6. Prototyping new instructions

- Motivation

- Introduction to motion estimation and SAD
- Senior assembly code for SAD
- New instruction selection for Senior

- Instruction set simulator basics

- Examples

We are considering implementing new instructions…

How much speedup in a real algorithm?

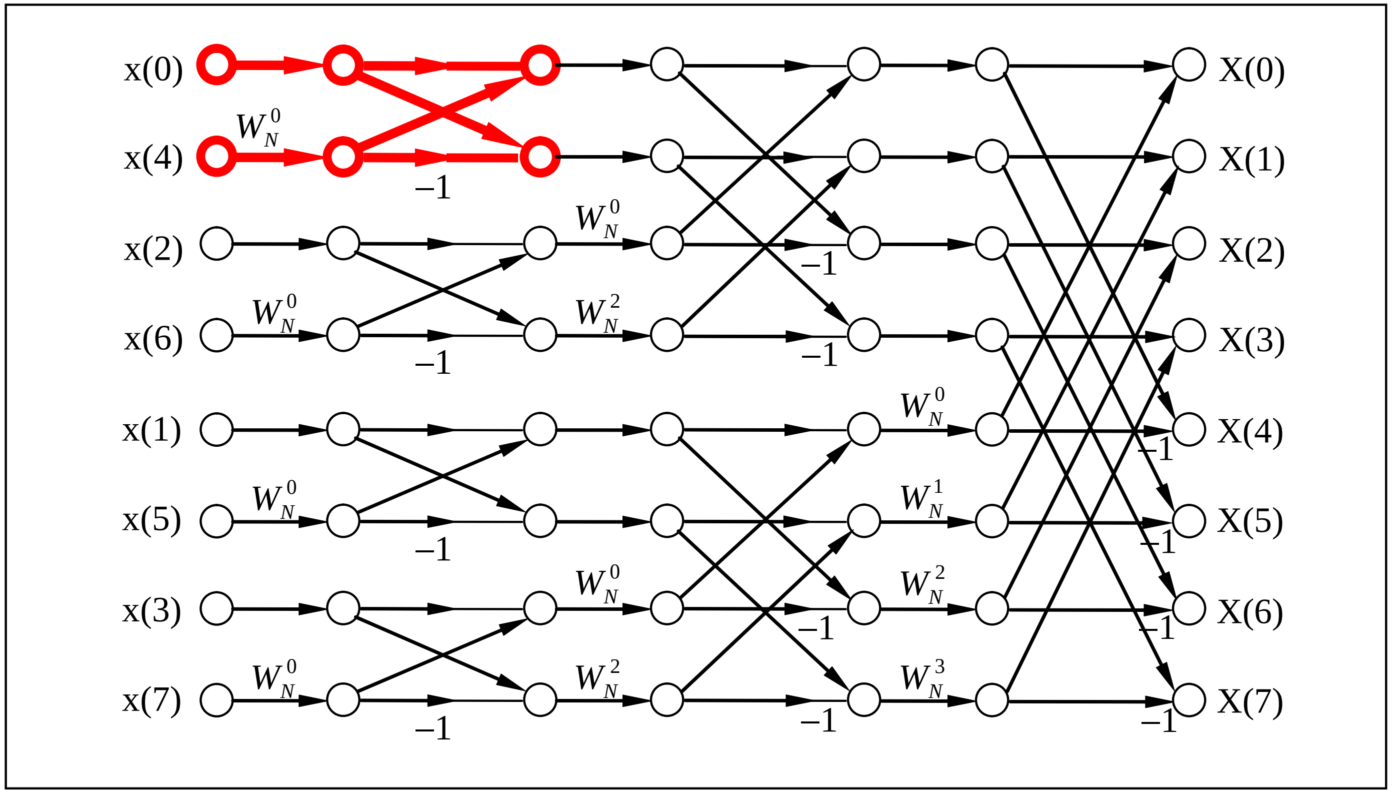
How much does it cost?

Other aspects:
Energy efficiency?
Memory usage?
Register pressure?
Compilation?
…
…

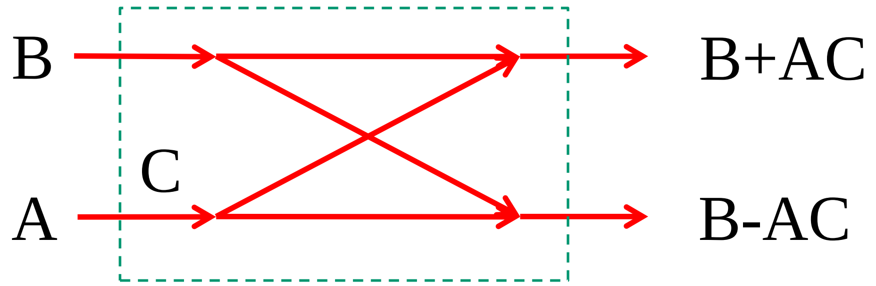
Accelerating an Application by adding new instructions

- Identify kernel components (profiling)
- Investigate if kernel can be accelerated at a reasonable hardware cost

