Can We Continue to Build Supercomputers Out of Processors Optimized for Laptops?

Richard C. Murphy

Sandia National Laboratories

DOE Institute for Advanced Architecture

rcmurph@sandia.gov

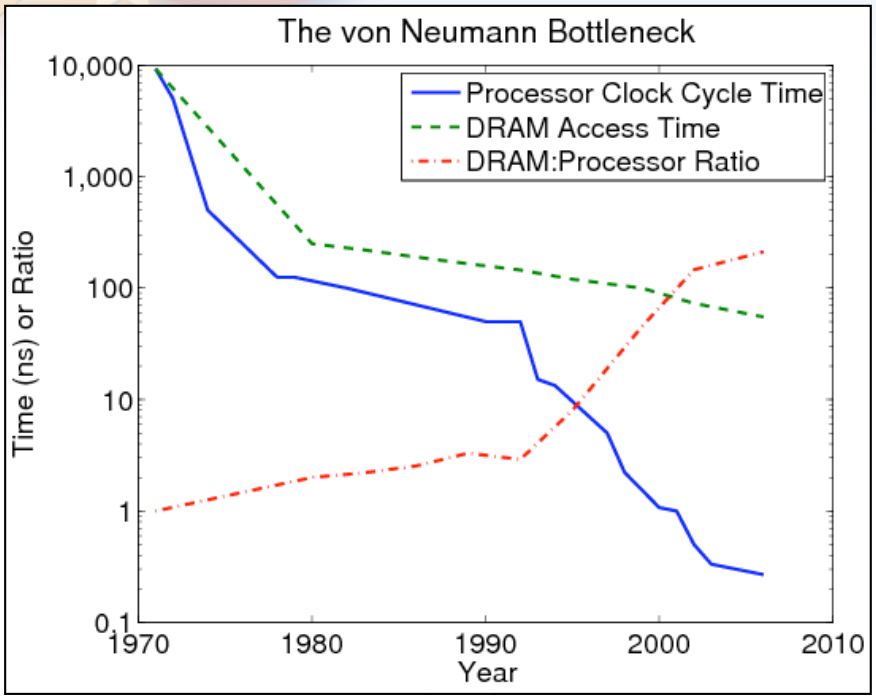
March 11, 2009

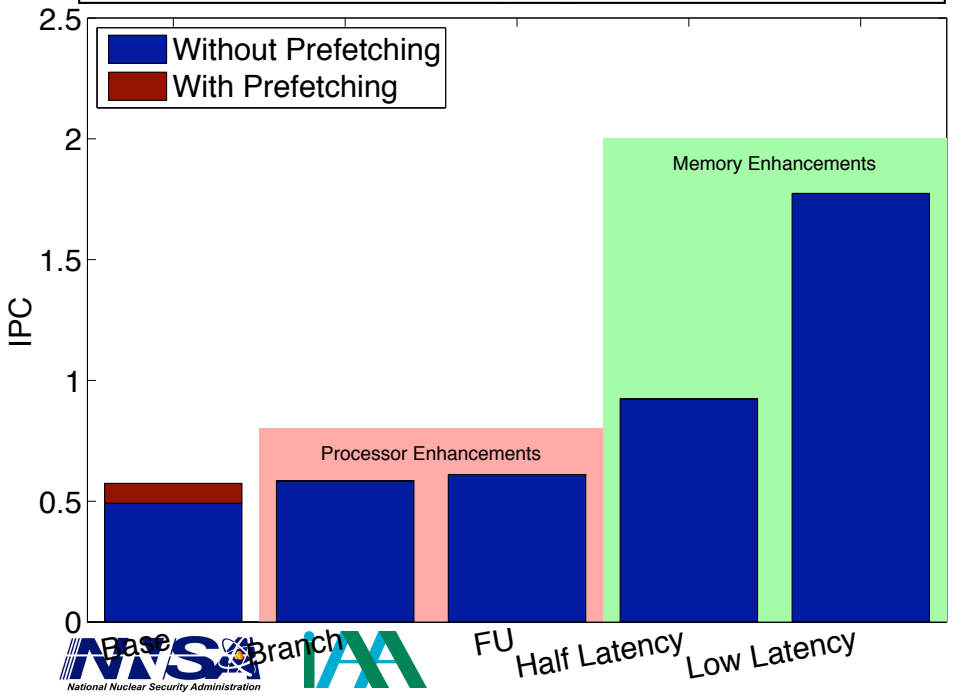
Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

National Nuclear Security Administration

Sandia National Laboratories

The Memory Wall





- **Historically:**
  - **Processor cycle time decreased FASTER than memory access access latency**
- **Technologically, this may not hold as cycle times go flat**
- **Definitely holds as memory hierarchies become more complex**
  - **Multicore exacerbates this problem**
- **Enhancements to "compute" not as effective as decreasing memory latency for problems that don't fit in cache!**

National Nuclear Security Administration

Sandia National Laboratories

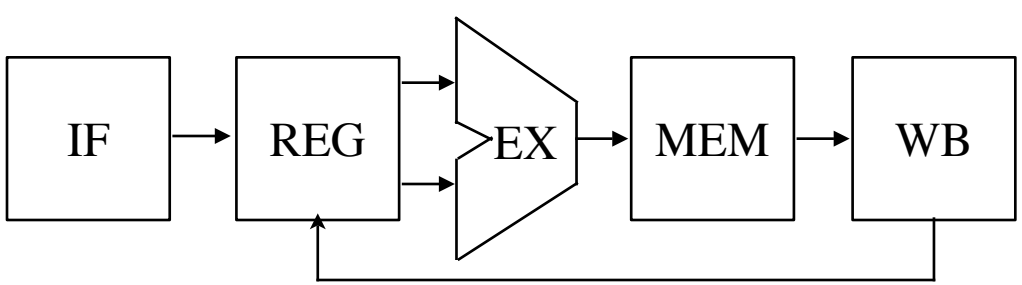
Motivating Example

It is singular how soon we lose the impression of what ceases to be constantly before us.

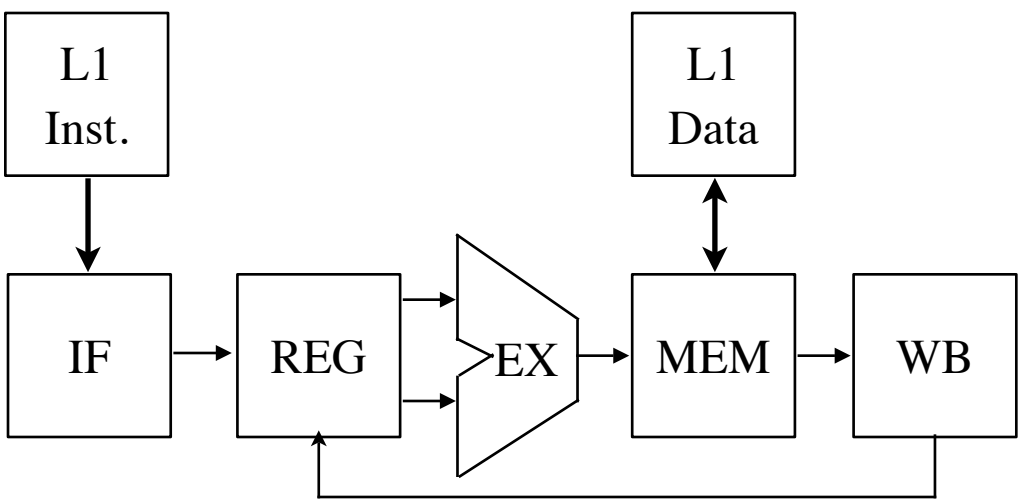
- Lord Byron

Consider THE Textbook 5-Stage Pipeline

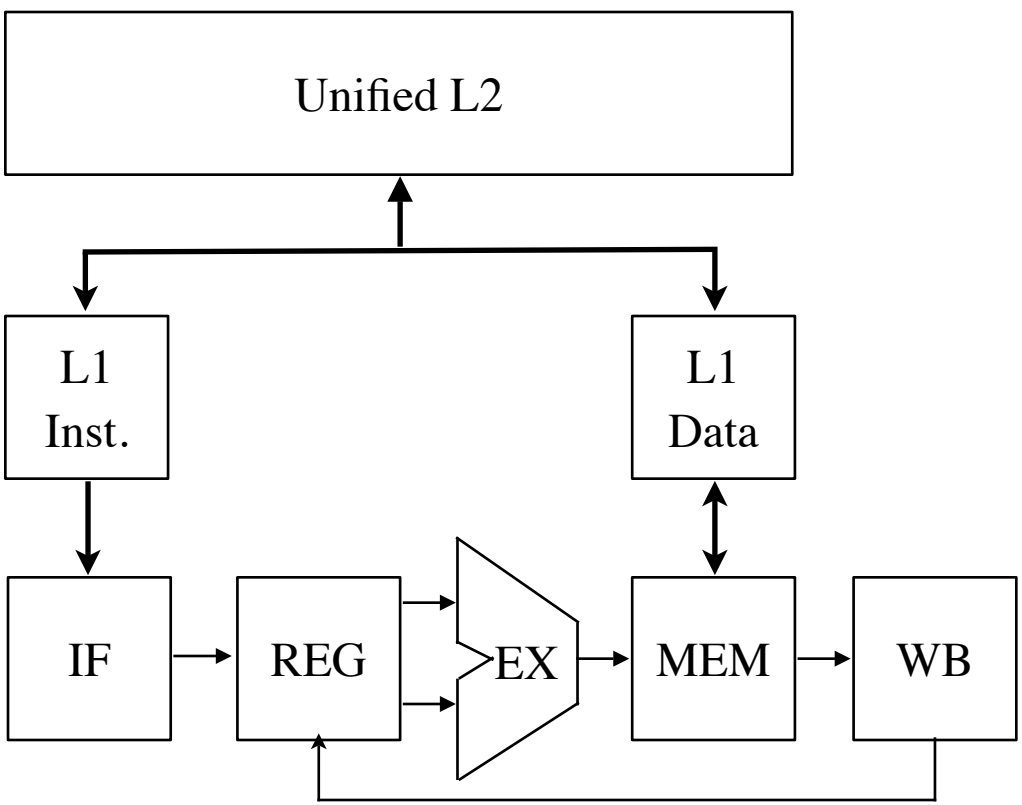
- The Stages Are:
  - IF: Instruction Fetch
  - REG: Register File Read
  - EX: Execute
  - MEM: Memory Access
  - WB: Write Back (to register file)
- Real pipelines are more complex
  - We will assume each stage takes one clock cycle



