Register Renamer

It is legal for the renamer to use physical register number zero. While architectural register number zero is not assigned a physical register. Architectural register number zero is always bypassed to zero and hence does not need a physical register number.

Physical target registers are assigned by Qupls\_reg\_renamer during the rename stage of the pipeline. Target registers for architectural register zero are not assigned.

I have forgotten the exact heuristic for the number of physical registers that should be present. It must be significantly more than the architectural number of registers. Since this architecture has a lot of registers, that means loads of them. Fortunately block RAM is used to implement the register file and can provide up to 512 physical registers. That many register are not needed. The design is restricted to 256 physical registers. This is about 3.5 times as many architectural registers, which should be plenty. The 256 registers not used in the block RAM are reserved for future usage.

The renamer uses four fifos that can each contain 64 rename register tags. The number of tags supported by all four fifos is thus 256, matching the number of physical registers. Each fifo may be used to allocate a target physical register every clock cycle. Therefore, up to four target registers may be assigned.

Alternate Register Renamer – low resource usage

* Circular buffer of register tags implemented with FPGA SRL’s.
* The renamer has to search for available register tags. It does this by rotating the SRL buffers until an available register tag shows up. A five times CPU clock is used for this search. In the worst case it may take 51 clock cycles to find an available register. (Having to rotate through all 255 potential registers). Not sure how to figure out what the average is. I tried the rotating renamer, and it ran for hundreds of instructions needing one single rotates. Really old registers tend to be freed up and they are the ones that are rotated into view first. When running in SIM it seems like there is always a register available. But it could be because to too short of a SIM run.
* If the same register were used 32 times in a row, (the size of the reorder buffer) it might take seven clock cycles to get past all the registers marked in use.

Register File

There are 320 logical registers required to support the full ISA register set including vector registers. Because of the large number of registers, the register set is implemented in block RAMs. To get the required depth of approximately 768 registers, two block RAMs are needed per register port. Approximately 128 block RAMs are needed to support 16 read ports and 4 write ports (16x4x2). The demo version uses only two write ports and 32 vector registers so only about 48 block RAMs are needed.

Physical Register zero is bypassed to the value zero.

The stack pointer register is banked depending on the operating mode. This is easily accomplished by adding the operating mode to the specified register. To be a little more efficient an ‘or’ operation is used instead of an add.

|  |  |  |
| --- | --- | --- |
| Vector Regno | Regno | Usage |
| 0 | 0 | Always written as zero, thus reads as zero |
| 0 to 3 | 1 to 30 | Programming model visible registers |
| 3 | 31 | “Safe” stack pointer (microcode), Alias for registers 32 to 35, looks like the stack pointer |
| 4 | 32 | Application / User stack pointer |
| 4 | 33 | Supervisor stack pointer |
| 4 | 34 | Hypervisor stack pointer |
| 4 | 35 | Machine Stack pointer |
| 4 | 36 | Micro code temporary |
| 4 | 37 | Micro code temporary |
| 4 | 38 | Micro code temporary |
| 4 | 39 | Micro code temporary |
| 5 | 40 | CTX |
| 5 | 41 | LR1 |
| 5 | 42 | LR2 |
| 5 | 43 | LR3 |
| 5 | 44 | LR4 |
| 5 | 45 | MCLR - Micro code link register |
| 5 | 46 |  |
| 5 | 47 |  |
| 5 | 48 | M0 |
| 5 | 49 | M1 |
| 6 | 50 | M2 |
| 6 | 51 | M3 |
| 6 | 52 | Vector – Global Mask Register |
| 6 | 53 | Vector restart mask |
| 6 | 54 | Vector exception |
| 6 | 55 | Card table address |
| 7 to 31 | 56 to 255 | 25 vector registers |

A ten-bit physical register tag is in use as there may be up to 768 or so registers needed. The last register tag of all ones is reserved for uninitialized registers. It is possible for the core to have a register as a source register before it has been properly loaded. In that event there would be no physical register assigned for it yet. This is represented with the tag of all one’s. Hardware forces this register valid so that the machine does not hang waiting for a register to be valid due to a software issue.