Example - Accelerating an FFT

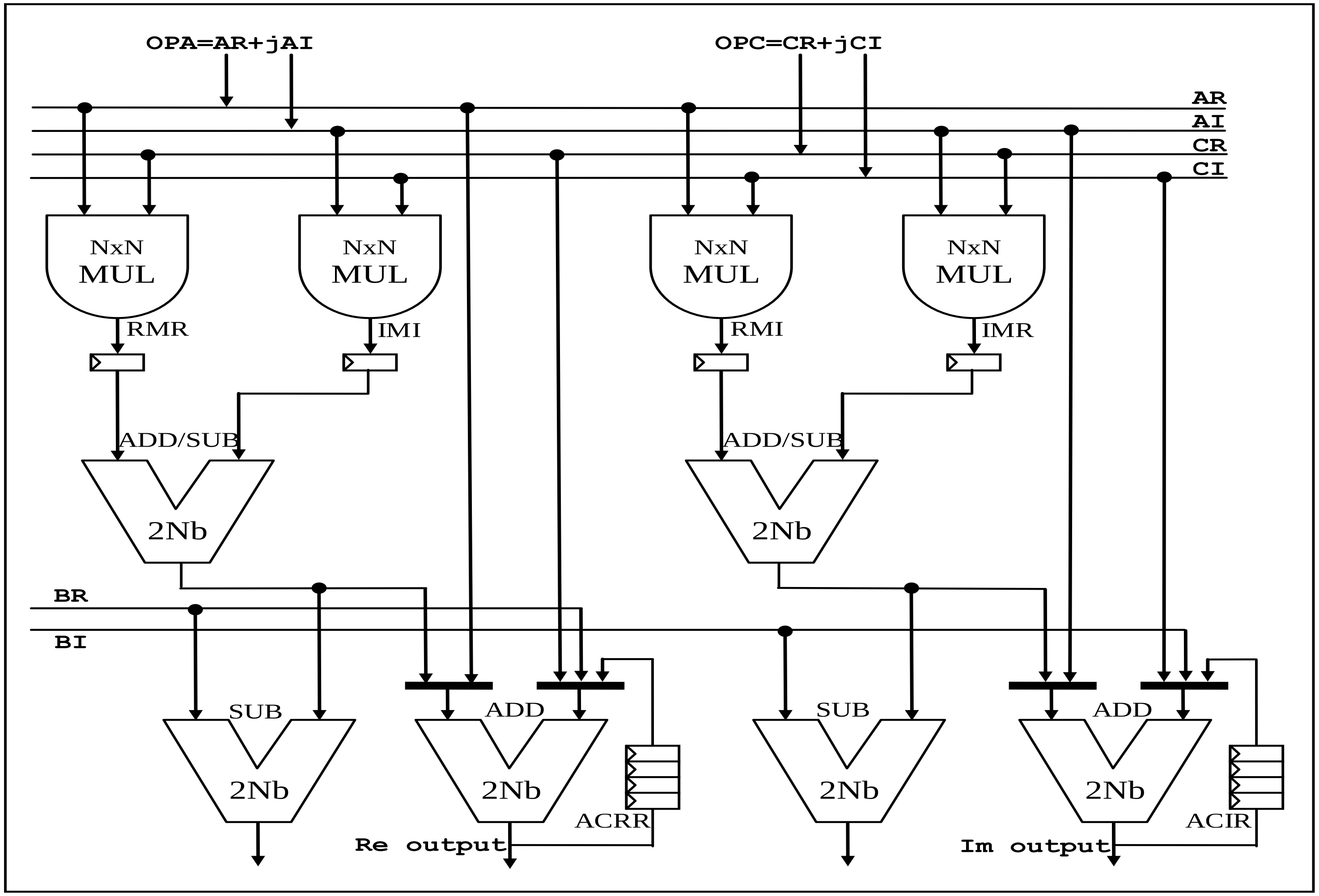


Implementing Complex Butterfly



- Behavioral description:
  - Output1Real = BR + (AR*CR – AI*CI);
  - Output1Imag = BI + (AR*CI + AI*CR);
  - Output2Real = BR – (AR*CR - AI*CI);
  - Output2Imag = BI – (AR*CI + AI*CR);
- More than 10 instructions using MAC
  - 4 MUL, 6 ADD
- Check how many butterflies to be computed ?
- Huge runtime cost ?
- What kind of hardware do we need to reduce the runtime cost to 1 instruction / butterfly ?

Acceleration by CMAC



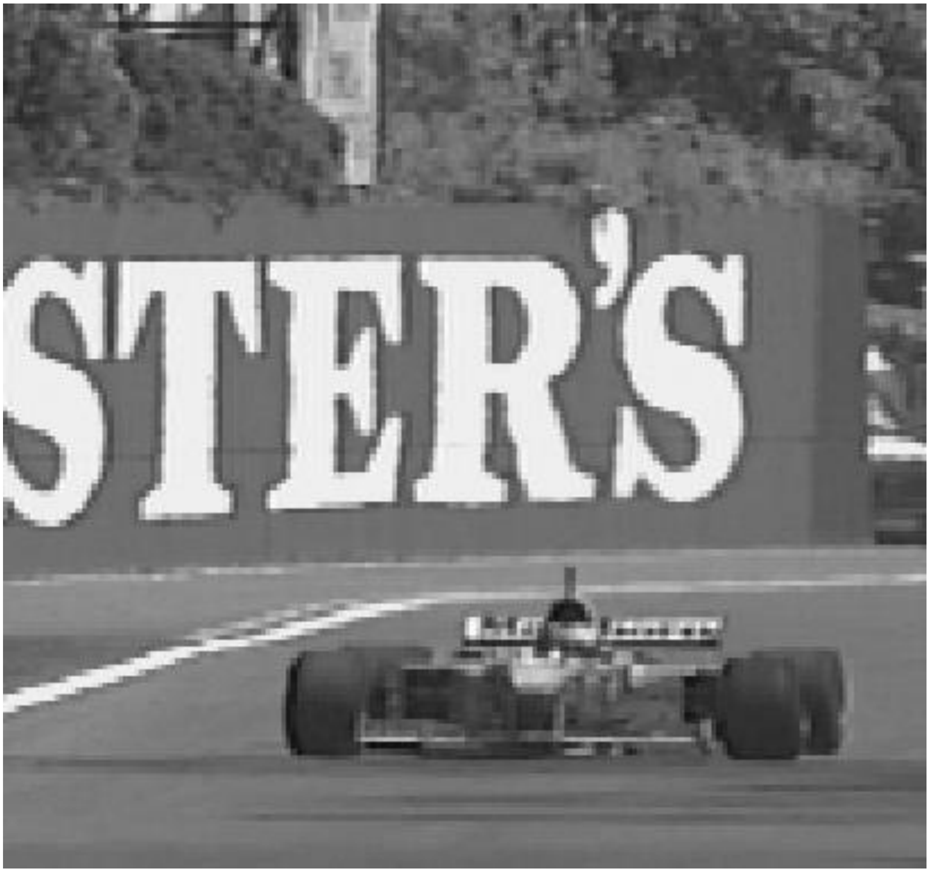
Other possible ASIP extensions

- Bit manipulation instructions
- Cryptographic/security
- Memory copying
- Vector/Matrix manipulation
- etc…

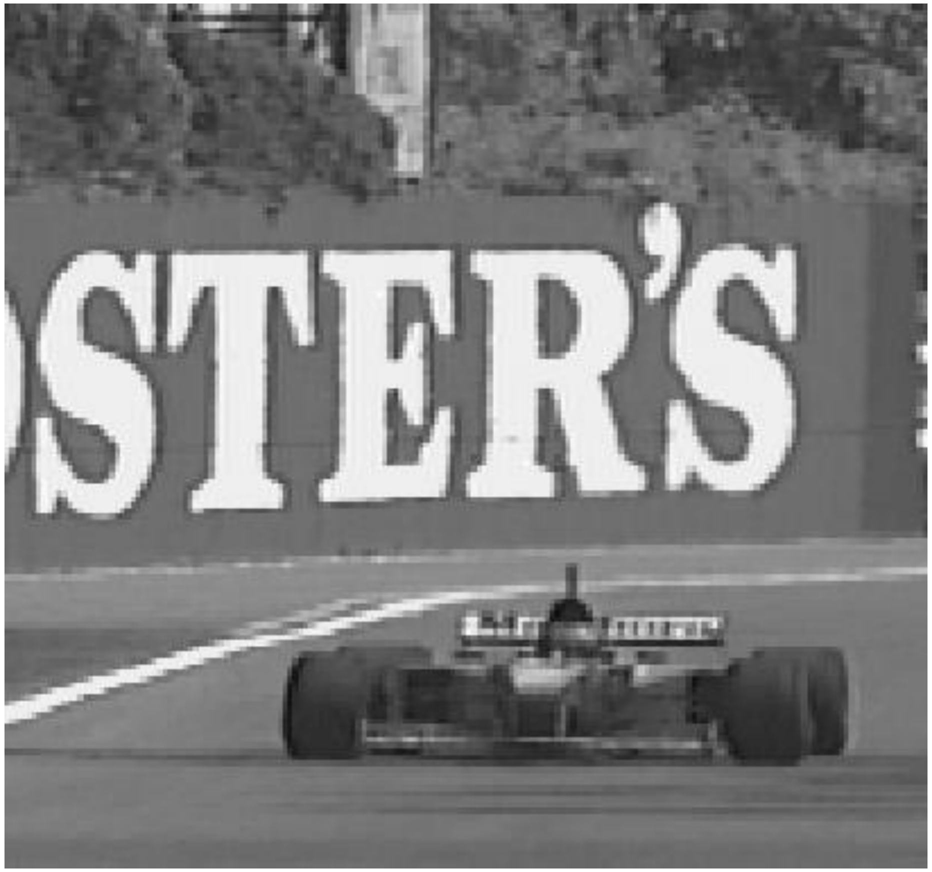
Our Application in Lab-4 (Motion Estimation)

- The intuition - Simple video encoder
  - Encode first image as a JPEG image
  - Calculate the difference between the current image frame and the previous image frame.
  - JPEG encode the difference