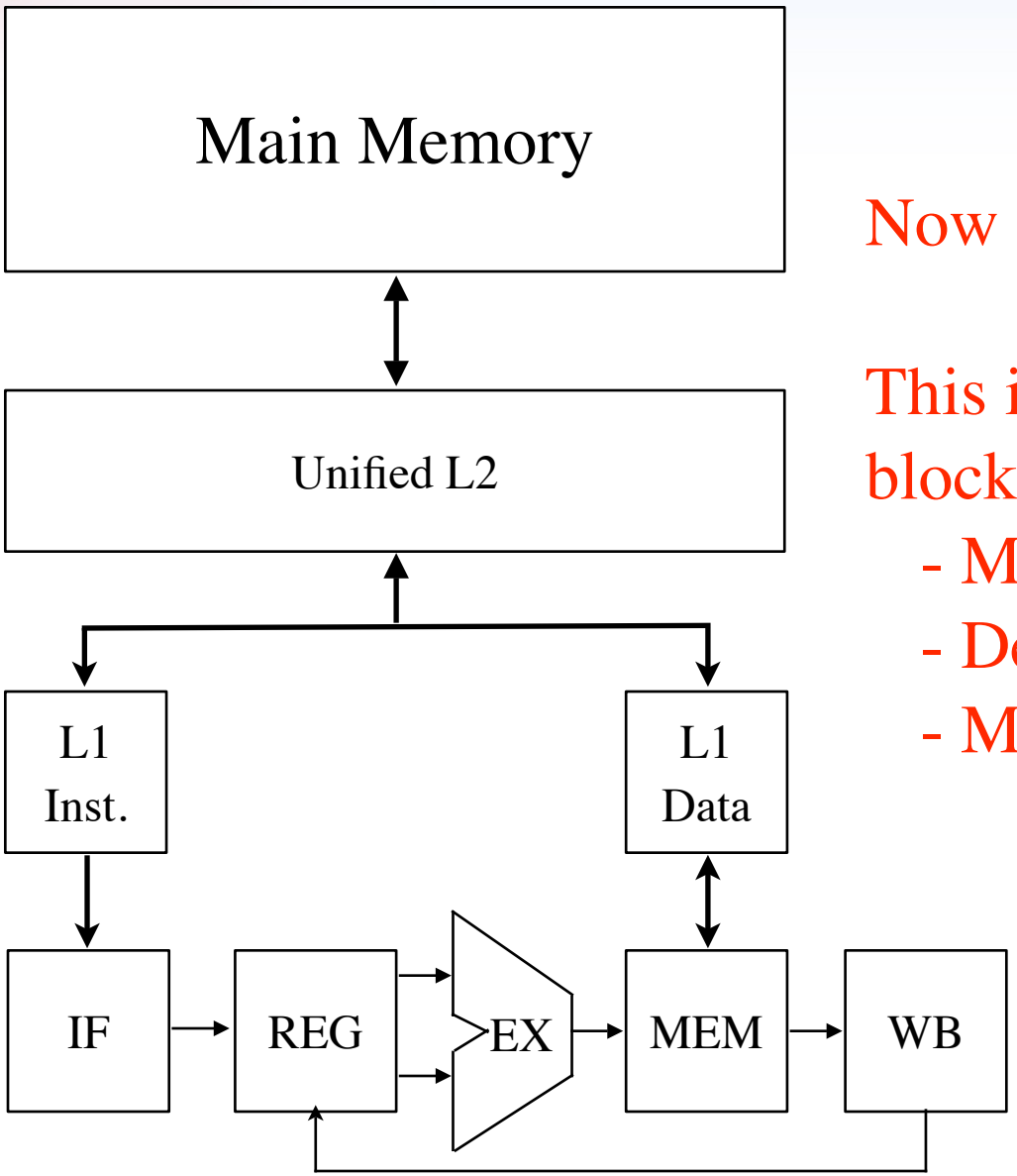
Add An L1 Cache



An L2 Cache



And Main Memory

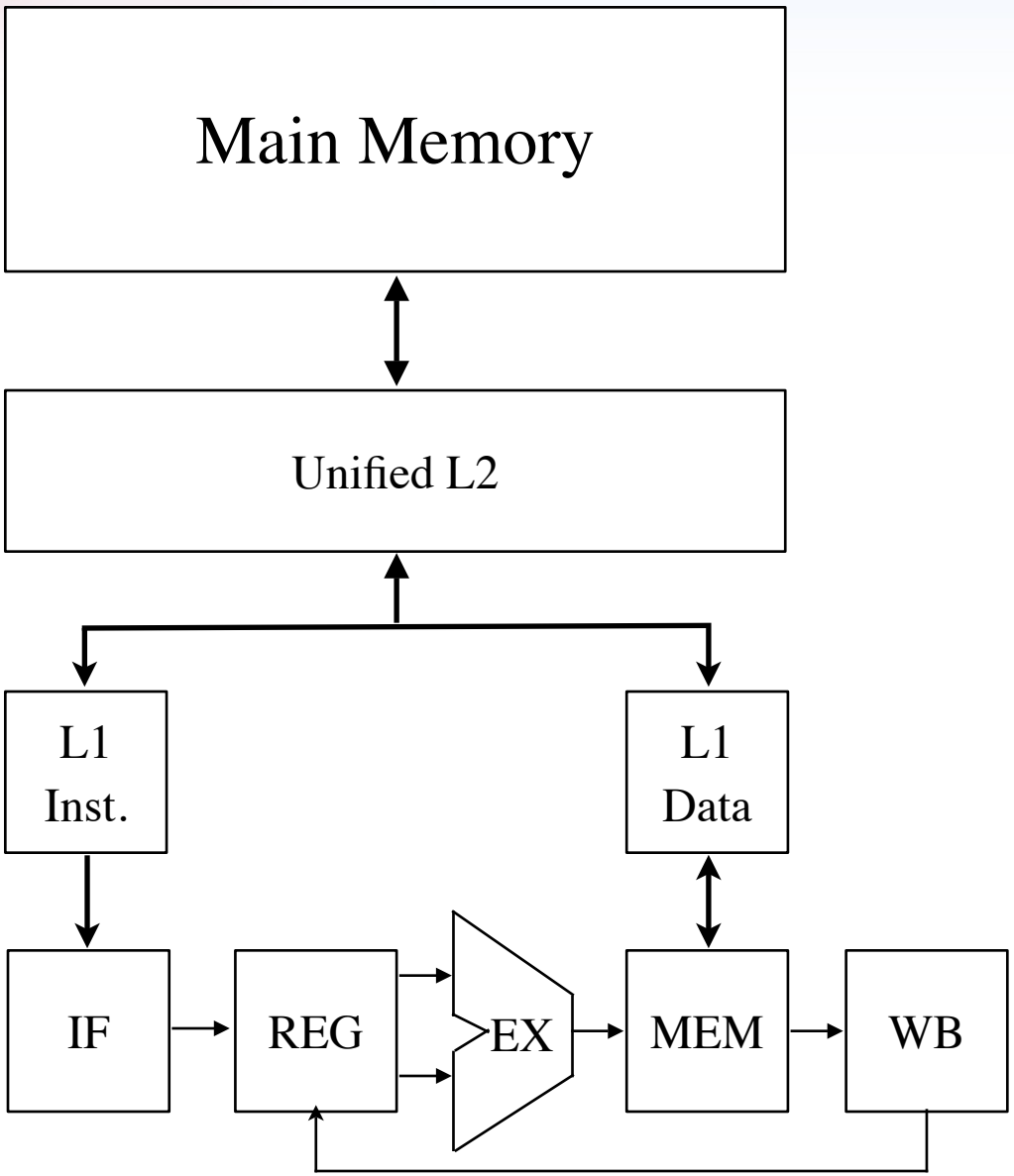


Now we have a computer (circa 1995)

This is looks a lot like the basic building block for today's machines, but they have

- Many of these
- Deeper pipelines
- More complex memory hierarchies

But latency's complicated...



O(100ns) - 200 clocks at 2GHz

O(10ns) - 20 clocks

O(2ns) - 4 clocks

O(<<1ns) - 1 clock

Example Code - Sparse Matrix Vector Multiplly

```
void matvec(int nnz, int n, double A_values[], int A_indices[], int A_offsets[],
double x[], double y[]) {
// Computes y = A*x: A is a sparse square matrix of dimension n w/ nnz nonzeros,
// x is a vector of known values and y (on exit) contains the result of A*x.
// nnz - Number of nonzero entries in sparse matrix A.
// n – Dimension of A, x and y.
// A_values – Nonzero matrix values stored contiguously row by row.
// A_indices – Column indices with matrix entries stored in A_values.
// A_offsets – Offsets of each row into A_values and A_indices.
// x – Input vector
// y – Output vector.

 int jstart = 0;
 int jstop = A_offset[0];
  for (int i=0; i<n; ++i) {
    jstart = jstop;
    jstop = A_offset[i+1];
    double sum = 0.0;
    for (int j=jstart; j<jstop; ++j)
        sum+= A_values[j]*x[A_indices[j]];
    y[i] = sum;
    }

 return;
}
```

NNSA National Nuclear Security Administration

iAA

Thanks to Mike Heroux, SNL for making this a real example

Sandia National Laboratories

Example Code - Sparse Matrix Vector Multiplly

```
void matvec(int nnz, int n, double A_values[], int A_indices[], int A_offsets[],
double x[], double y[]) {
// Computes y = A*x: A is a sparse square matrix of dimension n w/ nnz nonzeros,
// x is a vector of known values and y (on exit) contains the result of A*x.
// nnz - Number of nonzero entries in sparse matrix A.
// n – Dimension of A, x and y.
// A_values – Nonzero matrix values stored contiguously row by row.
// A_indices – Column indices with matrix entries stored in A_values.
// A_offsets – Offsets of each row into A_values and A_indices.
// x – Input vector
// y – Output vector.

 int jstart = 0;
 int jstop = A_offset[0];
  for (int i=0; i<n; ++i) {
    jstart = jstop;
    jstop = A_offset[i+1];
    double sum = 0.0;
    for (int j=jstart; j<jstop; ++j)
        sum+= A_values[j]*x[A_indices[j]];
    y[i] = sum;
    }

 return;
}
```

