Design of Embedded Hardware and Firmware

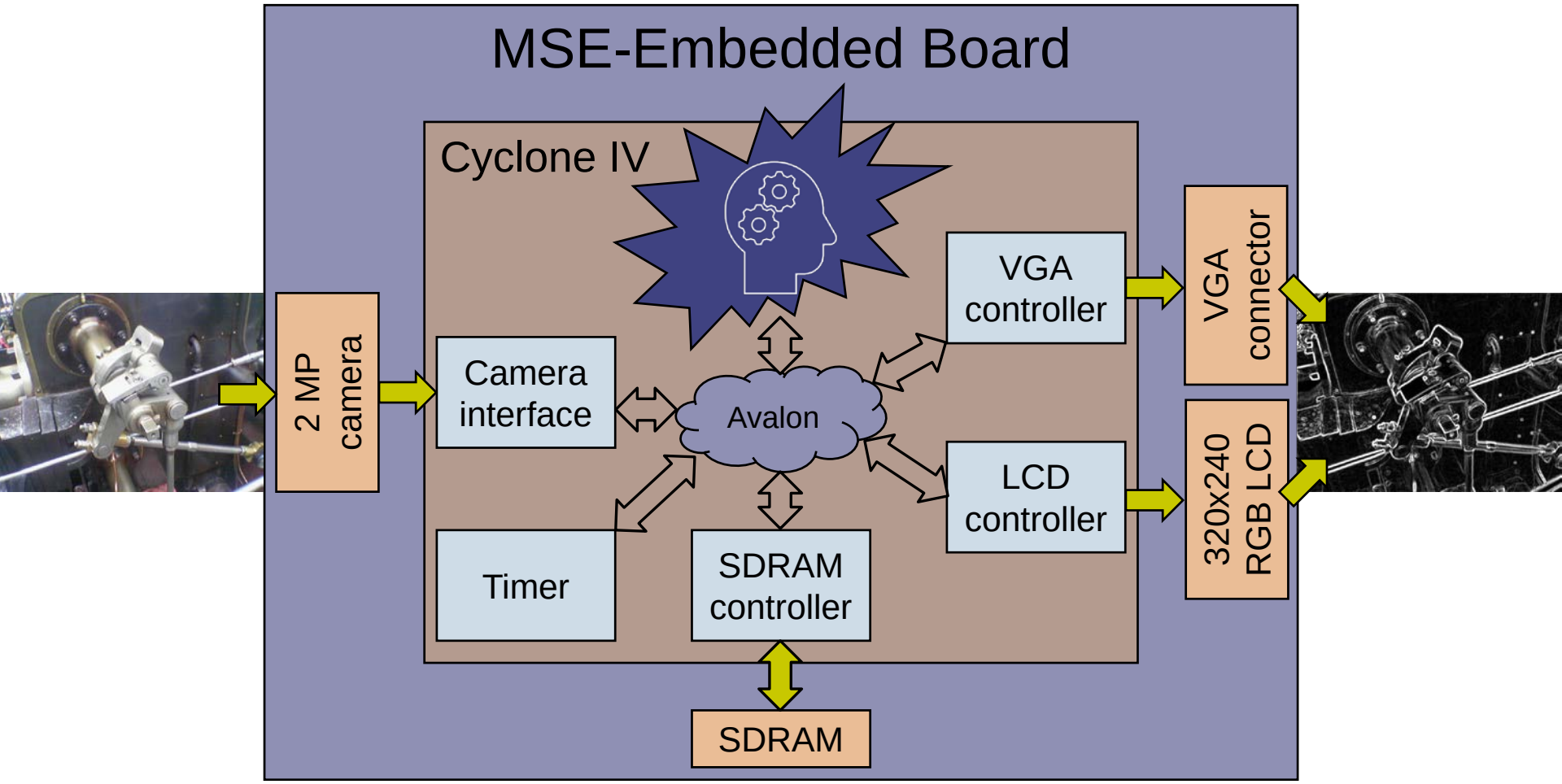
Software Optimization 2

Andrea Guerrieri

HES-SO//Genève

andrea.guerrieri@hesge.ch

Problem specifications



«Out-of-the-box» optimization

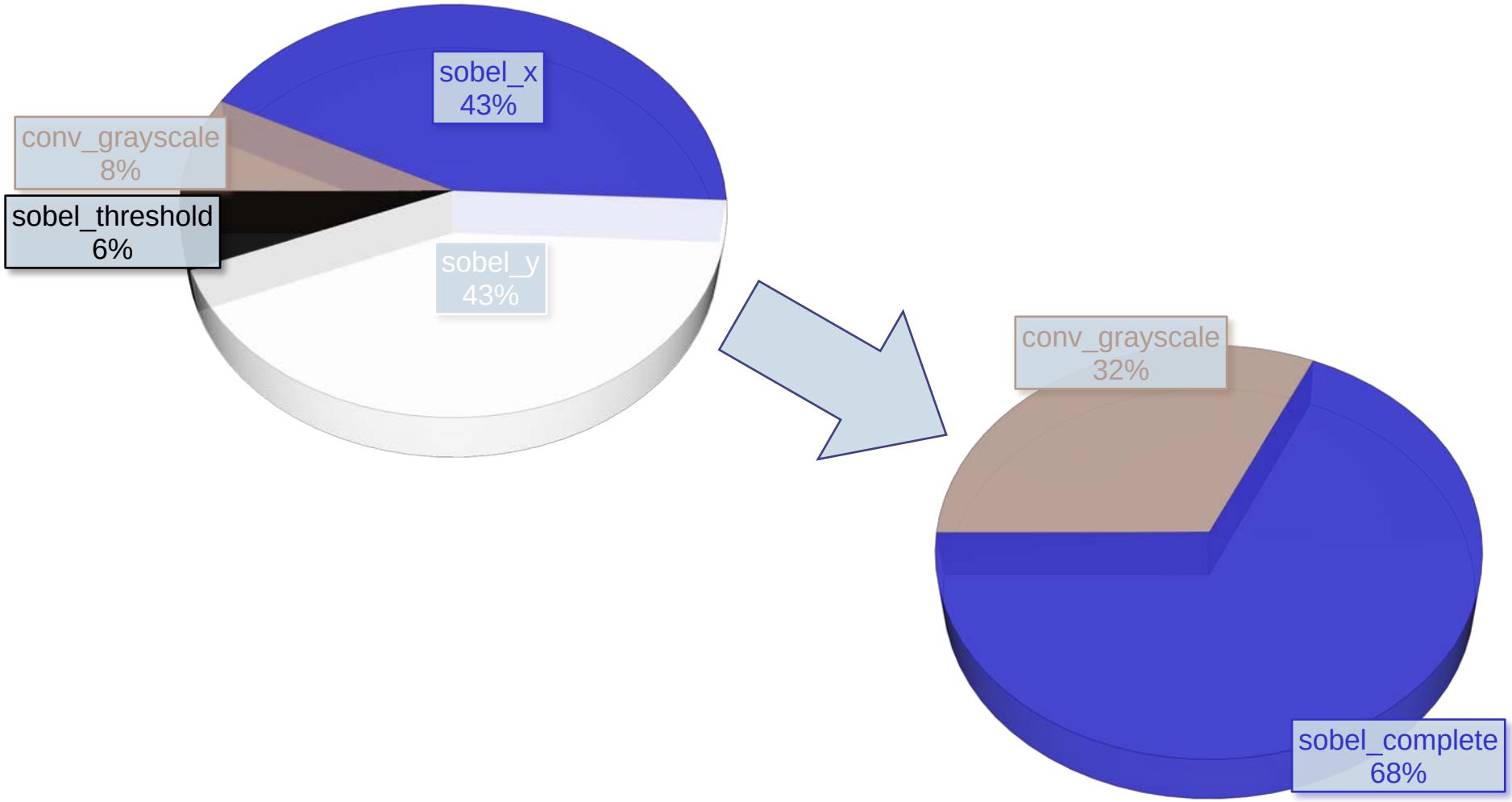
Function	-O0	-O1	-O2	-O3	
conv_grayscale (Time)	127336549 (2.547 s)	44161114 (0.883 s)	41296763 (0.826 s)	41194172 (0.824 s)	3.09x
sobel_x (Time)	729624798 (14.592 s)	129966322 (2.599 s)	109616923 (2.192 s)	29727206 (0.595 s)	24.54x
sobel_y (Time)	729677739 (14.594 s)	129964177 (2.599 s)	109607580 (2.192 s)	27249940 (0.545 s)	26.78x
sobel_threshold (Time)	102849150 (2.057 s)	33664258 (0.673 s)	32015554 (0.64 s)	32126201 (0.643 s)	3.20x
TOTAL (Time)	**1689488236 (33.79 s)**	**337755871 (6.754 s)**	**292536820 (5.85 s)**	**130297519 (2.607 s)**	**12.97x**