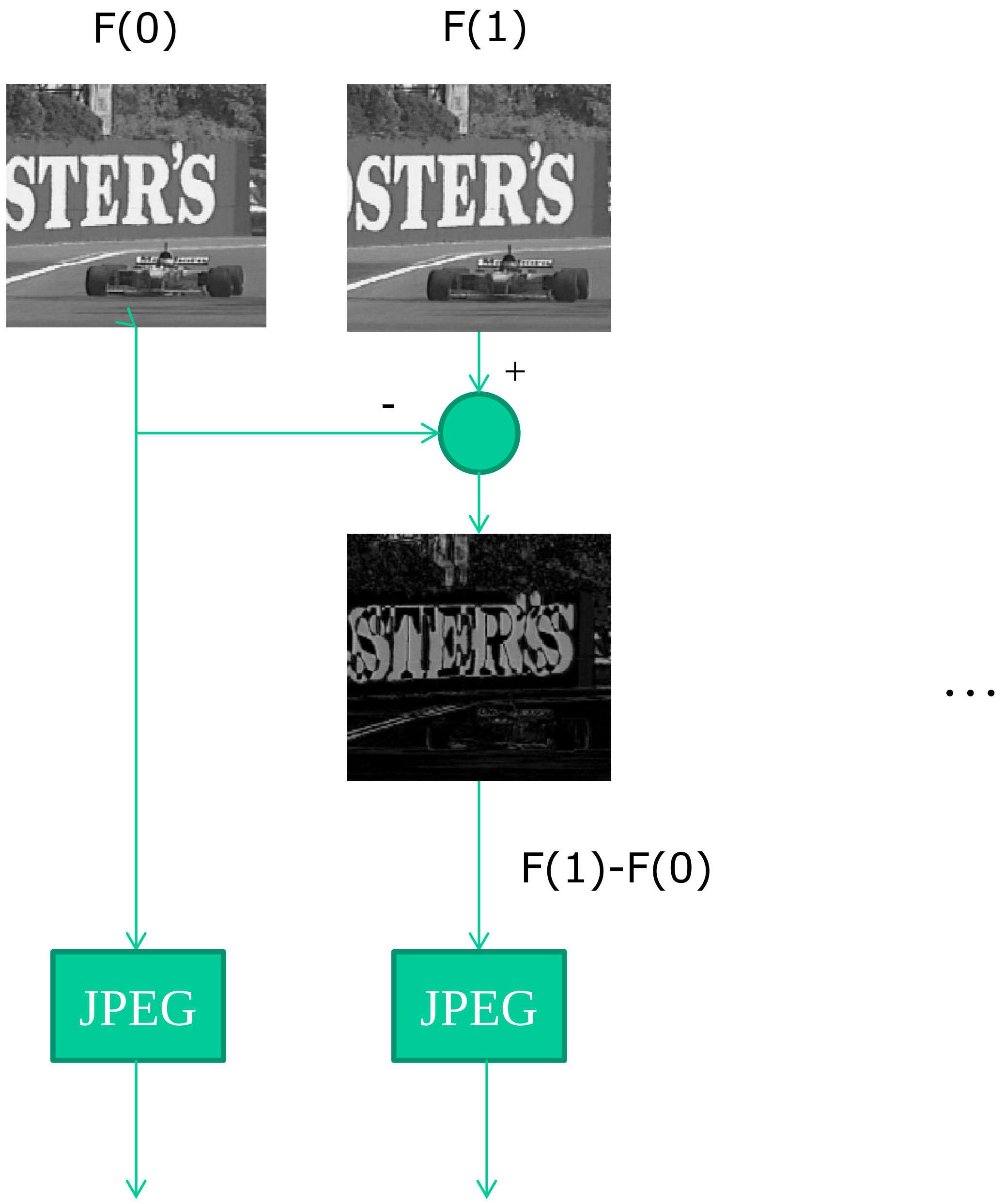
Sample Video Sequence



F(0)



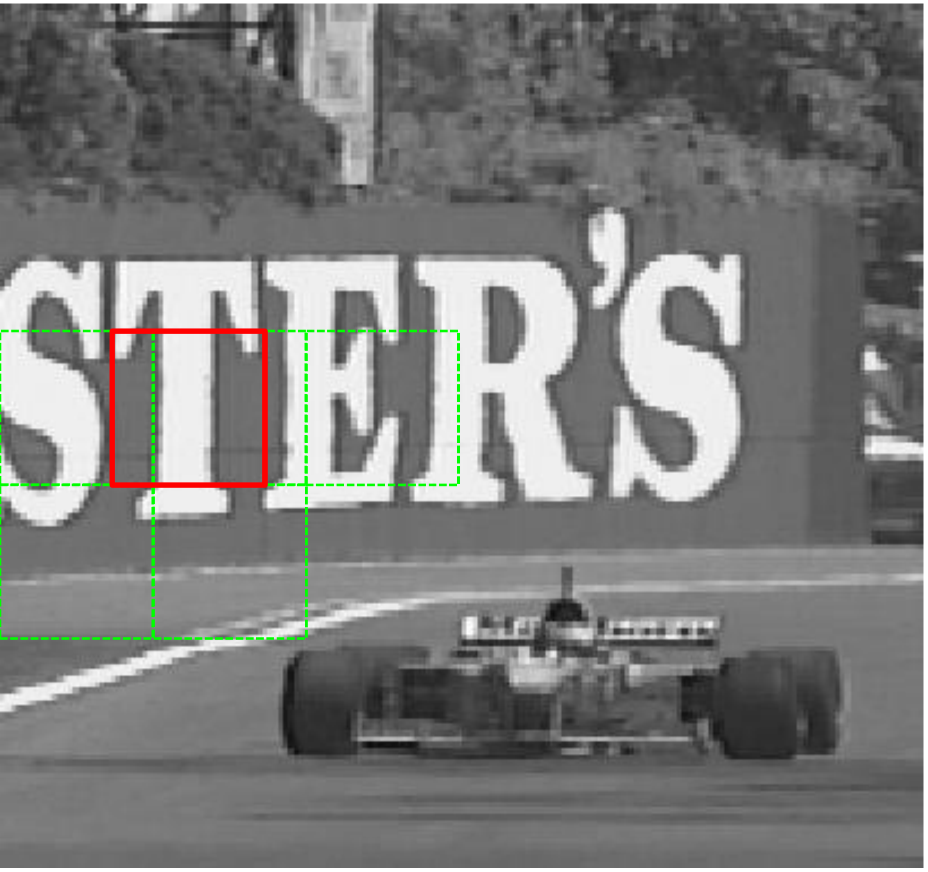
F(1)

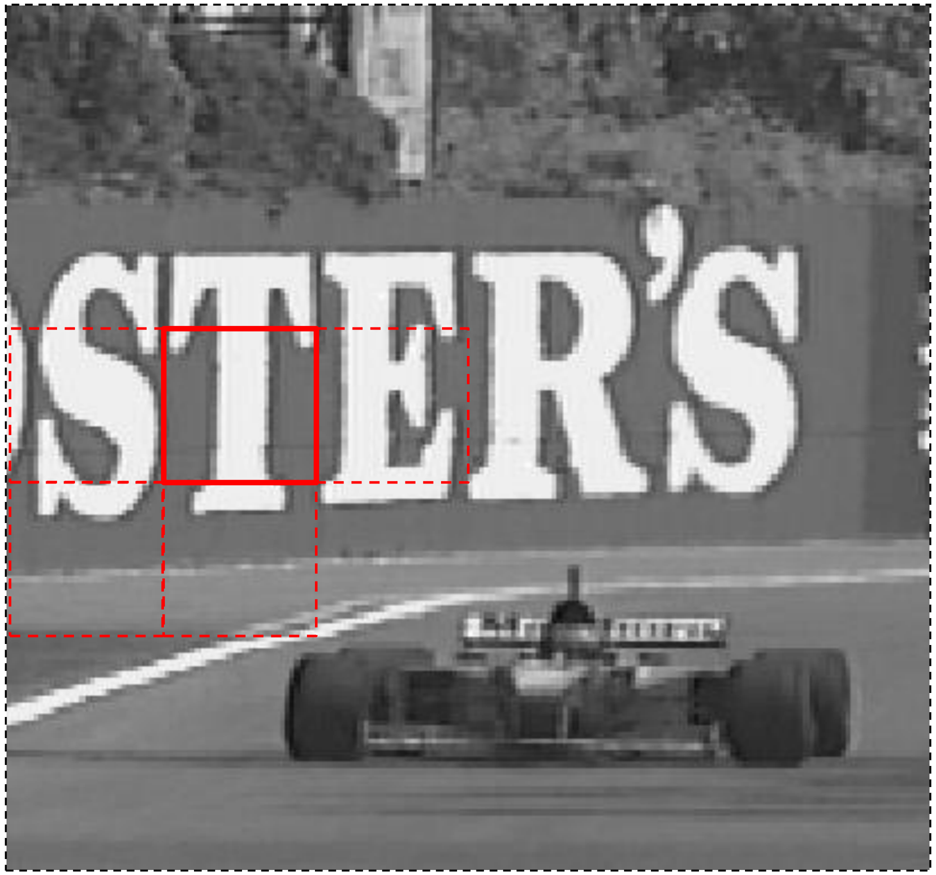

F(0)
F(1)
+
-
F(1)-F(0)
JPEG
JPEG
...

