ARM Cortex A8: A High Performance Processor for Low Power Applications

David Williamson

Consulting Engineer, ARM

AUTHOR BIO:

David Williamson is a Consulting Engineer at ARM where he has worked since 1999 on the development of various ARM processors. Most recently, he was the co-architect of the Cortex A8, and manager of the Cortex A8 performance modeling and validation teams. Prior to his work at ARM, David worked at AMD from 1995 on the development of the AMD Althon™ processor.

ACKNOWLEDGEMENTS

Writing this chapter has not been a singular effort and has involved the help of many others involved in the design and development of the Cortex A8 processor. Specifically, I would like to thank the Cortex A8 design team for their input and help in the development of this chapter.

[Contact Info:
David Williamson
7300 Twilight Shadow
Austin TX 78749
(512)653-8228
David.williamson@arm.com]

1 CORTEX A8

1.1 Overview

The ARM Cortex-A8 is a microprocessor targeted at systems that require high performance for both general purpose and media applications while maintaining a low, sub 1 Watt, power profile and a small silicon footprint. This processor targets a significantly higher performance point than any previous ARM processor. The increased level of performance for general purpose applications is realized through an energy-efficient balance of both increased operating frequency and improvements in machine efficiency as measured by Instructions Per Cycle (IPC). The increase in frequency is achieved using a deeper pipeline with less logic depth per stage when compared to previous ARM cores. The increase in IPC comes mainly from superscalar execution of instructions, but the improved branch prediction, efficient memory system, and other features contribute as well to the machine performance. Performance for media and graphics applications is increased even further than what is achieved for general purpose applications with a 64bit SIMD integer and floating point engine (NEON).

1.2 Instruction Set Architecture

The ARM architecture is a load/store architecture with an instruction set that it largely consistent with other RISC processors. Some special attributes worth mentioning include instructions capable of both shift and alu operations in the same instruction, the ability to use the program counter as a general purpose register, support for variable 16bit and 32bit instruction opcodes, and a fully conditional instruction set. The ARM integer register file includes 16 32bit registers. 13 of these registers are general purpose. The remaining three special purpose registers are the stack pointer, a link register, and the program counter. While these registers have special uses, they can also be used by most data processing and load/store instructions. The classic floating point and new NEON media instructions both use a second register file that contains 32 64bit registers. When used for integer SIMD operations, each register can contain a single 64bit value, two 32bit values, four 16bit values, or eight 8bit integer values. When used for floating point operations, each register can contain a single 64bit double precision value or two 32bit single precision values.

1.3 Basic pipeline description

Cortex A8 is an in-order, dual-issue superscalar processor with in-order instruction issue, execution and retire. The processor has a 13-stage main pipeline that is used for all instructions. This main pipeline can be broken into three decoupled, parts: Fetch, Decode, and Execute. Individual pipeline stages within each part are simply numbered: F1, F2, D0, D1 etc. The two Fetch stages at the front of the pipeline are responsible for predicting the instruction stream, fetching instructions from memory, and placing the fetched instructions into a buffer for consumption by the Decode pipeline. The five Decode stages take care of decoding, scheduling, and issuing instructions. They also deal with sequencing complex instructions and replaying instruction sequences when a memory stall occurs. The six Execute stages consist of two symmetric ALU pipelines, a load-store pipeline, and a multiply pipeline. In addition to the main pipeline, there is a 10 stage pipeline for the NEON SIMD execution engine, an eight stage pipeline for the Level-2 memory system, and a 13 stage pipeline for the debug trace generation. The 10 stage NEON pipeline includes four stages of instruction decode and issue and six stages for instruction execution. NEON instruction decode stages are numbered M0, M1, M2… and NEON execute stages are numbered N1, N2… Level-2 memory system pipeline stages are numbered L1, L2… A diagram including all stages in the full pipeline can be seen in Figure 1. Each main section of the pipeline will be discussed in more detail in the subsequent sections.

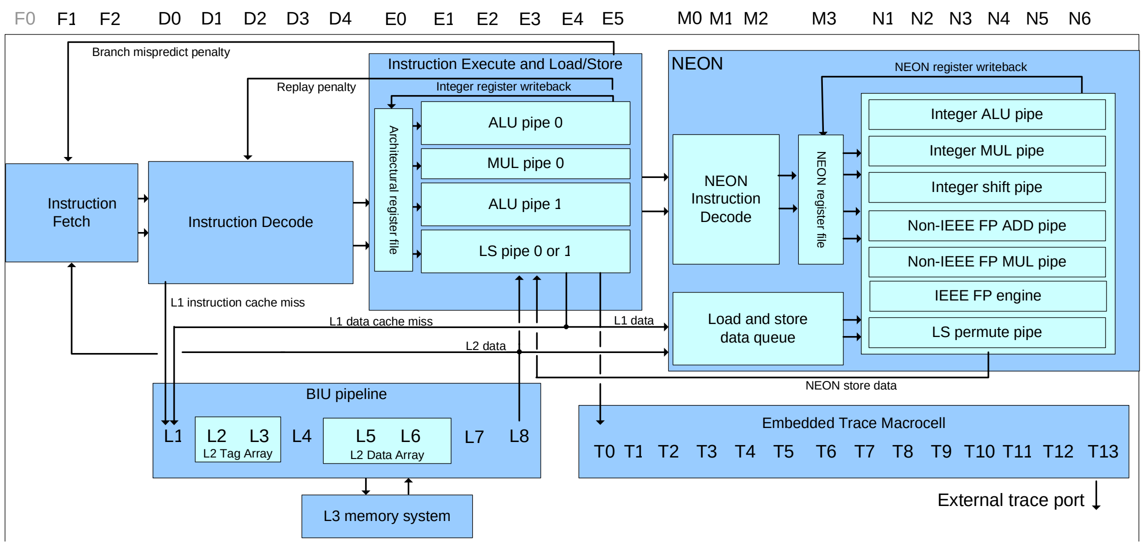

Figure 1 Cortex A8 full pipeline

2 INSTRUCTION FETCH

2.1 Instruction Fetch Pipeline Overview

The Instruction Fetch unit (I-Fetch unit) includes the entire Level-1 Instruction side memory system as well as dynamic branch prediction, and instruction queuing hardware. The Instruction Fetch pipeline runs decoupled from the rest of the processor, speculatively fetching up to four instructions per cycle along the predicted execution stream and placing them in the instruction queue to be consumed by the decode unit.

The fetch pipeline begins with the F0 stage where a new virtual address is generated. This address can either be a branch target address provided by a branch prediction for a previous instruction, or if there is no prediction made this cycle, the next address will be calculated sequentially from the fetch address used in the previous cycle. Note that the F0 Fetch stage is not counted as an official stage in the 13 stage main integer pipeline. This is because ARM processor pipelines have always counted stages beginning with the Instruction Cache access as the first stage.

Once an address has been calculated, it is used to access the Instruction Cache arrays to obtain data for the next set of instructions in the F1 stage. In parallel, the fetch address is also used in the F1 stage to access the branch prediction arrays to determine if a branch prediction should be made for the next fetch address.

In the final fetch pipeline stage, the F2 stage, instruction data is returned from the Instruction Cache (assuming a hit occurs) and placed into the instruction queue (or the queue bypass registers) for future consumption by the decode unit. Also in the F2 stage, if this instruction resulted in a branch prediction, the new target address is sent to the address generation unit to be used as the next fetch address.

When a branch prediction for a taken branch is made, the instruction currently in the F2 stage is changing the fetch address that is calculated in the F0 stage. Therefore, the instruction fetch currently in the F1 stage will need to be thrown away. This means there is a one cycle bubble in the fetch pipeline whenever a branch prediction is made for a taken branch. Typically, this bubble is not exposed since the fetch engine runs ahead of the rest of the machine, but it can be exposed in branch heavy code sequences or any taken branch that closely follows a branch misprediction.

2.2 Instruction Cache

The Instruction Cache is the largest component of the Instruction Fetch unit. It is a physically addressed, 4-way set associative cache capable of returning 64bits of data per access, and it is configurable to be either 16KB or 32KB in size. The cache line length is 64 bytes and line replacement is done using a random policy. The Instruction Cache also includes a 32-entry, fully associative TLB. TLB misses are serviced by a hardware table walk mechanism that is part of the Level 2 memory system.