Loop unrolling + in-lining

Function	-O0	-O0 (no loops)	-O0 (no loops, inline)
conv_grayscale (Time)	127336549 (2.547 s)	127071260 (2.541 s)	125935754 (2.519 s)
sobel_x (Time)	729624798 (14.592 s)	425314676 (8.506 s)	325845850 (6.517 s)
sobel_y (Time)	729677739 (14.594 s)	425378323 (8.508 s)	320545780 (6.411 s)
sobel_threshold (Time)	102849150 (2.057 s)	105635977 (2.113 s)	104956605 (2.099 s)
TOTAL (Time)	**1689488236 (33.79 s)**	**1083400236 (21.67 s)**	**877283989 (17.55 s)**
		1.56x	1.93x

1.31x (2.24x)

Loop unrolling + in-lining



Software Optimizations

So far only code-level modifications to remove the overhead…

Software Optimizations

So far only code-level modifications to remove the overhead…

How can we improve more?

RGB to grayscale

- First approximation:

  Grayscale = (R+G+B)/3

RGB to grayscale

- First approximation:

  Grayscale = (R+G+B)/3

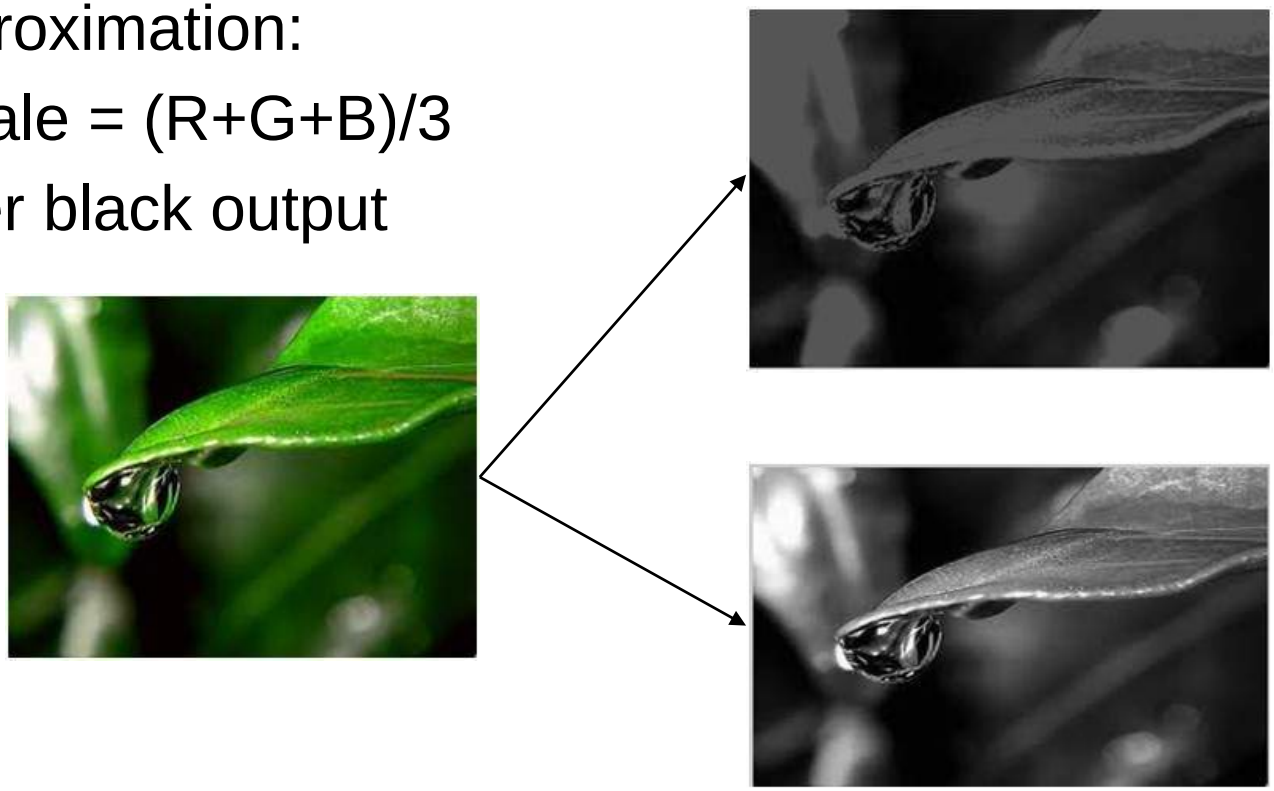
  - Rather black output

RGB to grayscale

- First approximation:

  Grayscale = (R+G+B)/3

  - Rather black output



- Luminance (ITU-R Rec.601), approximated:

  Grayscale = 0.3*R + 0.59*G + 0.11*B

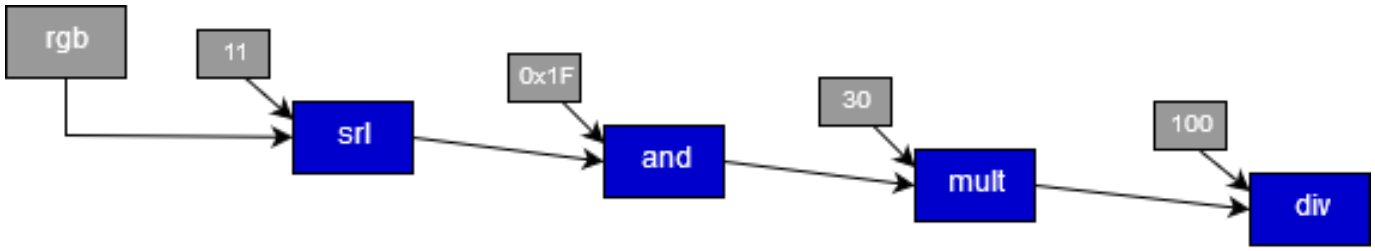
RGB to grayscale

Grayscale = 0.3*R + 0.59*G + 0.11*B

```
void conv_grayscale(void *picture,
                    int width,
                    int height) {
    int x,y,gray;
    unsigned short *pixels = (unsigned short *)picture , rgb;
    unsigned int temp;
    grayscale_width = width;
    grayscape_height = height;
    if (grayscale_array != NULL)
        free(grayscale_array);
    grayscale_array = (unsigned char *) malloc(width*height);
    for (y = 0 ; y < height ; y++) {
        for (x = 0 ; x < width ; x++) {
            rgb = pixels[y*width+x];
            temp = (rgb>>11)&0x1F; // red value
            gray = (temp*30)/100;
            temp = (rgb>>5)&0x3F; // green value
            gray += (temp*59)/100;
            temp = rgb&0x1F; // blue value
            gray += (temp*11)/100;
            IOWR_8DIRECT(grayscale_array,y*width+x,gray);
        }
    }
}
```

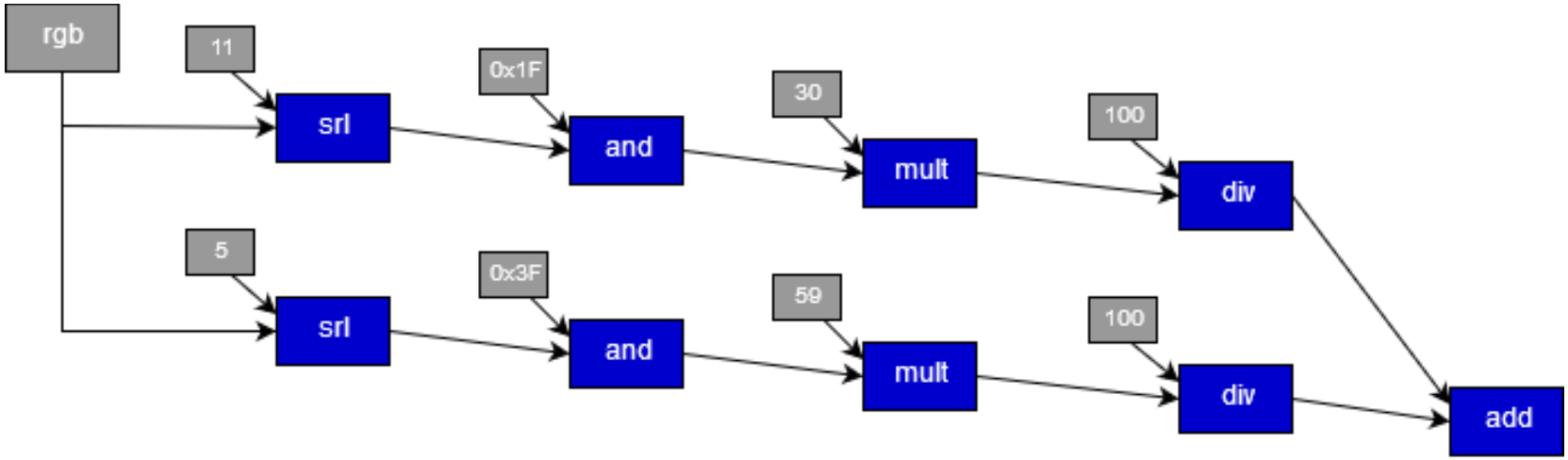
Data Flow Graph (DFG)

Grayscale = 0.3*R + 0.59*G + 0.11*B



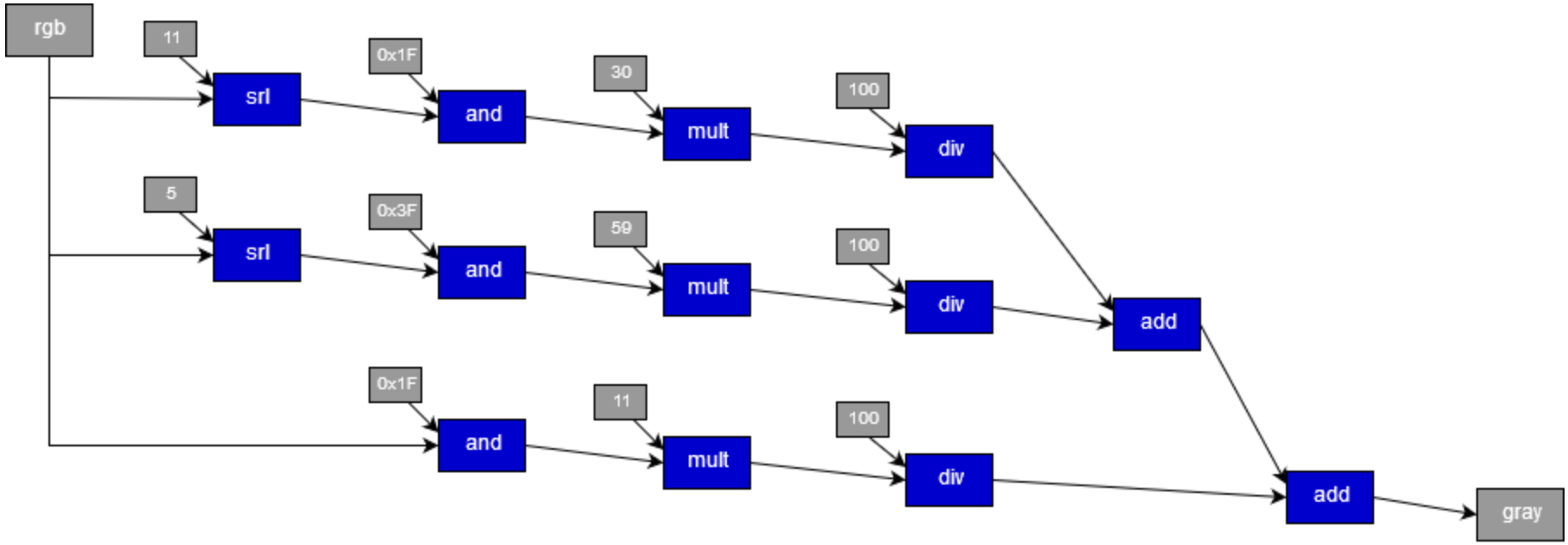
Data Flow Graph (DFG)

Grayscale = 0.3*R + 0.59*G + 0.11*B



Data Flow Graph (DFG)