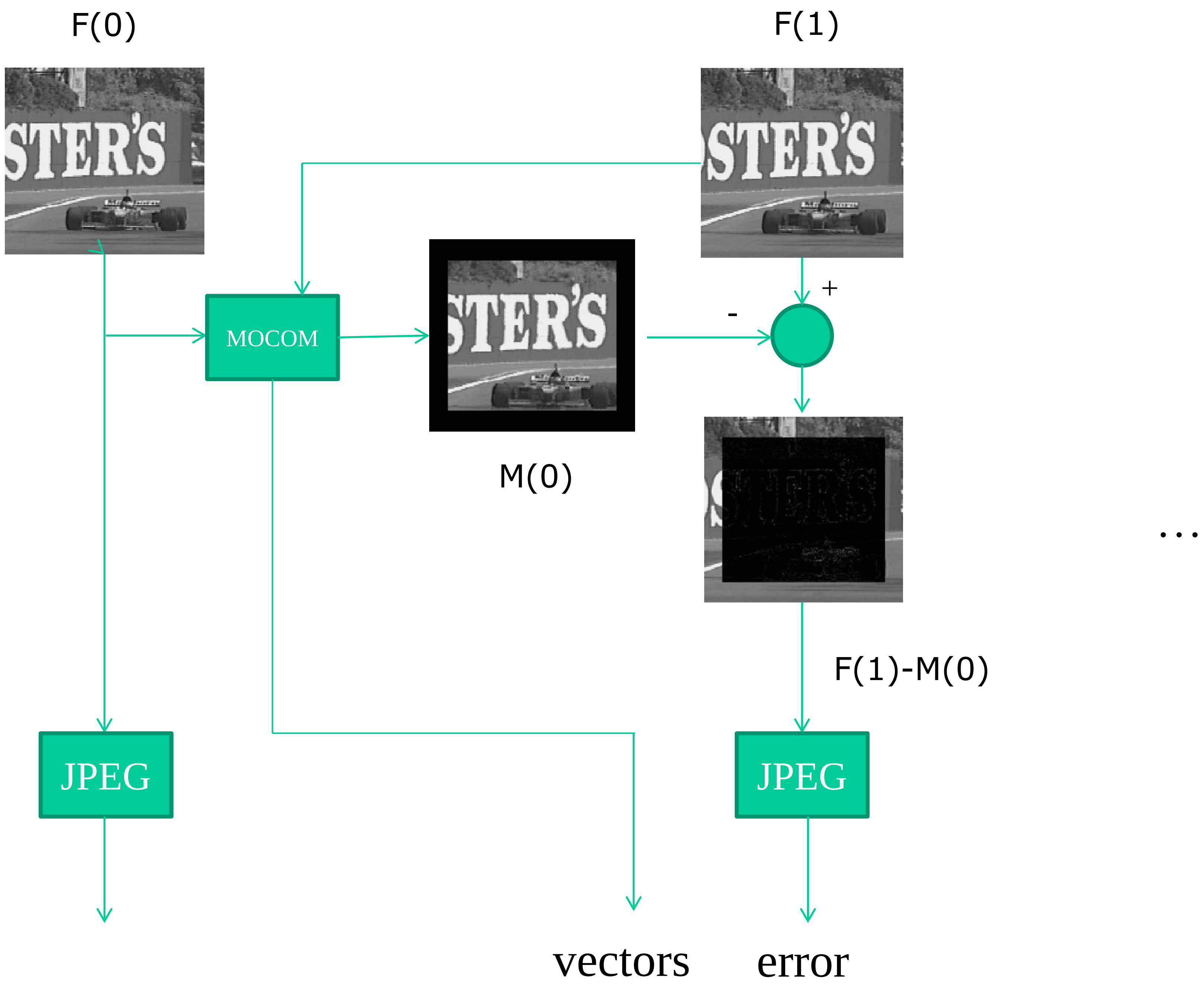
Our Application (Motion Estimation)

- More advanced video encoder
  - Encode first image as a JPEG image
  - Divide the second image into blocks
  - Find where each block is located in the first frame (motion estimation)
  - Encode motion information
  - Encode difference between motion compensated image and current image as a JPEG image

Motion estimation


F(n-1)

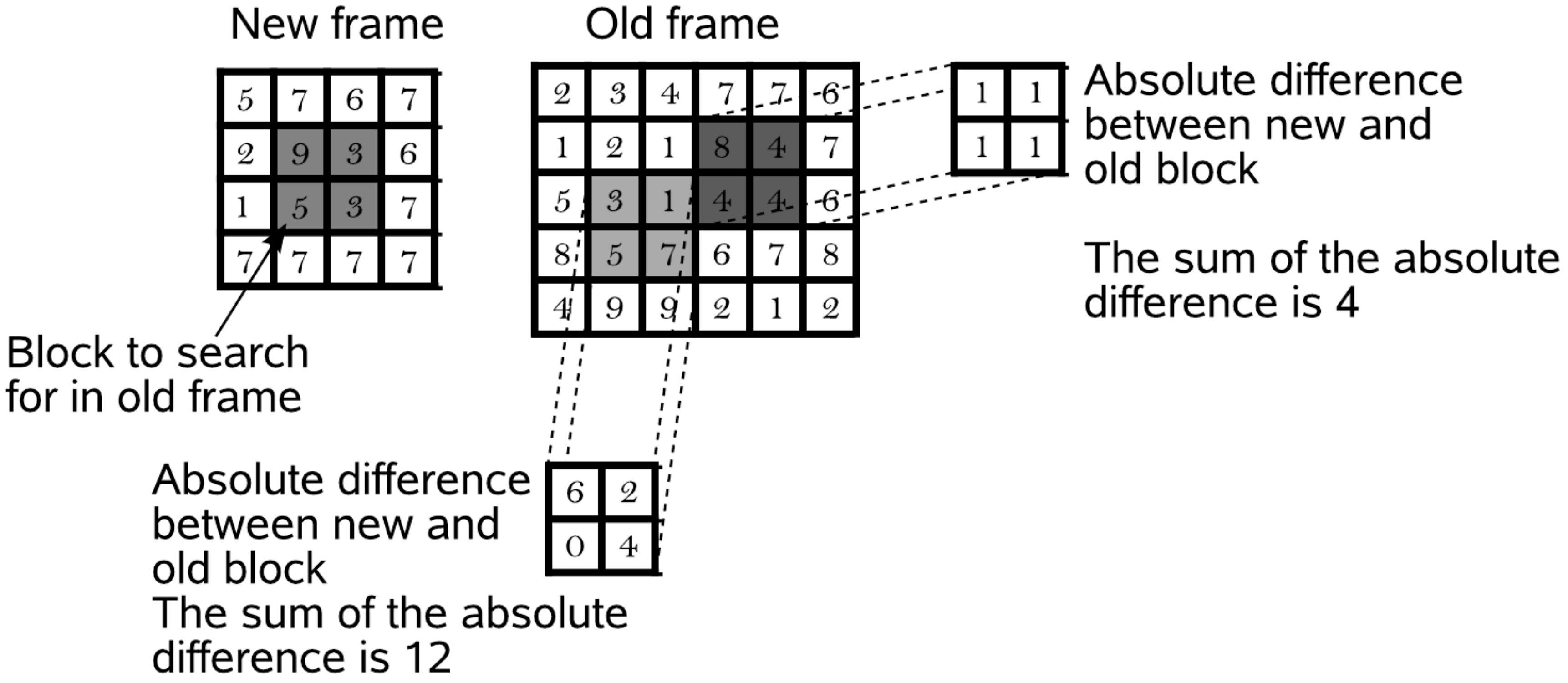

F(n)


F(0)
F(1)
MOCOM
+
-
M(0)
F(1)-M(0)
JPEG
JPEG
vectors
error
...