To minimize design effort, the Instruction and Data Caches are essentially identical, making use of the same array structures with only the minimum differences in the control logic that are needed to support each. In most ways, the level one cache design is a traditional cache design and it includes the typical data array, tag array, and TLB structures found in most processor caches. However, there is one additional structure in the Cortex A8 that is not present as part of a traditional cache design, the hashed virtual address buffer (HVAB) array.

A traditional set associative cache fires all ways of the data and tag RAMs in parallel at the same time that the physical address is being read from the TLB. This physical address is then compared with the values read from the tag array to determine which of the data RAM ways contains the required data and should be selected (or if a cache miss occurred). To avoid firing all these arrays in parallel, Cortex-A8 implements a way indication scheme based on a 6 bit hash of the virtual address and process ID of the access. This hash is used to index into the HVAB and determines the cache way that is likely to contain the data. This look up is done quickly and is available in time to prevent firing of all the ways in both the data and tag arrays. A TLB translation and tag compare is still required in order to validate the hit. If the hit proves to be incorrect then the access is flushed, the HVAB and Cache data is updated, and the access is repeated. Even though the TLB translation and tag read of the way containing the data are still required, they are removed from the critical path of the cache access which is another benefit of the HVAB array.

Since the HVAB only brings a benefit in power savings when its hits are correct and actually costs performance and energy when it predictions are wrong, the key to its success is a good hash function that has a low likelihood of generating false matches. Another key attribute of a hash function in this application is that it can be evaluated relatively quickly. The hash used in Cortex-A8s caches is a two-level XOR reduction that mixes the address bits and process ID in a carefully selected order. Modeling this function across a large range of applications has shown a negligible increase in additional cache misses from the use of the hash function.

2.3 Instruction Queue

Fetched instructions from the Instruction Cache are placed into the instruction queue (IQ), or registered for forwarding on to the D0 stage if the IQ is empty. The purpose of the IQ is to absorb instruction delivery and consumption discontinuities between the Instruction Cache and the Decode Unit. The decoupling afforded by the queue allows the I-Fetch unit to prefetch ahead of the rest of the integer unit and build up a backlog of instructions that are ready to be decoded. This backlog often hides the latency involved in predicting a change in the instruction stream and starting to fetch instructions from a new location. The queue also prevents stalls from the decode unit from propagating back into the prefetch unit in the same cycle that a stall condition is detected.

The IQ is organized physically as four parallel FIFOs, each six entries deep for a total of 24 entries. Each entry is 20 bits wide, holding 16 bits of instruction data and 4 bits of control state. Depending on the size of the instruction, an instruction may be contained within just one or two entries of the queue. Packing instructions in the queue in this manner can be complex to implement but it has efficiency advantages for the intermixed 16- and 32-bit long instructions that are common in the Thumb instruction set.

2.4 Branch Prediction

The branch predictor includes two arrays: the 512 entry Branch Target Buffer (BTB) and the 4096 entry Global History Buffer (GHB). The BTB indicates whether or not the current fetch address will return a branch instruction and, if so, gives the branch target address. The GHB contains 2-bit saturating counters that give the indication on whether conditional branches should be predicted taken or not taken.

The branch prediction arrays are both accessed in parallel with the Instruction Cache access in the F1 stage. The GHB entry is selected using a 10-bit global branch history and four low order bits of the PC. Branch history is created from the taken/not taken status of the ten most recent branches.   This information is saved in the Global History Register (GHR). Using branch history to determine prediction works well since heuristically instruction traces tend to take similar paths through a program creating different histories that predict the outcome on the next subsequent branch. The only flaw to a global history type of prediction is that it is possible to alias on two similar histories that differ in the n-th branch where n is the number of history bits + 1. To help prevent this type of aliasing, low order instruction address bits are also used to index the GHB. The GHB has 4096 entries, but is organized as a 256 entry by 32-bit array. So, only the upper eight bits of history are used to access the array and the final indexing based on remaining history and low order PC bits is done after the array is accessed. Each access to the GHB reads out 16 2-bit prediction values, each of which indicates whether the next branch should be predicted taken or not taken. The 16 values are multiplexed down to a single prediction using an XOR combination of the remaining history bits and the low order PC bits. GHB accesses always return a valid value and therefore there is no concept of a GHB miss. Instead, the GHB prediction is qualified by a hit from the BTB. To save power, the GHB is only accessed when the global history has changed. Since branch history is only updated on the prediction of a branch, the GHB array is not accessed any more often than necessary.

The BTB is indexed by the fetch address and contains branch target addresses and information about the branch type. The BTB stores predicted target addresses for both direct and indirect branches. On a BTB hit, if the entry is marked unconditional, the branch target address is used to fetch the next instruction. If the branch is conditional, then the value returned from the GHB access indicates whether or not the target address should be used. If the GHB lookup indicated that the branch was not taken, then the Instruction Cache continues fetching sequentially. On a BTB hit, the global history used to access the GHB is updated by shifting the taken/not taken status bit into the lowest order bit of the global history. This update to the GHB history is done in the D2 stage (two cycles after the prediction).

All branch predictions are resolved in the E4 stage at the end of the integer pipeline by comparing the predicted and calculated PCs. If the branch mispredicts, or a taken branch was not predicted, the pipeline is flushed and the BTB and GHB arrays are updated accordingly. The Global History Register, used to index the GHB, is updated as well to a non-speculative version of the global history to keep the predictor more accurate. In the case of a correct prediction, the GHB saturating counter and the non-speculative global history register are both updated. Figure 2 shows the Instruction fetch pipeline and the branch mispredict update path.

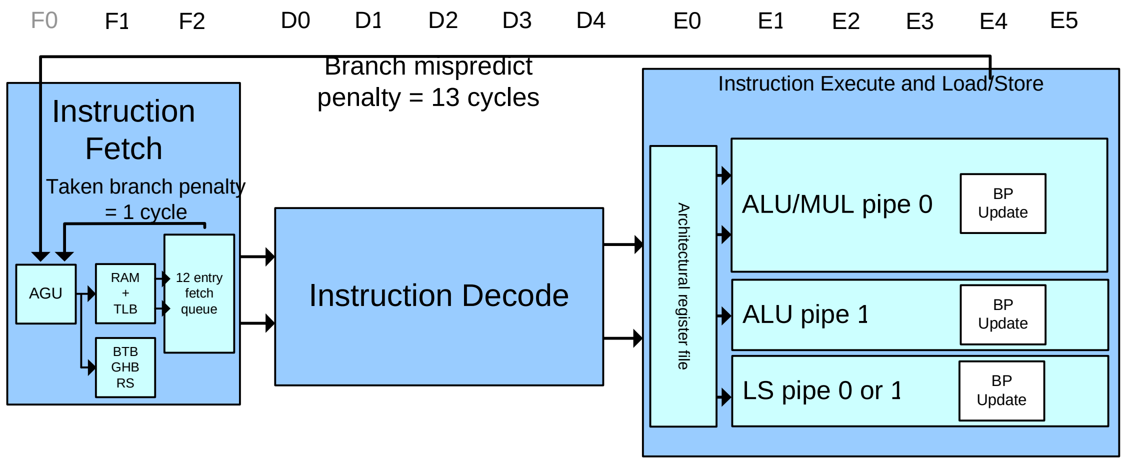

Figure 2: Instruction Fetch Pipeline and Branch Update

2.5 Return Stack

Cortex A8 also makes use of a return stack for subroutine prediction. The subroutine return stack depth is 8 entries. Return addresses are pushed onto the stack when the BTB lookup indicates that the branch is a subroutine call. When the BTB lookup indicates that the instruction is a subroutine return, the branch target address is popped from the return stack instead of being read from the BTB entry. Subroutines are often short and therefore it is important to support multiple push and pop operations in flight in the pipeline at a time. However, speculative updates to the return stack can be destructive since an update from the incorrect path can result in getting the return stack out of sync, generating multiple mispredictions. To achieve the performance benefits of speculative updates to the return stack without the performance cost, the Instruction Fetch unit maintains both a speculative and non-speculative return stack. The speculative return stack is read and updated immediately based on the information returned from the BTB access while the non-speculative stack is not updated until the branch is known to be non-speculative at the end of the pipeline. If a branch is mispredicted, then the state of the speculative stack is overwritten by the state in the non-speculative stack.