Consider just the inner loop

Pseudo-Assembly

```
sum += A_values[j]*x[A_indices[j]];
```

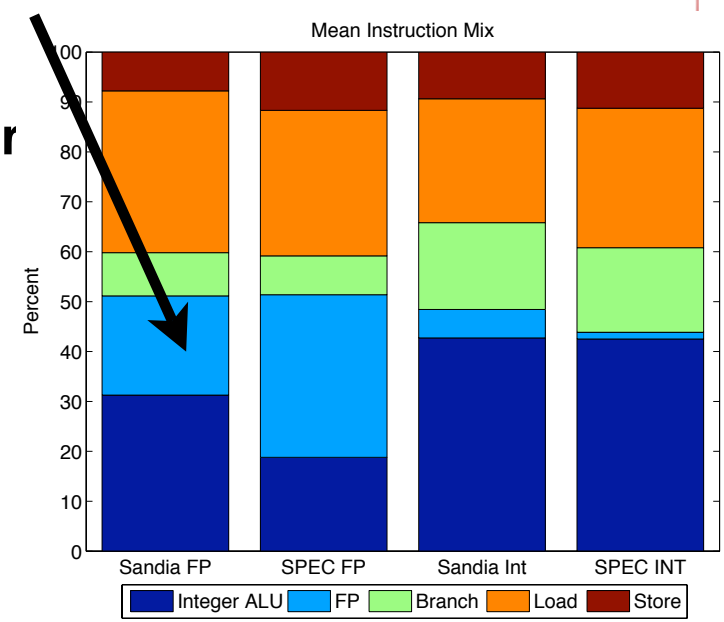
```
; compute j*4 for pointer math
slli $j_tmp, $j, 4 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 4 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
add $sum, $sum, $t3
```

- Assume 32-bit address space
- Registers
  - Base Address of A_values in $A_values
  - Base Address of A_indices in $A_indices
  - Base Address of x in $x
- Basic Values
  - sum in $sum
  - x in $x
- Temporary values labeled $t0...$tn
- Temporary Addresses labeled $a0...$an

Observations

```
; compute j*4 for pointer math
slli $j_tmp, $j, 4 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 4 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```

- **Real Floating Point Apps don't do many FLOPS**
  - **2 FLOPs (mulf, addf)**
  - **5 integer ops to compute addresses (slli, add, add, slli, add)**
  - **3 memory instructions (loads)**
    - **More if you have to write back sum before the end of the loop**
- **Real apps at Sandia**
  - **Usually < 10% FP**
  - **40-50% mem**
  - **30-40% integer**
  - **10% branch**



National Nuclear Security Administration

Observations (continued)

- Arun Rodrigues points out that we've talked a lot about floating point "resilience", and the potential ability to live with less accuracy, but your state bits are equally likely to be in error:
  - Inaccurate integer address calculations would be bad (40% of instructions)
  - Inaccurate branching would be worse (10% of instructions)
    - if(my_fp_calculation_was_in_error)...
  - I don't even know what an inaccurate load or store would be (40% of instructions), but it would be bad
  - Inaccurate instructions could really cause havoc (100% of instructions)

- As a computer architect, the idea of giving the wrong answer to improve performance makes me feel dirty

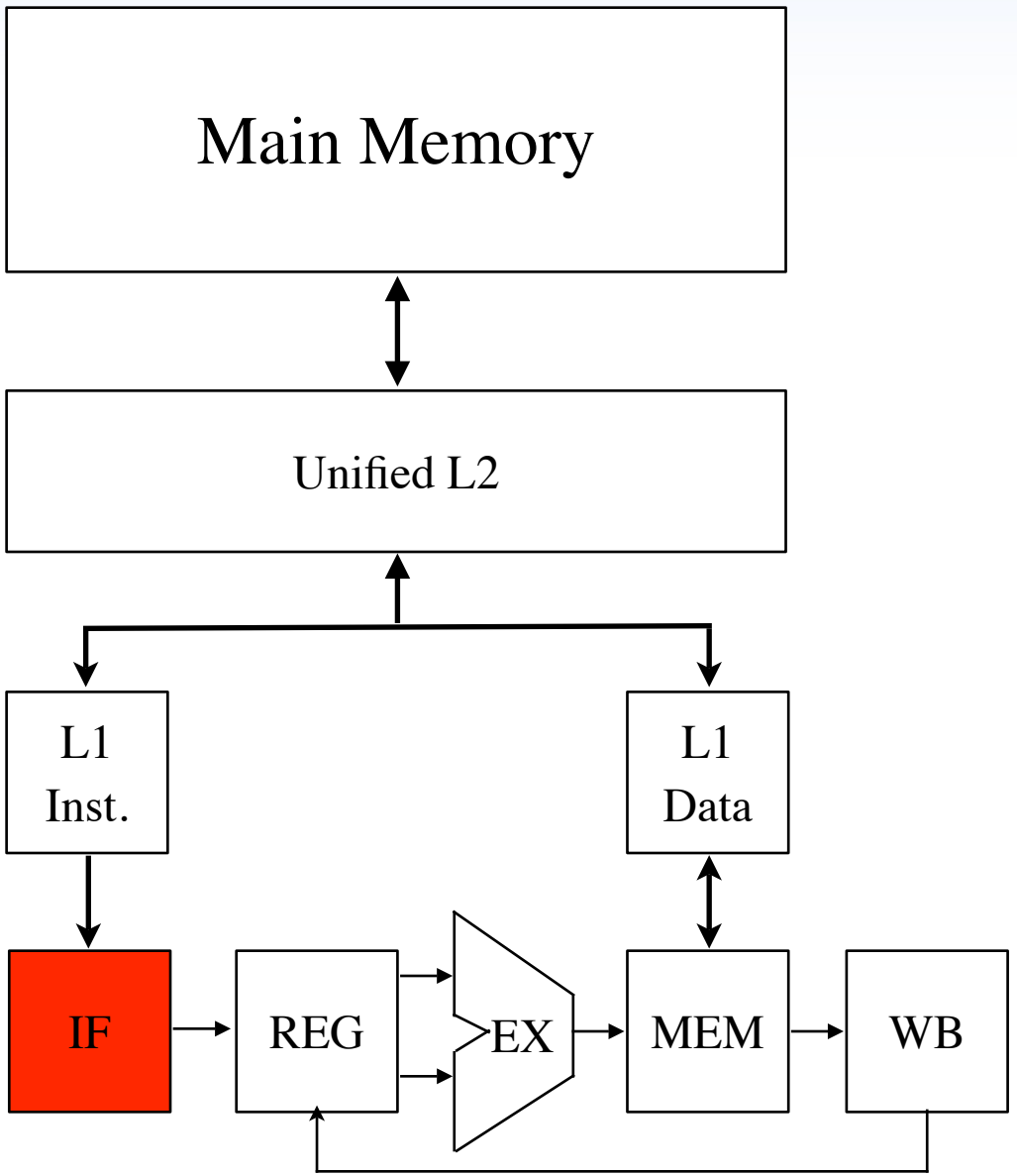
Observations (continued)

```
; compute j*4 for pointer math
slli $j_tmp, $j, 4 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 4 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```

- If the stride one accesses are large enough, they likely miss both caches to memory
  - A_values[j]
  - A_indices[j]
- X[A_indices[j]] introduces a data-dependent memory reference
  - A_indices[j] is required before the X value can be loaded
  - Exhibits *some* temporal and spatial locality