How to do Motion Estimation?

- For each block in the current image / frame, find the most similar looking block in the previous image
- What is the most similar looking block?
  - The block with the least difference
  - One metric: Sum of absolute difference (SAD)

Block Search using Sum of Absolute Difference (SAD)

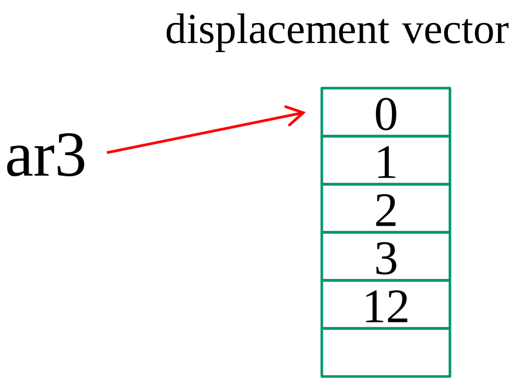

New frame
Old frame
5 7 6 7
2 9 3 6
1 5 3 7
7 7 7 7
2 3 4 7 7 6
1 2 1 8 4 7
5 3 1 4 4 6
8 5 7 6 7 8
4 9 9 2 1 2
1 1
1 1
6 2
0 4
Block to search for in old frame
Absolute difference between new and old block
The sum of the absolute difference is 4
Absolute difference between new and old block
The sum of the absolute difference is 12

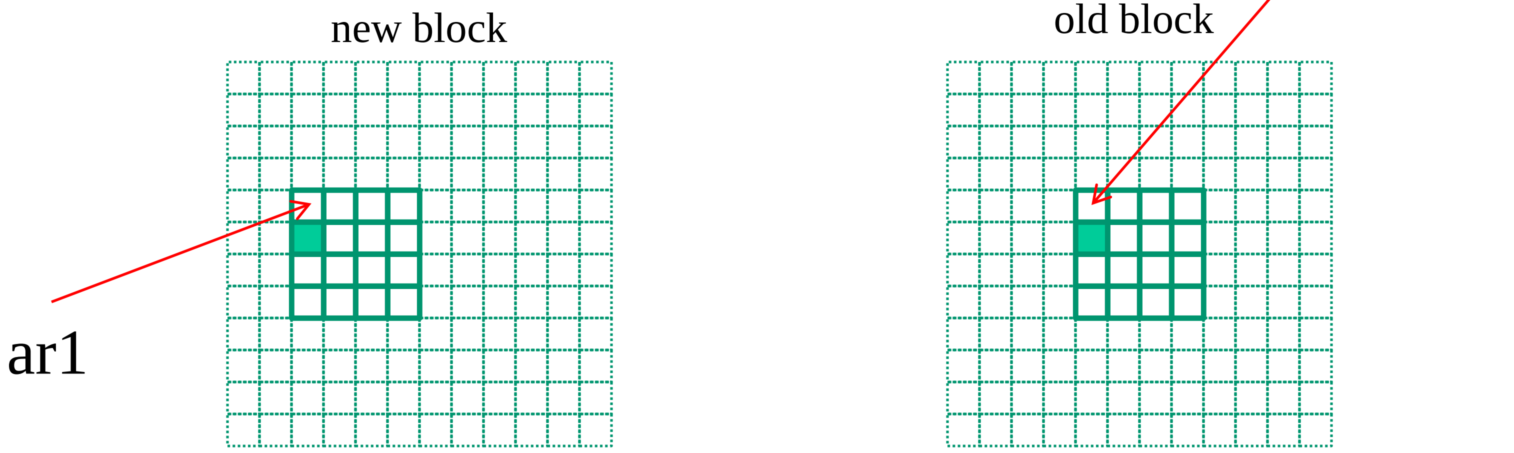
Pseudo code for Motion Estimation

```
for each block in the image{ // 4x4 blocks
    best_sad = Inf;
    for each candidate position{
           sad = compare_blocks(candidate_block, target_block);
           if (sad < best_sad) {
           best_sad = sad;
           best_block = candidate_block; }
    }
    output_position(best_block);
 }
compare_blocks(a,b){
    sum = 0;
    for each pixel p { // 16 pixels
           difference = a[p] - b[p];
           sum += abs(difference);
    }
    return sum;
}
```

Assembly Code for SAD Kernel

```
repeat sad_kernel_end,16
sad_kernel_start
   ld0 r0,(ar3++) ; Load displacement in image
   nop
   ld1 r1,(ar0,r0) ; Load pixel in new image
   ld0 r2,(ar1,r0) ; Load pixel in original image
   nop
   sub r1,r1,r2 ; Calculate difference
   abs r1,r1 ; Take absolute value
   add r4,r4,r1 ; Sum of absolute difference
sad_kernel_end
```





What to Accelerate Here?

- Could accelerate sub, abs
  - Absolute difference
- Could accelerate sub, abs, and add
  - SAD
- Could accelerate ld0 and ld1, sub, abs, and add
  - SAD with value loading
- Could accelerate ld0, ld0, ld1, sub, abs, and add
  - SAD with value loading and pixel offset
  - Would need dual port memory for mem0!
    - Probably not a good idea…

- Deterministic speedup
- Could be estimated without simulation

What about the Loop?

- Could do early abort if we have found a block which is obviously worse than the best block so far

- Data dependent speedup
- Hard to estimate without simulation

Instruction Set Simulators

- Program flow for an instruction set simulator
  - While there are no errors:
    - Update PC
    - Load instruction and decode it
    - Execute instruction
  - If error: Show debug information

- How to model pipeline effects?

Pipeline Accurate Simulation

- A pipeline accurate simulator is cumbersome to write and verify
  - ld0   r0,(ar3++)
  - add   r5,r5,r0; Not allowed due to the pipeline
- We would like to check for this without too much effort…

Emulating Pipeline Effects: the easy way

- uint16_t rf[32];
  - The register file
- int rf_busy[32];
  - If 0, access ok
  - If not 0, access is not ok
  - When updating the value of a register, update rf_busy[] to an appropriate value depending on how the pipeline looks like in the processor

Example: `ld0    r0,(ar3++) -> set rf_busy[0] = 2;`

Modified Simulation Flow

- While there are no errors:
  - Decrement rf_busy[] counters by 1
  - Update PC
  - Load instruction and decode it
  - Execute instruction
- If error: Show debug information

Updating PC

- Need to take care about:
  - Jumps
  - Delay slots
  - Loops
- You don’t have to modify this in the lab, but please try to understand the code anyway

Decoding Instructions

```
/* Check top bits for the type of instruction */
switch(insn & 0xc0000000) {
case 0x00000000: insn_moveloadstore(insn);
   // This is a move, load or store instruction
                                 break;
case 0x40000000: insn_type01(insn);
   // insn_type01() will figure out what this is
                                 break;
case 0x80000000: insn_pfc(insn);
   // Program Flow Control instruction
                                 break;
case 0xc0000000: insn_accelerated(insn);
                                 break;
}
```

Executing Instructions

```
opa = get_opa(insn);
opb = get_opb(insn);

switch (insn & 0x07800000) {  // Look at the instruction word
  case 0x00000000: result = opa & opb; break; // andn
  case 0x01000000: result = opa | opb; break; // orn
  case 0x02000000: result = opa ^ opb; break; // xorn
  default: sim_warning("Unimplemented logic instruction");
           return;
}

if(insn & 0x00800000) {
  update_flags(result);
}

set_reg(get_dreg(insn),result,0); // set_reg updates rf_busy!
```

Verification

- For lab-4, the result of “sad.asm” with accelerated instructions should be identical to the result without accelerated instructions
  - (This might not be true for all ASIP instructions)

Is it fast enough?