3 INSTRUCTION DECODE

3.1 Instruction Decode Pipeline Overview

The Instruction Decode unit is responsible for decoding, sequencing, and issuing instructions. It also handles sequencing of exceptions and other unusual events. The main blocks within the decode unit include the instruction decoders, the instruction sequencer, the pending and replay queue, and the instruction issue logic.

The logic of the decode unit occupies the D0 – D4 stages of the pipeline. Early instruction decoding is done in pipeline stages D0 and D1. In these stages the instruction type, the source and destination operands, and resource requirements for the instruction are all determined. Multi-cycle instructions are broken down by the sequencer into multiple single cycle operations in the D1 stage. These early stages generate all the decode information that is needed by the issue logic that lives in the D3 stage of the pipeline. The D2 stage is used to write instructions into and to read from the pending/replay queue structure. The instruction scheduling logic operates in D3. The scoreboard is read in this stage for all the operands of the next two instructions that could issue. These instructions are read from the pending queue or directly from the D2 stage if the pending queue is empty. The two instructions are also cross-checked against each other to check for any other dependency hazards that would not be detected by the scoreboard. The cross checks and scoreboard results are combined to determine whether 0, 1, or 2 instructions will be issued. Once this issue decision is made and the next set of instructions is issued across the D3/D4 boundary, these instructions cannot be stalled. After this point, instructions advance one pipeline stage per cycle and the replay mechanism will be used to handle any unpredictable hazards from the memory system (cache miss, store buffer full, etc.). The D4 stage performs final decode for all the control signals required by the I-Execute and Load-Store units. These are then registered and sent to the I-Execute and Load-Store units in E0.

Figure 3 outlines the structure and the pipeline of the Instruction Decode unit:

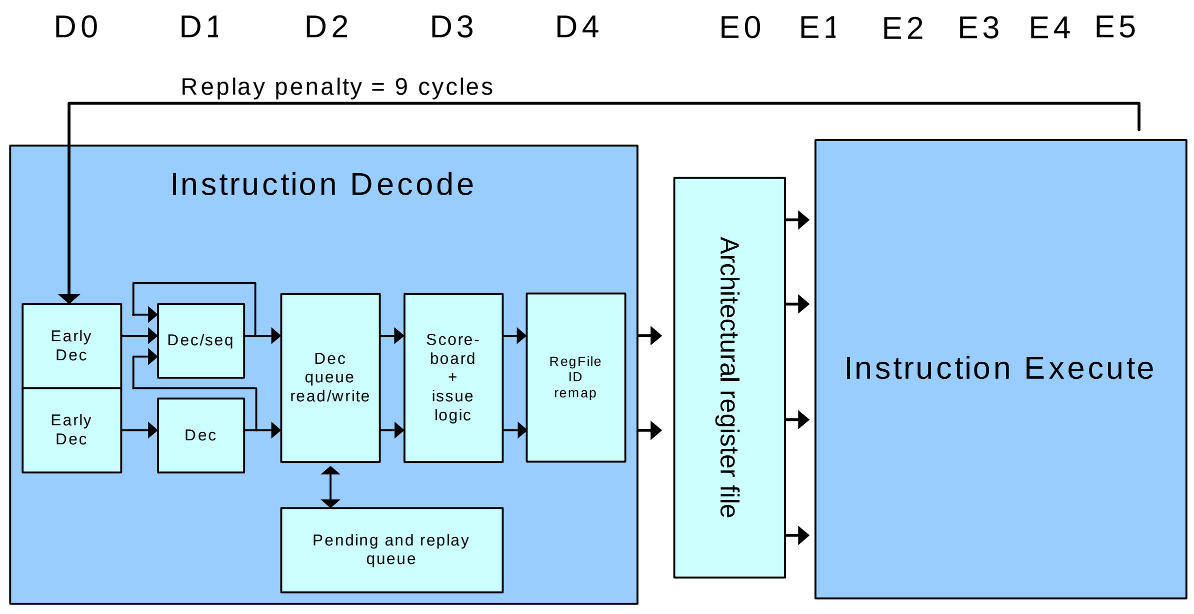

Figure 3: Instruction Decode pipeline

3.2 Static Scheduling Scoreboard

The scoreboard in CortexA8 is a predicting scoreboard that statically predicts when operands should become available. This is different and more complex than a traditional scoreboard that uses a single bit to indicate whether or not a source operand is ready for use. Rather than returning a single bit, the value read from the Cortex A8 scoreboard indicates the number of cycles until a valid result will be available for forwarding to a following instruction. This information is used in combination with when the source operand will be needed to determine if a dependency hazard should prevent the instruction from issuing. For example, we have an instruction being scheduled that requires the register R1 to be available as a source operand in the E2 stage. Since the scoreboard is accessed in the D3 stage, the E2 stage is four cycles away in the pipeline (D4, E0, E1, and E2). Thus, if the scoreboard indicates that the value for R1 will be available in 4 cycles or less, then no hazard exists on this source operand. However, if the scoreboard indicates that the value will not be available for 5 cycles or more, then the instruction could not issue due to the dependency. In order for this scoreboard to work properly, in addition to being updated whenever a new register is written, each entry in the scoreboard is also self updating on a cycle-by-cycle basis. The value in each scoreboard entry will count down by one each cycle until updated by a new register write or the value reaches zero, indicating that the value in now available directly from the register file. In addition to indicating when an operand will be available, each scoreboard entry also contains information to track which execution pipeline is producing the register result and what stage in the pipeline the producing instruction is currently in. This additional information is used to generate the forwarding multiplexer control signals that are passed along with the consuming instruction when it is issued.

As mentioned above, the static scoreboard implemented in CortexA8 is more complex than a traditional scoreboard. However, there are several advantages to using this method. First, when used in combination with the replay queue, it allows the implementation of a fire-and-forget pipeline with no stalls in the execution pipeline. This is important for removing fundamental speedpaths from the design that would otherwise prevent high frequency operation of the processor. Secondly, it is also a good technique to use in a low power design since knowing early in the pipeline which execution units will be required each cycle allows aggressive clock gating without the creation of speedpaths in the design.

3.3 Instruction scheduling

Cortex A8 is a dual issue processor with two main integer pipelines called ‘pipe0’ and ‘pipe1’. When two instructions are issued, pipe0 will always contain the older instruction in program order and pipe1 will always contain the younger instruction. This means that if the older instruction in pipe0 cannot issue, then the instruction in pipe1 will not issue, even if no hazard or resource conflicts exist for that instruction. When only one instruction is issued, it is always issued in pipe0. Furthermore, all issued instructions will progress in order down the execution pipeline and retire with results written back into the register file in the E5 stage. This in order nature of instruction issue and retire completely prevents WAR hazards and keeps tracking of WAW hazards and recovery from flush conditions straightforward.

If no RAW hazard is indicated from the scoreboard, the instruction in pipe0 should be free to issue. However, there are other constraints to consider besides just the scoreboard indicators for dual issuing a pair of instructions. The first thing to consider is the instruction types for the two instructions and whether or not that combination is supported. In a given cycle, the following combinations of dual issue are supported:

- Any two data processing instructions
- One load-store instruction and one data processing instruction, in any order

- Older multiply instruction with a younger load-store or data processing instruction

In addition to the constraints listed above, only one of the two instructions issued can change the program counter. Instructions that change the PC include traditional branch instructions as well as any data processing or load instruction with the PC as the destination register.

