمان Deadline خاتمه یابد، چراکه در غیر این صورت Slack-time بلا استفاده داریم و میتوانستیم از آن برای بهبود کارایی استفاده کنیم!
بر این مبنی رابطه زیر را متصور میشویم (فرکانس پردازش در زمان $f(t) = \text{t}$)

$$\text{تعداد چرخه ساعت} = \int_{t=0}^{\text{Deadline=8ms}} f(t)dt \geq T_{\max} f_{\max} = (6\text{ms}) f_{\max}$$

$$\int_{t=0}^{8\text{ms}} \frac{f(t)}{f_{\max}} dt = \int_{t=0}^{8\text{ms}} \rho(t)dt = \int_{t=0}^{8\text{ms}} \left(\rho_0 + (1-\rho_0)\frac{t}{8\text{ms}}\right)dt = 8\text{ms}\ \rho_0 + \frac{1-\rho_0}{8\text{ms}}\frac{(8\text{ms})^2}{2} = 8\text{ms}\ \left(\rho_0 + \frac{1-\rho_0}{2}\right)$$

$$= 4\text{ms}\ (\rho_0 + 1) \geq 6\text{ms} \Rightarrow \rho_0 \geq 0.5 \Rightarrow \rho(t) = \frac{t}{16\text{ms}} + 0.5$$

حال با توجه به نتیجه بدست آمده، باید امیدریاضی انرژی مصرفی را بدست آوریم.

$$\text{مشابه قبل} : \int_{t=0}^{\mathbb{T}=\text{ زمان اجرا با احتساب DVFS}} \rho(t)dt \geq T$$

$$\Rightarrow \int_{t=0}^{\mathbb{T}} \rho(t)dt = \int_{t=0}^{\mathbb{T}} \left(\frac{t}{16\text{ms}} + 0.5\right)dt = \frac{\mathbb{T}^2}{32\text{ms}} + 0.5\mathbb{T} \geq T \Rightarrow \frac{1}{32\text{ms}}\mathbb{T}^2 + 0.5\mathbb{T} - T \geq 0$$

$$\text{حذف دیمانسیون ms} \Rightarrow \mathbb{T} = \frac{-0.5 \pm \sqrt{0.25 + \frac{T}{8}}}{\frac{1}{16}} \Rightarrow \mathbb{T} = 16\sqrt{0.25 + \frac{T}{8}} - 8$$

$$\mathbb{E}(\text{EN}) = \mathbb{E}\left(\int_{t=0}^{\mathbb{T}} \rho^3(t)dt\right) = \mathbb{E}\left(\int_{t=0}^{\mathbb{T}} \left(\frac{t}{16} + 0.5\right)^3 dt\right)$$