به نام خداوند بخشنده مهربان

طراحی سیستم های دیجیتال

دکتر اجلالی

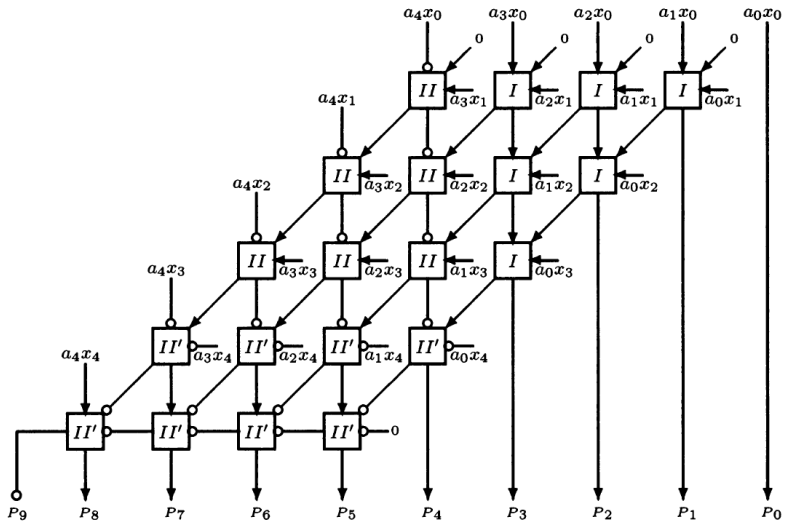
۴۰۱۱۰۶۳۲۸ - نیکا قادری
بهار ۱۴۰۳

تاریخ گزارش: ۲۹ اردیبهشت ۱۴۰۳ **طراحی یک** $arrayMultiplier$ سوال اول

فهرست مطالب

۱ معرفی و بررسی هدف تمرین

در این بخش، ابتدا عملکرد کلی این ضرب کننده را شرح خواهیم داد و در مرحله بعدی، به شرح ماژول های مربوطه خواهیم پرداخت. سپس در مرحله آخر، کد وریلاگ[۱] را با استفاده از کد پایتون تولید می کنیم.
اساس کار ضرب کننده آرایه ای[۲]، بر ضرب بیت به بیت ورودی های یک جایگاه در هم و انتقال خروجی های این ضرب به جایگاه بعدی می باشد. همان طور که در شکل ۱ پیداست، برای ضرب $n * m$ بیتی، به $(m-1) * n$ واحد جمع کننده نیاز خواهیم داشت که می توان آن را در یک آرایه تک بعدی نمایش داد. هر واحد جمع کننده اندیسی دارد. قرارداد می کنیم که همیشه باید از سمت راست و بالا شروع به شماره گذاری کرد.



شکل ۱: ضرب کننده آرایه ای

[۱] verilog
[۲] Array Multiplier

به طور مثال در شکل ۱ ، اندیس واحدی که یکی از ورودی هایش $a_۲x_۲$ هست، ۶ می باشد.
حال کافی است که به تعداد کافی از این واحد ها نمونه[۳] گرفته و درگاه[۴] های مرتبط را به هم وصل کنیم. مثالی از این مورد به طور کامل در بخش بعدی بررسی شده است.

۲ ماژول های وریلاگ و توضیحات

در این بخش یک ساختار یک نمونه ۵ * ۵ که با استفاده از کد پایتون تولید شده است بررسی می شود.