Grayscale = 0.3*R + 0.59*G + 0.11*B



RGB to grayscale

Grayscale = 0.3*R + 0.59*G + 0.11*B

```
void conv_grayscale(void *picture,
                    int width,
                    int height) {
    int x,y,gray;
    unsigned short *pixels = (unsigned short *)picture , rgb;
    unsigned int temp;
    grayscale_width = width;
    grayscape_height = height;
    if (grayscale_array != NULL)
        free(grayscale_array);
    grayscale_array = (unsigned char *) malloc(width*height);
    for (y = 0 ; y < height ; y++) {
        for (x = 0 ; x < width ; x++) {
            rgb = pixels[y*width+x];
            temp = (rgb>>11)&0x1F; // red value
            gray = (temp*30)/100;
            temp = (rgb>>5)&0x3F; // green value
            gray += (temp*59)/100;
            temp = rgb&0x1F; // blue value
            gray += (temp*11)/100;
            IOWR_8DIRECT(grayscale_array,y*width+x,gray);
        }
    }
}
```

~945 cycles/pixel

RGB to grayscale

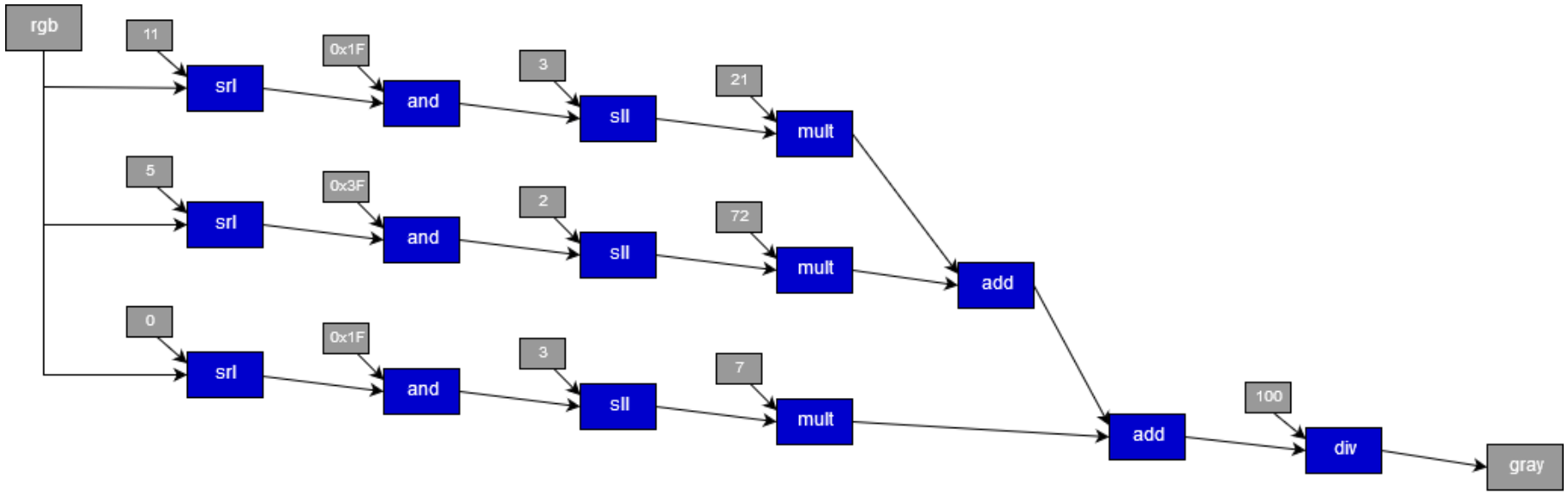
Grayscale ~= 0.33*R + 0.56*G + 0.11*B

```
void conv_grayscale(void *picture,
                    int width,
                    int height) {
    int x,y,gray;
    unsigned short *pixels = (unsigned short *)picture , rgb;
    grayscale_width = width;
    grayscape_height = height;
    if (grayscale_array != NULL)
        free(grayscale_array);
    grayscale_array = (unsigned char *) malloc(width*height);
    for (y = 0 ; y < height ; y++) {
        for (x = 0 ; x < width ; x++) {
            rgb = pixels[y*width+x];
            gray = (((rgb>>11)&0x1F)<<3)*21; // red part
            gray += (((rgb>>5)&0x3F)<<2)*72; // green part
            gray += (((rgb>>0)&0x1F)<<3)*7; // blue part
            gray /= 100;
            IOWR_8DIRECT(grayscale_array,y*width+x,gray);
        }
    }
}
```

~800 cycles/pixel

Data Flow Graph (DFG)

Grayscale ~= 0.33*R + 0.56*G + 0.11*B



Algorithm optimization

Objdump:

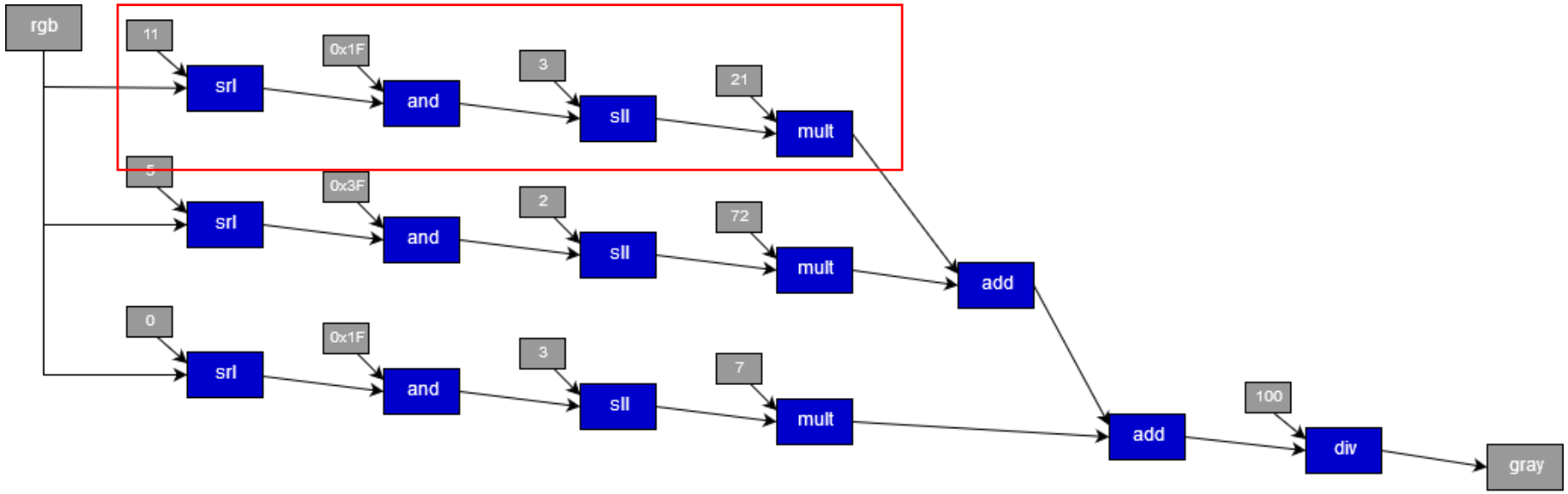
```
b48:    1080000b        ldhu    r2,0(r2)
b4c:    e0bffb0d        sth     r2,-20(fp)
                gray = (((rgb>>11)&0x1F)<<3)*21; // red part
b50:    e0bffb0b        ldhu    r2,-20(fp)
b54:    1004d2fa        srli    r2,r2,11
b58:    10bfffcc        andi    r2,r2,65535
b5c:    100490fa        slli    r2,r2,3
b60:    10803fcc        andi    r2,r2,255
b64:    10800564        muli    r2,r2,21
b68:    e0bffc15        stw     r2,-16(fp)
                gray += (((rgb>>5)&0x3F)<<2)*72; // green part
b6c:    e0bffb0b        ldhu    r2,-20(fp)
b70:    1004d17a        srli    r2,r2,5
b74:    10bfffcc        andi    r2,r2,65535
b78:    1085883a        add     r2,r2,r2
b7c:    1085883a        add     r2,r2,r2
b80:    10803fcc        andi    r2,r2,255
b84:    10801224        muli    r2,r2,72
b88:    e0fffc17        ldw     r3,-16(fp)
b8c:    1885883a        add     r2,r3,r2
b90:    e0bffc15        stw     r2,-16(fp)
                gray += (((rgb>>0)&0x1F)<<3)*7; // blue part
b94:    e0bffb0b        ldhu    r2,-20(fp)
b98:    100490fa        slli    r2,r2,3
b9c:    10803fcc        andi    r2,r2,255
ba0:    108001e4        muli    r2,r2,7
ba4:    e0fffc17        ldw     r3,-16(fp)
ba8:    1885883a        add     r2,r3,r2
bac:    e0bffc15        stw     r2,-16(fp)
                gray /= 100;
bb0:    e0bffc17        ldw     r2,-16(fp)
bb4:    01401904        movi    r5,100
bb8:    1009883a        mov     r4,r2
bbc:    00022c80        call    22c8 <__divsi3>
bc0:    e0bffc15        stw     r2,-16(fp)
                IOWR_8DIRECT(grayscale_array,y*width+x,gray);
bc4:    d0a6cd17        ldw     r2,-25804(gp)
bc8:    e13ff917        ldw     r4,-28(fp)
```

