05 – Microarchitecture, RF and ALU

Oscar Gustafsson

Microarchitecture Design

- Step 1: Partition each assembly instruction into microoperations, allocate each microoperation into corresponding hardware modules.
- Step 2: Collect all microoperations allocated in a module and specify **hardware multiplexing** for RTL coding of the module
- Step 3: Fine-tune intermodule specifications of the ASIP architecture and finalize the top-level connections and pipeline.

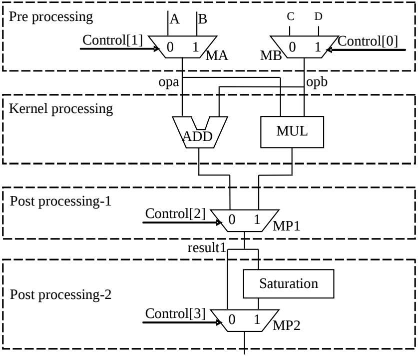
Hardware Multiplexing

- Reusing one hardware module for several different operations
- Example: Signed and unsigned 16-bit multiplication

Hardware Multiplexing

Possible functions

1. A + C
2. A + D
3. B + C
4. B + D
5. A * C
6. A * D
7. B * C
8. B * D
9. SAT(A + C)
10. SAT(A + D)
11. SAT(B + C)
12. SAT(B + D)
13. SAT(A * C)
14. SAT(A * D)
15. SAT(B * C)
16. SAT(B * D)

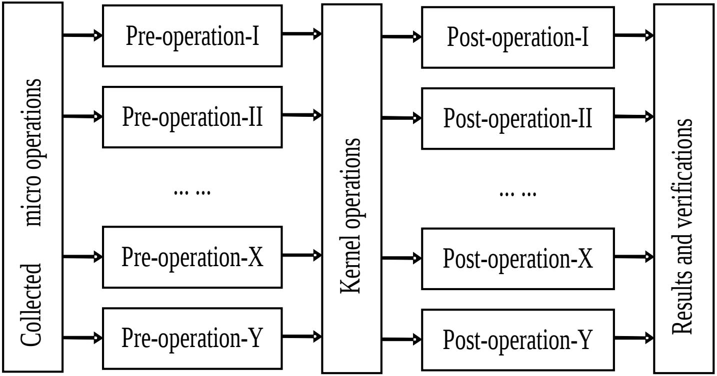


[Liu2008]

Hardware multiplexing

- Hardware multiplexing can be implemented either by SW or by configuring the HW
- A processor is basically a very neat design pattern for multiplexing different HW units
- Perhaps the most important skill of a good VLSI designer

Typical design pattern for datapath modules



[Liu2008]

Discussion break

- Which of these units is most expensive in terms of area?
    - 17×17 bit multiplier
    - 32-bit adder/subtracter
    - 32-bit 16-to-1 mux
    - 32-bit adder
    - 8 KiB memory (32 bits wide)

Area properties (a.k.a. what to optimize)

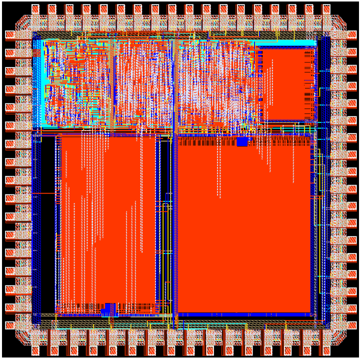
- Relative areas of a few different components
  - 32-bit adder: 0.2 to 1 area units
  - 32-bit adder/subtracter: 0.3 to 2 area units
  - 32-bit 16-to-1 mux: 0.5 to 0.6 area units
  - 17×17 bit multiplier: 1.3 to 3.7 area units
  - **8 KiB memory (32 bits wide): 33 area units**
- Exam hint: You are typically supposed to minimize the area of the units you design. That is, do not use more multipliers than necessary, avoid extra adders, do not worry about small 2-to-1 multiplexers. (And do not add extra SRAM memories if you can avoid it…)

Performance properties

- Relative maximum frequencies
  - 32-bit adder: 0.1 to 1
  - 32-bit adder/subtracter: 0.1 to 0.9
  - 32-bit 16-to-1 mux: 0.31 to 0.9
  - 17×17 bit multiplier: 0.11 to 0.44
  - 8 KiB memory (32 bits wide): 0.53

Optimizing memory size is often the most important task

- MP3 decoder example
- All memories in the chip are 3 time the size of the DSP core itself
- (I/O pads are also larger than the DSP core itself)



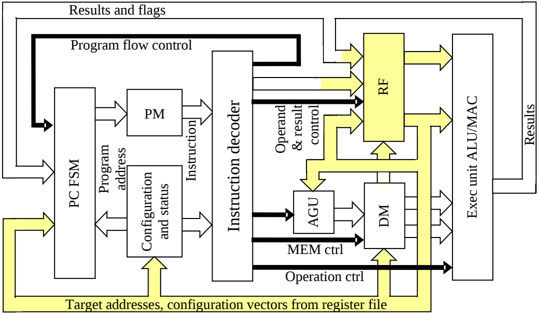
Microarchitecture design of an instruction