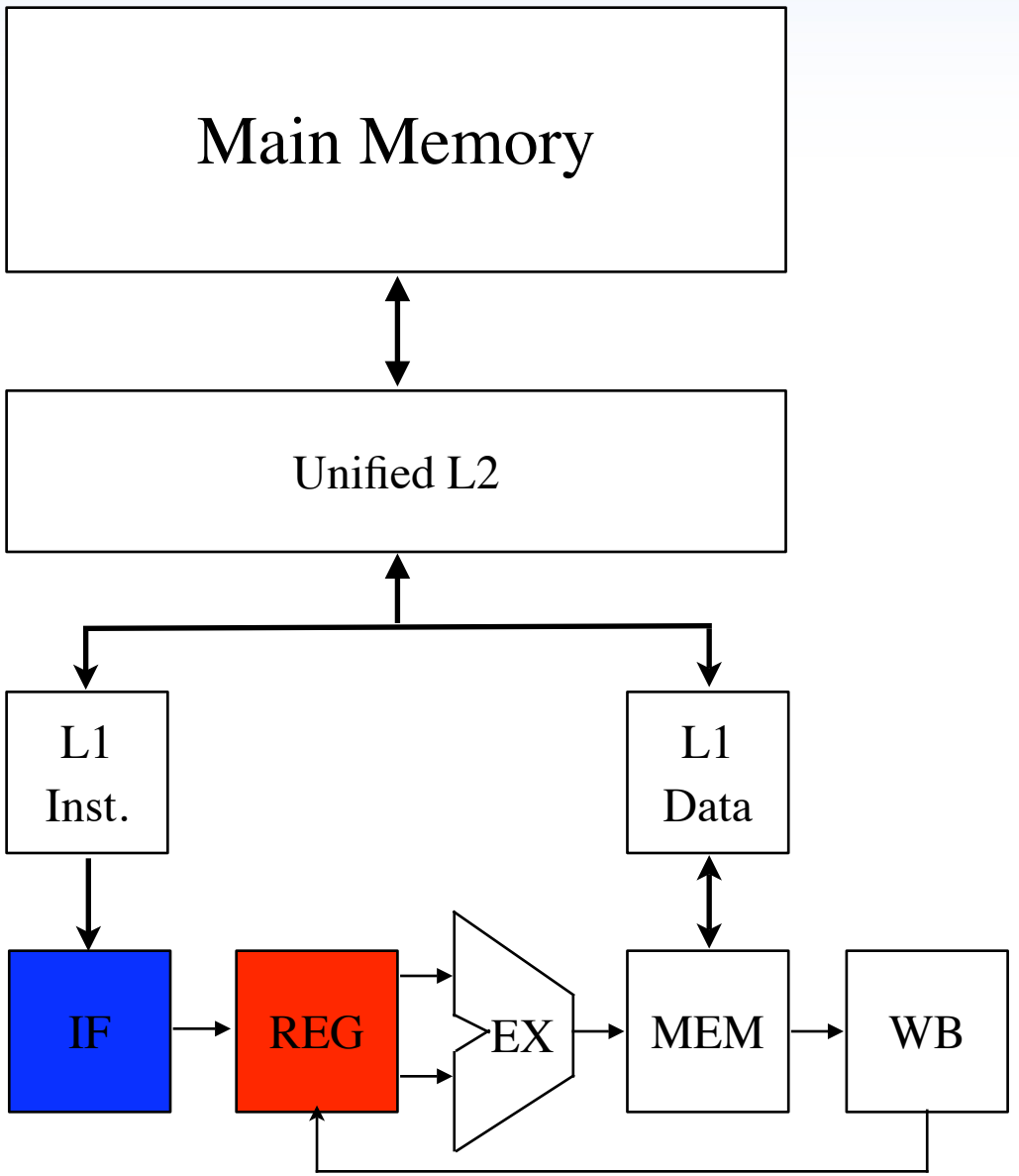
Cycle 1

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



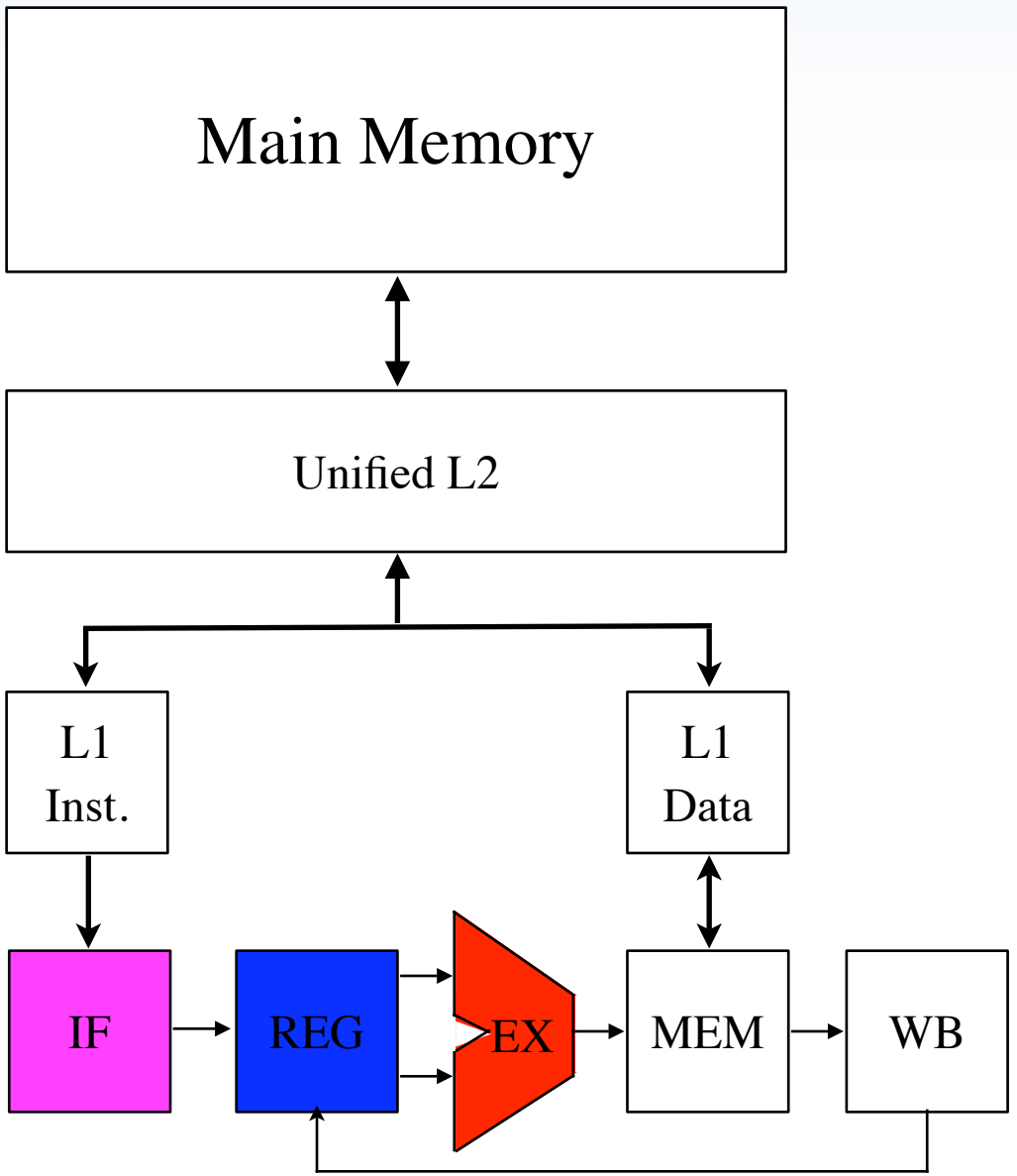
Cycle 2

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



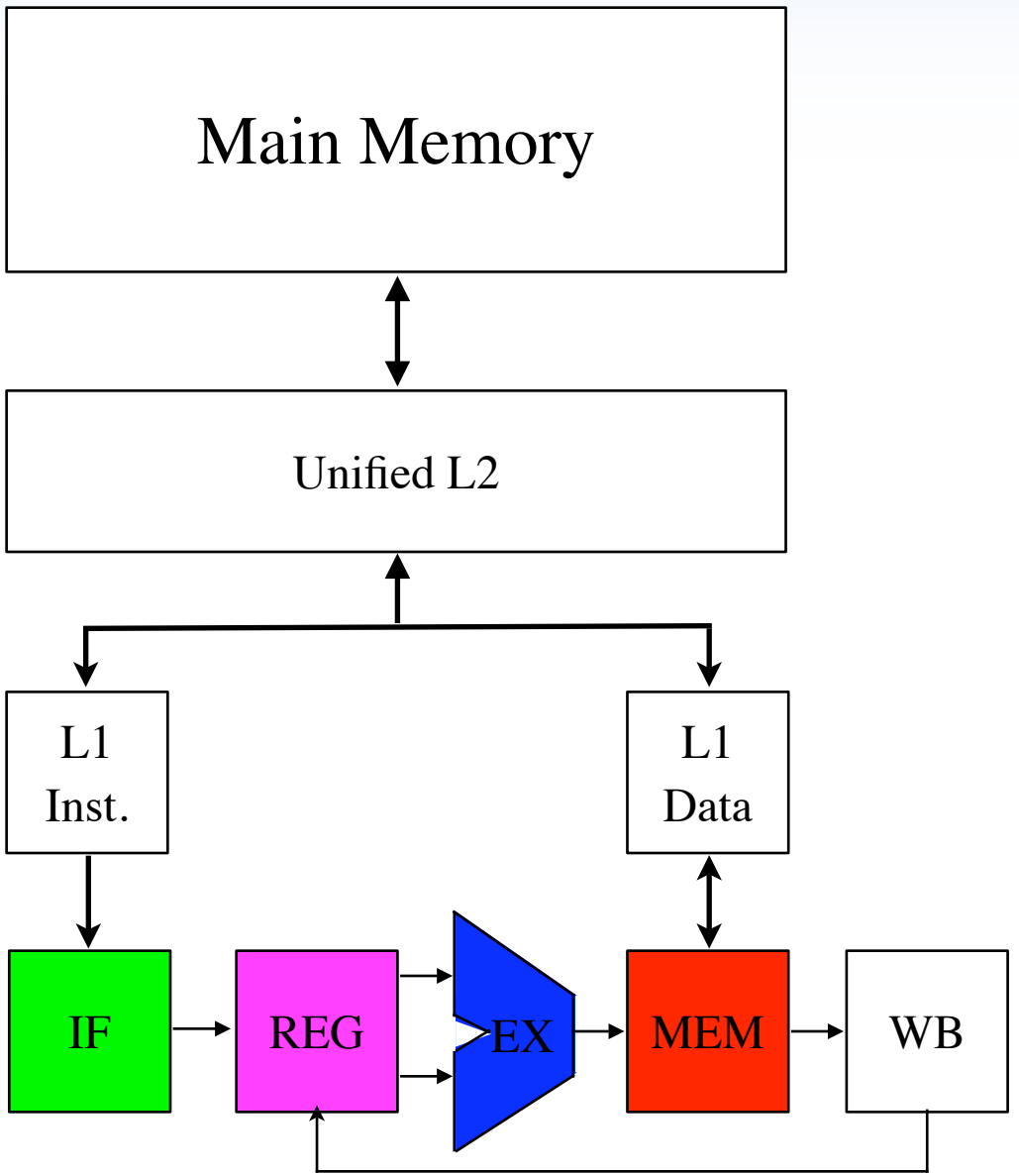
Cycle 3

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



