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.

Register File

Because there are only four write ports, writing to architectural register zero bypasses the value to zero on a write. Otherwise much more bypassing would be required to bypass the value read to zero on read ports.

The stack pointer register is banked depending on the operating mode. This is easily accomplished by adding the operating mode to the specified register. Registers 65 to 68 are used as the stack pointers.

|  |  |
| --- | --- |
| Regno | Usage |
| 0 | Always written as zero, thus reads as zero |
| 1 to 62 | Programming model visible registers |
| 63 | Alias for registers 65 to 68, looks like the stack pointer |
| 64 | Not used, looks like zero in some cases. |
| 65 | Application stack pointer |
| 66 | Supervisor stack pointer |
| 67 | Hypervisor stack pointer |
| 68 | Machine stack pointer |
| 69 | Micro code temporary |
| 70 | Micro code temporary |
| 71 | Micro code temporary |
| 72 | Micro code temporary |
| 73 | Micro code stack pointer |

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.