- Required microoperations for a typical convolution instruction:
  - `conv ACRx,DM0(ARy++%),DM1(ARz++)`
- Required microoperations:
  - Instruction decoding
  - Perform addressing calculation
  - Read memories
  - Perform signed multiplication
  - Add guard bits to the result of the multiplication
  - Accumulate the result
  - Set flags
  - For a combined `repeat/conv` instruction:
    - PC $<=$ PC while in the loop
    - PC $<=$ PC + 1 as the last step in the loop
    - No saturation/rounding during the iteration
    - Saturate/round after final loop iteration

The register file (RF)

- The RF gets data from data memories by running load instructions while preparing for an execution of a subroutine.
- While running a subroutine, the register file is used as computing buffers.
- After running the subroutine, results in the RF will be stored into data memories by running store instructions.

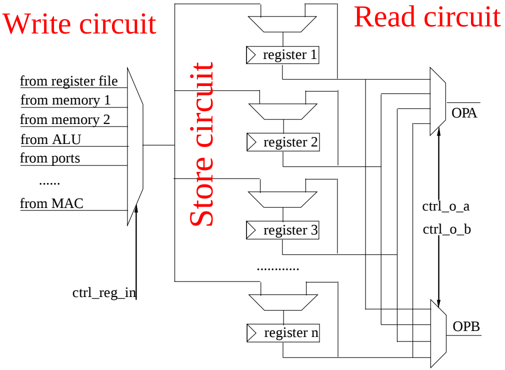
General register file



[Liu2008]

- Connected to almost all parts of the core

Register file schematic



[Liu2008]

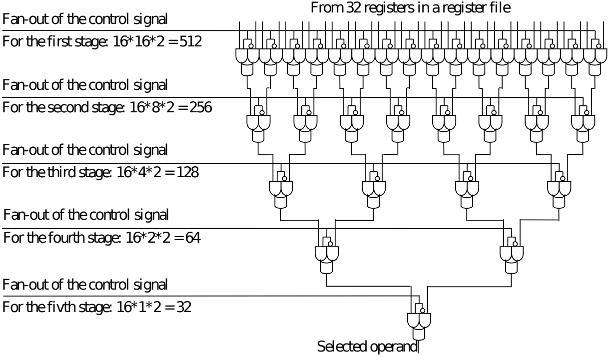
Register file speed

- Almost (but not quite) the same speed as a very fast 32-bit adder (in this particular technology)
- Also note that it is possible to use special register file memories (but at an increased verification cost)

Read before write or write before read

- A processor architect has to decide how the register file should work when reading and writing the same register
- Read before write
  - The old value is read
- Write before read
  - The new value is read (more costly in terms of the timing budget)

Physical design: fan-out problem



[Liu2008]

Register File in Verilog

```
reg [15:0] rf[31:0]; // 16 bit wide RF with 32 entries
always @(posedge clk) begin
    if(we) rf[waddr] <= wdata;
end

always @* begin
    op_a = rf[opaddr_a];
    op_b = rf[opaddr_b];
end
```

Special purpose registers

- Sometimes we need special purpose registers (SPR or SR)
  - BOT/TOP for modulo addressing
  - AR for address register
  - SP
  - I/O
  - Core configuration registers
  - etc
- Should these be included in the general purpose register file?

Special purpose registers as normal registers

- Convenient for the programmer. Special purpose registers can be accessed like any normal register.
  - Example: `add bot0,1 ; Move ringbuffer bottom one word`
  - Example 2 (from ARM): `pop pc`
- Drawbacks:
  - Wastes entries in the general purpose register file
  - Harder to use specialized register file memories

Special purpose registers needs special instructions

- Special instructions required to access SR:s
- Example:
  - `move r0,bot0` ; Move ringbuffer bottom one word
  - `(nop) ; May need nop(s) here`
  - `add r0,1`
  - `(nop) ; May need nop(s) here`
  - `move bot0,r0`
  - (Move is encoded as move from/to special purpose register here)
- Advantage:
  - Easier to meet timing as special purpose registers can easier be located anywhere in the core
  - Can scale easily to hundreds of special purpose registers if required. (Common on large and complex processors such as ARM/x86)
- Drawback:
  - Inconvenient for special registers you need to access all the time

