06 – MAC

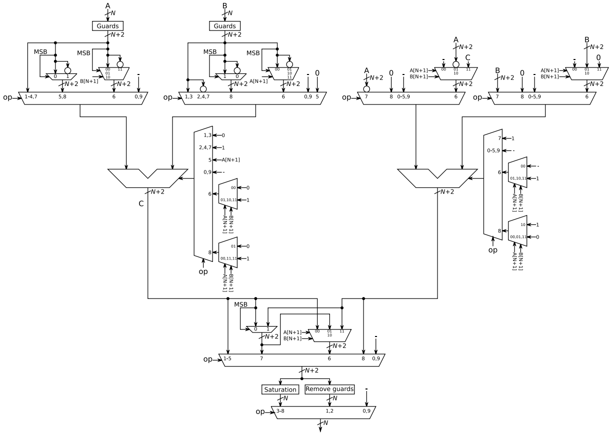
Oscar Gustafsson

LINKÖPING UNIVERSITY

A few remaining issues from last lecture

- ALU example
- Hardware for $|x - y|$
- Shifts in ALU:s

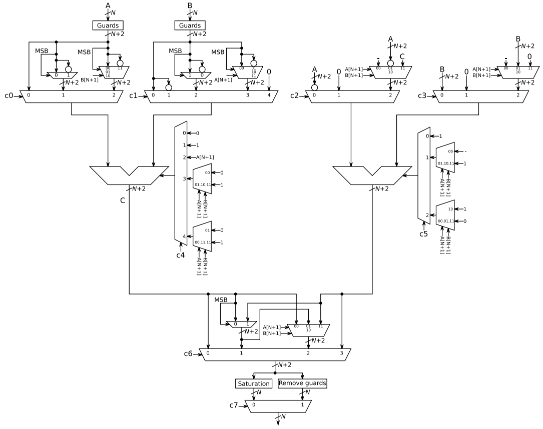
ALU example from lecture 5



ALU example from lecture 5

```
-- purpose: Select operand 1 for adder 1
-- type   : combinational
-- inputs : operand, a, b
-- outputs: adder1op1
adder1op1select: process (operand, a, b)
begin  -- process adder1op1select
case operand is
  when 1|2|3|4|7 => adder1op1 <= a;
  when 5|8 => if (a(wordlength+1) = '0') then
                adder1op1 <= a;
              else
                adder1op1 <= not(a);
              end if;
  when 6 => if (a(wordlength+1) = '0' or b(wordlength+1) = '0') then
              adder1op1 <= a;
            else
              adder1op1 <= not(a);
            end if;
  when others => adder1op1 <= (others => '-');
end case;
end process adder1op1select;
```

ALU example from lecture 5

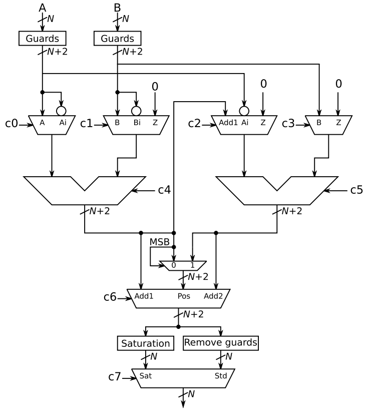


ALU example from lecture 5

```
adder1op1select: process (operand, a, b)
begin  -- process adder1op1select
case c1 is
  when 0 => adder1op1 <= a;
  when 1 => if (a(wordlength+1) = '0') then
              adder1op1 <= a;
            else
              adder1op1 <= not(a);
            end if;
  when others => if (a(wordlength+1) = '0' or b(wordlength+1) = '0') then
                   adder1op1 <= a;
                 else
                   adder1op1 <= not(a);
                 end if;
end case;
end process adder1op1select;

c1adder1op1select: process (operand)
begin  -- process c1adder1op1select
case operand is
  when 1|2|3|4|7 => c1 <= 0;
  when 5|8 => c1 <= 1;
  when 6 => c1 <= 2;
  when others => c1 <= 0;
end case:
```

