21-05-2015

Statically scheduled, 2-way superscalar CPU with flag stack and decomposed branches

Christian Bering Bøgh - s952606@student.dtu.dk

Abstract

Performance enhancing features for statically scheduled CPU architectures are proposed in this project.

The main goal is to develop and evaluate these features, which necessitates the development of a prototype CPU, a computer platform to plug it into and an assembler to develop code for it.

Additional characteristics and a secondary goal of the designed CPU includes a small code footprint - meaning code density is high since each bundle of two instructions is packed in only 32 bits.

A 32-bit, 4-stage, statically scheduled, 2-way superscalar CPU has been designed using VHDL. A platform, including memory, UART and VGA has also been developed.

The complete system has been synthesized using Altera Quartus II v13.0.1 and tested on a Terasic DE0 FPGA prototyping board. The FPGA is an Altera Cyclone III EP3C16F484C6.

An assembly language has been defined and an assembler has been written in Java and used for writing test programs, including a Mandelbrot fractal renderer.

Introduction

Increasing CPU performance, by going superscalar, obviously exacerbates any difficulties in keeping the pipeline(s) filled.

This is due to hazards incurring their penalties across several pipelines instead of just one. For instance, a single branch delay slot will become two slots in a dual-issue pipeline - making the compiler's job of filling these slots with useful work even harder. Same goes for load delay slots.

To some degree, control hazards can be overcome through prediction, speculation and buffering, removing the need for ISA defined branch delay slots. ISA enhancements, like predication, further limits the loss of useful work from mispredicts, by simplifying control flow.

Data hazards can either be attacked through out-of-order execution and dynamic scheduling, or through an increase in the scope for code optimization and static scheduling (big ISA visible register files).

Common for these approaches is that they seek to mask the existing hazards, while in some cases, making the underlying hazards worse due to lengthening of the pipeline.

The proposed architecture takes a different approach. Instead of masking the effects of hazards, they are either removed entirely, or moved such that they can be overcome more easily with just static scheduling.

The decomposition of branches into a push-target-address-on-stack instruction and a branching bit, present in every 32-bit instruction bundle, makes the actual branching so fast that the need for a branch delay slot can be eliminated.

Arranging condition flags as a stack, makes powerful multi-operand flag logic operations feasible, which can eliminate conditional branches.

Despite register operand fields only taking three bits, 16 registers are available in a split register file.

Besides doubling the number of registers, the splitting into 8 integer and 8 pointer registers and limiting the available pointer operations has the added benefit of simplifying forwarding such that effective address calculation can be done in the decode stage.

This in turn allows memory access to be moved to the execute stage, thus shortening the pipeline and avoiding the need for a load delay slot.

Branch decomposition

Branch target caches and call-return stacks are enhancements added to modern architectures to mask control hazards and even achieve zero-cycle branches when correctly predicted.

Some ISAs have hints in their branch instructions, telling whether the BTB should be used or not, but usually it isn't ISA visible.

In the proposed architecture, branches are split into two instructions.

One is a push instruction, which pushes the branch target address onto an 8 entry circular branch target stack, abbreviated BTS in the following.

Push instructions execute in the decode stage with full forwarding to the branch unit in the fetch stage. While this requires more hardware, it eliminates the control hazard.

The other instruction is the actual branch, and because it gets the target address from the top of the BTS, it can be very small and fast. In fact, it's only a single bit, and it's checked during instruction fetch.

In other words, there is no branch instruction. It has been reduced to a single control bit, present in every 32-bit instruction bundle.

This is particularly beneficial if the top of BTS can be reused, effectively giving zero cost branches.

The basic structure or "pattern" of a loop with counting variable is illustrated in the following example:

```
      push pc, @loop_begin + 1    & ...       # Address 0x800
      set i0 = count - 1                      # Address 0x804
loop_begin:
      sub.f i0 = i0, 1            & ...       # Address 0x808
      ...                         & ...       # Address 0x80C
loop  ...                         & ...       # Address 0x810
```

In the first bundle, instruction slot 0 pushes the 32-bit value 0x809 onto the BTS. The least significant two bits aren't used for addressing, since the 32-bit bundles are naturally aligned.

