Fujitsu A64FX

Amritesh Dasari, Jakob Arend, Pratik Sampat and Sudhanshu Agarwal
History

- Launched in 2019 for use in the Fugaku supercomputer
- Fugaku was rated as the *most powerful* supercomputer in June 2020 (since been overtaken)

General Pipeline

Instruction fetch

ARMv8 instructions

Decode

μOP instructions

Execution - Load/store - Commit
Instruction Fetch Stage

Fetches 8 instructions simultaneously

Predicts four branch directions per cycle

Predicts direction of one taken branch per cycle

Source: A64FX Microarchitecture Manual v1.8
Decode Stage

Takes in up to 6 ARMv8 instructions per cycle

ARMv8 instructions decoded into μOP instructions

Broken down μOPs sent in-order to execution stage
Execution Stage: Overview

Reservation Stations

- Execute instructions out of order

Pipelining

- Exploits ILP while executing instructions

Source: A64FX Microarchitecture Manual v1.8
Execution Stage: Reservation Stations

**Int/floating/predicate**
- 2 stations
- 20 entries each

**Address calculation**
- 2 stations
- 10 entries each

**Branch prediction**
- 1 station
- 19 entries

Source: A64FX Microarchitecture Manual v1.8
Execution Stage: Pipelining

Five separate pipelines for integer, SIMD floating point and SVE, predicate, branch, and load/store operations respectively.

Pipelines are 11-24 stages deep and support operand bypassing.

Source: A64FX Microarchitecture Manual v1.8
Load/Store Stage

Executes one store or two loads at once

Uses virtual load and store ports

Load/store both pipelined for performance

Source: A64FX Microarchitecture Manual v1.8
Commit Stage

Uses a commit stack to maintain proper execution

Checks exceptions and branch prediction results

Commits 4 μOPs per cycle

Source: A64FX Microarchitecture Manual v1.8
Memory Hierarchy
Memory Hierarchy

- The processor employs Single Instruction Multiple Data Compute method with 512 bits as the width.
- L1 I-Cache is 3MiB (64KiB/Core)
- L1 D-Cache is 3MiB (64KiB/Core)
- L2 Cache is 32MiB (8MiB*4)
- This 4x multiplier is due to the CMGs that this processor has (*CMG is Core Memory Groups).
- The Memory Accesses between these CMGs is in NUMA config.

Source: Fujitsu Publication
Scalar Vector Extensions and HMB2 Memory

- This architecture implements 128/256/512 Bits of SVE.
- SVE improves the suitability of the architecture for High-Performance Computing and Machine Learning applications.
- HBM2:
  - Total Capacity: 32GiB
  - No. of stacks per package: 4
  - Memory Capacity: 8GiB
  - Bandwidth: 256GB/s
- One Memory Access Controller per CMG
- Processor known for its Extensive Data Integrity with its Unique 128,400 Error Checkers to detect and correct 1-bit errors on a chip.

Source: Fujitsu Publications
Memory Management Unit

- Translation Lookaside Buffer (TLB)
  - TLB consists of 2 parts: Instruction TLB and Data TLB
  - Each of these have 2 levels:
    - L1: Fully associative and FIFO replacement algorithm.
    - L2: 4-way set associative and LRU replacement algorithm.

- Translation Table Cache
  - A Translation Table Cache is implemented for temporarily storing table descriptors.
  - Diff between TTC and TLB is that TLB is to suppress the occurrence of the table walk and TTC is to reduce latency caused by memory access during table walk.

<table>
<thead>
<tr>
<th></th>
<th>For Instruction</th>
<th>For Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>Association</td>
<td>Full associative</td>
</tr>
<tr>
<td></td>
<td>Number of entries</td>
<td>16 entries</td>
</tr>
<tr>
<td></td>
<td>Replacement</td>
<td>FIFO</td>
</tr>
<tr>
<td>L2</td>
<td>Association</td>
<td>4-way set associative</td>
</tr>
<tr>
<td></td>
<td>Number of entries</td>
<td>1,024 entries</td>
</tr>
<tr>
<td></td>
<td>Replacement</td>
<td>LRU</td>
</tr>
</tbody>
</table>