۱.۲ cell

```
module Cell(output Cnext, Sthis, input xn, am, Slast, Cthis);

  wire t;
  and (t, xn, am);

  xor (Sthis, t, Slast, Cthis);
  xor (t1, Slast, Cthis);
  and (t2, t, t1);
  and (t3, Cthis, Slast);
  or (Cnext, t2, t3);

endmodule
```

شکل ۲: cell

ماژول $cell$ عملا یک جمع کننده است که دو ورودی و یک $carry\ in$ - که با نام $cthis$ مشخص شده- می گیرد و هر سه را با هم جمع می‌کند.
تنها تفاوتش با یک $fulladder$ عادی، ورودی دوم آن می باشد که در واقع دو مقدار an و xn است. ماژول ابتدا $an.xn$ را محاسبه می کند و سپس با ورودی های دیگر جمع می کند. این عملکرد به دلیل کابرد این مدار در این سوال خاص می باشد که تقریبا همیشه یکی از ورودی ها به شکل and از دو بیت ورودی برنامه است.

۲.۲ DoubleCell

```
module DoubleCell(output Cnext, Sthis, input xn, am, xn2, am2, Cthis);

  wire t1, t2, t3, t4, t5;
  and (t1, am, xn);
  and (t2, am2, xn2);

  xor (Sthis, t1, t2, Cthis);
  xor (t3, t2, Cthis);
  and (t4, t1, t3);
  and (t5, Cthis, t2);
  or (Cnext, t4, t5);

endmodule
```

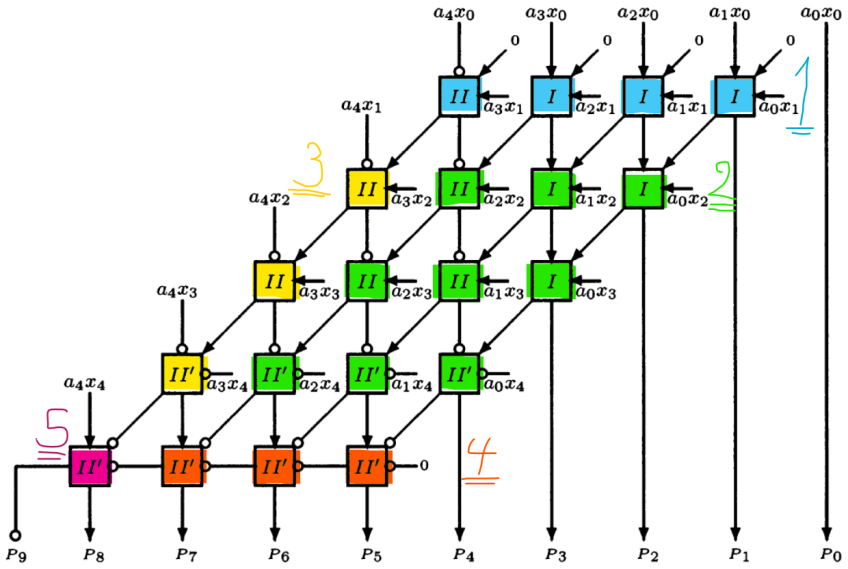
شکل ۳: doubleCell

دقیقا مثل ماژول بالا طراحی شده است با این تفاوت که اینجا به جای ورودی اول $sthis$، دو بیت $an_۲$ و $xn_۲$ داده شده که ابتدا باید با هم and شوند. در واقع دوتا and داریم. کابرد این ماژول در ستون های آخر واحدها می باشد که در آن ورودی اول به جای خروجی s یکی از جمع کننده های بالایی، مستقیما از بیت های ورودی محاسبه می شود.

۳.۲ arrayMultiplier

این ماژول که ماژول اصلی می باشد، شامل تعدادی نمونه[۵] از ماژول های جمع کننده $cell$ و $doubleCell$ می باشد، که خروجی ها و ورودی های متناظر برخی به هم متصل شده اند.
دو آرایه کلی به نام های $s - partial$ و $c - partial$ تعریف می کنیم که خروجی های تمام جمع کننده ها را در خود ذخیره، و جمع کننده ها را به هم متصل می کنند. اندازه هر دو آرایه، برابر با $n * (m - ۱)$ می باشد.
برای درک ساز و کار این ماژول، این ساختار به بخش های مختلفی تقسیم شده است؛ که با اعداد ۱ تا ۵ درشکل ۴ مشخص شده اند.