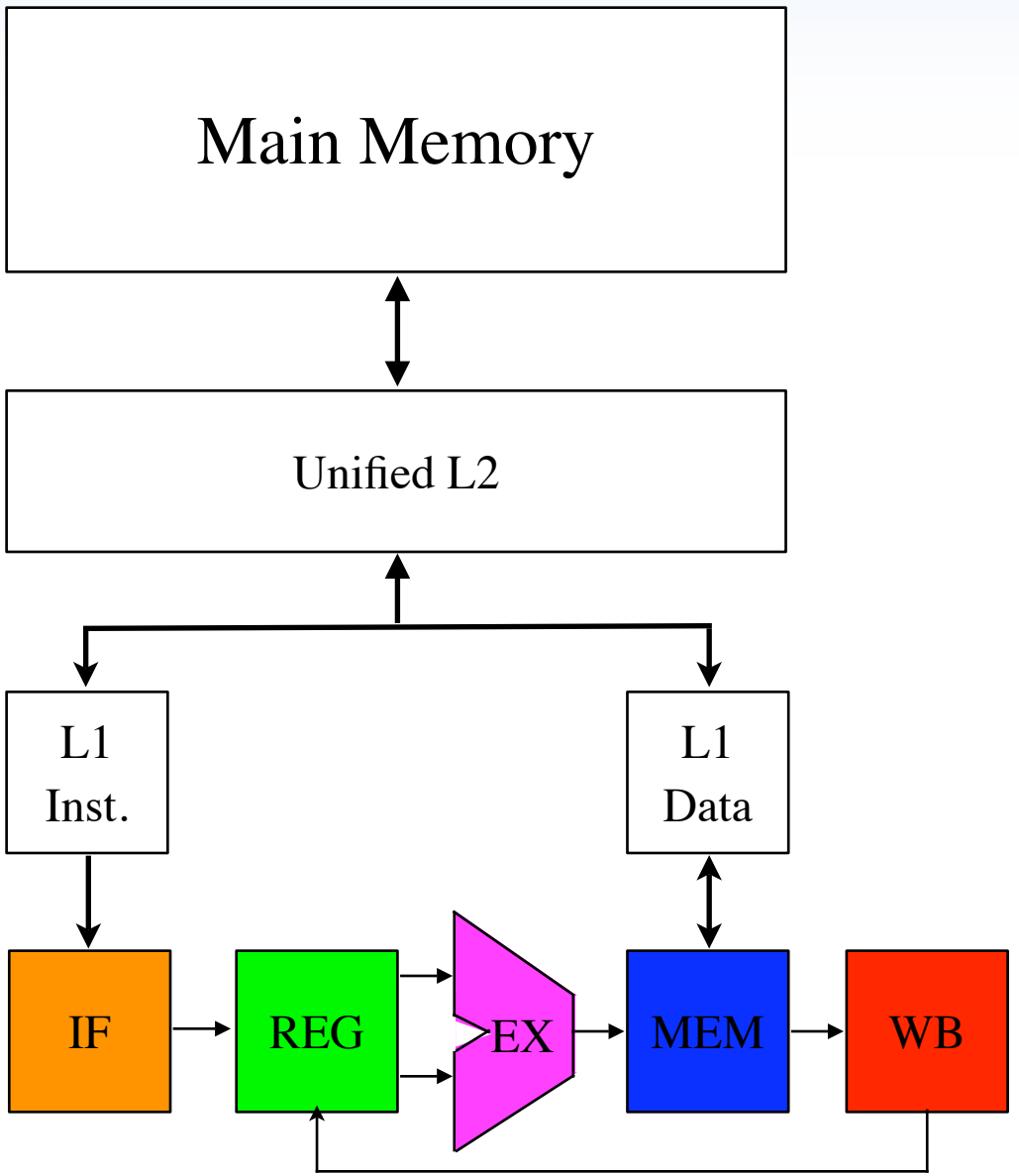
Cycle 4

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



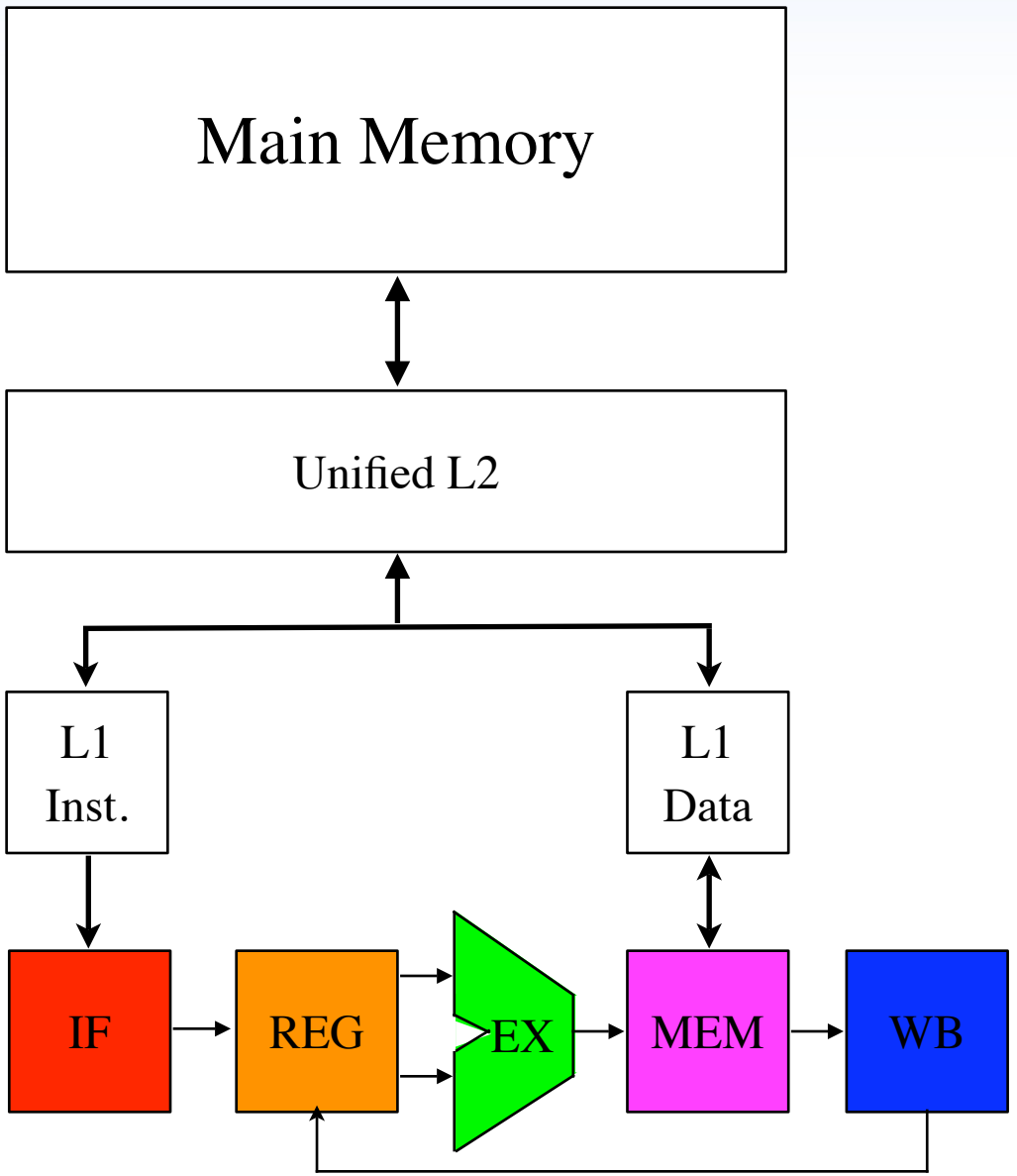
Cycle 5

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



Cycle 6

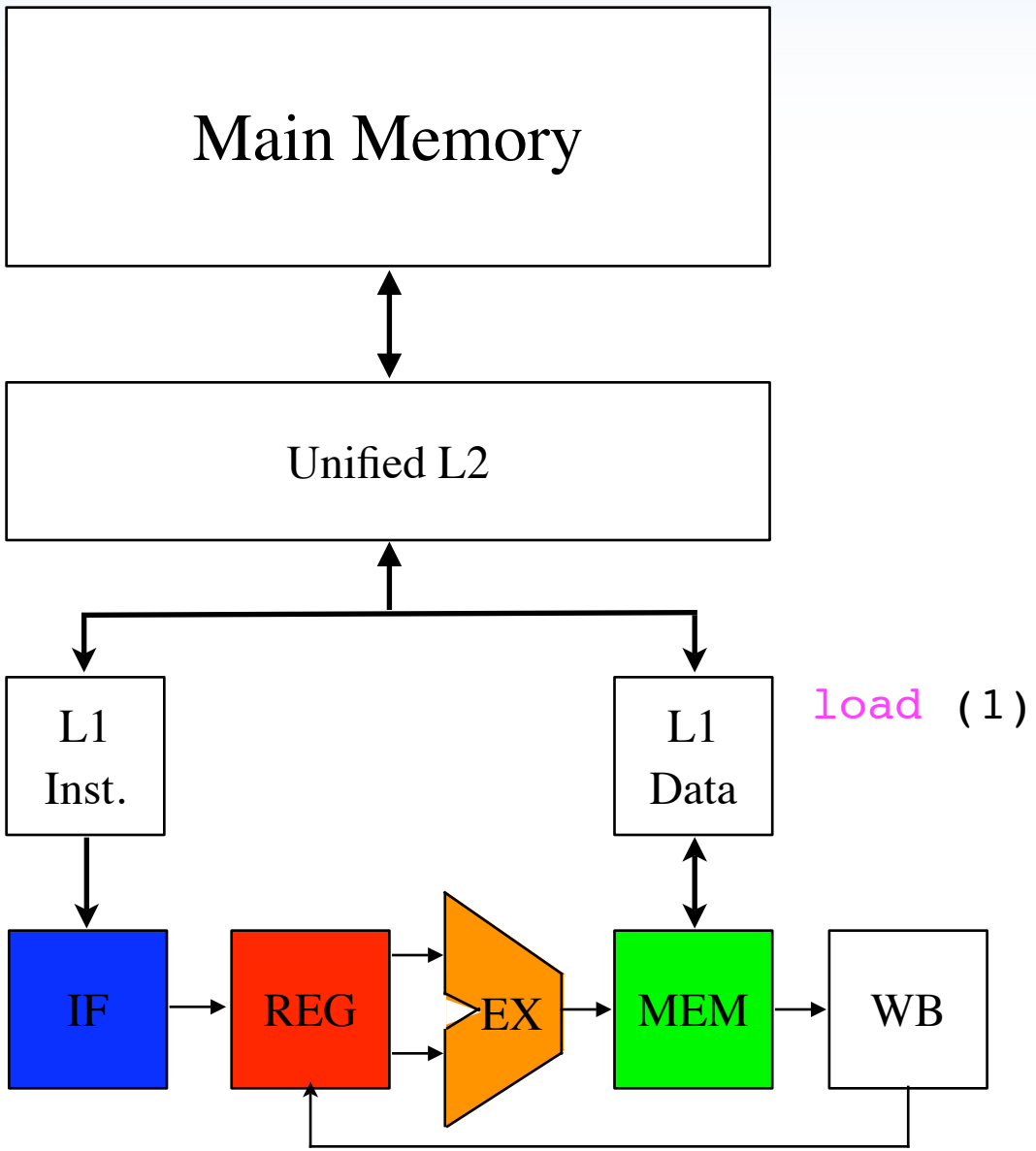
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



load $t0, 0($a0) will MISS to memory!