ALU example from lecture 5



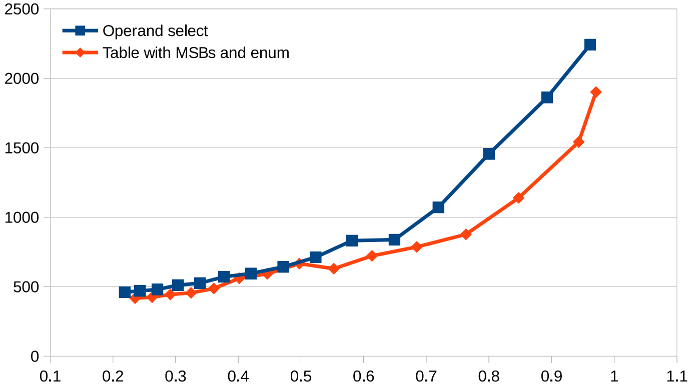
ALU example from lecture 5

```
type c0options is (An, Ai);
...
adder1op1select: process (c0, a)
begin  -- process adder1op1select
case c0 is
  when An => adder1op1 <= a;
  when others => adder1op1 <= not(a);
end case;
end process adder1op1select;

c0adder1op1select: process (operand, asign, bsign)
begin  -- process c1adder1op1select
case operand is
  when 1|2|3|4|7 => c0 <= An;
  when 5|8 => if asign = '0' then
                c0 <= An;
              else
                c0 <= Ai;
              end if;
  when 6 => if asign = '1' and bsign = '1' then
              c0 <= Ai;
            else
              c0 <= An;
            end if;
  when others => c0 <= An;
```

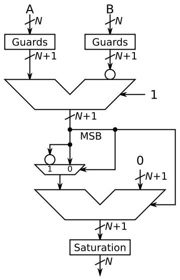
ALU example from lecture 5

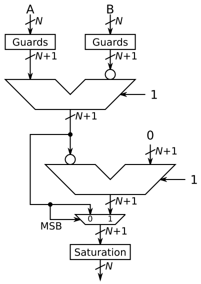
Results for 8-bit ALU in 65 nm: area vs f_{clk} (GHz)

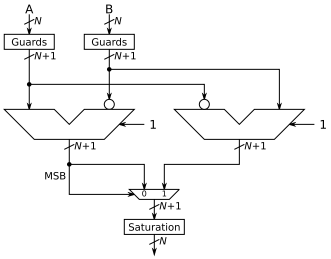


16-bit $|A - B|$

Three approaches

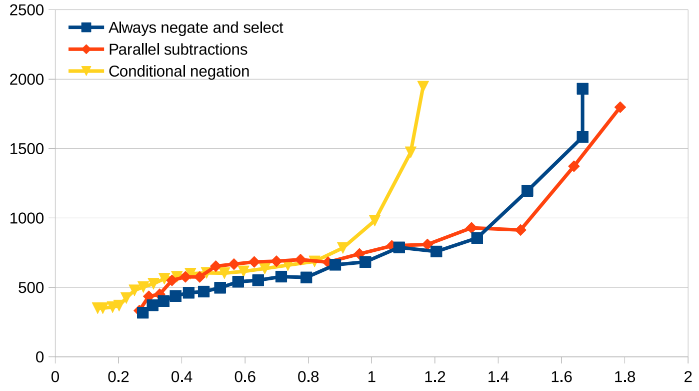




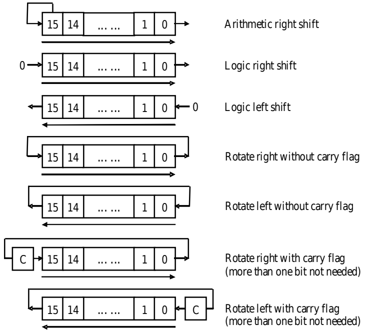


16-bit $|A - B|$

Results for 16-bit $|A - B|$ in 65 nm: area vs f_{clk} (GHz)