Instead they tell the branch unit how to behave:

00	Unconditional branch. Pops the top of the BTS. Doesn't touch the flag stack
01	Branch on flag = 0. Only pops the BTS if not taken - e.g. on loop exit. Pops the top flag of the flag stack
10	Branch on flag = 0. Always pops the BTS - e.g. if-then-else. Pops the top flag of the flag stack
11	Reserved

The push instruction adds the PC-relative (indicated by “@”) value of "loop_begin" plus 1 to the program counter and pushes it. The instruction's immediate field is 9 bits, which allows for offsets between -64 and 63 bundles.

The 2nd bundle is occupied by a single extended instruction. The immediate field is a 16-bit signed value. If count-1 is 7 or less, a normal small set instruction can be used.

The 3rd bundle's slot 0 updates the loop counter and pushes the resulting sign bit onto the flag stack. It's the "**.f**" suffix which tells the assembler to set the bundle's flag generate bit.

The flag generation is placed two bundles before the branch. This is because the moving of the branching decision to the fetch stage means flag forwarding from the execute stage can't reach the branch unit in the same cycle.

Essentially, the control hazard has been converted to a data hazard. But this hazard is much easier to deal with.

The 5th bundle uses the **loop** keyword to tell the assembler to set the bundle's branch bit. The **jump**, **call**, **return** or **branch** keywords could also have been used. They are all synonymous and exist purely for convenience and code readability.

What if the loop body is only two bundles? It would seem there's no room for a flag-generate-to-branch delay slot - causing a single cycle stall of the fetch stage on every iteration.

There's a trick to avoiding stalls without resorting to loop unrolling:

```
      push pc, @loop_begin + 1   &   flag 0b00000000
      set i0 = count - 2
loop_begin:
      ...                        &   ...
loop  sub.f i0 = i0, 1           &   ...
      flag.pop 0b11001100        &   ...
```

A loop prologue is added to push a '0' flag onto the flag stack. This is necessary because flag generation is now one iteration behind.

Similarly, an epilogue is added to pop the excess flag after loop exit.

Note that in bundle 4, **loop** causes a flag to be popped, while at the same time, a new flag is generated and pushed - effectively just overwriting the old flag.

What if the loop body is only a single bundle?

This trick leverages the structure of the flag stack, which is really just a 32-bit shift register. Of course, this puts a tight limit on the magnitude of count.

```
      ...                        &   flag 0b11111111
      push pc, @loop_begin + 1   &   shiftl fr = fr, count - 1
loop_begin:
loop  ...                        &   ...
```

First a '1' flag (stop bit) is pushed, then count-1 zeroes are pushed by left shifting the flag register.

Only the first iteration experiences a stall, which is due to the shift instruction writing the flags right before the loop.

If possible, put another bundle in between, to avoid this stall.

The pattern for a subroutine call:

```
     push pc, @subroutine
call push pc, 4                 &  ...
```

The extended push instruction has a 24-bit signed immediate field giving an offset of +/- 8Mbyte.

For larger offsets or absolute addresses, a pointer register can be initialized and pushed.

The return address is pushed simultaneously with the popping of the subroutine address - effectively overwriting it.

Making the call conditional:

```
     push pc, @subroutine + 2
call push pc, 8                 &  ...
     ...                        &  add sp, 4
```

Adding 4 bytes to the BTS stack pointer has the effect of popping the unused return address on call-not-taken.

As per tradition, the stack grows downwards.

Smallest possible stall free if-then-else construct:

```
       copy.f i3 = i3           &  ...
       push pc, @else_label + 2 &  ...
branch push pc, @end_if_label   &  ...
branch ...                      &  ...       # Then-bundle
else_label:
branch ...                      &  ...       # Else-bundle
end_if_label:
```

A zero test is performed on integer register 3. If zero, the then-clause is executed.

Note that the else-clause also branches to end_if_label. This is just a cheap way of popping the BTS.

Also note that 3 instruction slots in the beginning need to be filled with useful work. If only one instruction can be found, the copy and first push may as well be put in the same bundle.

That will cause a stall, but save two nops - i.e. one bundle.