In addition to the resource check, the two instructions are also cross-checked against each other to determine whether there are any RAW or WAW hazards between the two instructions that would prevent dual issue. If both instructions are writing to the same destination register (a WAW hazard), or if the younger instruction needs a destination register before it is produced from the older instruction (a RAW hazard), then dual issue is prevented. It's worth noting that when checking for a RAW hazard, it is not enough to simply check if the second instruction requires a source operand that is a destination register form the first instruction. In a similar fashion to what is done when accessing the scoreboard, a comparison is done between when the data will be produced by the older instruction and when it is needed by the younger instruction. If the data isn't needed until one cycle or more after it is produced, then the dependent pair can still be dual issued. Some examples of cases where this occurs include:

- Compare or subtract instruction that sets the flags followed by a flag-dependent conditional branch instruction
- Any ALU instruction followed by a dependent store of the ALU result to memory
- A move or shift instruction followed by a dependent ALU instruction

This dual issue of dependent instruction pairs is a key feature in the design that provides a significant performance increase since the pairs of dependent instructions mentioned above are quite common in typical code sequences.

3.4 Replay and pending queue

The D2 stage of the decode pipeline is where instructions are inserted into and extracted from the pending/replay queue structure. This is a single structure with multiple read and write pointers used to hold instructions for two purposes. The pending portion of the queue holds new instructions that have not yet been issued while the replay entries in the queue hold instructions that have recently been issued and are still in flight in the execution pipeline.

The pending queue serves two purposes. First, it prevents the stall signal generated from the issue logic from rippling any further up the pipeline. The determination in D3 of how many instructions will be issued occurs very late in the cycle. Therefore, it is important to limit the fanout of this signal to maintain the high frequency operation of the design. The second purpose of the pending queue is to pack the pending instructions as close as possible so there are always two instructions available to consider for dual issue. In the case where only one instruction is issued, it is important that the next two instructions can be considered together, even though they were originally sent from the fetch unit in different cycles. The packing done by the pending queue makes this possible and maximizes the opportunities for dual issue of instructions.

When instructions are issued, they move naturally from the pending queue to the replay queue simply by manipulating the queue pointers. The replay queue is the other key component, along with the static scheduling scoreboard, in the Cortex A8 fire-and-forget pipeline. As mentioned in the scheduling section, instructions are statically scheduled in the D3 stage based on a prediction of when the source operand will be available. However, there are some cases, due to stalls from the memory system, when the data will not be available as expected. To handle these cases a recovery mechanism is used to flush all subsequent instructions in the execution pipelines and re-issue (replay) them. The replay queue records every instruction in flight in the integer execution pipeline. Instructions are placed in the queue before they are issued and removed as they write back their results and retire. The replay queue tracks the information necessary to restart instruction execution cleanly from the issue point.

The most common memory system stall is a Level-1 Data Cache miss. Therefore, the time taken to replay is balanced to be equal to the minimum Level-2 cache hit latency of eight cycles. i.e. If the instruction were stalled, rather than replayed, it would still spend eight cycles waiting for the data to be returned from the L2 cache. Most other causes of replay have latencies longer than eight cycles before they are resolved. To prevent multiple replays from occurring on a longer stall condition, an indicator from the memory system is used to hold the first replayed instruction until the appropriate time for it to issue.

The position of the replay queue in D2/D3 is a trade-off between performance and area/power costs. The earlier the queue in the pipe the deeper but narrower it is and the greater the cycle cost of replay. At one extreme replay could be implemented by re-fetching instructions from the I-cache. However, that would result in an unacceptably long replay penalty. At the other extreme, replay could be implemented using a queue of all the control signals crossing the decode-execute issue point, but that would result in an unacceptably large structure required for the replay queue and would also not allow it to be combined as easily with the pending queue. The best trade off is to position the queue in D2. This allows for the combined structure with the pending queue and also matches well to the minimum L2 cache hit latency.

3.5 Multi-cycle Instructions

The ARM instruction set is considered a RISC instruction set and therefore the large majority of commonly executed instructions have a single opcode and make one pass through the execution pipeline. However, there are a few, less commonly executed complex instructions that must be broken down into multiple instruction opcodes and make multiple passes through the execution pipeline. These instructions are called multi-cycle instructions. Multi-cycle instructions are unrolled into micro-ops in the sequencer which operates in the D1 stage. The micro-ops then move on to D2 to be queued in the pending and replay queue. Each micro-op of a multi-cycle operation is effectively treated as if it were an independent instruction. To help in the expanding of multi-cycle instructions into multiple micro-ops, a temporary register (Rtmp) is used to pass data from one micro-op to a subsequent one. Rtmp is scheduled using the scoreboard and issue logic just like any other register, but it is only used to pass intermediate results between micro-ops that are part of a single multi-cycle instruction. When micro-ops from a multi-cycle instruction are issued, they will issue at the rate of one micro-op per cycle. However, it is possible to pair the last micro-op with a following independent instruction.

3.6 NEON SIMD Instructions

Although the bulk of Advanced SIMD instruction decode takes part in the NEON unit itself, there is still some amount of work for I-Decode to perform on these instructions. First, valid NEON and floating point instructions are recognized and tagged for routing to the NEON engine and any undefined NEON instructions are trapped. Second, all NEON loads, stores and register transfers are decoded and scheduled since the main pipeline is responsible for providing data to the NEON unit for these instructions. Third, before issuing NEON instructions, the decode unit must first check the number of NEON instructions currently in flight that have not yet been consumed by the NEON unit. If the NEON queue could overflow from too many instructions, then issue must stall until a slot in the NEON Instruction queue is free. All decoding and scheduling of NEON instructions besides these cases is handled within the NEON unit.

4 INTEGER EXECUTE

4.1 Integer Execute Pipeline Overview

The Integer Execution unit is pipelined across the E0 to E5 stages. It is responsible for doing the full execution of all ARM data processing, multiply, and traditional branch instructions. It is also responsible for maintaining the program counter, resolving condition codes for conditional instructions, generating addresses for load/store instructions, and prioritizing all potential pipeline flushes due to exceptions, branch misprediction, or a memory replay. NEON data processing instructions pass straight through the execution pipeline and are passed into the NEON instruction queue after the E5 stage.

The register bank is accessed in E0. Up to 6 registers can be read from the register file for 2 instructions (a maximum of 4 sources per instruction). After the register file is accessed, instructions are sent to one of two symmetrical ALU pipelines named Pipe-0 and Pipe-1. All instructions have an instruction component that is sent down either Pipe-0 or Pipe-1, even load/store, NEON, and multiply instructions that will not make use of the shifter or ALU. This component of the instruction is used for maintaining program order for instruction flushing, PC tracking, and data forwarding. The multiply-accumulate unit is bound to ALU Pipe-0 and therefore multiplies will always be the older instruction if dual issued. The load-store pipeline can be combined and used with either Pipe-0 or Pipe-1 on an instruction-by-instruction basis. This allows loads and stores to issue as either the older or the younger instruction, increasing the dual issue opportunities.

In the two symmetric ALU pipelines, the E1 stage contains the shifter and the E2 stage includes the ALU. Most data processing operations are completed using either the shifter logic in E1 or the ALU present in E2. Some ARM instructions require use of both the shifter and the ALU which is why these blocks are pipelined instead of being implemented in parallel in a single execute stage. The E3 stage is used to complete saturation arithmetic used by some ARM data processing instructions. In E4, any change in control flow including branch mispredictions, exceptions, and memory system replays are prioritized and processed. The multiply pipeline does the actual multiply operation in the E1, E2, and E3 stages. Multiply accumulate instructions do the accumulate operation in the E4 stage. All results of ARM instructions are written back into the register file in the E5 stage. Even when a result for an instruction is available early, it is always written to the register file in order in the E5 stage and made available early using forwarding logic. This prevents the creation of any WAW hazards.