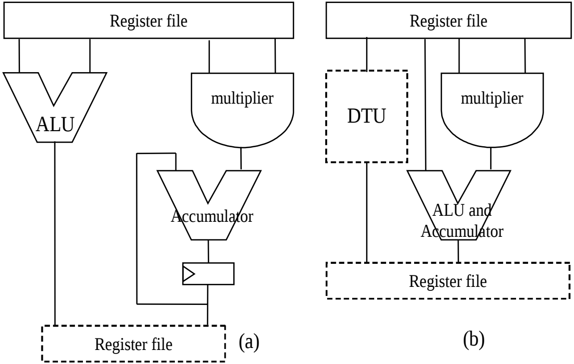
Conclusions: SPRs

- Only place SPRs as a normal register if you believe it will be read/written via normal instructions very often

ALU in general

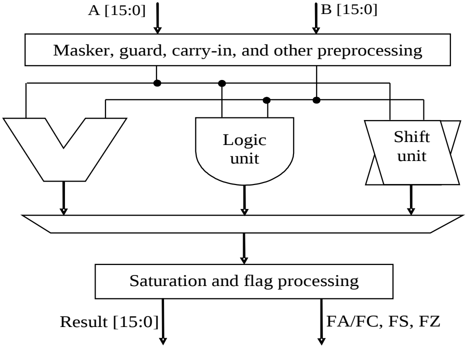
- ALU: Arithmetic and Logic Unit
  - Arithmetic, logic, shift/rotate, others
  - No guard bits for iterative computing
  - One guard bit for single step computing
  - Get operands from and send result to RF
  - Handles single precision computing

Separate ALU or ALU in MAC



[Liu2008]

ALU high level schematic



[Liu2008]

Pre-processing

- Select operands: from one of the sources
  - Register file, control path, HW constant
- Typical operand pre processing:
  - Guard: one guard
    - (does not support iterative computing)
  - Invert: Conditional/non-conditional invert
  - Supply constant 0, 1, –1
  - Mask operand(s)
  - Select proper carry input

Post-processing

- Select result from multiple components
  - From AU, logic unit, shift unit, and others
- Saturation operation
  - Decide to generate carry-out flag or saturation
  - Perform saturation on result if required
- Flag operation
  - Flag computing and prediction

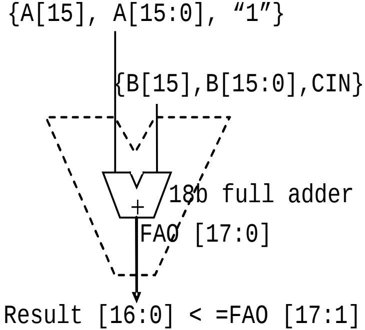
General instructions

	Operation	opa	opb	Carry in	Carry out
ADD	Addition	+	+	0	Cout/SAT
SUB	Subtraction	+	-	1	Cout/SAT
ABS	Absolute	+/-		A[15]	SAT
CMP	Compare	+	-	1	SAT
NEG	Negate	-		1	SAT
INC	Increment	+	1	0	SAT
DEC	Decrement	+	-1	0	SAT
AVG	Average	+	+	0	SAT

Special Instructions

Mnemonic	Description	Operation
MAX	Select larger value	`RF <= max(OpA,OpB)`
MIN	Select smaller value	`RF <= min(OpA,OpB)`
DTA	Difference of two absolute values	`GR <= |OpA| - |OpB|`
ADT	Absolute of the difference of two values	`GR <= |OpA-OpB|`

Adder with carry in for RTL synthesis (safe solution)



[Liu2008]

- Full adder may have no carry in
- One guard bit
- We need 2 extra bits in the adder
- LSB of the 18b result will not be used
- MSB of the 18b result will be the guard
- Works on all synthesis tools

Adder for RTL synthesis (modern version)

-
```
{Cout,R[15:0]}={1'b0,A[15:0]}+{1'b0,B[15:0]}+Cin;
```
- `Cout` is 1 bit wide
- **Important: `Cin` is 1 bit wide!**
- Modern synthesis tools can usually handle this case without creating two adders

Example: Implement an 8-bit ALU

Instructions	Function	OP
NOP	No change of flags	0
A+B	$A + B$ (without saturation)	1
A-B	$A - B$ (without saturation)	2
SAT(A+B)	$A + B$ (with saturation)	3
SAT(A-B)	$A - B$ (with saturation)	4
SAT(ABS(A))	$\|A\|$ (absolute operation, saturation)	5
SAT(ABS(A+B))	$\|A + B\|$ (absolute operation, saturation)	6
SAT(ABS(A-B))	$\|A - B\|$ (absolute operation, saturation)	7
SAT(ABS(A)-ABS(B))	$\|A\| - \|B\|$ (absolute operation, saturation)	8
CLR S	Clear S flag (other flags unchanged)	9

- There should be a negative, zero, and saturation flag!
- Discussion topic: How many adders are needed for each operation?
- Discussion topic: How many guard bits are needed for each operation?

Oscar Gustafsson

www.liu.se

LINKÖPING
UNIVERSITY