- You will gain a substantial speedup
- Is it worth the extra hardware cost?

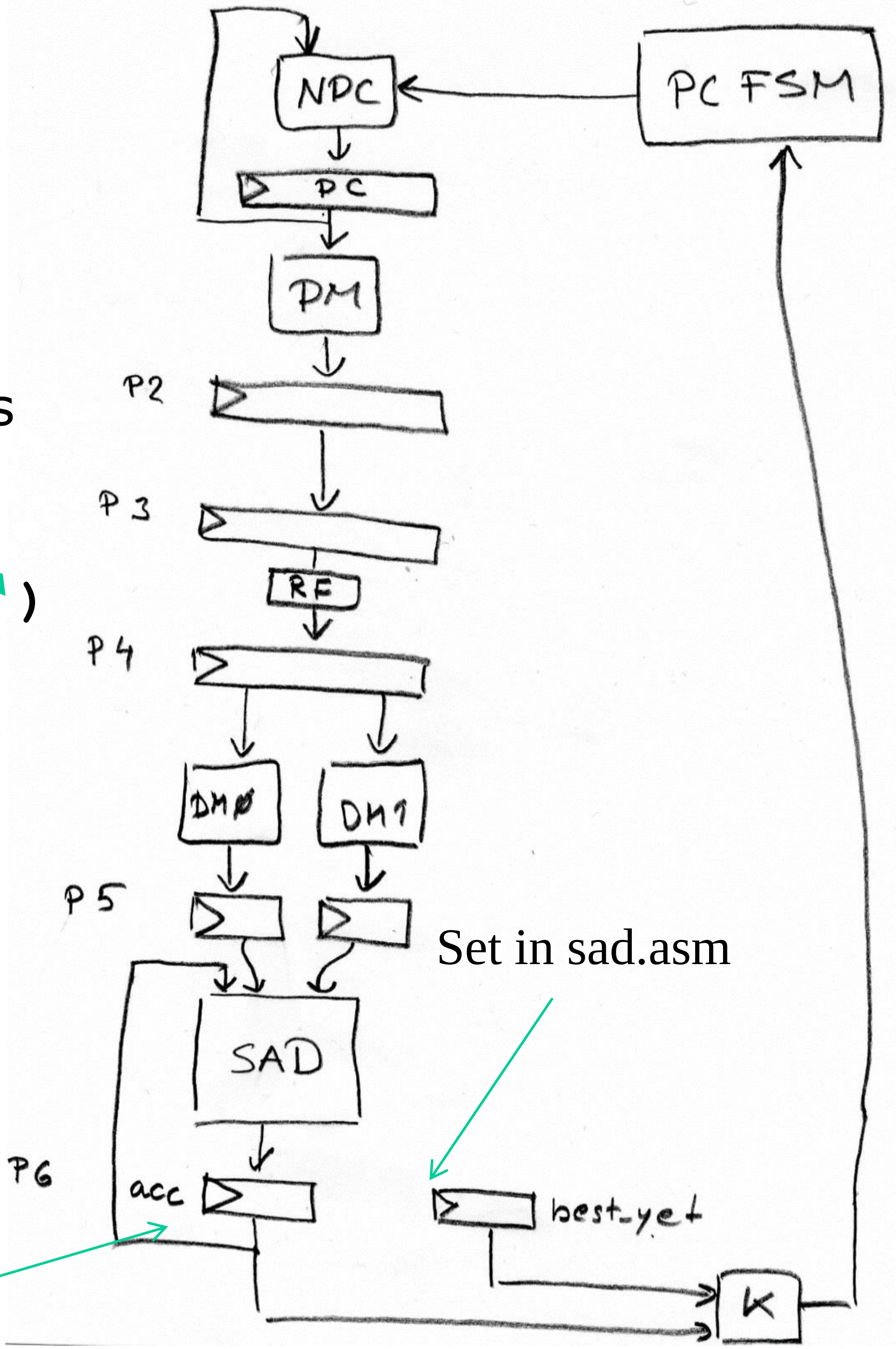
Counting clockcycles

```
accel_sad r4,r0
…
sub r0,r4,r5
```

Nr of nops

```
set_reg(get_dreg(insn),val, )

(will set rf_busy for you)
```



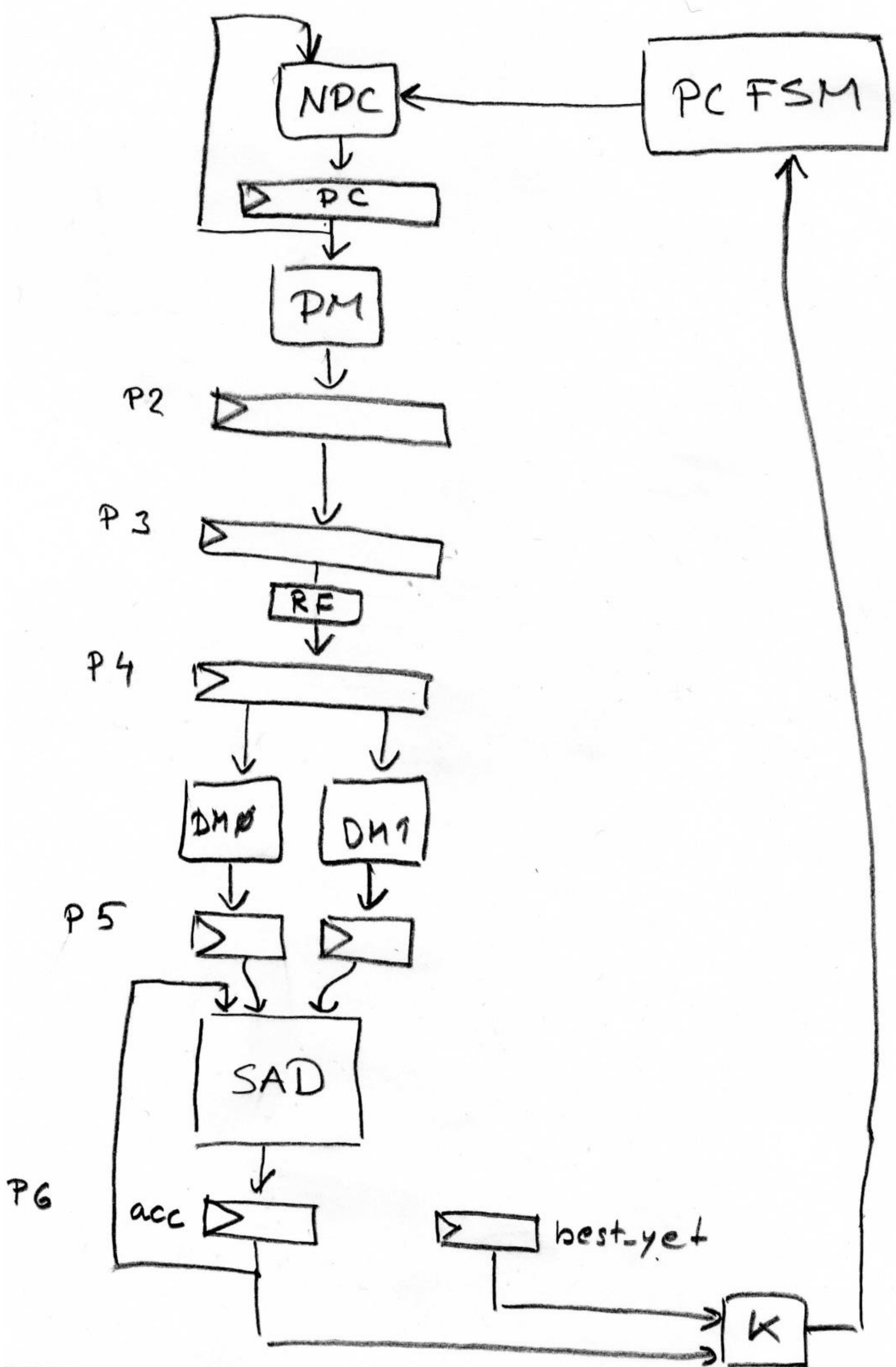
Set in sad.asm

cleared in sad.asm

Counting clockcycles

pipeline delay

`repeat_sad_stop();`



Exercises!