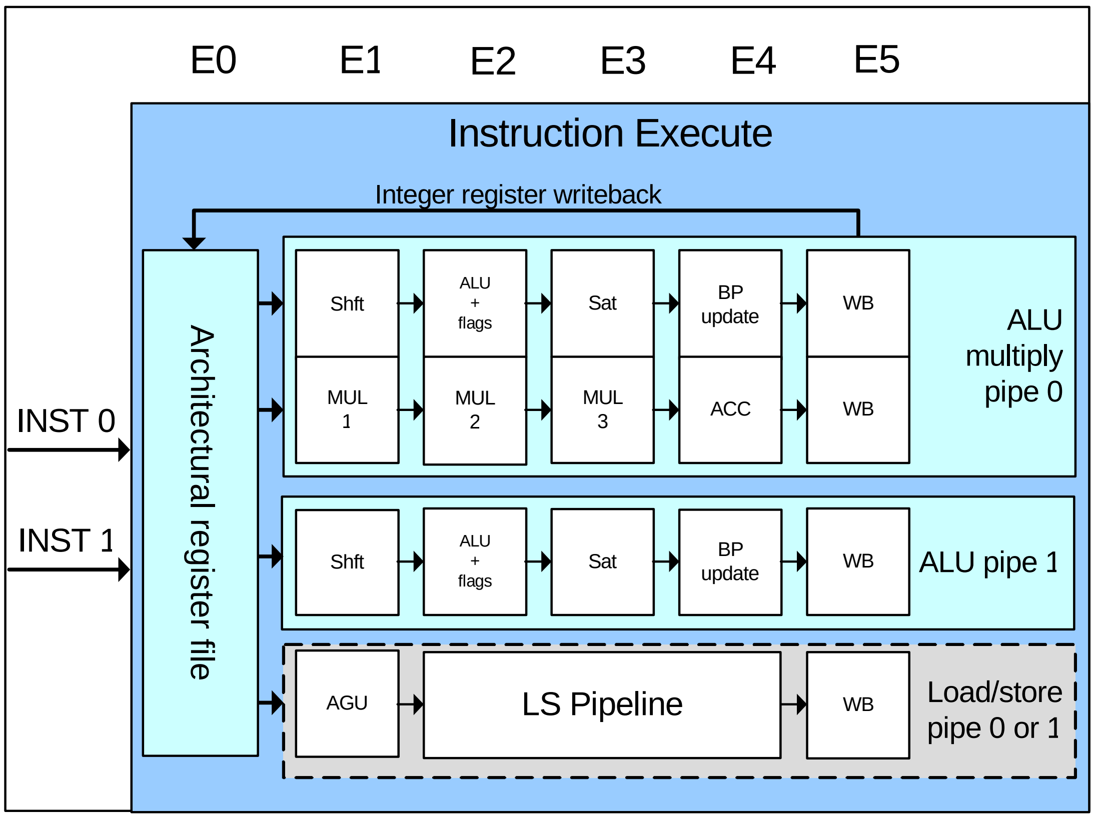

Figure 4: Execution pipeline

4.2 Processing flags and conditional instructions

Like most processor architectures, the ARM instruction set defines arithmetic flags such as carry, overflow, sign, etc. that are used to resolve branches and other conditional instructions. There are four of these flags and they are part of a larger processor status register called the CPSR. These flag bits are difficult to scoreboard in the traditional fashion since some instructions set just a subset of the flags. To scoreboard them properly, each individual bit would have to be tracked in the scoreboard as a separate register. This is too much overhead and complexity. Therefore, flags are not placed in the scoreboard like other registers. Instead, maintenance of the latest values of the flags is handled within the execute unit in the E2 stage. To keep this maintenance as straightforward as possible, all instructions that update the flags or read the flags must do so in one of the two ALU pipelines in the E2 stage. This way, the latest copy of the flags can live physically in the E2 stage, generated each cycle by a merging of the flag outputs from the pipe0 ALU and the pipe1 ALU. Since pipe0 is guaranteed to contain be the older instruction, it is always possible to create the correct final set of flags for use in the next cycle by merging the output from both instructions. Allowing two instructions to update the flags in parallel is particularly relevant to the Thumb instruction set where many instructions are only available in a flag-setting variant.

As mentioned in the architecture discussion, one of the unique features of the ARM instruction set is that almost all ARM instructions can be made conditional. Generally speaking, the ARM compiler makes fairly heavy use of conditional instructions to reduce the number of short branches. Therefore, it is important to handle the execution of conditional instructions in an efficient manner. From an architectural point of view, when an instruction fails its condition codes and is not executed, it becomes a NOP (no operation). However, late conversion of an instruction to a NOP does not work well in a statically scheduled machine. Therefore, conditional instructions are implemented on Cortex A8 such that they always generate a result regardless of whether they pass or fail their condition codes. This is done by reading the old value of the destination register as a source operand for the instruction. If a conditional instruction fails its condition code check, the old destination value will be passed on as the result of the instruction that failed its condition codes. Like all uses of the flags, condition code resolution is done in the E2 stage of the pipeline local to the two ALUs. Since flags are forwarded between the ALU pipelines in the same cycle that they are generated, back-to-back condition code setting and use in a conditional instruction is fully supported.

On occasion, the entire processor status register (CPSR) will need to be read by an instruction, including the latest value of the flags that are stored in the E2 stage of the execution pipeline. When a read of the full CPSR is required, the execution pipeline must first be allowed to drain of all outstanding operations so that the latest value of the flags can be read and merged in with the rest of the CPSR register. This means that an instruction that reads the CPSR register will take a few additional cycles to execute. However, these instructions are not common, so this is a good tradeoff to make in order to keep updates to the flags and condition code resolution as efficient as possible.

4.3 Forwarding paths

One of the most important features in Cortex A8 for efficient execution of code is the extensive support of key forwarding paths. Result data is forwarded from the outputs of both shifters, both ALUs, the multiplier, and the Data Cache. Data from each of these sources is made available for any consumer instruction that requires it as soon as it is produced. Also, once data is available for forwarding, it will continue to be available while the instruction is in flight until its result is written into the register file in the E5 stage. Supporting all of these forwarding cases requires a significant amount of data multiplexing. However, high frequency operation is still achievable since the controls to steer the data to the correct pipeline are available early. Also, any non-critical sources of data (data that was produced in a previous cycle) can be combined together early to reduce the number of critical inputs into the forwarding multiplexer to a maximum of 6 in all cases.

4.4 Exceptions and branches

All control flow changes for mispredicted branches and exceptions are handled and prioritized at one time in the E4 stage. Resolving all branches and exceptions together allows for a relatively simple prioritization scheme and a single path for communicating all changes in instruction stream to the prefetch and decode units. This simplifies the branch resolution logic, the interface to the prefetch unit, and reduces the number of potential sources of the next fetch address. The downside of processing all branches in one stage is that you have to pick a point that is late enough in the pipeline for all branches to have their outcome resolved. This means a consistent, but high, branch mispredict penalty for all types of mispredicted branches. However, in most cases, a conditional branch is immediately preceded by the operation that will set the condition codes. Therefore, the number of times the branch could be resolved early is fairly small. Also, there is some upside in performance to late branch resolution since this allows both the flag setting instruction and the dependent branch to be issued in parallel.

5 MEMORY SYSTEM

5.1 Memory System Pipeline Overview

The Cortex A8 memory system consists of the integer load/store pipeline, the Level-1 Data Cache, the level-2 Data Cache, and the bus interface unit. All data memory transfers are handled by the memory system including all NEON and floating point load and store operations. All Instruction Cache misses are also handled by the level 2 cache and bus interface unit.

The Load-store pipeline runs in parallel with the two ALU pipelines in stages E1 – E3. In the E1 stage, the memory address is generated in the AGU from the base and index register. In the E2 stage, the address is applied to the cache arrays. In the case of a load operation, data is returned for formatting and forwarding in the E3 stage. In the case of a store operation, the store data is formatted and ready to be written into the cache in the E3 stage. Forwarding load data in the E3 stage results in a one cycle load-use penalty for the common case of load data forwarded to the ALU. The penalty is two cycles if forwarding to a multiply, shift, or address calculation, and the penalty is zero cycles in the case of forwarding to a subsequent store instruction.

In the case of an L1 Data Cache load miss, a replay sequence occurs as described in the decode section and the miss request is sent to the level-2 cache pipeline. The latency for a level-2 cache access is a minimum of 8 cycles. The L1 cycle is an arbitration stage where the next request to process is chosen among all pending requests from the load-store unit, Instruction Fetch Unit, and many other cases. The L2 and L3 stages are where the tag ram is accessed. In the L4 stage, it is determined whether or not a cache hit has occurred. If the access resulted in a cache hit, the data RAM is accessed in the L5 and L6 stages. In the L7 stage, data is returned to the load/store unit, and it is formatted and forwarded to the load instruction in the L8 stage.