[۳] instance
[۴] port
[۵] instance



شکل ۴: مثال ۵ * ۵

بخش اول: شامل سطر اول آرایه جمع کننده ها است. به تعداد $m-1$ جمع کننده داریم که m تعداد بیت های ورودی اول است. بیت های نقلی[۶] تمام این سری از جمع کننده ها صفر می باشد، و دو عدد ورودی آن ها در واقع ترکیبی از ۴ بیت ورودی های برنامه است: اگر آن ها را به ترتیب a و x بنامیم، بیت های $a[i]$،$a[j]$،$x[k]$ و $x[l]$ ورودی های این جمع کننده هستند که الگوی دقیق نمایه [۷] آنها بر اساس شکل مشخص می شود. در اینجا با توجه به چهار بیت ورودی، از ماژول $doubleCell$ استفاده می کنیم.

بخش دوم: شامل سطر های میانی به جز ستون های آخر می شود. کافی است در کد پایتون دو حلقه تودرتو بسازیم تا تمام نمونه ها را بسازند. ورودی نقلی هر کدام از آن ها به جمع کننده متناظر سطر قبلشان وابسته است و یکی از ورودی ها نیز ترکیبی از دو بیت $a[i]$ و $x[j]$ می باشد. ورودی دوم از خروجی $s-partial$ جمع کننده ای ساخته می شود که نمایه آن $m-2$ واحد از نمایه فعلی کمتر می باشد. با توجه به نوع ورودی ها، از ماژول $cell$ استفاده می کنیم.

بخش سوم: ستون های آخر سطر های میانی. وروی نقلی آن ها برابر با خروجی نقلی جمع کننده هایی است که در سطر قبلی (ولی در همان موقعیت سطری) قرار دارند. ورودی های این جمع کننده ها دوباره ترکیبی از ۴ بیت a و x می شود.
تعداد این جمع کننده ها $m-2$ است که با توجه به ساختارشان، با ماژول $doubleCell$ ساخته می شوند.

بخش چهارم: جمع کننده های سطر آخر به جز آخرین ستون را در بر می گیرد. ورودی آنها از بیت نقلی جمع کننده قبلی و $s-partial$ واحدی قرار می گیرد که $m-2$ خانه قبل تر قرار دارد. همچنین بیت ورودی نقلی از واحد موجود در سطر قبلی تامین می شود. در این بخش نیز از $cell$ استفاده می کنیم، با این تفاوت که یکی از ورودی های a یا x را $1'b1$ می دهیم و دیگری را برابر با بیت نقلی واحد قبلی می گذاریم تا نتیجه به درستی تعیین شود.
لازم به ذکر است که اولین واحد این سطر بیت ورودی نقلی ندارد که آن را با $1'b0$ مقداردهی می کنیم و جدا می نویسیم.