The following function should be implemented on the Senior processor.

```
// matrixptr is in r0, vectorptr in r1 and resultptr in r2
function rotate_vector(matrixptr, vectorptr, resultptr)
    A = dm0[matrixptr]
    B = dm0[matrixptr+1]
    C = dm0[matrixptr+2]
    D = dm0[matrixptr+3]
    repeat 50
        X = dm0[vectorptr++]
        Y = dm0[vectorptr++]
        ROTATEDX = A*X+B*Y
        ROTATEDY = C*X+D*Y
        dm1[resultptr++] = ROTATEDX
        dm1[resultptr++] = ROTATEDY
    endrepeat
endfunction
```

Constraints:

- `matrixptr`, `vectorptr`, and `resultptr` are available in general purpose registers when the function is called.
- `A`, `B`, `C`, `D`, `X`, `Y`, `ROTATEDX`, and `ROTATEDY` are 16 bit fractional numbers.
- You don't need to worry about saturation and rounding in this exercise.
- You may not add any ports or change the width of either DM0 or DM1.

Tasks:

Your task is to create a special unit present in the pipeline of the Senior processor so that the function `ROTATE_VECTOR` can be executed in less than 130 clock cycles. You will also need to select suitable instructions to implement this function. You can assume that the processor has the required program flow control and addressing modes required to support this function.

a) Select a set of new instructions that will allow you to execute `ROTATE_VECTOR` in less than 130 clock cycles and translate `ROTATE_VECTOR` into assembler. For each of your new instructions you also need to describe any AGU or memory operation that it may perform.

b) Draw a hardware schematic of your vector rotation unit. You should minimize the amount of multipliers. You may also use as many gates, multiplexers and adders as you want to (within reason).

c) Draw a control table for your hardware where you include all instructions that you selected in task a.

Note: In a real scenario you would probably reuse some hardware in the MAC unit for the vector rotation instruction(s). For simplicity reasons we ignore this fact in this exercise.

```
X = dm0[vectorptr++]
Y = dm0[vectorptr++]
ROTATEDX = A*X+B*Y
ROTATEDY = C*X+D*Y
dm1[resultptr++] = ROTATEDX
dm1[resultptr++] = ROTATEDY
```

DM0	ROT	DM1
newX=DM0		
Y=DM0, X=newX		
newX=DM0	ROTX=AX+BY	
Y=DM0, X=newX	ROTY=CX+DY	
newX=DM0	ROTX=AX+BY	DM1=ROTX
Y=DM0, X=newX	ROTY=CX+DY	DM1=ROTY
	ROTX=AX+BY	DM1=ROTX
	ROTY=CX+DY	DM1=ROTY
		DM1=ROTX
		DM1=ROTY

48

```
X = dm0[vectorptr++]
Y = dm0[vectorptr++]
ROTATEDX = A*X+B*Y
ROTATEDY = C*X+D*Y
dm1[resultptr++] = ROTATEDX
dm1[resultptr++] = ROTATEDY
```

DM0	ROT	DM1
newX=DM0		
Y=DM0, X=newX		
newX=DM0	ROTX=AX+BY	
Y=DM0, X=newX	ROTY=CX+DY	DM1=ROTX
newX=DM0	ROTX=AX+BY	DM1=ROTY
Y=DM0, X=newX	ROTY=CX+DY	DM1=ROTX
	ROTX=AX+BY	DM1=ROTY
	ROTY=CX+DY	DM1=ROTX
		DM1=ROTY

48

DM0	ROT	DM1
newX=DM0		
Y=DM0, X=newX		
newX=DM0	tmp0=AX+BY	
Y=DM0, X=newX	tmp1=CX+DY	DM1=tmp0
newX=DM0	tmp0=AX+BY	DM1=tmp1
	tmp1=CX+DY	DM1=tmp0
		DM1=tmp1

49

- **rot1:** newX = dm0[ar0++];
- **rot2:** Y = dm0[ar0++]; X = newX;
- **rot3:** tmp0 = OpA*X+OpB*Y; newX = dm0[ar0++];
- **rot4:** dm1[ar1++] = tmp0; tmp1 = OpA*X+OpB*Y; Y = dm0[ar0++]; X = newX;
- **rot5:** dm1[ar1++] = tmp1; tmp0 = OpA*X+OpB*Y; newX = dm0[ar0++];
- **rot6:** dm1[ar1++] = tmp0; tmp1 = OpA*X+OpB*Y;
- **rot7:** dm1[ar1++] = tmp1;

The following assembly program will work:

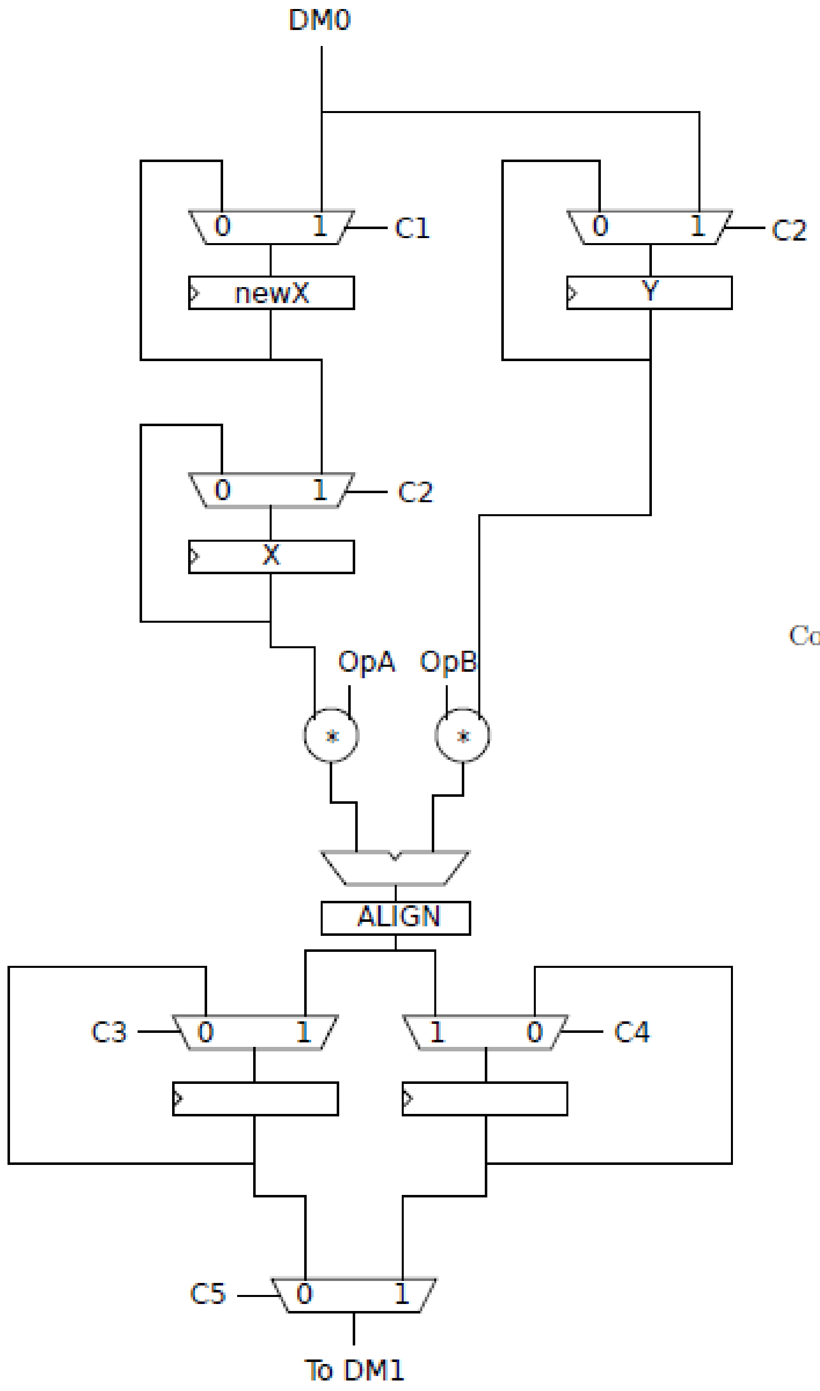
```
rotate_vector:
    ld   r4,dm0[r0]
    ld   r5,dm0[r0+1]
    ld   r6,dm0[r0+2]
    ld   r7,dm0[r0+3]
    move ar0,r1
    move ar1,r2

    rot1   ; Prologue
    rot2
    rot3   r4,r5

    repeat 49, endloop
    rot4   r6,r7
    rot5   r4,r5
endloop:

    rot6   r6,r7 ; Epilogue
    rot7

    ret
```