Source: AA64FX Microarchitecture Manual v1.8
Cache Specifications

<table>
<thead>
<tr>
<th></th>
<th>For Instruction</th>
<th>For Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Association method</td>
<td>4-way set associative</td>
<td>4-way set associative</td>
</tr>
<tr>
<td>Capacity</td>
<td>64 KiB</td>
<td>64 KiB</td>
</tr>
<tr>
<td>Hit latency (load-to-use)</td>
<td>4 cycles</td>
<td>5 cycles(integer)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 cycles</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(SIMD&amp;FP / SVE in short mode)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 cycles</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(SIMD&amp;FP / SVE in long mode)</td>
</tr>
<tr>
<td>Line size</td>
<td>256 bytes</td>
<td>256 bytes</td>
</tr>
<tr>
<td>Write method</td>
<td>---</td>
<td>Writeback</td>
</tr>
<tr>
<td>Index tag</td>
<td>Virtual index and physical tag (VIPT)</td>
<td>Virtual index and physical tag (VIPT)</td>
</tr>
<tr>
<td>Index formula</td>
<td>index A = (A mod 16,384) / 256</td>
<td>index A = (A mod 16,384) / 256</td>
</tr>
<tr>
<td>Protocol</td>
<td>SI state</td>
<td>MESI state</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>For instruction and data (by shared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Association method</td>
<td>16-way set associative</td>
</tr>
<tr>
<td>Capacity</td>
<td>8 MiB</td>
</tr>
<tr>
<td>Hit latency (load-to-use)</td>
<td>46 to 56 cycles</td>
</tr>
<tr>
<td>Line size</td>
<td>256 bytes</td>
</tr>
<tr>
<td>Write method</td>
<td>Writeback</td>
</tr>
<tr>
<td>Index and tag</td>
<td>Physical index and physical tag (PIPT)</td>
</tr>
<tr>
<td>Index formula</td>
<td>index &lt;10:0&gt; = ((PA&lt;36:34&gt; xor PA&lt;32:30&gt; xor PA&lt;31:29&gt; xor PA&lt;27:25&gt; xor PA&lt;23:21&gt; &lt;&lt; 8) xor PA&lt;18:8&gt;)</td>
</tr>
<tr>
<td>Protocol</td>
<td>MESI state</td>
</tr>
</tbody>
</table>

Source: AA64FX Microarchitecture Manual v1.8
Cache Architecture

- L1 caches are implemented in units of processor cores. L2 caches are implemented in units of CMGs.
- Cache Coherence explained later.
- A memory unit is connected to only the L2 cache in a CMG.
- Read/Write requests from the L2 cache are sent to the memory unit via the MAC.

<table>
<thead>
<tr>
<th>Direction</th>
<th>Bus Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1D</td>
<td>L2 to L1D: 64 bytes / cycle (per Core)</td>
</tr>
<tr>
<td></td>
<td>L1D to L2: 32 bytes / cycle (per Core)</td>
</tr>
<tr>
<td>L2</td>
<td>L2 to L1D: 512 bytes / cycle (per CMG)</td>
</tr>
<tr>
<td></td>
<td>L1D to L2: 256 bytes / cycle (per CMG)</td>
</tr>
<tr>
<td>L2</td>
<td>L2 to L2: 64 bytes / cycle (per Ring)</td>
</tr>
<tr>
<td>L2</td>
<td>Memory to L2: 128 bytes / cycle (per CMG)</td>
</tr>
<tr>
<td>L2</td>
<td>L2 to Memory: 64 bytes / cycle (per CMG)</td>
</tr>
</tbody>
</table>

Source: AA64FX Microarchitecture Manual v1.8
Cache Coherence Protocol and Performance

- In the A64FX, coherence between the caches is guaranteed by Hardware. A common MESI protocol is adopted as the protocol for coherence.

The Performance values represent performance per CMG

<table>
<thead>
<tr>
<th>Condition</th>
<th>State</th>
<th>Possible Cause of State</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Modified</td>
<td>Data has been modified from main memory values (Dirty). Other caches at the same level do not have the data.</td>
</tr>
<tr>
<td>E</td>
<td>Exclusive</td>
<td>Data matches main memory values (Clean). Other caches at the same level do not have the data.</td>
</tr>
<tr>
<td>S</td>
<td>Shared</td>
<td>Data matches main memory values (Clean). Other caches at the same level also have the data.</td>
</tr>
<tr>
<td>I</td>
<td>Invalid</td>
<td>A cache line is invalid. Other caches request data when the data in the E/M state. Data writeback.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local memory latency (load-to-use)</th>
<th>Memory Access Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest core</td>
<td>135.5 ns (@ CPU 2GHz)</td>
</tr>
<tr>
<td>Longest core</td>
<td>144.5 ns (@ CPU 2GHz)</td>
</tr>
</tbody>
</table>

| Read throughput | Peak                  | 256 GB/s (per MAC) (@ CPU 2GHz) |
| Write throughput | Peak                  | 128 GB/s (per MAC) (@ CPU 2GHz) |

Source: AA64FX Microarchitecture Manual v1.8
Scalable Vector Extensions

TOFU Interconnect
Scalable Vector Extensions (SVE) for SIMD

- Architectures generally allow only fixed bits for vectorized operations via SIMD.

- SVE ISA defines vector length between 128 to 2048 bits and hardware decides the vector length. FX64 defines 512 bits Vector Registers: Z0 - Z31.

- Software has no vector length definition. Code can run and scale automatically on different vector length implementations. Improves portability.