Now imagine what it would look like without decomposed branches:

```
     copy.f i3 = i3           &  ...
     branch.c @else_label     &  ...
     ...                      &  ...
     branch @end_if_label     &  ...      # Then-bundle 1
     ...                      &  ...      # Then-bundle 2
else_label:
     ...                      &  ...      # Else-bundle
end_if_label:
```

The push instructions are replaced by conventional (but imagined) branch instructions with one branch delay slot.

The bundle count goes from 5 to 6. Half a bundle goes to the then-clause, which may be excellent if 3 instructions are exactly what are needed.

The problem is the first branch delay slot, which means 4 instead of just 3 useful instructions must be found. If only two can be found, there's no option of trading a bundle for a stall.

A limitation of the BTS is that only one value can be pushed per cycle. If two push instructions are in the same bundle, only the one in slot 1 will execute.

Another limitation is that only instruction slot 1 can push pointer registers and modify the stack pointer.

Direct manipulation of the BTS stack pointer through adding/subtracting of offsets takes precedence over other attempted stack pointer movements.

Modifying the previous subroutine call example:

```
     push pc, @subroutine
call push pc, 4               &  sub sp, 4
```

The difference is that now the return address doesn't overwrite the subroutine address - making reuse possible.

This is because the "sub sp" instruction both ignores the pop of the **call** and executes before the **push**.

As a general rule, there should be no instruction interdependence within a bundle, but here a common data structure is structurally modified by 3 instructions (including the **branch**) in the same bundle.

The chosen behavior is the one that seems the most useful and intuitive, while also not being expensive to implement.

The flag stack

With such very simple branches, the question arises; which flag should a conditional branch test?

The answer is; the top of the flag stack - i.e. the LSB of the 32-bit flag register.

This limitation would likely be troublesome on deep pipelines with long penalties. They use their flat flag file to calculate flags well in advance of them being needed - often speculatively.

But the proposed ISA enhancements are meant to particularly benefit short in-order pipeline architectures.

On the plus side, a stack makes advanced flag manipulation possible, even with tiny 15-bit instructions.

Such manipulation can significantly reduce the overall number of branches.

The **flag** instruction contains 8 bits of truth-table plus a pop enable bit, which tells whether the used flags should be popped off the stack.

The truth-table is applied to the 3 top-of-stack flags - i.e. the 3 LSBs of the flag register.

When popping is enabled, it would be very unhelpful if it was always 3 flags that got popped.

Suppose one wanted to perform an or-operation on two flags, pop them and then push the result.

This instruction would do that:

flag.pop 0b11101110

No matter what the value of the 3rd flag is, it has no impact on the result. This is clearly seen from the repeating pattern of the truth-table.

The flag unit sees this too and only pops two flags. A minimum of one flag is popped when popping is enabled.

If a bundle contains two flag instructions, both are executed and process the same 3 flags. Conceptually, the result from instruction slot 1 is pushed first, then slot 0.

The instruction wanting to pop the most flags gets its way. A branch, wanting to pop one flag, is included in this max calculation.

Another example:

flag.pop 0b11110000

This one pops 3 flags and pushes the 3rd flag back on the stack. Effectively just removing the two top flags.

To see how flag-operations can cut down on the number of branches, here is an example code snippet from the Mandelbrot renderer:

```
branch add.f i3 = i6, i7     &   add i2 = i2, i4
       neg i2 = i2           &   nop
positive_xy:
       sub.f p2 = p2, 1      &   rotatel i3 = i3, 4
       and.f i3 = i3, 0xF    &   add i2 = i2, i1
       nop                   &   flag.pop 0b11111101
```

For each pixel, the inner loop of the renderer iterates a formula until one of two exit conditions is met.

To test these, 3 flags are generated by the 3 instructions with "**.f**" suffixes.

One can see how they are mixed with unrelated instructions and how they are combined by a **flag** instruction into a single loop control flag.

Another example is from the multiply demo:

```
sub.f i0 = i0, 48
sub.f i1 = i0, 10
add.f i3 = i3, i0         &   nop
flag.pop 0b11111011       &   nop
```

After a new input character has been received from the user over the UART, some input validation needs to be performed.