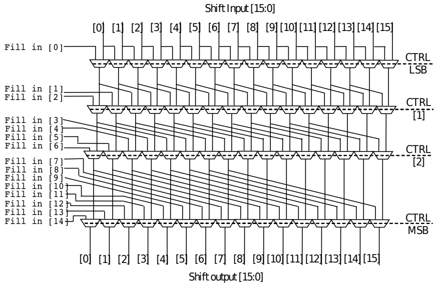


Typical ALU shift operations


15
14
... ...
1
0
Arithmetic right shift
0
Logic right shift
0
Logic left shift
Rotate right without carry flag
Rotate left without carry flag
C
Rotate right with carry flag (more than one bit not needed)
C
Rotate left with carry flag (more than one bit not needed)

[Liu2008]

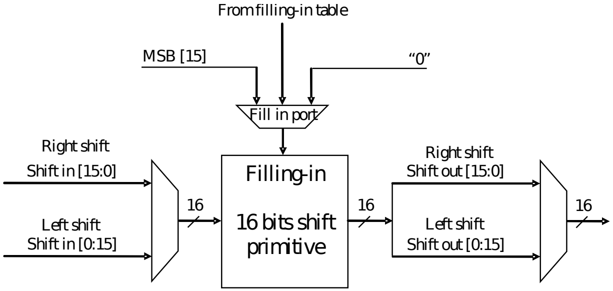
Shifter primitive



[Liu2008]

- Note: Barrel shifters based on 4-to-1 multiplexers may be more efficient

Hardware multiplexing in shifter



[Liu2008]

- Note: Fill in table may be complicated for some shift operations

Bitwise logic operations

- AND, OR, XOR
- More or less trivial

Other ALU operations

- Find leading one: Returns the most significant bit set to one
- Find leading zero: Returns the most significant bit set to zero
- Population count: Returns the number of bits set to one

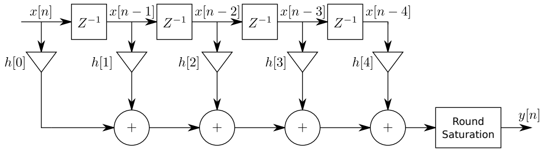
```
// RTL code for find leading one
casez(opa)
   16'b1???????????????: msbbit = 16;
   16'b01??????????????: msbbit = 15;
   16'b001?????????????: msbbit = 14;
endcase
```

MAC - Introduction

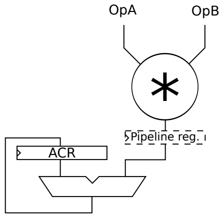
- MAC - Multiply and ACcumulate
- One of the most important parts of a DSP processor

Why MAC?

- Convolution based algorithms
  - FIR, IIR, Auto correlation, Cross correlation
- Linear algebra based algorithms
  - Inner product, Matrix multiplication
- Support most transformation algorithms
  - FFT, DCT
- Example below: FIR implemented with 4 MACs and one MULT



Basic structure of a MAC unit



- Why ACR instead of normal RF?
  - Reduce the number of RF ports
  - No register/accumulator size mismatch
  - No need to write to RF after a very long pipeline
    - During convolution: (Address generation, memory read, multiply, add)

Multiplier basics

- Create a schematic for a 16×16 multiplier with support for:
  - OP1: Signed/unsigned integer multiplication
  - OP2: Signed/unsigned integer multiplication (read out high part)
  - OP3: Signed fractional multiplication (with proper rounding)
- The instruction decoder will set the C_s signal according to whether signed or unsigned multiplication is desired

MAC unit basics

- Example: Create a schematic for a MAC unit 16-bit inputs, 8 guard bits and the following operations:
  - OP0: NOP
  - OP1: ACR = 0
  - OP2: ACR = Signed/unsigned multiplication
  - OP3: ACR = Signed/unsigned multiply and accumulation
  - OP4: ACR = SAT(RND(ACR)) // Rounding/saturation for fractional inputs/outputs
- Note: A fairly common mistake on the exam is to forget the NOP instruction!