The Load-store and L2 pipelines can be seen in the following pipeline diagram.

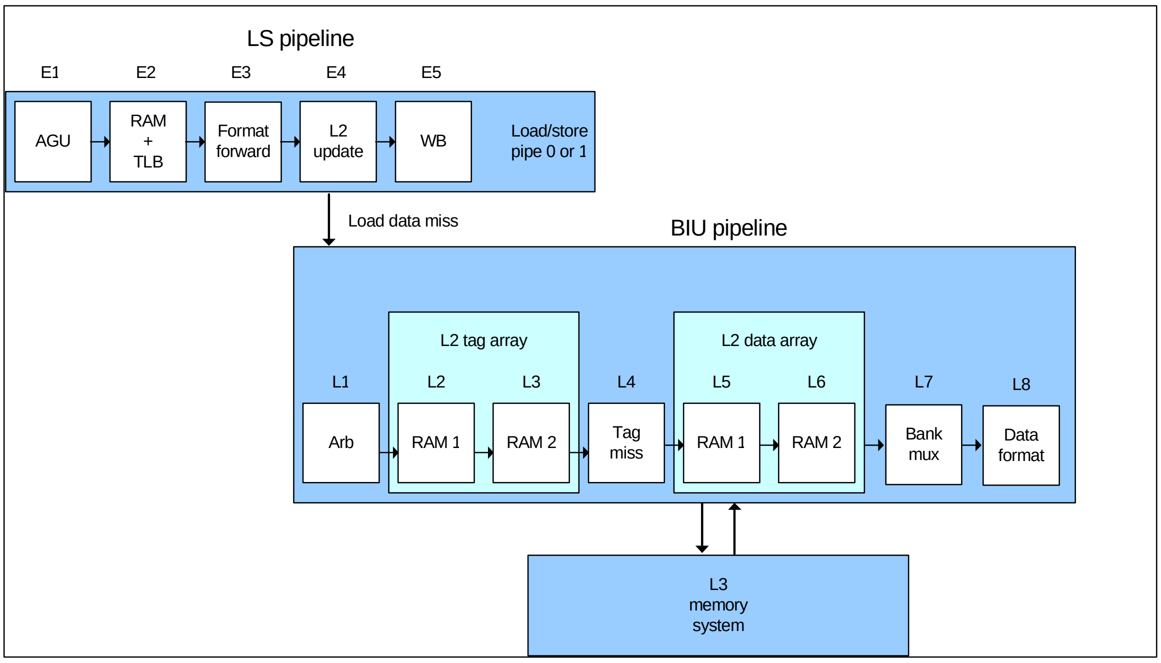

Figure 5: Memory system pipeline

5.2 Level-1 Data-side memory system structure

The D-side memory system uses many of the same components found in the Instruction Cache including using the same data array, the tag array, the TLB, and the HVAB array. Like the Instruction Cache, the Data Cache size is configurable between 16KB and 32KB. Also like the Instruction Cache, it is 4-way set associative, physically addressed, has a 32 entry TLB, and uses a Hash Virtual Address Buffer (HVAB) way indicator scheme to improve timing and reduce power dissipation. While many things are similar, there are also other additional components in the data side memory system that are not required for the Instruction Cache. These include the integer store buffer, the NEON store buffer, and the data alignment and forwarding logic.

ARM integer stores pass through a 3 entry, 64 bit store buffer with byte gathering (merging) on all entries before entering the Data Cache. This buffer is needed since stores are still speculative in the E3 stage and data should not be written into the cache until it is non-speculative two cycles later. In addition, the buffer merges any subsequent stores that are to the same 64bit location as one of the buffer entries. This store merging saves power since it reduces the number of cache access required. Full forwarding from the store buffer to subsequent load instructions is supported. So, no performance is lost from waiting to commit stores to the Data Cache. NEON stores pass through an 8 entry, 128 bit store buffer. The NEON stores have a deeper buffer than integer stores since the NEON pipeline follows the ARM pipeline and store data will not be provided from the NEON engine, allowing the buffer entry to retire, until the N3 stage of the NEON pipeline. Once a store is placed in the NEON store buffer in the E3 stage, data hazard checking is performed to compare with all subsequent memory operations (both ARM integer and NEON). Assuming no hazard condition exists, these subsequent operations are allowed to complete and the NEON store is non-blocking. If an address hazard is detected, then the subsequent memory request will replay and wait for the NEON store to complete.

The Level-1 Data-side memory system supports reading or writing 64-bits per cycle for integer load and store instructions and 128-bits per cycle for NEON load and store instructions. Non-word-aligned reads are supported without additional latency in cases where the entire access falls completely within a naturally address-aligned 128-bit region. Non-word-aligned writes are supported without additional latency in cases where the access falls completely within a naturally address-aligned 64-bit region.

5.3 Non-blocking NEON loads

NEON load operations that miss in the Level 1 Data Cache do not generate a replay sequence. Instead, the missed request is passed from the level 1 memory system to the level 2 memory system. When data is returned from the level 2 memory system, it will be sent directly to the NEON Load data queue and will not be returned to the level 1 memory system. In this way, NEON can stream data directly from the Level-2 cache as if it was a level 1 cache. This is very useful for NEON applications since media code tends to work on streaming datasets that are more likely to miss in the L1 and often even the L2 Cache. Therefore, it is beneficial for NEON performance to stream accesses with multiple misses outstanding with data coming from the L2 and also from the external memory system directly to NEON.

5.4 Level-2 Cache structure

The Level-2 cache is physically addressed, has a 64-byte line size, and is 8-way set associative with a random replacement policy. Write back, write through and write-allocate policies are all supported, but write-allocate is the default, high performance option. The cache size is configurable to be anything from 1MB to 128KB in size. A 0KB size is also supported in the case where no L2 cache is desired. In the 0KB scenario, the cache control logic is also removed from the design for additional area and power savings. The cache is pipelined to support multiple accesses in flight at once but RAM accesses take at least 2 cycles so 4-way banking is implemented to allow back-to-back access for streaming data. For large cache sizes it is possible to configure wait states to allow more than 2 cycles to access the tag and data RAMs. The external bus interface is configurable to be either 128- or 64-bits wide and it supports multiple outstanding requests. The interface to the Level-1 memory system includes a 128-bit read interface and a 128-bit write interface. This allows for efficient linefill operations that can complete in just 4 cycles.

5.5 Memory System Request Buffers

The memory system contains request buffers which all arbitrate for access to the Level-2 cache and to the external memory system. This arbitration is done in the first stage of the Level 2 cache pipeline. All requests, whether an initial request from the Level-1 memory system, or a linefill request from the external memory system, arbitrate for cache access at this same point and then flow through the L2 pipeline. This simplifiies hazard checking and reduces the complexity required to maintain proper memory transaction ordering rules.

There are three types of buffers that arbitrate for cache access: miss buffers, write buffers, and victim buffers.

5.5.1 Miss Buffers

The L2 unit includes three sets of miss buffers to track outstanding Level 1 read misses for a total of 14 possible pending L2 read requests:

- IMB, Instruction side miss buffer
- DMB, Integer data side miss buffer
- NMB, NEON (data) miss buffer

The 1-entry IMB holds the pending read request from the Instruction side memory subsystem. There can be only one outstanding request on the Instruction side. The 1-entry DMB and 12-entry NMB hold pending read requests from the data side memory subsystem. The DMB contains any pending read request for an ARM integer load instruction. The NMB holds up to 12 outstanding read requests for NEON and floating point load instructions. ARM integer code can only have one outstanding read miss since the level-1 memory system is blocking after the first load miss for ARM integer loads. However, as mentioned in the earlier section, the level 1 memory system is non-blocking for NEON load misses to allow streaming of data from the L2 cache or external memory to NEON.

5.5.2 Write-Combining and Write Buffer

The L2 unit has an 8-entry, 128-bit wide write buffer and a 2-entry 128-bit wide write combining buffer to handle sub-block writes. Write combining is performed on incoming write requests to the same quad-word. If the incoming write request is to a different quad-word, the current contents of the Write-Combining Buffer is transferred to the write data buffer and the incoming request is placed in the write combining buffer. This write combining capability saves power by reducing multiple writes to the same quadword down to a single store, and it also improves performance for the same reason.