بخش پنجم: آخرین واحد جمع کننده است. از $Cell$ استفاده می کنیم. فقط به جای $s-partial$ ورودی، این بار دو بیت از a و x را انتخاب می‌کنیم. تصاویر زیر، بخشی از پیاده سازی توضیحات فوق را نمایش می دهند.

```
// last row exept for the last and first columns


Cell c_result1(c_partial[17], s_partial[17],
    1'b1, c_partial[16], s_partial[14], c_partial[13]);


Cell c_result2(c_partial[18], s_partial[18],
    1'b1, c_partial[17], s_partial[15], c_partial[14]);


// last column of the last row
Cell c_result_ll(c_partial[19], s_partial[19],
    a[4], x[4], c_partial[18], c_partial[15]);
```

```
// last columns except for the last row


DoubleCell c_last0(c_partial[7], s_partial[7],
    a[3], x[2], a[4], x[1], c_partial[3]);


DoubleCell c_last1(c_partial[11], s_partial[11],
    a[3], x[3], a[4], x[2], c_partial[7]);


DoubleCell c_last2(c_partial[15], s_partial[15],
    a[3], x[4], a[4], x[3], c_partial[11]);


// first column of the last row
Cell c_result(c_partial[16], s_partial[16],
    1'b0, 1'b0, s_partial[13], c_partial[12]);
```

```
// first line of the multiplier:


DoubleCell c_first0(.Cnext(c_partial[0]), .Sthis(s_partial[0]),
    .xn(x[0]), .am(a[1]), .xn2(x[1]), .am2(a[0]), .Cthis(1'b0));


DoubleCell c_first1(.Cnext(c_partial[1]), .Sthis(s_partial[1]),
    .xn(x[0]), .am(a[2]), .xn2(x[1]), .am2(a[1]), .Cthis(1'b0));


DoubleCell c_first2(.Cnext(c_partial[2]), .Sthis(s_partial[2]),
    .xn(x[0]), .am(a[3]), .xn2(x[1]), .am2(a[2]), .Cthis(1'b0));


DoubleCell c_first3(.Cnext(c_partial[3]), .Sthis(s_partial[3]),
    .xn(x[0]), .am(a[4]), .xn2(x[1]), .am2(a[3]), .Cthis(1'b0));
```

حال با استفاده از بافر[۸]ها خروجی نهایی را در آرایه ای به نام $product$ با طول $m+n$ ذخیره می کنیم. به این صورت که ابتدا واحدهای مورب (در این مثال، شماره های ۰، ۴، ۸ و ۱۲) که نمایه آنها مضرب $(m-1)$ است را در بیت های ۱ تا $n-1$ آرایه خروجی قرار می دهیم. (n تعداد بیت های ورودی دوم است) سپس، $m-1$ بیت بعدی آرایه $product$ همان سطرهای آخر آرایه $s-partial$ می شوند و آخرین بیت نتیجه هم برابر با بیت نقلی خروجی آخرین واحد جمع کننده می شود. توجه کنید که تا اینجا، تکلیف $product[0]$ هنوز مشخص نشده است. برای همین در ابتدای کد به طور دستی واحدی به نام $p-first$ و از نوع $cell$ کار گذاشته شده که حاصل $a.x$ را محاسبه می کند و در $s-partial[0]$ قرار

[۶] carry
[۷] index
[۸] buffer

می دهد. بنابراین کافی است این بیت را برابر با $product[0]$ بگذاریم.
تصاویر زیر نحوه مقداردهی های نهایی را نشان می دهند:

```
// product bits from the last line of cells


buf (product[5], s_partial[16]);

buf (product[6], s_partial[17]);

buf (product[7], s_partial[18]);

buf (product[8], s_partial[19]);

// msb and lsb of product
buf (product[9], c_partial[19]);
buf (product[0], p0);
```

```
// product bits from first and middle cells


buf (product[1], s_partial[0]);

buf (product[2], s_partial[4]);

buf (product[3], s_partial[8]);

buf (product[4], s_partial[12]);
```

۳ کد تست بنچ

از آنجایی که تست بنچ همراه با کد اصلی تولید شده است، آن را مستقیم بعد از ماژول $arrayMultiplier$ بررسی می کنیم. در تست بنچ پس از نمونه گرفتن از ماژول اصلی، در ده مرحله بر اساس n و m ورودی که توسط کد پایتون داده شده است، اعداد تصادفی به عنوان ورودی به ضرب کننده داده می شوند و خروجی آن در یک سیم جدا به نام p قابل مشاهده است. ساختار این ماژول در شکل ۵ قابل مشاهده است.

```
module tb;
wire [9:0] p;
reg [4:0] a;
reg [4:0] x;

ArrayMultiplier am (p, a, x);
initial $monitor("a=%b, x=%b, p=%b", a, x, p);

initial begin

#10
a = 5'b01101;
x = 5'b00001;

#10
a = 5'b00000;
x = 5'b01011;

#10
a = 5'b10011;
x = 5'b11001;

#10
a = 5'b01111;
x = 5'b00100;
```