MAC unit basics

- Something is missing in the specification.
- (Something which was previously a common error on the exam…)

MAC unit basics

- We need to read back the result of the accumulation:
  - Op5: `RF = ACR[31:16]`
  - Op6: `RF = ACR[15:0]`
  - Op7: `RF = ACR[39:32]`
- (In fact, we need to be able to save/restore the complete state of the MAC unit.)
  - We need to be able to do context switches!

MAC unit basics - Discussion break

- Question: If we have 16-bit fractional inputs and want a 16-bit fractional output, which bits of the accumulator do we need to read out? How do we perform saturation and rounding?

MAC unit basics - Discussion break

- Question: If we have 16-bit fractional inputs and want a 16-bit fractional output, which bits of the accumulator should we need?
- OP8: `RF = ACR[30:15]`

MAC unit basics

- Alternatives:
    - Support fractional multiplication by scaling right by one step directly after the multiplication. Now both (saturated) integer and (saturated) fractional MAC is supported by reading out only ACR[31:16] and ACR[15:0].

MAC unit basics

- Alternatives:
  - Allow post operations on the ACR value during readout:
    - `RF = SAT(ROUND(ACR[31:16]))`
    - Drawback: Potentially a long critical path.
  - Sign-extended readout of guard bits: `RF = { {8{ACR[39]}}, ACR[39:32]};`
  - Allow a few other different kinds of readout options

MAC unit basics

- We probably also want to be able to set the ACR to arbitrary values:
  - OP9: `ACR[31:16] = OpA`
  - OP10: `ACR[15:0] = OpA`
  - OP11: `ACR[39:32] = OpA[7:0]`
- We may want some other variants as well:
  - `ACR = { {8{OpA[15]}}, OpA[15:0], 16'b0};` (Load high and sign extend)
  - `ACR = { {9{OpA[15]}}, OpA[15:0], 15'b0};` (Load fractional value and sign extend)
  - `ACR = { {8{OpA[15]}}, OpA[15:0], OpB[15:0]};` (Load high and low part simultaneously)
- `OpA` and `OpB` may be either a memory or register file

Scaling

- Scaling is typically desired in a MAC unit
    - A scaling operation allows us to handle more than just fractional/integer multiplication
- OP12: Scaling by 0.5 (easy)
- OP13: Scaling by 0.25 (easy)
- OP14: Scaling by 2 (easy)
- OP15: Scaling by 4 (easy)
- etc...
- OP16: Scaling by 1.5 (easy or hard?)
- OP17: Scaling by 0.75 (easy or hard?)

Scaling

- WARNING: Arithmetic right shift is not identical to division by a power of two!
- Not a big concern if you round the result properly

```
#include <stdio.h>
int main(int argc, char **argv){
    int a=-1;
    printf("%d,  %d\n", a/2, a >> 1);
    return 0;
}
// Output from this program on x86_64 (using GCC): 0, -1
```

- Nitpick: Right shift on a negative number is actually not specified by the C standard! (It is an implementation defined behavior although it is unlikely to use anything but an arithmetic right shift.)

Wide multiplication operation

- You may sometimes want to run a 32×32 bit wide multiplication on a MAC unit with a 16 bit wide multiplier.
- Straightforward method:

```
  $signed(A[31:0]) * $signed(B[31:0]) =
= $unsigned(B[15:0]) * $unsigned(A[15:0]) +
+ ($unsigned(B[15:0]) * $signed(A[31:16])) << 16 +
+ ($signed(B[31:16]) * $unsigned(A[15:0])) << 16
+ ($signed(B[31:16]) * $signed(A[31:16])) << 32)
```

- One reason why Senior has separate `mulxx/macxx/convxx` instructions, where `x` can be either `s` (signed) or `u` (unsigned).