Is the character between ASCII '0' and '9'? Is an overflow generated when updating the value currently being input by the user?

Finally, branches can sometimes be eliminated completely:

```
copy i6 = p3            &   flag.pop 0b11101110
shiftr i6 = i6, 16-6    &   and i4 = fr, 1
add i6 = i6, i5         &   shiftl i4 = i4, 16+6
mul.u pi4 = i2, i0      &   add i6 = i6, i4
```

This snippet is from the Mandelbrot renderer. Logic, shift and rotate instructions in slot 1 can access the flags as a 32-bit register.

Slot 0 can only access the flag register directly in one way, namely the shift amount of a shift right instruction can be the flag register.

This makes possible the easy application of 32 entry truth-tables:

```
...                    &   load i0 = p7, truthtable-vars
shiftr.f i1 = i0, fr   &   rotatel fr = fr, 32 - 5
```

In the above example, a 32-bit truth-table is loaded into i0 and shifted according to the 5 top flags of the flag stack.

The bit ending up in position zero is pushed to the flag stack, but only AFTER the 5 flags are popped by rotating the flag register.

Note that slot 1 can only shift or rotate left, thus a shift right needs to be emulated as shown.

The conceptual order in which results are written to the flag register is:

1. Stack movement due to popping.
2. Push result from slot 1 "flag" instruction.
3. Push result from slot 0 "flag" instruction.

4. Direct flag register write by slot 1 instruction.
5. Push result from slot 0 flag generation - i.e. instruction with "**.f**" suffix.

Note that a slot 0 flag-instruction can take the "**.f**" suffix, causing it to generate a flag equal to its ordinary result.

This can be used to either push two identical flags or make the result survive a direct flag register write by slot 1.

Flag generation

The proposed ISA has no dedicated compare-instructions to generate flags. Instead it relies on flag generation as a side-effect of slot 0 arithmetic and logic instructions.

The "**.f**" suffix tells the assembler to set the flag generate bit of the bundle.

- **Add** instructions will generate an overflow flag.
- **Sub** and **neg** will generate a flag equal to the true 33'th-bit extended sign. It is more useful than the MSB of the result, because it can be used directly as a less-than flag.
- **Or**, **and**, **xor**, **not**, **copy**, **swap** and pointer-pointer subtraction generates a zero-test flag.
- **Shiftr** (shift right) generates a flag equal to bit 0 after shifting. **Shiftra** (shift right arithmetic) inverts the flag.
- **Flag** instructions can produce a duplicate of its ordinary result.

Memory access and addressing modes

Only instruction slot 1 can execute loads and stores. In slot 0, the same opcodes encode SIMD multiply instructions.

Only two addressing modes are available.

- **Pointer register + offset** adds a 3-bit scaled unsigned constant to a pointer to get the effective address. Only naturally aligned access is available.
- **Pointer register with scaled post-update** uses the pointer directly, but adds a 3-bit sign-extended constant to it after access.

The effective address is calculated in the decode stage, to have it ready for the execute stage where memory access occurs. This means that more advanced modes, like **pointer register + integer register index** are too slow to be feasible. The whole point of the split register file is to keep integer registers away from effective-address calculation. The forwarding network is more than twice as wide for integers compared to pointers, when looking at the forwarding multiplexors.

Since both memory access and flag generation is in the execute stage, forwarding has been moved to the decode stage. This means decode is by far the most complex stage.

Example of memory access with post-update:

```
     set i0 = 16-2
     flag 0                    &  push pc, @palette_copy+1
palette_copy:
     nop                       &  load.+ i1 = p6, 4
loop sub.f i0 = i0, 1          &  store.+ i1 -> p5, 4
```

The code snippet is taken from the Mandelbrot renderer where the VGA palette is initialized by a small block copy. Notice how address related offsets are always given in bytes. The assembler will divide values by 4 and give a warning if not an even multiple of 4. This is to avoid any problems that might otherwise arise should byte addressing instructions be added in the future.

Instruction format

Instructions are organized in 32-bit bundles usually containing two 15-bit instructions. A branch bit and a flag generate bit takes up the remaining two bits. Opcodes are 6 bits with one unused bit – i.e. only half the opcode space is currently used.