شکل ۵: تست بنچ

۴ توضیحات کد پایتون

در این برنامه، ابتدا با دادن $prompt$ از کابر ورودی های n و m گرفته می شود و یا در صورت ورودی اشتباه، خطای مناسب داده می شود. سپس تابع $generate\_array\_multiplier$ اجرا می شود که درون آن نیز چک می شود که ابعاد معنی دار باشند:

```
if __name__ == "__main__":
    try:
        m = int(input("Enter the number of bits for a - first number with 'm' bits: "))
        n = int(input("Enter the number of bits for x - second number with 'n' bits: "))
        generate_array_multiplier(n, m)
    except ValueError:
        print("Invalid input. Please provide valid integer values for n and m.")
```

```
def generate_array_multiplier(n, m):
    # Validate input: Ensure n and m are positive integers
    if not (isinstance(n, int) and isinstance(m, int) and n > 0 and m > 0):
        print("Invalid input. Please provide positive integer values for n and m.")
        return
```

در مرحله بعدی کد وریلاگ را به صورت متنی در یک $string$ $formatted$ ذخیره می کنیم و هرجا نیاز به ابعاد شد به صورت پارامتری از ورودی های m و n استفاده می کنیم. حلقه های هر پنج بخش را در پایتون پیاده سازی می کنیم تا در کد تولید شده نیازی به استفاده از $generate$ و $forloop$ نباشد و کد کاملا به صورت ساختاری باقی بماند.
تنها نکته باقیمانده، نام گذاری نمونه های تولید شده درون حلقه ها می باشد به طوری که تکراری نباشند. برای این پیاده سازی می توان از شماره کنونی حلقه در نام گذاری استفاده کرد.
تصاویری از نحوه تولید کد:

```
    # Generate Verilog code for the array multiplier
    verilog_code = f"""
module DoubleCell(output Cnext, Sthis, input xn, am, xn2, am2, Cthis);

  wire t1, t2, t3, t4, t5;
  and (t1, am, xn);
  and (t2, am2, xn2);

  xor (Sthis, t1, t2, Cthis);
  xor (t3, t2, Cthis);
  and (t4, t1, t3);
  and (t5, Cthis, t2);
  or (Cnext, t4, t5);

endmodule
```

```
// first line of the multiplier:
"""
    for i in range(m - 1):
        verilog_code += f"""

DoubleCell c_first{i}(.Cnext(c_partial[{i}]), .Sthis(s_partial[{i}]),
    .xn(x[0]), .am(a[{i + 1}]), .xn2(x[1]), .am2(a[{i}]), .Cthis(1'b0));
"""
```

```
// middle lines except for the last columns in each row:

"""
    for j in range(n - 2):
        for k in range(m - 2):
            verilog_code += f"""

Cell c_middle{j}{k}(c_partial[{(m - 1) * (j + 1) + k}], s_partial[{(m - 1) * (j + 1) + k}],
    a[{k}], x[{j + 2}], s_partial[{(m - 1) * j + k + 1}], c_partial[{(m - 1) * j + k}]);
"""
```

```
// product bits from the last line of cells

"""

    for i in range(n - 1, n + m - 2):
        verilog_code += f"""

buf (product[{i + 1}], s_partial[{(m - 1) * (n - 1) + i - (n - 1)}]); """

    verilog_code += f"""

// msb and lsb of product
buf (product[{m + n - 1}], c_partial[{(m - 1) * (n - 1) + m - 2}]);
buf (product[0], p0);
```

