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.

Register File

There are 512 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 1536 registers, three block RAMs are needed per register port. Approximately 192 block RAMs are needed to support 16 read ports and 4 write ports (16x4x3). 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 or interrupt level. 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 7 | 1 to 62 | Programming model visible registers |
| 7 | 63 | Alias for registers 65 to 68, looks like the stack pointer |
| 8 | 64 | Machine stack pointer |
| 8 | 65 to 71 | Interrupt stack pointers |
| 9 | 72 | Application / User stack pointer |
| 9 | 73 | Supervisor stack pointer |
| 9 | 74 | Hypervisor stack pointer |
| 9 | 75 | Safe stack pointer |
| 9 | 76 | Micro code temporary |
| 9 | 77 | Micro code temporary |
| 9 | 78 | Micro code temporary |
| 9 | 79 | Micro code temporary |
| 10 | 80 to 511 | 54 vector registers |

An eleven-bit physical register tag is in use as there may be up to 1536 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.

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.

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 a single byte, 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 (bit=1), or schedule the instruction as a copy target on the ALU (bit = 0).

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.