Source: https://ieeexplore.ieee.org/document/7924233
Traditional vectorization techniques vs SVE

> Example: Processing 10 chunks of 4 bytes of data = 40 bytes of data to process

Scalar Architecture
- 10 Iterations over a 4 byte register

Fixed width Vector Architectures:
- 2 iterations Vectorized
- 2 iterations Scalar

SVE Vector Length Agnostic
- Adjustable predicate to determine the size of the vector used
- Reduces the binary as there is no need for the scalar loop
- 3 Iterations, all vectorized

Main Loop - Vectorized: $16B \times 2 = 32B$

Drain Loop - Scalar: $4B \times 2 = 8B$

Predicate bits determine which parts of the vector must be processed

Source: [Introduction to Arm SVE](#)
Torus interconnect

● Similar to the Mesh - each node connected to multiple neighbours
● Performance of mesh may depend of placement in the middle or edge
● The Torus connects the edges as well

+ Has higher path diversity
  + Leads in lower hops
  + Better Fairness

- Complex - Difficult to have equal link lengths
- Cost

Source: Interconnects Lecture - ETH Zurich
TOFU (Torus Fusion) Interconnect

- 6D Network to appear as 3D Torus
  - Groups of ABC 3D torus with size $2 \times 3 \times 2$ connected by an XYZ 3D torus

- Topology aware: Torus Rank Mapping
  - Give Rank to processes to optimize communication between nearest neighbours

- Highly fault tolerant design


- Can scale over 10,000 nodes
- Interfaced using Open MPI
Gather and Scatter

Non-sequential data accesses commonly used in sparse vector linear algebra operations

Combined Gather

- A64FX optimizes for gather operations by dividing vector elements into 128-bit spaces and loading pairs of data from these spaces concurrently.

- Essentially, if two elements lie in the same 128-bit space, they are loaded together.

- So, in an ideal case, the latency for gather accesses is halved.
ld1d z2.d, p3/z, [x0, z1.d]

Initial state

Memory: ...

| 0 | 1 | 3 | 5 | 4 | 6 | 2 | 7 | ...

z2

1st flow

Memory: ...

A pair of elem 0 and 1 is written in one flow.

z2

Element 6 is inactive.

Fujitsu A64FX and Fugaku

Launched in 2019 primarily for the Fugaku supercomputer

4th flow

Memory ⋮ 0 1 3 5 6 2 7 ⋮

\( z_2 \)

5th flow

Memory ⋮ 0 1 3 5 4 6 2 7 ⋮

\( z_2 \)

A pair of elem 4 and 5 is written in one flow.

Figure 7-9  Summary of Elements for Gather Instruction

Single Core Performance and Power Efficiency
Focus on single core performance

- A64FX focuses on improving single core performance by using complicated branch predictors

<table>
<thead>
<tr>
<th>Role</th>
<th>Branch Predictor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch direction &amp; Branch target address prediction</td>
<td>Small Taken Chain Predictor (S-TCP)</td>
</tr>
<tr>
<td>Branch direction prediction</td>
<td>Branch Weight Table (BWT)</td>
</tr>
<tr>
<td></td>
<td>Loop Prediction Table (LPT)</td>
</tr>
<tr>
<td></td>
<td>Return Address Stack (RAS)</td>
</tr>
<tr>
<td>Branch target address prediction</td>
<td>Branch Target Buffer (BTB)</td>
</tr>
</tbody>
</table>

Small taken chain predictor

- Detects a chain of *taken* branches and prefetches instructions. This is useful when executing long running loops.

Figure 3-3  Chain Structure Consisting of Multiple Taken Branch Instructions

Energy Efficiency

- Fine grained energy analysis for CPU cores and caches

Fujitsu High Performance CPU for the Post-K Computer. Toshio Yoshida, 2018
Power Tuning

- Application level hardware control for power optimization using built-in counters

Fujitsu High Performance CPU for the Post-K Computer. Toshio Yoshida, 2018
Any questions?
Fujitsu A64FX and Fugaku

Launched in 2019 primarily for the Fugaku supercomputer

References

- The ARM Scalable vector Extensions: https://ieeexplore.ieee.org/document/7924233
- Introduction to ARM SVE: https://www.youtube.com/watch?v=eGCcPo4UAHs
- Optimizing HPC Applications using SVE: https://developer.arm.com/documentation/101726/0400
- The Tofu Interconnect: https://ieeexplore.ieee.org/document/6041538
- Fujitsu High Performance CPU for the Post-K Computer. Toshio Yoshida, 2018
Improvements to auto vectorization

> How to program when we do not know how wide the vectors are?

- Per Lane prediction
  - Map operations to the lane of the vector

- Predicate driven loop control
  - Process partial vectors so that loops heads and tails are not performed in scalar

- Fault-tolerant speculative vectorization:
  - Suppress memory faults if it is not invoked by the first element of the vector

```
for (i = 0; i < n; i++) ->
```

```
Vector 1
1 2 0 0
5 6 0 0
1 1 0 0
```

```
Vector 2
Predicate bits
```

```
= 6 8 0 0
```

```
1 1 0 0
```

```
for (i = 0; i < n; i++) ->
When n misaligned with the vector
```

```
Will not fault
```

Source: Introduction to Arm SVE