در ادامه، با دادن $prompt$ به کاربر، از او پرسیده می شود که آیا مایل به تولید اتوماتیک تست بنچ برای ماژول ساخته شده نیز هست یا نه. که در صورت تائید، بر اساس ابعاد ورودی تست بنچی ساخته می شود. در بازه های زمانی یکسان، اعداد باینری به طور تصادفی ساخته می شوند و به عنوان ورودی در تست بنچ نوشته می شوند. ساز و کار این کد در زیر آورده شده است:

همچنین در نهایت، همه رشته های لازم روی فایل های خروجی نوشته می شوند:

```
    prmt = input("Would you like to get a testbench as well? enter y for yes, n for no.\n")
    if prmt.strip() == 'y':
        testbench_code = f"""
module tb;
wire [{n+m-1}:0] p;
reg [{m-1}:0] a;
reg [{n-1}:0] x;

ArrayMultiplier am (p, a, x);
initial $monitor("a=%b, x=%b, p=%b", a, x, p);

initial begin
"""
        for i in range(10):
            a_generated = "".join(str(random.randint(0, 1)) for _ in range(m))
            x_generated = "".join(str(random.randint(0, 1)) for _ in range(n))
            testbench_code += f"""
#10
a = {m}'b{a_generated};
x = {n}'b{x_generated};
"""

        testbench_code += """
end
endmodule
"""
```

شکل ۶: ساخت تست بنچ

```
        testbench_code += """
end
endmodule
"""
        with open("tb.v", "w") as f:
            f.write(testbench_code)
        print(f"""process finished: multiplier code generated for m = {m} and n = {n} and testbench created""")
    else:
        print(f"""process finished: multiplier code generated for m = {m} and n = {n}""")
```

شکل ۷: ذخیره نهایی

۵ اجرا و شکل موج

برای مثال، یک ضرب کننده ۶ * ۴ می سازیم. در نتیجه، ابتدا کد و تست بنچ را با کد پایتون تولید می کنیم:

```
Run:  main
D:\verilog\EXAM\SOAL1PYTH\ds\Scripts\python.exe D:\verilog\EXAM\SOAL1PYTH\main.py
Enter the number of bits for a - first number with 'm' bits: 4
Enter the number of bits for x - second number with 'n' bits: 6
Verilog code written to arrayMultiplier.v successfully!
Would you like to get a testbench as well? enter y for yes, n for no.
y
process finished: multiplier code generated for m = 4 and n = 6 and testbench created

Process finished with exit code 0
```

شکل ۸: تولید کد

حال اگر به پوشه کنونی مراجعه کنیم می توان کد تولید شده را در دو فایل $tb.v$ و $arrayMultiplier.v$ مشاهده کرد:

main.py | arrayMultiplier.v | tb.v

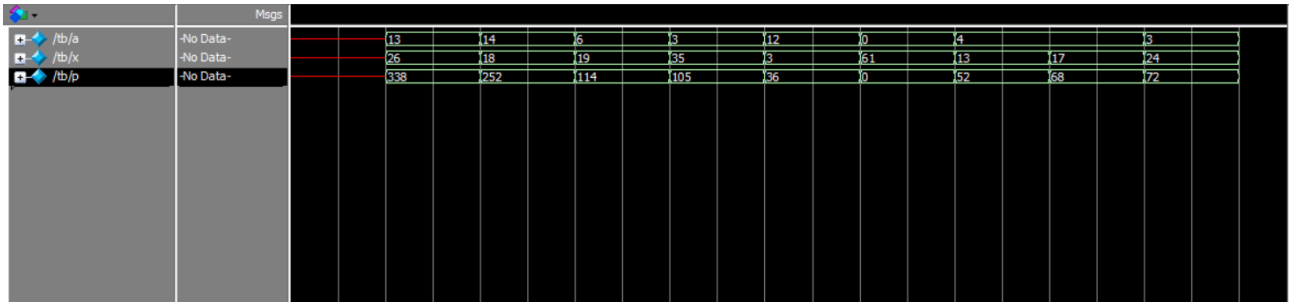
```
module DoubleCell(output Cnext, Sthis, input xn, am, xn2, am2, Cthis);

  wire t1, t2, t3, t4, t5;
  and (t1, am, xn);
  and (t2, am2, xn2);

  xor (Sthis, t1, t2, Cthis);
  xor (t3, t2, Cthis);
  and (t4, t1, t3);
  and (t5, Cthis, t2);
  or (Cnext, t4, t5);

endmodule


module Cell(output Cnext, Sthis, input xn, am, Slast, Cthis);

  wire t;
  and (t, xn, am);

  xor (Sthis, t, Slast, Cthis);
  xor (t1, Slast, Cthis);
  and (t2, t, t1);
  and (t3, Cthis, Slast);
  or (Cnext, t2, t3);

endmodule


module ArrayMultiplier(product, a, x);
```