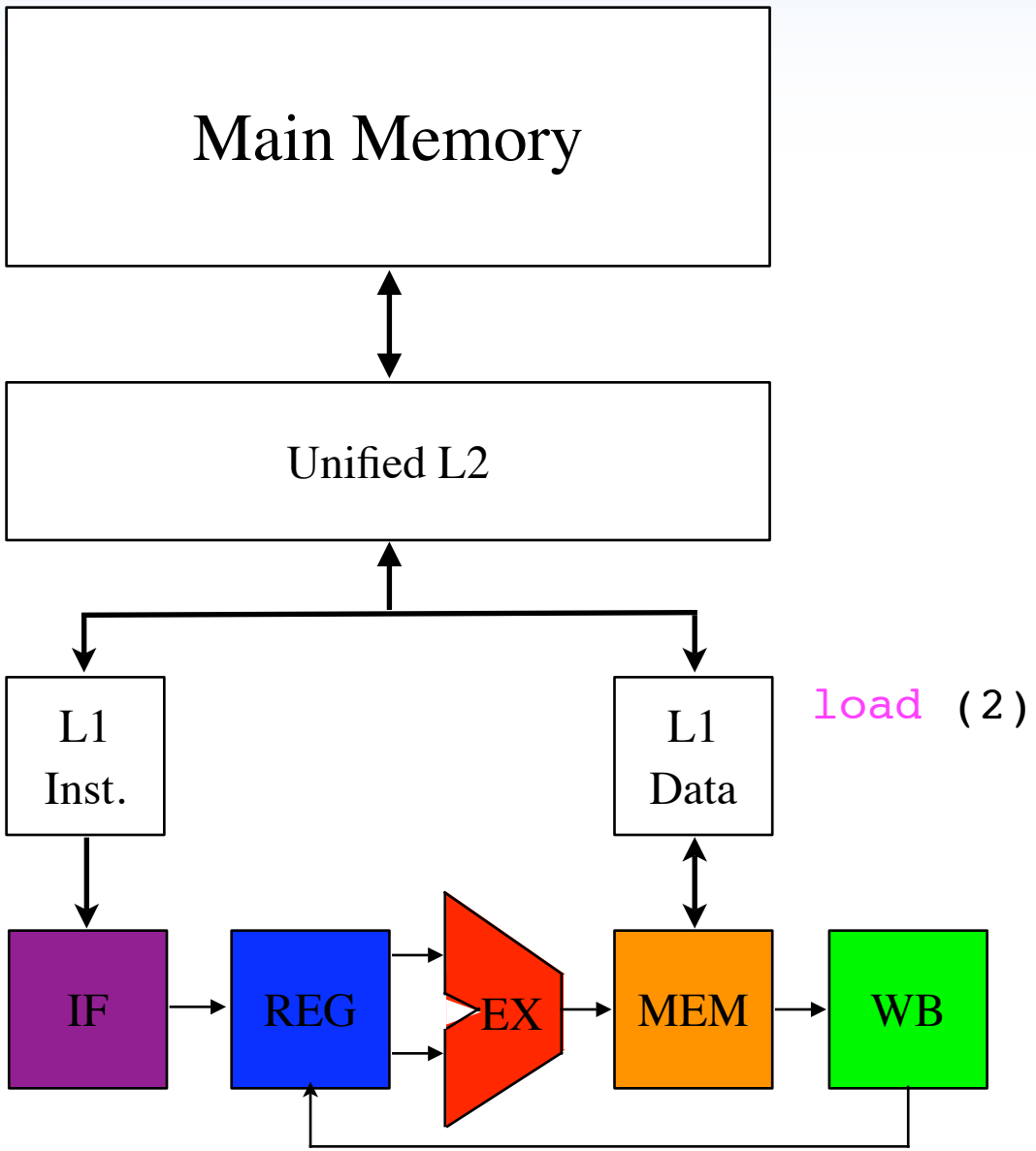
Cycle 7

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



Cycle 8

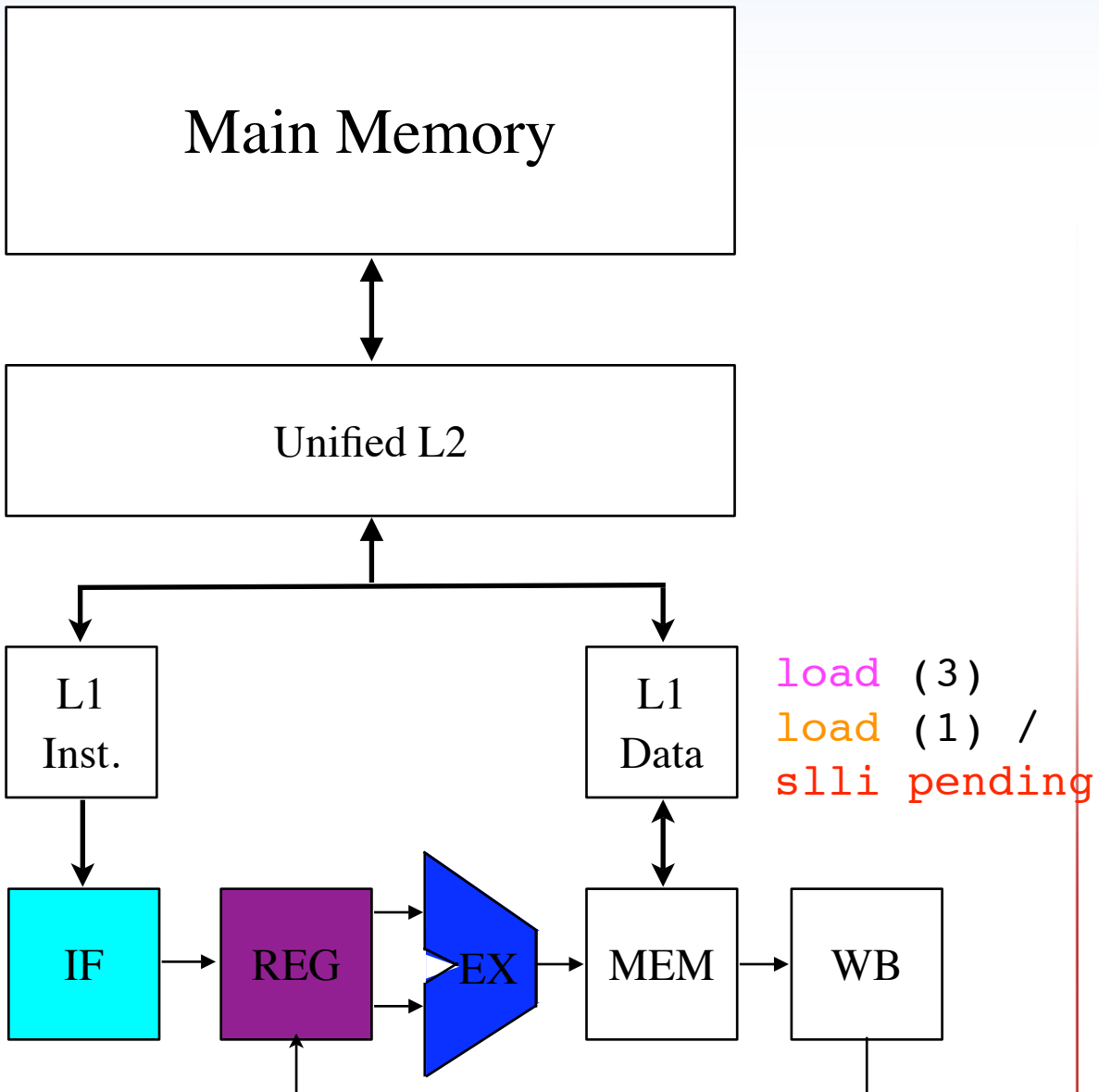
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



load $t1, 0($a1) will MISS to memory!

Cycle 9

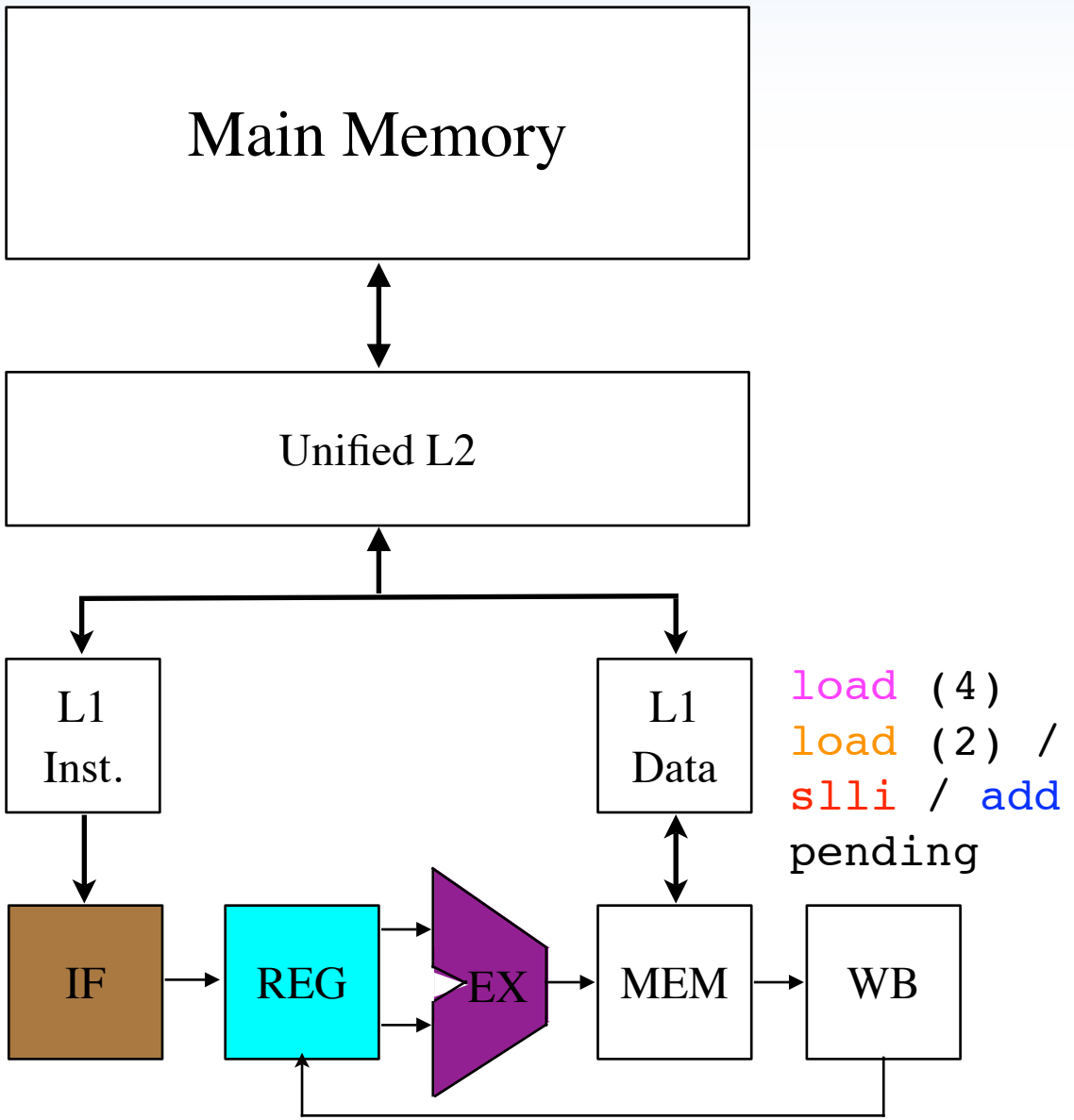
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



slli depends on A_indices[j]