31	30 25	24 22	21 19	18 16	15	14 9	8 6	5 3	2 0
B	opcode1	D1	B1	A1	F	opcode0	D0	B0	A0

Extended instructions like 16-bit immediate add, sub and set take up an entire bundle:

31	30 16	15	14 9	8 6	5 3	2 0
B	Immediate	F	opcode0	D0	B0	H-S-I

There are several other instruction formats. Some instructions can only be executed by one of the two pipelines. In summary:

Pipeline 0:

- Can generate flags.
- Can read the program counter and top of BTS as arguments to pointer arithmetic.
- Add and subtract constants to the top of BTS.

- SIMD multiplications. Two pairs of 16-bit values contained in two integer registers are multiplied and the two 32-bit results are stored in same numbered integer and pointer registers. Not enough write ports to write both results to the integer file. Both signed and unsigned multiply is available.
- Right shifts. Both logic and arithmetic.

Pipeline 1:

- Can directly read and write the flag register using logic and shift/rotate instructions.
- Can push pointer registers to the BTS.
- Add/subtract offsets to the BTS stack pointer
- Memory access - load/store.
- Left shift and left rotate.

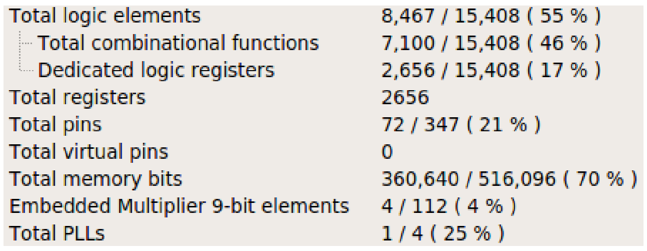
This asymmetry rarely causes any problems when writing code and is well worth it considering the savings in hardware.

Common instructions executable by both pipelines:

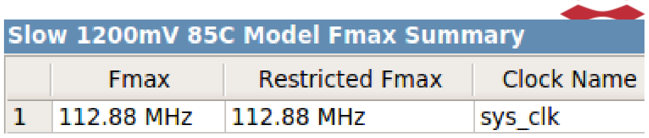
- Integer logical:
  - Or, and, and imm, xor, not.
- Integer arithmetic:
  - Add, add imm, sub, sub imm, neg.
- Pointer arithmetic:
  - Add int, add imm, sub int, sub imm, sub ptr.
- Push program counter + offset on BTS
- Flag logic
- Copy, swap, set, nop.

Synthesis results

Full design, including main memory, UART and 320x240x16 VGA display. Optimized for speed:

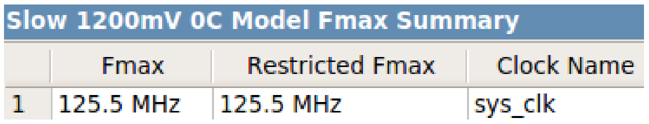


Total logic elements	8,467 / 15,408 (55 %)
Total combinational functions	7,100 / 15,408 (46 %)
Dedicated logic registers	2,656 / 15,408 (17 %)
Total registers	2656
Total pins	72 / 347 (21 %)
Total virtual pins	0
Total memory bits	360,640 / 516,096 (70 %)
Embedded Multiplier 9-bit elements	4 / 112 (4 %)
Total PLLs	1 / 4 (25 %)



Slow 1200mV 85C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name
1	112.88 MHz	112.88 MHz	sys_clk



Slow 1200mV 0C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name
1	125.5 MHz	125.5 MHz	sys_clk

Conclusion

The developed ISA works well for the few applications hand-written in assembler. It gives very dense code and not very many nops – e.g. close to a CPI of 0.5 for the Mandelbrot renderer. But how would a compiler cope?

The ISA seems awkward in the beginning, but one quickly gets used to it. In this paper a fair bit of focus has been put on code examples and "pattern" equivalents to traditional code.

The focus, when writing the VHDL, has solely been on speed. Simply to see what's achievable. Rewriting for area and optimizing for area during synthesis could likely cut logic usage by 30-40%.