Wide multiplication operation

- Somewhat trickier in practice (64-bit result)

```
AH = $signed(A[31:16]);    AL = $unsigned(A[15:0])
BH = $signed(B[31:16]);    BL = $unsigned(B[15:0])

ACR = AL * BL;
R0 = ACR[15:0];
ACR = ACR >> 16; // Question to audience: Do we
                 // need rounding here?
ACR += BL*AH;
ACR += AL*BH;
R1 = ACR[15:0];
ACR = ACR >> 16;

ACR += HL*AL;
R2 = ACR[15:0];
R3 = ACR[31:16];
```

Long Arithmetic Operations

- There is a wide adder in the MAC unit. This may be used for long addition/subtraction:
  - `ACRz = ACRx + ACRy;` (`x`, `y`, and `z` are numbers from 0 to $N-1$, where N is the number of accumulator registers)
  - `ACRz = ACRx - ACRy;`
  - `ACRz = ABS(ACRx);`
  - Compare (set flags according to `ACRx - ACRy`)

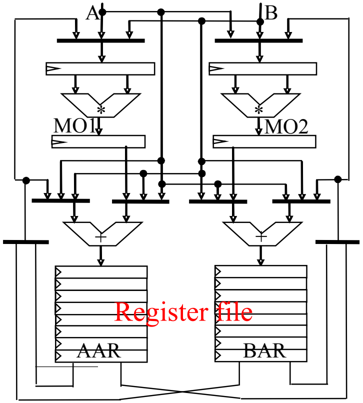
Long Arithmetic Operations

- It is also convenient to be able to add immediates or values from the register file:
  - `ACRz = {{24{OpA[15]}}, OpA[15:0]};`
  - `ACRz = {{8{OpA[15]}}, OpA[15:0], 16'b0};`
  - `ACRz = {{8{OpA[15]}}, OpA[15:0], OpB[15:0]};`
- We may want the following variant to make it easy to work with fractional data:
  - `ACRz = {{9{OpA[15]}}, OpA[15:0], 15'b0};`

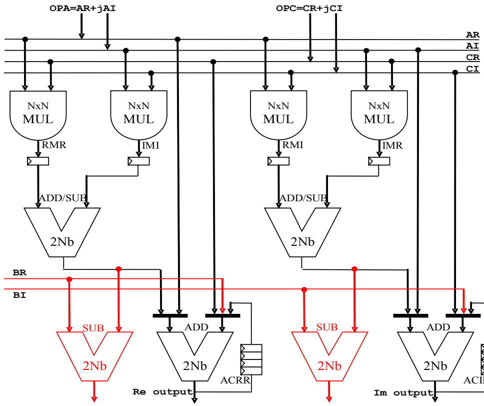
Flags

- We probably want all the normal flags in the MAC unit
- Zero (Z)
- Negative (N)
- Overflow (V)
- Carry (C) *(Probably not so important)*
- We may also want a sticky version of the overflow flag (*(S)*) dealing with the full accumulator (not overflow as in saturation)
  - Rationale: Do a number of calculations, go to error handling code if an overflow occurred at any point in the calculation.

Advanced MAC architectures: Dual MAC



Advanced MAC architectures: Complex MAC



Complex multiplication algorithms

- Normal algorithm:
  - $(a + bi)(c + di) = (ac - bd) + i(ad + bc)$
  - 4 multiplications, 2 additions
- If one factor $(c + di)$ is constant:
  - Usecase: FFT, complex FIR/IIR, etc
  - Still 4 multiplications and 2 additions
- Critical path: multiplier $\rightarrow$ adder