Content of ALIGN: `assign out[15:0] = in[30:15];`

Insn	C1	C2	C3	C4	C5
rot1	1	0	0	0	-
rot2	0	1	0	0	-
rot3	1	0	1	0	-
rot4	0	1	0	1	0
rot5	1	0	1	0	1
rot6	0	0	0	1	0
rot7	0	0	0	0	1

2

The function FIR_3 is responsible for 70% of the time in a hypothetical application running on the Senior processor. Your task is to evaluate the hardware cost of speeding up this function by designing a custom instruction that is able to execute FIR_3 in one clock cycle. Additionally, it is necessary to initialize the values used by the FIR_3 function by using the INITFIR_3 function. However, it is expected that the FIR_3 function will be executed around 1000 times as often as the INITFIR_3 function. This means that the INITFIR_3 function does not need to execute quickly.

```
function FIR_3()
    samples[2] = samples[1]
    samples[1] = samples[0]
    samples[0] = dm0[inputptr]
    inputptr = inputptr + 1

    tmp = 0
    tmp = tmp + samples[0] * coefficients[0]
    tmp = tmp + samples[1] * coefficients[1]
    tmp = tmp + samples[2] * coefficients[2]

    dm1[outputptr] = SATURATE(tmp)
    outputptr = outputptr + 1
endfunction


function INITFIR_3(val1, val2, val3, val4, val5)
    samples[0] = 0
    samples[1] = 0
    samples[2] = 0
    inputptr = val1
    outputptr = val2
    coefficients[0] = val3
    coefficients[1] = val4
    coefficients[2] = val5
endfunction
```

Constraints:

- Function parameters are passed in general purpose registers
- `samples` contains 16 bit values in signed integer format
- `coefficients` contains 16 bit values in signed integer format
- The `tmp` variable has a suitable number of guard bits.
- You don't need to pipeline this unit for maximum clock frequency.
- You should be able to issue one `FIR_3()` instruction every clock cycle.

Tasks:

Your task is to design and implement the function `FIR_3()` as a special instruction on the Senior processor. You also need to implement support for the `INITFIR_3()` function and you will probably need to add a couple of instructions to do this.

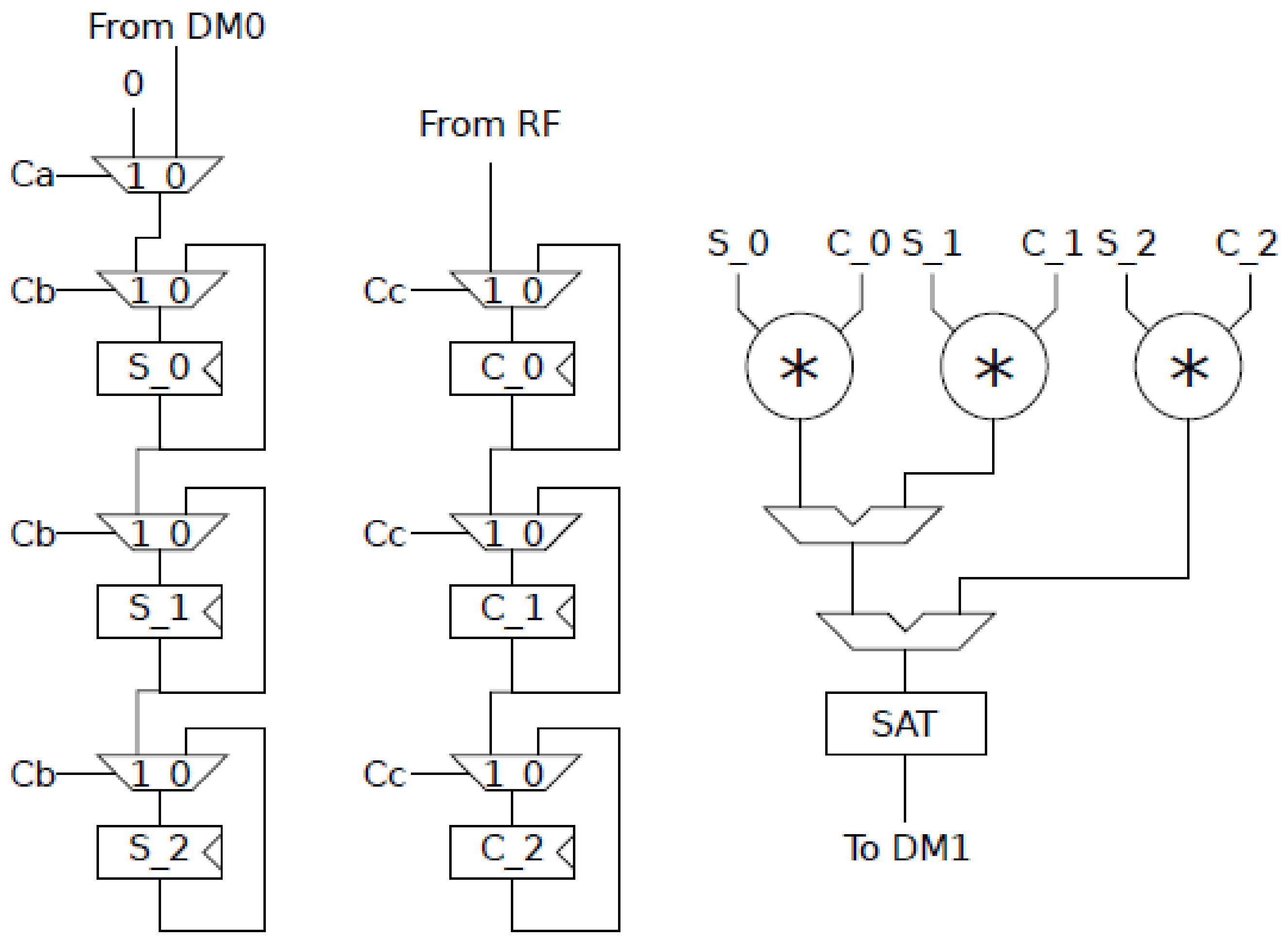
a) Draw a hardware schematic of the modified parts of the pipeline. You don't need to annotate bit widths. You don't need to annotate the contents of a *SATURATE* box.

b) Draw a control table for your hardware where you include a `NOP` instruction, the `FIR_3` instruction and the instructions necessary to implement `INITFIR_3`. You should also write pseudo assembler code for `INITFIR_3`.

How to modify the pipeline:

You will need to modify the MAC unit and make sure that it can send the result back to DM1. (There is no need to modify the AGU unit as the addressing used in this code snippet are standard post increment addressing modes that any normal DSP processor would have.)

Schematics



Control table

Operation	Ca	Cb	Cc
NOP	-	0	0
CLEARSAMPLES	1	1	0
MOVE COEFF0, REG	-	0	1
FIR_3	0	1	0

```
INIT_FIR:
    CLEARSAMPLES
    CLEARSAMPLES
    CLEARSAMPLES
    move COEFF0, R5     ; val5
    move COEFF0, R4     ; val4
    move COEFF0, R3     ; val3
    move AR0, R1        ; In the AGU
    move AR1, R2        ; In the AGU
    ret
```