Data Flow Graph (DFG)

Grayscale ~= 0.33*R + 0.56*G + 0.11*B

Operations:

7



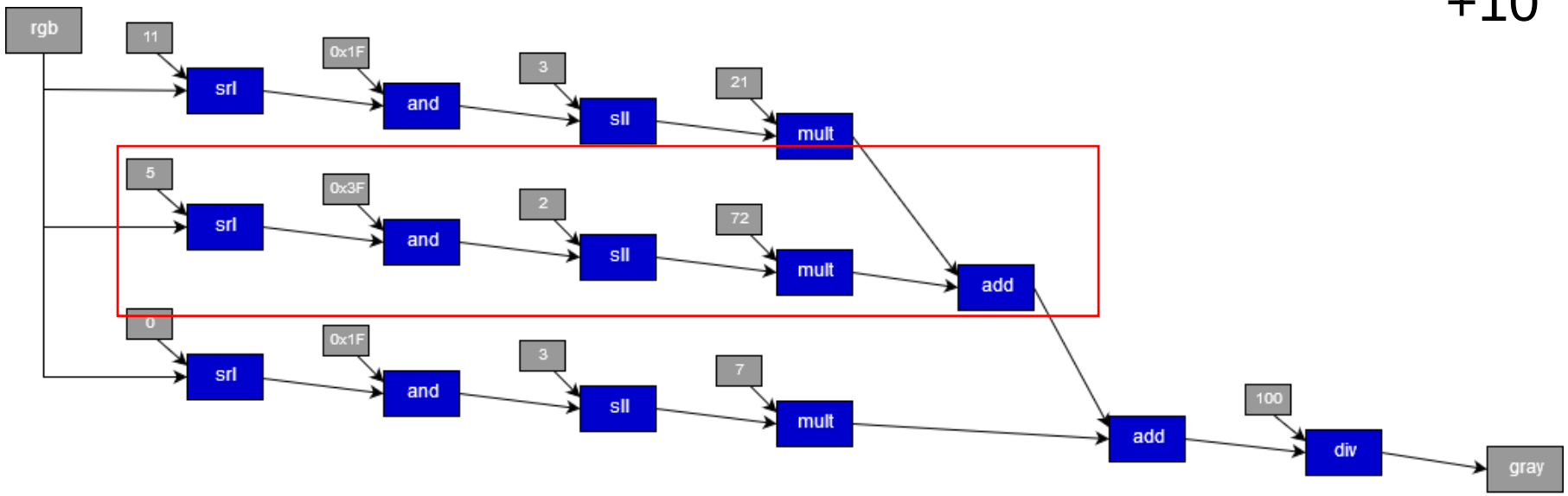
Data Flow Graph (DFG)

Grayscale ~= 0.33*R + 0.56*G + 0.11*B

Operations:

7

+10



Data Flow Graph (DFG)

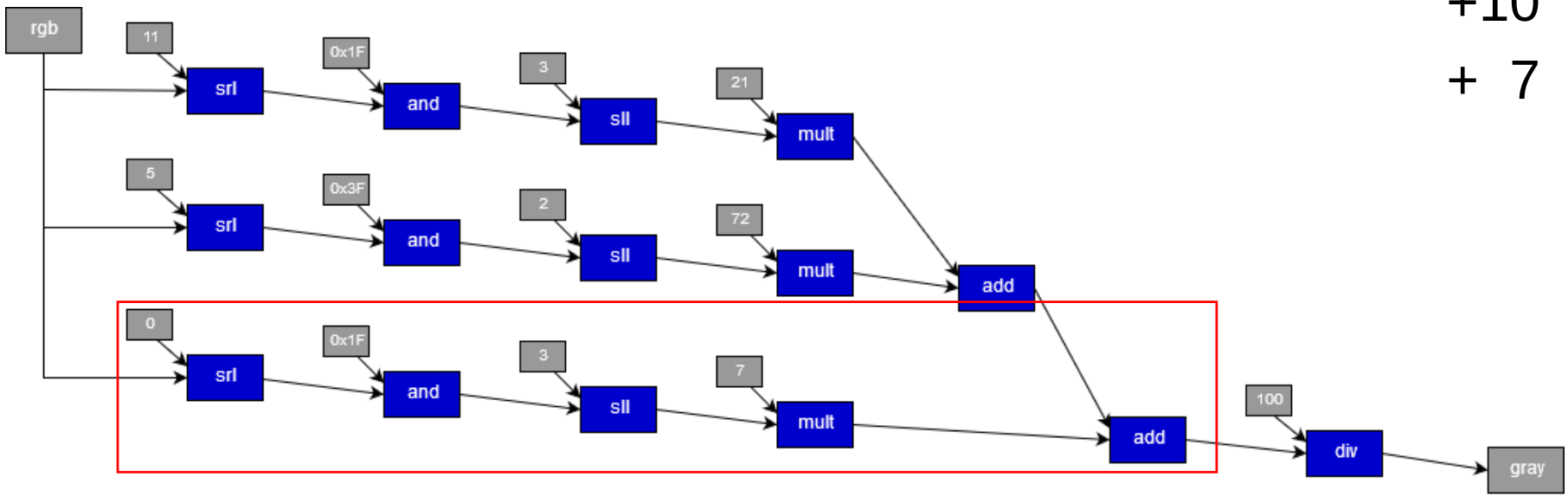
Grayscale ~= 0.33*R + 0.56*G + 0.11*B

Operations:

7

+10

+ 7



Data Flow Graph (DFG)

Grayscale ~= 0.33*R + 0.56*G + 0.11*B

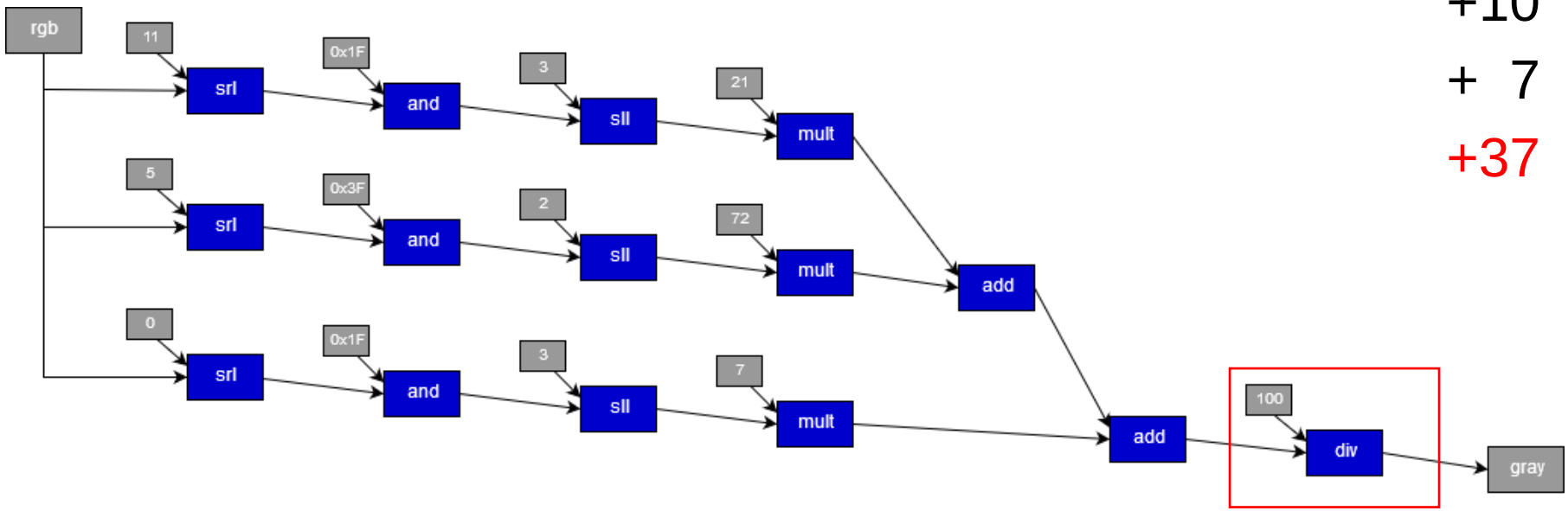
Operations:

7

+10

+ 7

+37



Algorithm optimization

Division:

33 operations

```
000022c8 <__divsi3>:
    22c8:       20001b16        blt     r4,zero,2338 <__divsi3+0x70>
    22cc:       000f883a        mov     r7,zero
    22d0:       28001616        blt     r5,zero,232c <__divsi3+0x64>
    22d4:       200d883a        mov     r6,r4
    22d8:       29001a2e        bgeu    r5,r4,2344 <__divsi3+0x7c>
    22dc:       00800804        movi    r2,32
    22e0:       00c00044        movi    r3,1
    22e4:       00000106        br      22ec <__divsi3+0x24>
    22e8:       10000d26        beq     r2,zero,2320 <__divsi3+0x58>
    22ec:       294b883a        add     r5,r5,r5
    22f0:       10bfffc4        addi    r2,r2,-1
    22f4:       18c7883a        add     r3,r3,r3
    22f8:       293ffb36        bltu    r5,r4,22e8 <__divsi3+0x20>
    22fc:       0005883a        mov     r2,zero
    2300:       18000726        beq     r3,zero,2320 <__divsi3+0x58>
    2304:       0005883a        mov     r2,zero
    2308:       31400236        bltu    r6,r5,2314 <__divsi3+0x4c>
    230c:       314dc83a        sub     r6,r6,r5
    2310:       10c4b03a        or      r2,r2,r3
    2314:       1806d07a        srli    r3,r3,1
    2318:       280ad07a        srli    r5,r5,1
    231c:       183ffa1e        bne     r3,zero,2308 <__divsi3+0x40>
    2320:       38000126        beq     r7,zero,2328 <__divsi3+0x60>
    2324:       0085c83a        sub     r2,zero,r2
    2328:       f800283a        ret
    232c:       014bc83a        sub     r5,zero,r5
    2330:       39c0005c        xori    r7,r7,1
    2334:       003fe706        br      22d4 <__divsi3+0xc>
    2338:       0109c83a        sub     r4,zero,r4
    233c:       01c00044        movi    r7,1
    2340:       003fe306        br      22d0 <__divsi3+0x8>
    2344:       00c00044        movi    r3,1
    2348:       003fee06        br      2304 <__divsi3+0x3c>
```

Nios II Reference Guide

Table 60. Hardware Multiply and Divide Details for the Nios II/f Core

ALU Option	Hardware Details	Cycles per Instruction	Result Latency Cycles	Supported Instructions
No hardware multiply or divide	Multiply and divide instructions generate an exception	–	–	None
Logic elements	ALU includes 32 x 4-bit multiplier	11	+2	`mul, muli`
				continued...
32-bit multiplier	ALU includes 32 x 32-bit multiplier	1	+2	`mul, muli, mulxss, mulxsu, mulxuu`
16-bit multiplier	ALU includes 3 16 x 16-bit multiplier	1	+2	`mul, muli`
16-bit multiplier	ALU includes 4 16 x 16-bit multiplier	2	+2	`mul, muli, mulxss, mulxsu, mulxuu`
Hardware divide	ALU includes SRT Radix-2 divide circuit	35	+2	`div, divu`