Cycle 9

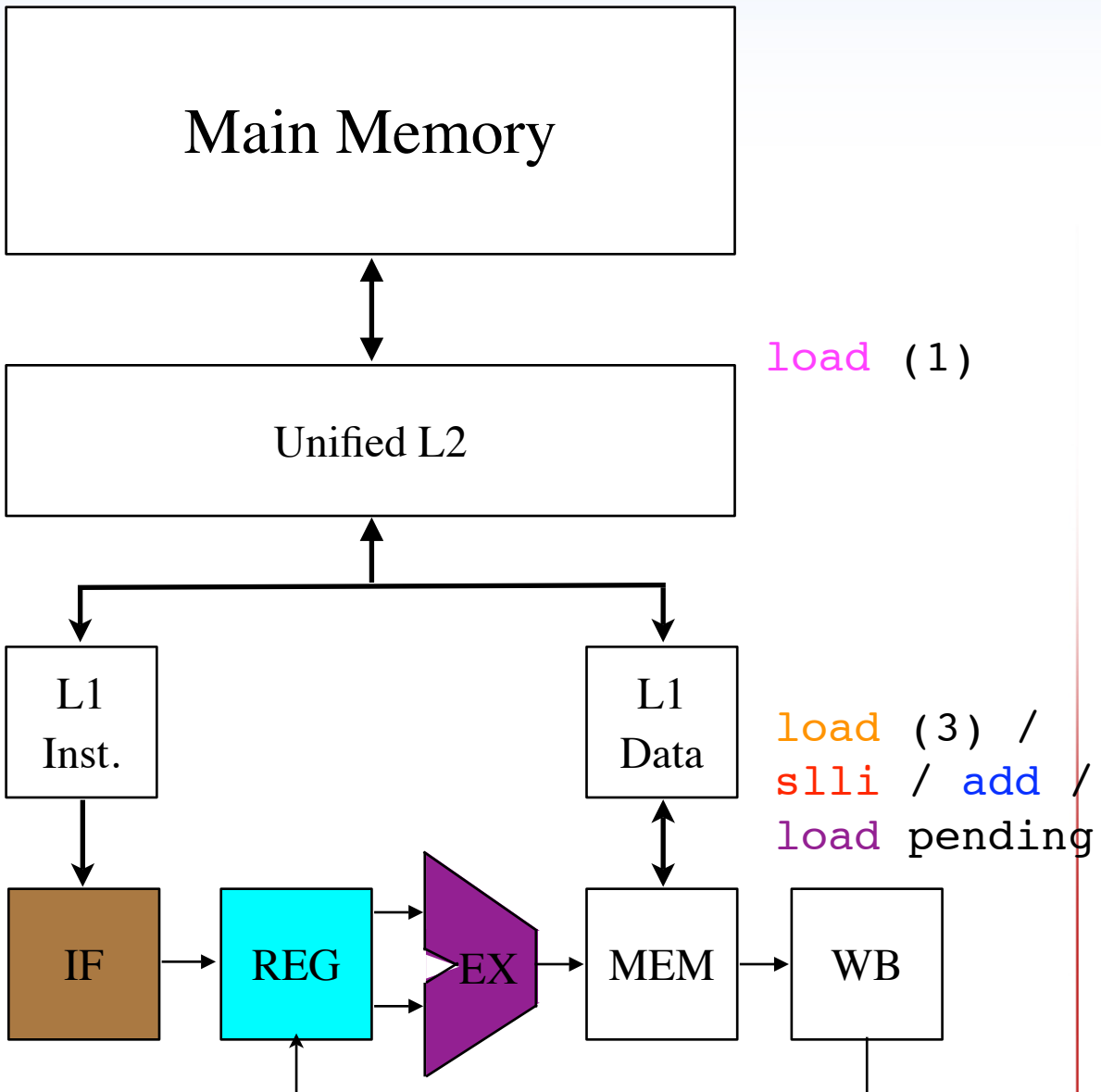
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



add depends on A_indices[j]

Cycle 10

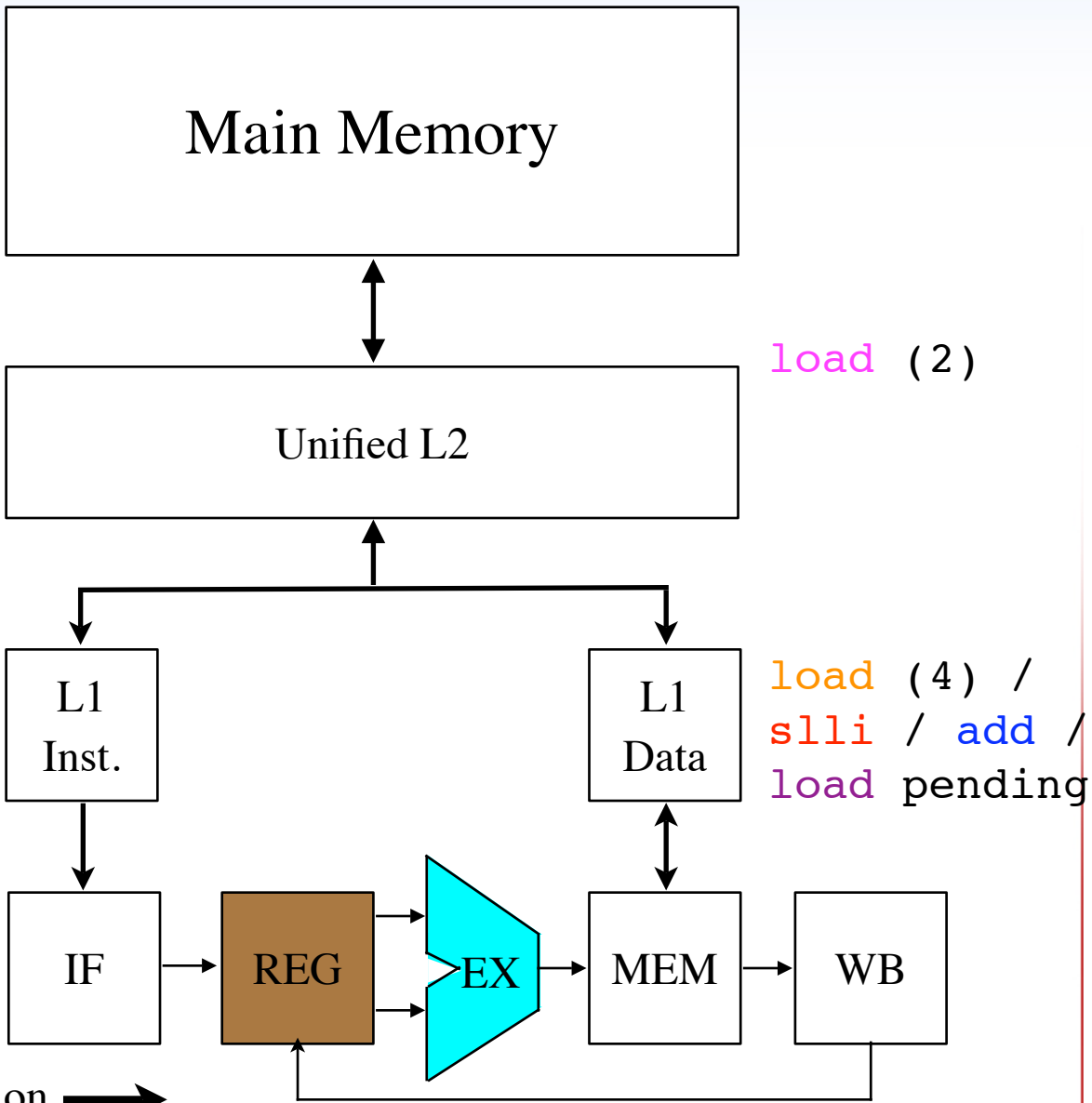
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



load depends on A_indices[j]