Note that block RAM is faster than a lot of the CPU logic. This can be made use of by multiplexing the BRAM write ports using a five times clock. This allows four write ports to be accessed during a single CPU clock and reduces the number of block RAMs required.

Decoder

Need to know if the architectural register is register zero in several places. So the decoder decodes this status into a single bit.

Instruction Extract

The BSR / BRA instruction is trapped in the extract stage and causes an immediate change of the IP. At decode, the BSR / BRA is flagged as done already and thus is never scheduled for execution.

It is too costly resource wise (50kLUTs!) to pack expanded vector elements in the instruction expansion table. Instead, vector instructions are expanded into fixed positions in the table, and the remainder of the table is filled with NOP instructions. Doing this requires 1/50th the resources, but has a performance impact as there are NOPs inserted into the instruction stream. For the typical case of no vector instructions, the table is packed with four scalar instructions. There are always at least four instructions present in the table. They may be NOPs.

Using longer vectors is more efficient than shorter ones because there are fewer NOPs processed. Using a sequence of consecutive vector instructions is better than mixing vector and scalar instructions.

May look at partially packing the table at some point.

Predicated Execution

Predicated execution of instructions and masking of vector operations is handled using a PRED instruction modifier. The modifier is placed in code before the instructions it applies to. Using the PRED modifier is more code dense than having a predicate register field in every instruction. The PRED modifier shows up only when needed, which is not for most instructions. A single PRED modifier applies for up to eight following instructions. A mask field in the PRED modifier allows instructions to ignore the predicate if the modifier is to be applied to fewer than eight instructions.

The PRED modifier modifies the scheduling of subsequent instructions. Up to eight following instructions may check the predicate status of the PRED instruction.

The PRED modifier is scheduled and executes like any other instruction. It amounts to a bit extract from a register then a case statement based on a mask. It is handled by ALU logic. The PRED modifier writes its result, which is 64-bits, to the ROB entry for the PRED instruction. The ROB was selected as the place to store the predicate result because the result is temporary and needed only by the scheduler for subsequent instructions. Scheduling of subsequent instructions checks for a prior PRED modifier. If found, the appropriate predication bit is then read from the ROB and used to either schedule the instruction on its functional unit (all bits in group have a non-zero value), or schedule the instruction as a copy target on the ALU (all bits in group = 0).

Vector Elements

A vector element is a 64-bit wide slice of a vector register which is treated as a single 64-bit register by the CPU. There are eight 64-bit wide elements to a vector register for a total of 512-bits.

Predicate Groups

When a predicate is applied, each vector element has a predicate byte associated with it. Each bit of the predicate byte is reserved for one or more bytes of the element.

Predicate values are grouped into groups of eight bits. Each bit represents a byte in a register. Each predicate byte represents a vector element. Since there are eight elements in a vector register eight bytes are required or 64-bits. If the element contains a 64-bit value then only the least significant bit of the byte for the element acts as a predicate bit. If the element contains two 32-bit values then the least significant two bits of the predicate byte for the element act as predicate bits. And so on.

|  |  |
| --- | --- |
| Lanes in Element | Bits Checked for Predication |
| 1x64 bit | 0 |
| 2x32 bit | 0 and 1 |
| 4x16 bit | 0 to 3 |
| 8x8 bit | 0 to 7 |
|  |  |

To set a true predicate for all lanes in a vector, where the lanes are 64-bit elements, the least significant bit of each byte of the predicate value must be set. The predicate value in this case would be 0101010101010101h. If other bits of the predicate are set they will be ignored. So, the predicate register may be loaded with all ones for instance, when lanes are 64-bit elements. The value FFFFFFFFFFFFFFFFh works as well.

Note that one of the set instructions will set the predicate bits appropriately for the size of lanes being compared.