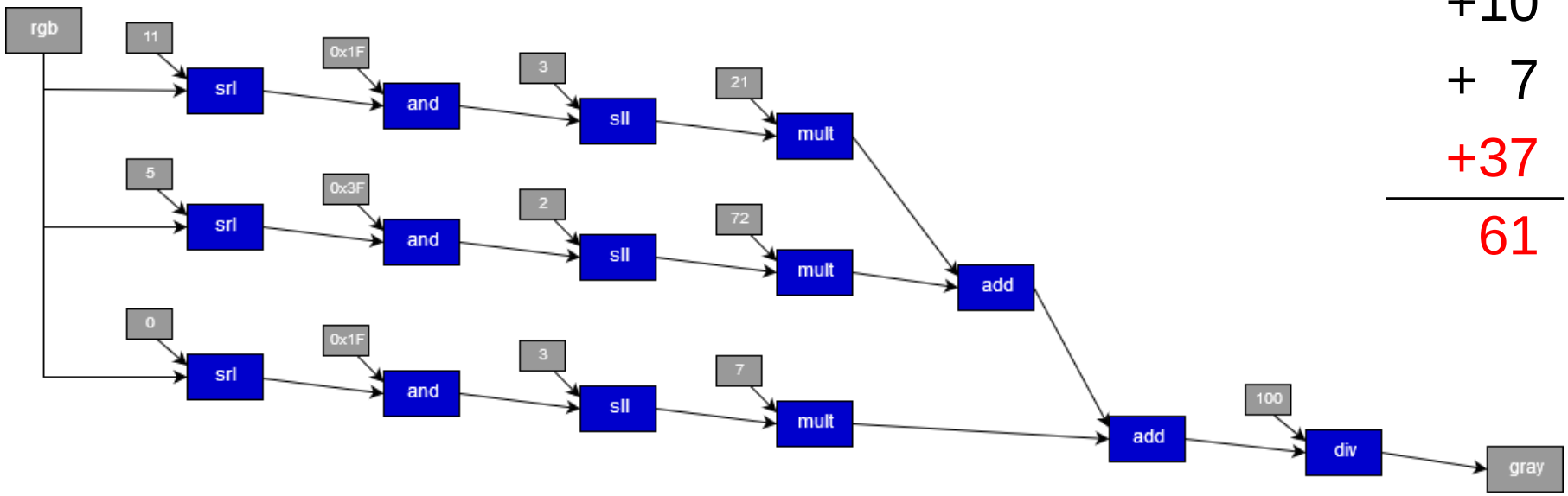
Data Flow Graph (DFG)

Grayscale ~= 0.33*R + 0.56*G + 0.11*B

Operations:

7
+10
+ 7
+37

61



Algorithm optimization

- Do we need a lot of precision in the division?

Algorithm optimization

- Do we need a lot of precision in the division?

- Not really, we could substitute the division by a bit shift

Algorithm optimization

- Do we need a lot of precision in the division?
- Not really, we could substitute the division by a bit shift
- It is important to:
  - Check constant values
  - Check Underflow and Overflow conditions
  - Minimize the introduced error

Algorithm optimization

- Do we need a lot of precision in the division?
- Not really, we could substitute the division by a bit shift
- It is important to:
  - Check constant values
  - Check Underflow and Overflow conditions
  - Minimize the introduced error
- By removing the division, you can gain from ~800 to ~580 cycles/pixel (1.38x)

Algorithm optimization

- With this optimization, what happened with the players: area, performance, power consumption, time-to-market?

Algorithm optimization

- With this optimization, what happened with the players: area, performance, power consumption, time-to-market?

- Can we optimize more?

Nios II Reference Guide

Table 60. Hardware Multiply and Divide Details for the Nios II/f Core

ALU Option	Hardware Details	Cycles per Instruction	Result Latency Cycles	Supported Instructions
No hardware multiply or divide	Multiply and divide instructions generate an exception	–	–	None
Logic elements	ALU includes 32 x 4-bit multiplier	11	+2	`mul, muli`
				continued...
32-bit multiplier	ALU includes 32 x 32-bit multiplier	1	+2	`mul, muli, mulxss, mulxsu, mulxuu`
16-bit multiplier	ALU includes 3 16 x 16-bit multiplier	1	+2	`mul, muli`
16-bit multiplier	ALU includes 4 16 x 16-bit multiplier	2	+2	`mul, muli, mulxss, mulxsu, mulxuu`
Hardware divide	ALU includes SRT Radix-2 divide circuit	35	+2	`div, divu`

Algorithm optimization

- With this optimization, what happened with the players: area, performance, power consumption, time-to-market?

- Can we optimize more?
  - Using optimized multipliers?

Algorithm optimization

- With this optimization, what happened with the players: area, performance, power consumption, time-to-market?

- Can we optimize more?
  - Using optimized multipliers?
  - Other parts of the code?

Other optimizations?

Grayscale ~= 0.33*R + 0.56*G + 0.11*B

```
void conv_grayscale(void *picture,
                    int width,
                    int height) {
    int x,y,gray;
    unsigned short *pixels = (unsigned short *)picture , rgb;
    grayscale_width = width;
    grayscape_height = height;
    if (grayscale_array != NULL)
        free(grayscale_array);
    grayscale_array = (unsigned char *) malloc(width*height);
    for (y = 0 ; y < height ; y++) {
        for (x = 0 ; x < width ; x++) {
            rgb = pixels[y*width+x];
            gray = (((rgb>>11)&0x1F)<<3)*21; // red part
            gray += (((rgb>>5)&0x3F)<<2)*72; // green part
            gray += (((rgb>>0)&0x1F)<<3)*7; // blue part
            gray /= 100;
            IOWR_8DIRECT(grayscale_array,y*width+x,gray);
        }
    }
}
```

~800 → ~480 cycles/pixel

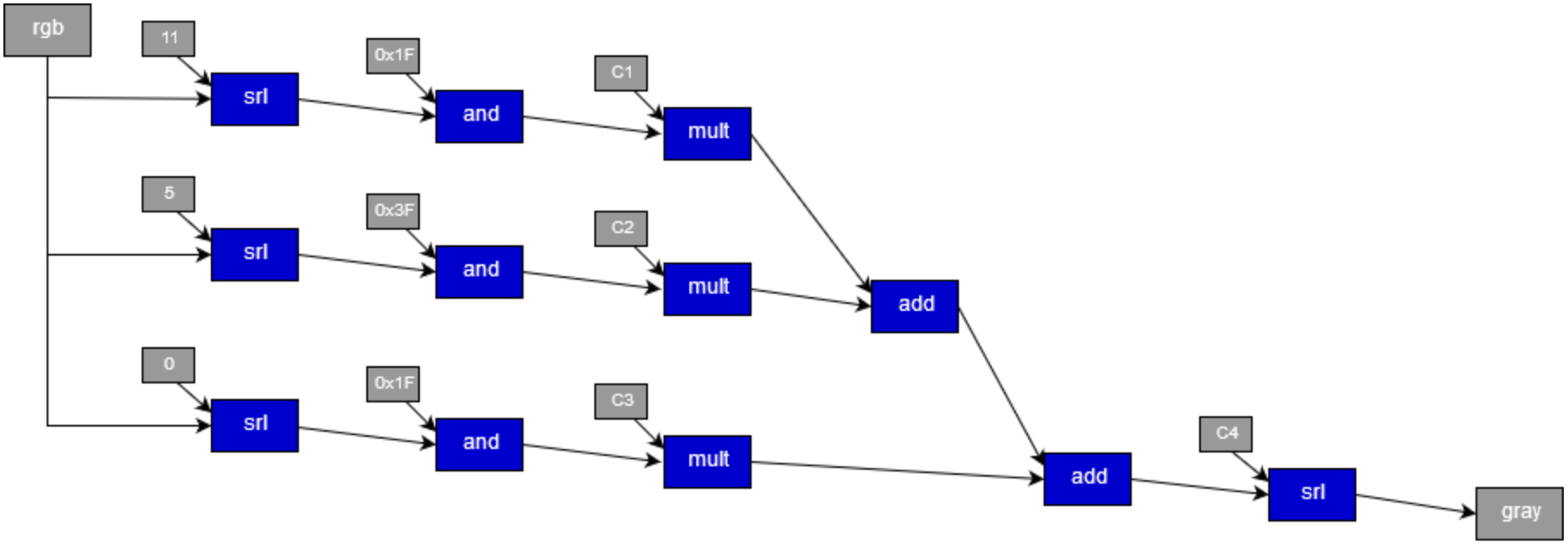
Algorithm optimization

- With this optimization, what happened with the players: area, performance, power consumption, time-to-market?