Complex multiplication algorithms

- Gauss' algorithm
  - $k_1 = c(a+b)$
  - $k_2 = a(d-c)$
  - $k_3 = b(c+d)$
  - $(a+bi)(c+di) = (k_1 - k_3) + i(k_1 + k_2)$
  - 3 multiplications, 5 additions
  - Drawback: Needs slightly wider multipliers/adders
- If one factor $(c+di)$ is constant:
  - Precompute $d-c$ and $c+d$
  - 3 multiplications and 3 additions
- Critical path: adder $\rightarrow$ multiplier $\rightarrow$ adder

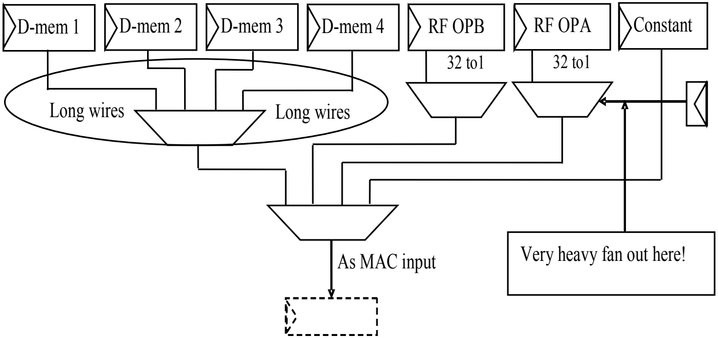
Wide multiplication: Karatsuba's variant

- A similar trick can be used for performing real-valued multiplications
- Could be useful for handling multiplications wider than the native datawidth
- If you are interested, search for Karatsuba's algorithm
  - Wikipedia has a good introduction
- A very similar trick can also reduce the computational complexity of FIR filters (Search for Fast Fir algorithm, FFA)
  - We might look at this in a later lecture
  - Talk to Oscar if you think this is really interesting! (from earlier year's slides...)

Floating-point MAC

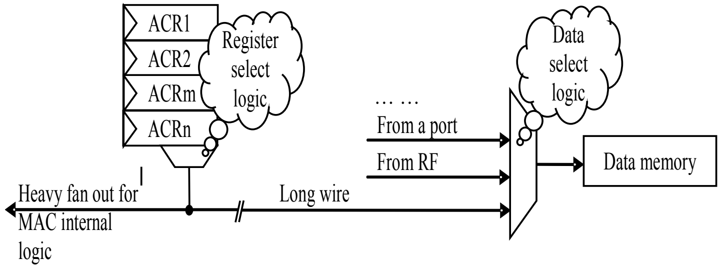
- Potential problem:
  - Floating-point addition is cumbersome
  - The accumulation part is hence cumbersome
  - Easiest solution (from HW point of view): Use several accumulators and use loop unrolling/software pipelining
- Floating point MAC is sometimes called FMA (Fused Multiply and Add)
  - In this case rounding is only done once instead of twice
  - Alignment can be performed in parallel with the multiplication

Critical Path issues in MAC unit



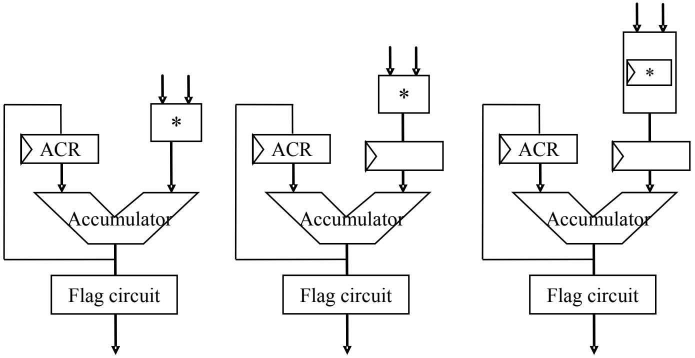
[Liu2008]

Critical Path issues in MAC unit



[Liu2008]

Critical Path issues in MAC unit



(a) MAC in one clock cycle (b) MAC using two clocks (a) MAC using three clocks

[Liu2008]

Oscar Gustafsson

www.liu.se

LINKÖPING UNIVERSITY