Example two: 16-bit lanes are being used. Predicate value 0F0F0F0F0F0F0F0Fh will enable all lanes. The value FFFFFFFFFFFFFFFFh works as well. To mask off lane zero the value 0F0F0F0F0F0F0F0Eh would be used.

Sync

For the demo version, to reduce the logic footprint, any sync instruction will cause a pipeline flush. Demo sync does not resolve before and after fields of the instruction. This guarantees the sync will work at the cost of performance.

Quad Precision

Since the CPU is a 64-bit machine with 64-bit registers some means must be arrived at to perform 128-bit quad precision operations. The solution used is to perform the operation using register pairs. The pair of registers is specified by a combination of the quad precision instruction and an instruction modifier, QFEXT, dedicated to performing quad precision operations. The modifier supplies registers to hold the upper 64-bits of the quad precision value.

The quad precision operation then borrows an ALU port to act as a venue to be able to store the quad precision value. A quad precision operation uses the ALU as a holding place to store values. The scheduler sees the quad precision modifier and schedules it for the ALU. The modifier is does not complete its execution until the quad precision operation is complete. The scheduler schedules a quad precision operation as a pair of operations, one on an ALU used for passthrough, and one on the floating-point unit.

ALU Pair Instructions

ALU instructions that require a pair of ALUs are issued to two ALUs at the same time. Both ALUs see the same instruction. However, the ‘C’ register port is used as the target register for the high-order ALU. For instance, the MULW, multiply widening instruction, causes both ALUs to perform the multiply however high order product bits are written by ALU #1 while low order product bits are written by ALU #0.

Stomping on Instructions

If the instruction is stomped on before the enqueue stage (rename etc) then registers are not renamed and the instruction is not enqueued, so that it does not waste queue slots. However, if the instruction about to be queued is stomped, it is allowed to be queued as the renamer has already assigned registers, and it is difficult to undo the assignment. So, things are left as is in that case, but the instruction is marked as a copy target instruction.

If there was a cache miss at the fetch stage, it must percolate down to the subsequent stages as the pipeline is advanced. A fetch from micro-code is not considered to be a miss even if there is a cache miss.

Scheduler

Window size – the scheduler does not look at every instruction in the ROB to choose what to issue. It has a fixed size window that looks backward into the queue from the oldest instruction towards the newest one. Since newly queued instructions are unlikely to be ready to execute, to reduce hardware cost the scheduler does not look at them. Older instructions are considered for issue before newer ones.

Signal Naming Conventions

Many signals are named depending on the pipeline stage.

|  |  |
| --- | --- |
| Stage Output | Signal Postifx |
| Fetch | \_f |
| Extract | \_x |
| Decode | \_d |
| Rename | \_r |
| Enqueue | \_q |

Micro-code

Micro-code instructions short circuit the first two pipeline stages fetch and align as fetch and align are not required for micro-code.

Fetch continues but instructions are sourced from the micro-code store.

The fetch address continues to increment. At the end of the micro-code function a branch is made back to the next instruction after the macro-instruction that triggered the micro-code.

Memory Model

Q+ uses virtual addresses to access memory and I/O.

Relocation of sections of memory.

Protecting memory from access by the wrong program.

Sharing sections of memory between different programs.

It is critical that sections of memory associated with a program are relocatable.

Relocation of Sections

There are several different types of sections that may be relocated. These include code and data. Data includes stack, constants, pre-initialized data and uninitialized data.

Code. Program code can be made relocatable by using only relative addressing within a program. Subroutines that are to be externally visible need to be tracked with a map in the operating system.

Protecting Memory

Memory is protected from access by the wrong program via the use of a memory key associated with each page of memory. Unless the program has a key to the memory it is not able to access it. Each page of memory also has access rights associated with it. It may be readable, writeable or executable.

Capabilities

Capabilities instructions execute on the floating-point unit as that unit has access to register pairs for 128-bit wide operations. The quad-float register extension prefix instruction can be used to select the high-order half of the capabilities register.