- Can we optimize more?
  - Using optimized multipliers?
  - Other parts of the code?
  - Anything else?

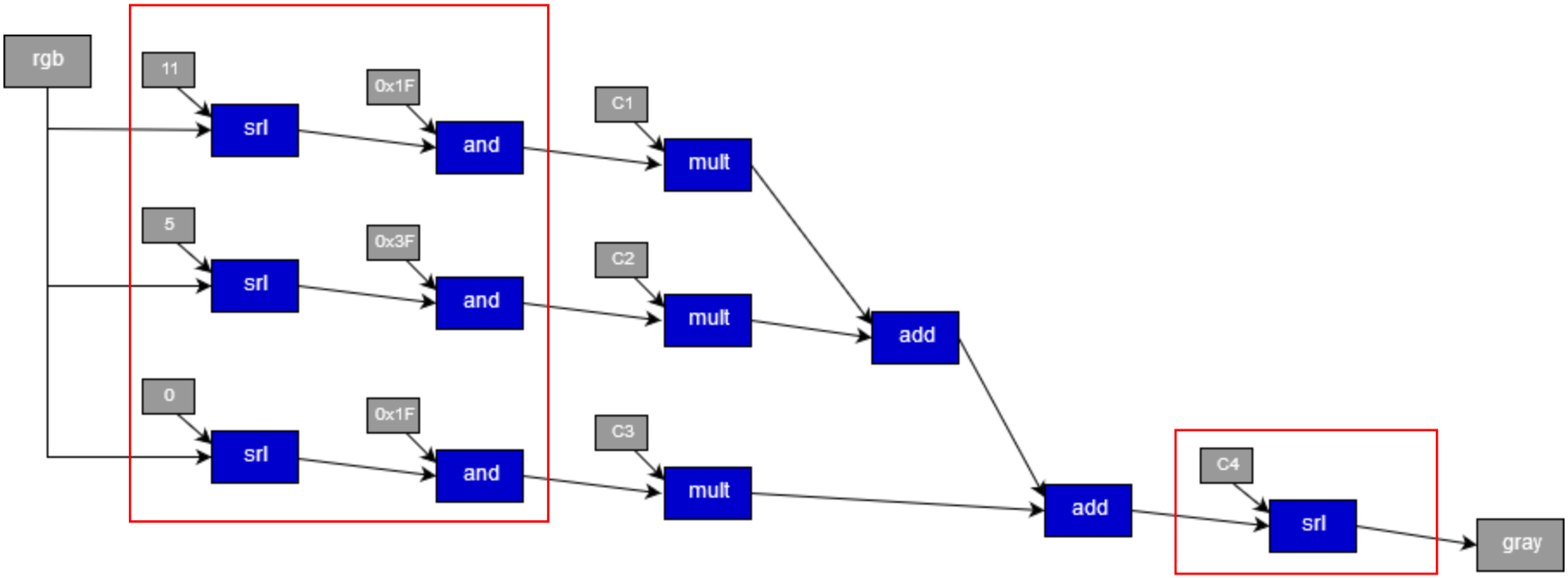
Algorithm optimization

- How would you implement it in HW?



Algorithm optimization

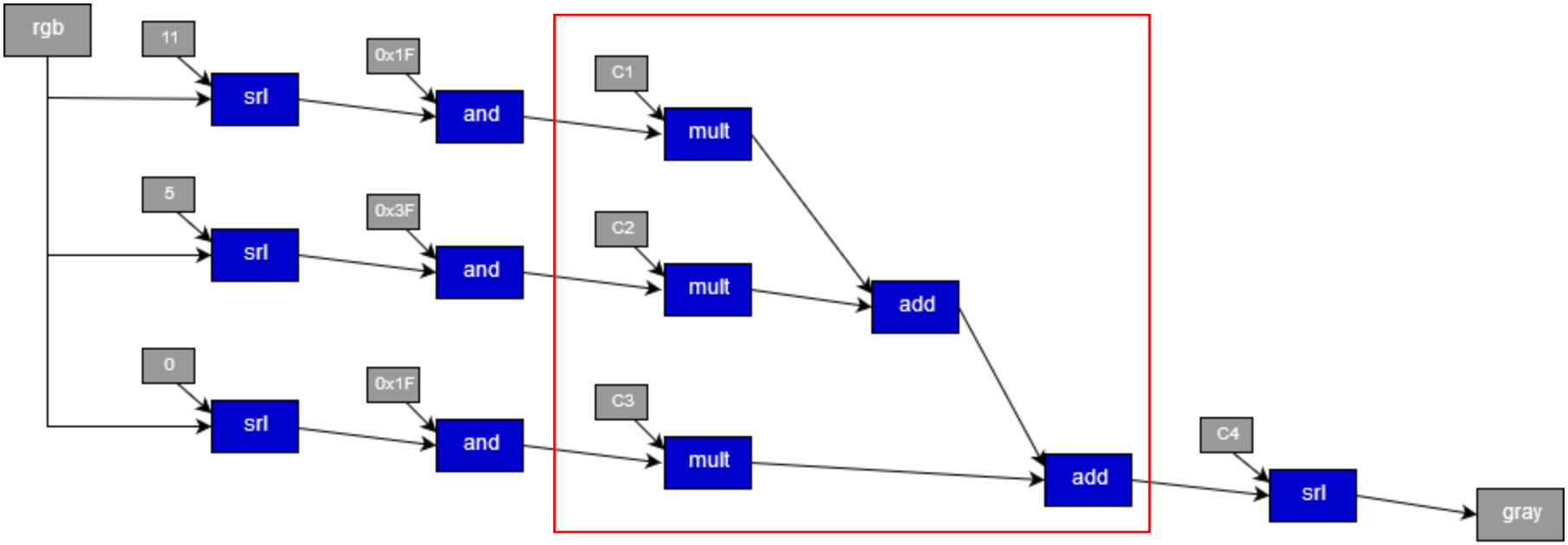
- How would you implement it in HW?



- Shifting and slicing

Algorithm optimization

- How would you implement it in HW?



- Shifting and slicing
- Multipliers and adders

Algorithm optimization

- How would you implement it in HW?
  - Shifting and slicing
  - Multipliers and adders
  - Parallelism

Algorithm optimization

- How would you implement it in HW?
  - Shifting and slicing
  - Multipliers and adders
  - Parallelism

- In SW:
  - 1 LOAD + 1 OPERATION + 1 STORE
  - + loop control

Algorithm optimization

- How would you implement it in HW?
  - Shifting and slicing
  - Multipliers and adders
  - Parallelism

- In SW:
  - 1 LOAD + 1 OPERATION + 1 STORE
  - + loop control

- What happened with the players: area, performance, power consumption, time-to-market?