main.py | arrayMultiplier.v | tb.v

```
a = 4'b0110;
x = 6'b010011;

#10
a = 4'b0011;
x = 6'b100011;

#10
a = 4'b1100;
x = 6'b000011;

#10
a = 4'b0000;
x = 6'b111101;

#10
a = 4'b0100;
x = 6'b001101;

#10
a = 4'b0100;
x = 6'b010001;

#10
a = 4'b0011;
x = 6'b011000;

#10
a = 4'b0010;
x = 6'b011010;

end
```

شکل ۹: خروجی های تولید شده

اکنون می توان در $modelsim$ تست بنچ را اجرا کرد و عملکرد کد را مورد بررسی قرار داد. پس از کامپایل کردن همه فایل ها و در قسمت شبیه سازی[۹] می توان موج های مربوط به دو ورودی و خروجی را اضافه کرد و مبنا[۱۰] آنها را نیز به اعداد بدون علامت[۱۱] تغییر داد. بدین صورت علاوه بر ترمینال، می توان خروجی ها را به صورت شکل موج نیز مشاهده کرد. با استفاده از تست بنچی که در مرحله قبل تولید کردیم، شکل موج به صورت زیر در می آید:
همچنین خروجی ترمینال نیز به صورت زیر می باشد که مقادیر را به صورت باینری نمایش می دهد:



شکل ۱۰: شکل موج: a و x ورودی هستند و p خروجی ضرب این دو هست.

```
VSIM 4> run
# a=xxxx, x=xxxxxx, p=xxxxxxxxxx
# a=1101, x=011010, p=0101010010
# a=1110, x=010010, p=0011111100
# a=0110, x=010011, p=0001110010
# a=0011, x=100011, p=0001101001
# a=1100, x=000011, p=0000100100
# a=0000, x=111101, p=0000000000
# a=0100, x=001101, p=0000110100
# a=0100, x=010001, p=0001000100
# a=0011, x=011000, p=0001001000
```

شکل ۱۱: خروجی ترمینال

۶ نتیجه گیری

در این سوال، با عملکرد ضرب کننده آرایه ای آشنا شدیم و کد وریلاگ آن را به صورت ساختاری نوشتیم، که شامل نمونه گرفتن از ماژول های متفاوت می باشد. به دلیل این که ابعاد ضرب کننده متغیر هست، ممکن است تعداد این نمونه ها بسیار زیاد شود به طوری که کد وریلاگ حجیم شده و عملا به صورت دستی قابل نوشتن نیست. برای حل این مشکل می توان به زبان برنامه نویسی دیگری، مثل پایتون، کدی نوشت که الگوریتم ضرب کننده را خود بنویسد و فقط ابعاد ورودی را از کاربر بگیرد. با استفاده از حلقه های پی در پی و محاسبه پارامتریک نمایه ها موفق شدیم این کد را با حجم بسیار کم و به طور یکتا بنویسیم. سپس برای کد مربوطه، تست بنچ هم ساختیم و آن را در محیط $ModelSim$ آزمایش کردیم.

پایان

[۹] simulation
[۱۰] radix
[۱۱] unsigned