Once valid, an entry in the write buffer will arbitrate for access to the L2 cache. If the write access results in a cache miss, the L2 unit supports write-allocate. Therefore, a victim buffer will be allocated and an L3 linefill request will be initiated similar to an L2 read miss.

As an additional optimization, if the full cache line is written, the Level-2 line is simply marked dirty and no external memory requests are required. Of course, write accesses for the victim line may still be required. This optimization results in both a power and performance improvement in store streaming scenarios.

5.5.3 Victim Buffer

The L2 unit has a 4-entry victim buffer to handle linefills and evictions. Each entry in the victim buffer (VB) can hold a full cache line of data (64 bytes). When a load or store request misses in the L2, a victim buffer is allocated to perform the linefill request. Once data has been returned from the L3 system, the VB will arbitrate for the L2 cache to read out the victim data and write in the fill data, essentially performing a swap. If the L2 victim is dirty, the VB will schedule a write-back of the L2 data to the L3 memory system. Each victim buffer uses a unique ID on the external bus which allows for multiple requests outstanding at the same time and also allows the data from these requests to be returned out of order.

6 NEON MEDIA PROCESSING ENGINE

6.1 NEON Pipeline Overview

The NEON unit processes the Advanced SIMD instruction set that consists of 32-bit SIMD integer and floating point instructions that can operate on 128-bit, 64-bit, 32-bit, 16-bit, or 8-bit data values. These instructions were added to greatly accelerate processing of media-style workloads such as audio and video filters and codecs. The NEON unit also executes all ARM floating point instructions from the pre-existing VFP instruction set. It makes sense for the NEON unit to execute both the new Advanced SIMD instructions and the pre-existing VFP floating point instructions since they share the same register file and load/store instructions.

The NEON media processing engine's pipeline starts at the end of the main integer pipeline. As a result, all instruction speculation is resolved before instructions reach the NEON pipeline. This reduces the complexity of the NEON Unit and also allows for a zero-cycle load-use penalty in most cases, even when data is returned from the L2 cache, due to the decoupling buffers used to hold pending instructions and data. The cost of putting the NEON engine at the end of the main integer pipeline is the longer delay required for writing values from the NEON register file to the ARM register file. However, this forwarding of data from a NEON register to an ARM register is not commonly seen in media code.

The NEON unit is decoupled from the ARM integer pipeline by the 16 entry NEON instruction queue (NIQ). NEON can receive up to two valid NEON instructions from the Integer Execute Unit. Once there are enough instructions sent to NEON to fill the queue, the decode unit will wait until entries have been de-allocated from the NIQ before sending any more instructions. In a similar fashion to what is seen on the instruction side, NEON is also decoupled from the memory system by the 12 entry load data queue. The load data queue can receive data from either the Level 1 Data Cache or from Level 2 Memory System and this data can be received out of order. Store data is written out from the N3 stage in the NEON execution pipeline to the NEON store buffer. All the interfaces between the NEON engine and the other components of the processor can be seen in Figure 6.

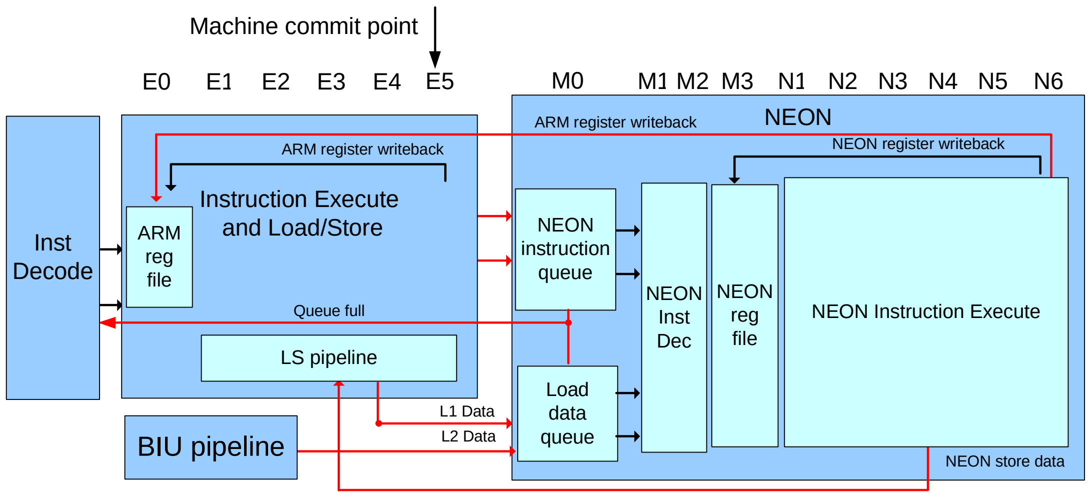

Figure 6: NEON Engine Interfaces

6.2 NEON Pipeline

The NEON engine has its own pipeline that begins at the end of E5 and is a 10 stage pipeline. NEON instructions are either read from the instruction queue or directly from the E5 pipestage when the queue is empty. Instructions are always issued and retired in-order.

NEON has 4 decode stages, M0-M3, and 6 execute sages, N1-N6. The four decode stages in M0-M3 are very similar in structure and design to the four decode stages, D0-D4, seen in the main pipeline. The first two stages are used to decode the instruction resource and operand requirements and the later two stages are used for instruction scheduling. A static scoreboard with a fire-and-forget issue mechanism is used for the NEON pipeline in a similar way to what is seen in the ARM integer pipeline with the key difference being no requirement for a replay queue since there are no conditions under which a pipeline flush can occur.

The NEON decode logic is capable of dual issuing any LS permute instruction with any non-LS permute instruction. Dual issuing these combinations of instructions requires fewer register ports than what would be needed for dual issuing two data processing instructions since LS data is provided directly from the load data queue. It is also the most useful pairing of instructions to dual issue since significant load/store bandwidth is required to keep up with the Advanced SIMD data processing operations.

The 32-entry register file is accessed in the M3 stage when the instruction(s) are issued. Once issued, an instruction will be sent to one of seven execution pipelines: Integer multiply, integer shift, integer alu, NFP Add, NFP multiply, IEEE floating point, or load/store permute. All execution datapath pipelines are balanced at six stages. The 10 stages of the NEON pipeline including all the execution pipelines, can be seen in Figure 7.

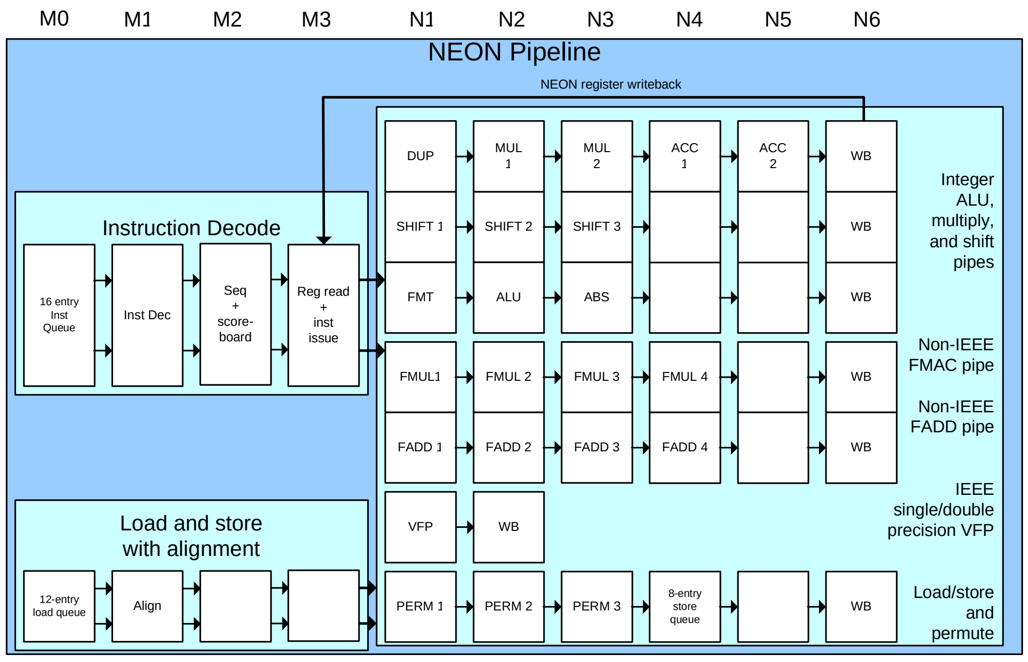

Figure 7: NEON Pipeline Stages

6.3 NEON Execution pipelines

NEON has three SIMD integer pipelines, a load-store/permute unit, two SIMD single-precision floating-point pipelines, and a non-pipelined IEEE compliant floating point engine. All NEON and floating point instructions are processed by one or more of these execution pipelines.

6.3.1 NEON Integer Pipelines

There are three execution pipelines responsible for executing NEON integer instructions, an integer multiply-accumulate (MAC) pipeline, an integer Shift pipeline, and an integer ALU pipeline. The MAC unit consists of two 32x16 multiply arrays with two 64-bit accumulate units. The 32x16 multiplier array can perform four 8x8, two16x16, or one 32x16 multiply operation in each cycle. The accumulate units have dedicated register read ports for the accumulate operand. The MAC unit is also optimized to support one multiply accumulate operations per cycle for high performance on a sequence of MAC operations with a common accumulator.

The Shift pipeline consists of three stages and is therefore a bit shorter than the other NEON pipelines. When only a shift result is required, it is made available early for subsequent instructions at the end of the N3 stage. Some instructions do a shift and accumulate operation. For these instructions, the result from the shift pipeline is forwarded to the MAC pipeline to complete the accumulate operation.

The NEON Integer ALU consists of two 64-bit SIMD ALUs operating in parallel, with four 64-bit inputs. The first stage of the ALU pipeline, N1, is responsible for formatting the operands to be used in the next cycle. This includes inverting operands as needed for subtract operations, multiplexing vector element pairs for folding operations, and sign/zero-extend of operands. The N2 stage contains the main ALU which is responsible for all NEON SIMD integer add, subtract, logical, count leading-sign/zero, count set, and sum of element pairs operations. If this is the only operation required by the instruction, then the result is available for forwarding at the end of the N2 stage. The flags are also calculated in the N2 stage, to be used in the following stage. The N3 stage is responsible for the compare, test, and max/min operations and for saturation detection. It also has a SIMD incrementor for generating 2’s complement and rounding operations, and data formatter for high-half and halving operations. Similar to the shift pipeline, the ALU pipeline can also make use of the MAC accumulator in stages N4 and N5 for completing the final stages of the absolute-difference-accumulate operation.

6.3.2 NEON Load-Store/Permute Pipeline

The permute pipeline is fed by the load data queue (LDQ). The LDQ holds all data associated with NEON load accesses prior to entering the NEON permute pipeline. It is 12 entries deep and each entry is 128-bit wide. Data can be placed into the LDQ from either L1 cache or L2 memory system. Accesses that hit in the L1 cache will return and commit the data to the LDQ. Accesses that miss in the L1 cache will initiate an L2 access. A pointer is attached with the load request as it proceeds down the L2 memory system pipeline. When the data is returned from the L2 cache, the pointer is used to update the LDQ entry reserved for this load request.

Each entry in the LDQ has a valid bit to indicate valid data returned from L1 cache or L2. Entries in the LDQ can be filled by L1 or L2 out-of-order, but valid data within the LDQ must be read in program order. Entries at the front of the LDQ are read off in-order. If a load instruction reaches the M2 issue stage before the corresponding data has arrived in the LDQ, then it will stall and wait for the data.

L1 and L2 data that is read out of the LDQ is aligned and formatted to be useful for the NEON execution units. Aligned/formatted data from the LDQ is multiplexed with NEON register read operands in the M3 stage, before it is issued to the NEON execute pipeline.

The NEON LS/Permute pipeline is responsible for all NEON load/stores, data transfers to/from the integer unit, and data permute operations. One of the more interesting features of the NEON instruction set is the data permute operations that can be done from register to register or as part of a load or store operation. These operations allow for the interleaving of bytes of memory into packed values in SIMD registers. For example, when adding two eight byte vectors, you may wish to interleave all of the odd bytes of memory into register A and the even bytes into register B. The permute instructions in NEON allow you to do operations like this natively in the instruction set and often with only using a single instruction.

This data permute functionality is implemented by the load-store permute pipeline. Any data permutation required is done across 2 stages, N1-N2. In the N3 stage, store data can forwarded from the permute pipeline and sent to the NEON Store Buffer in the memory system.

6.3.3 NEON Floating-Point pipelines

The NEON Floating-Point (NFP) has two main pipelines: a 6-stage multiply pipeline and a 6-stage add pipeline. The add pipeline adds two single-precision floating-point numbers, producing a single-precision sum. The multiply pipeline multiplies two single-precision floating-point numbers, producing a single-precision product. In both cases, the pipelines are 2-way SIMD which means that two 32bit results are produced in parallel when executing NFP instructions. Most classic ARM vfp single precision floating point instructions can also be executed in the NFP pipeline when IEEE compliant mode is not enabled. When executing multiply-accumulate instructions, both pipelines are used back-to-back to produce the final result.

6.3.4 IEEE compliant floating point engine

The IEEE compliant floating point engine is a non-pipelined implementation of the ARM Floating-Point instruction set targeted for medium performance IEEE 754-compliant and double precision floating-point. It is designed to provide general-purpose floating-point capabilities for a Cortex A8 processor. This engine is not pipelined for most operations and modes, but instead iterates over a single instruction until it has completed. A subsequent operation will be stalled until the prior operation has fully completed execution and written the result to the register file. The IEEE compliant engine will be used for any floating point operation that cannot be executed in the NEON floating point pipeline. This includes all double precision operations and any floating point operations run with IEEE precision enabled.

7 IMPLEMENTATION AND DEPLOYMENT

ARM is a provider of intellectual property. This means ARM does not manufacture any silicon, but instead provides complete processor designs to the customer, typically a silicon vendor. The silicon vendor will then integrate the ARM processor as an embedded component in one or more of their System-on-Chip (SoC) products. ARM will typically provide the same processor design to multiple customers. This creates additional design challenges since each silicon vendor will have a different fabrication process and therefore will need a slightly different mask set to fabricate the design. This is normally dealt with by providing ‘soft’ deliverables for the design. Soft IP means the customer is provided with an RTL level description of the design along with a set of implementation scripts that can be used to synthesize, place, and route the design in a way specific to their design libraries and process.

Delivering Soft IP gives a lot of flexibility to the customer, including the ability to configure some features in the design such as external bus widths and cache sizes. However, there are limitations as well. Delivering Soft IP means that the design will need to be fully synthesized and not make use of any of the advanced implementation techniques that are commonly used in the development of high performance processors.

For Cortex A8 to hit the top performance targets desired by its customers, support for advanced implementation techniques would be required. At the same time, Cortex A8 would need to still support the traditional soft IP delivery method for customers to whom time to market and flexibility were more important than achieving top performance. Customers that wish to quickly implement and deploy the processor with a small team can do so using the soft IP flow. While customers that wish to target higher performance levels can do so as well by using a larger team to build portions of the design using an advanced implementation flow. Designs implemented using both the soft and advanced flows are logically identical and share the same RTL code base. In this way, the requirements of the full customer base can be met with a single design. To help minimize the cost of building an advanced implementation, ARM provides a reference design to customers that want to take advantage of the advanced implementation flow. This reference design minimizes the effort required from the customer to only the activities that are unique to their specific process. Also, to keep the cost of the advanced implementation as low as possible, the advanced techniques are used strategically only in the areas that provide the most benefit. A large majority of the design can still be implemented using a standard synthesis flow and the additional benefits from using the advanced implementation will still be realized.

8 CONCLUSION

This chapter has gone through the details of the Cortex A8 microarchitecture describing the design along with some of the background behind the decisions made along the way. The end result is a processor design that is uniquely placed in the market because of the fine balance it achieves between high performance and low power operation.