Cycle 11

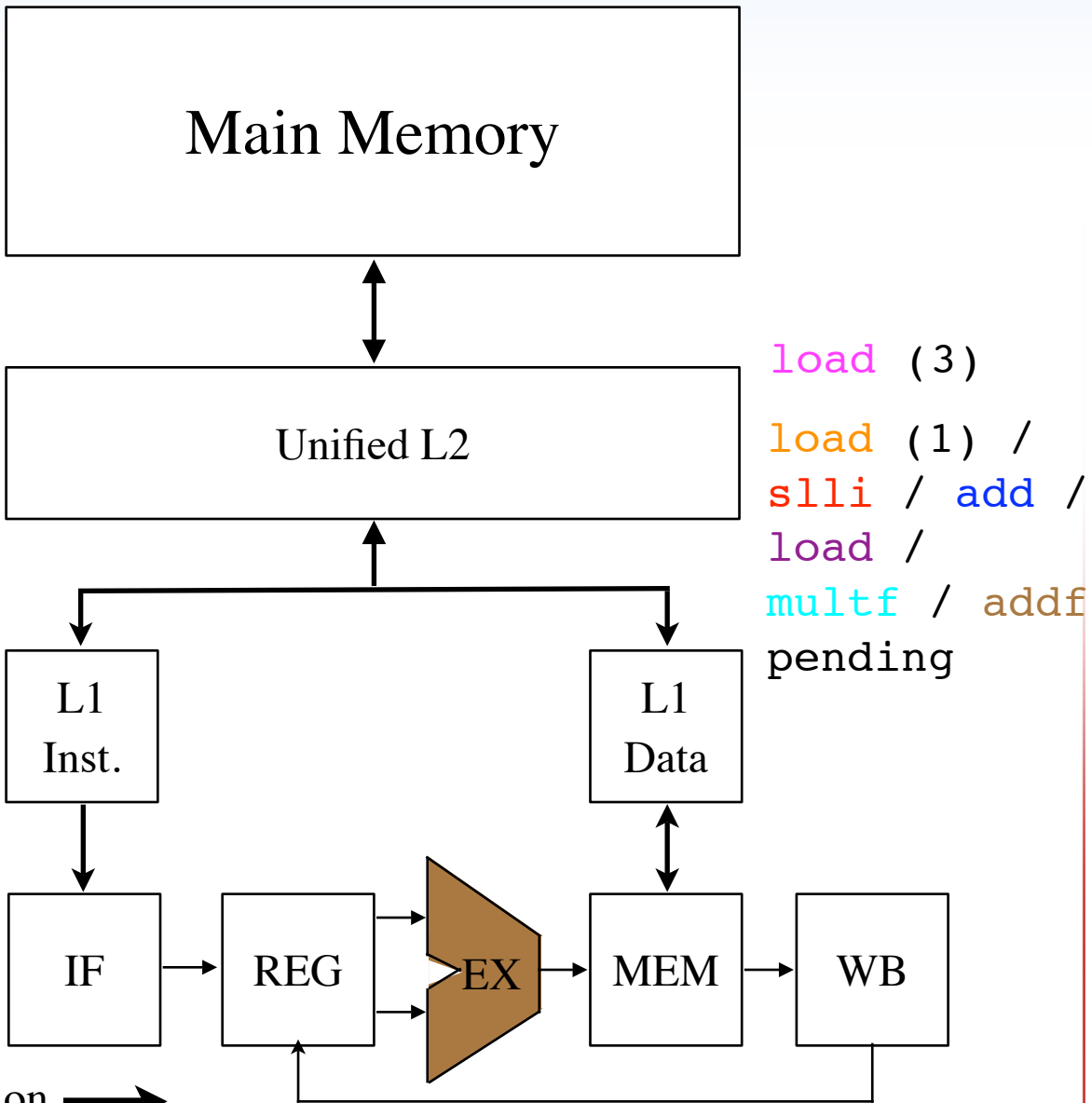
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



multf depends on both loads!

Cycle 12

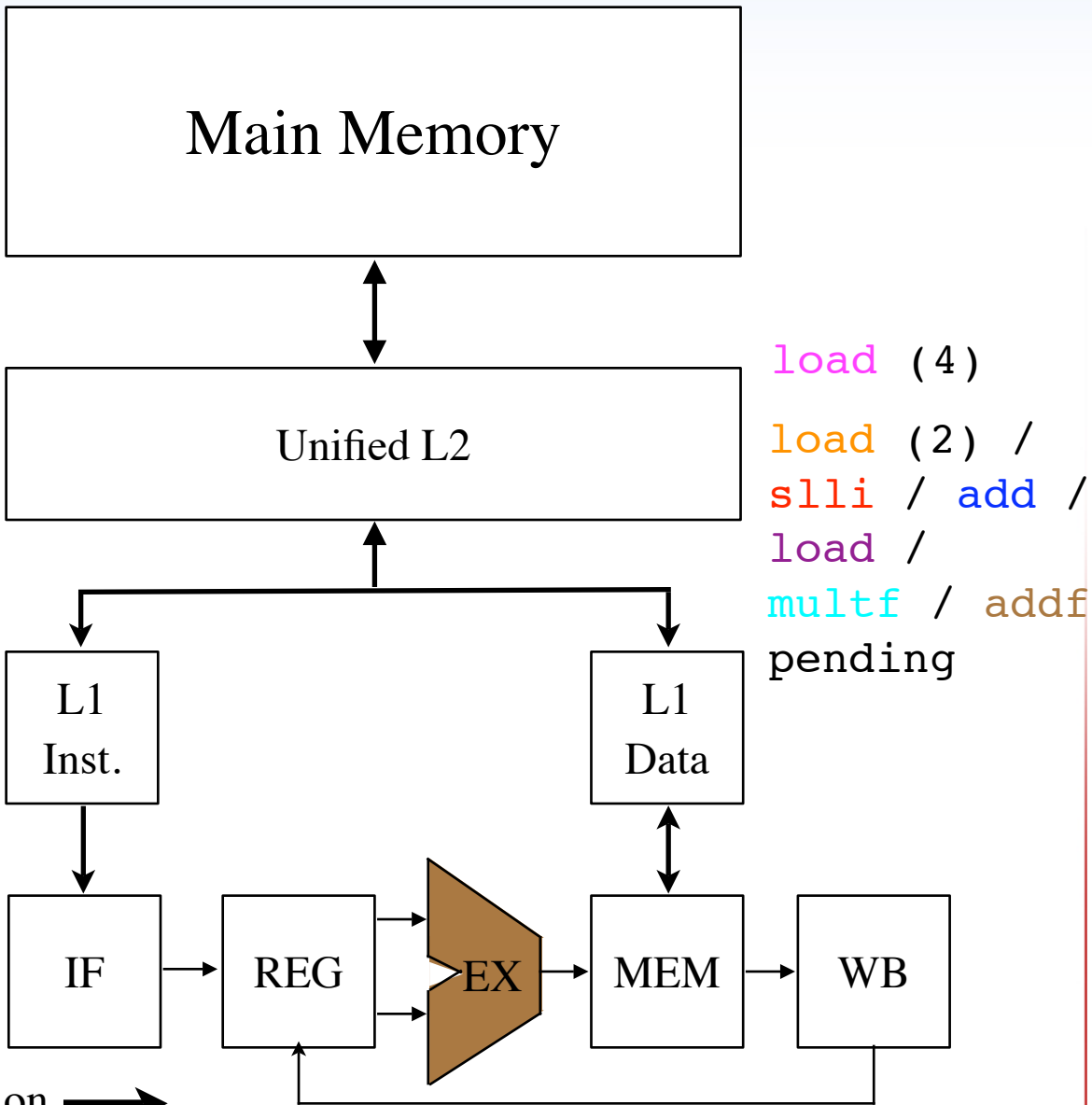
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



addf depends on the multf

Cycle 13

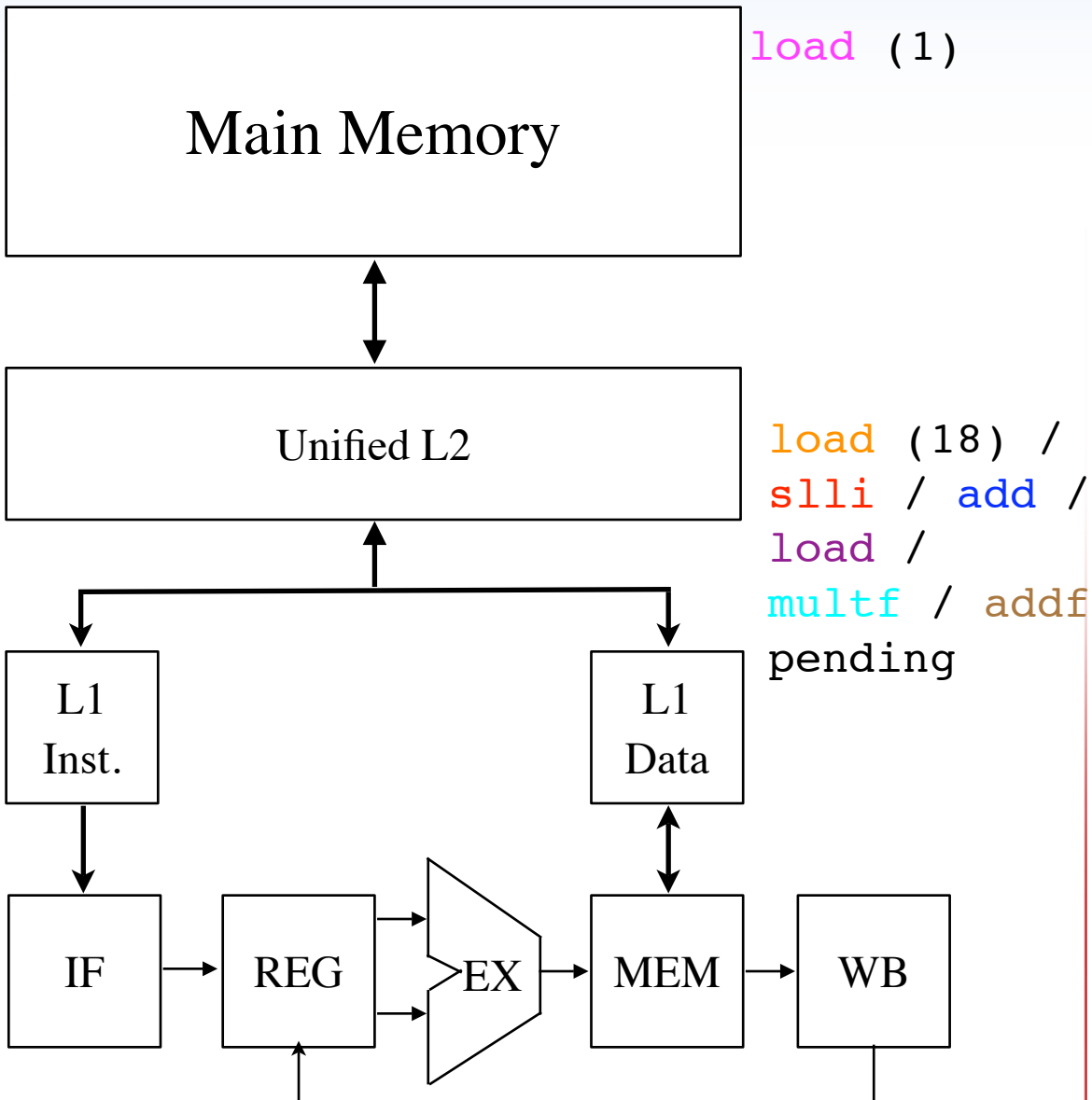
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



addf depends on the multf

Cycle 29

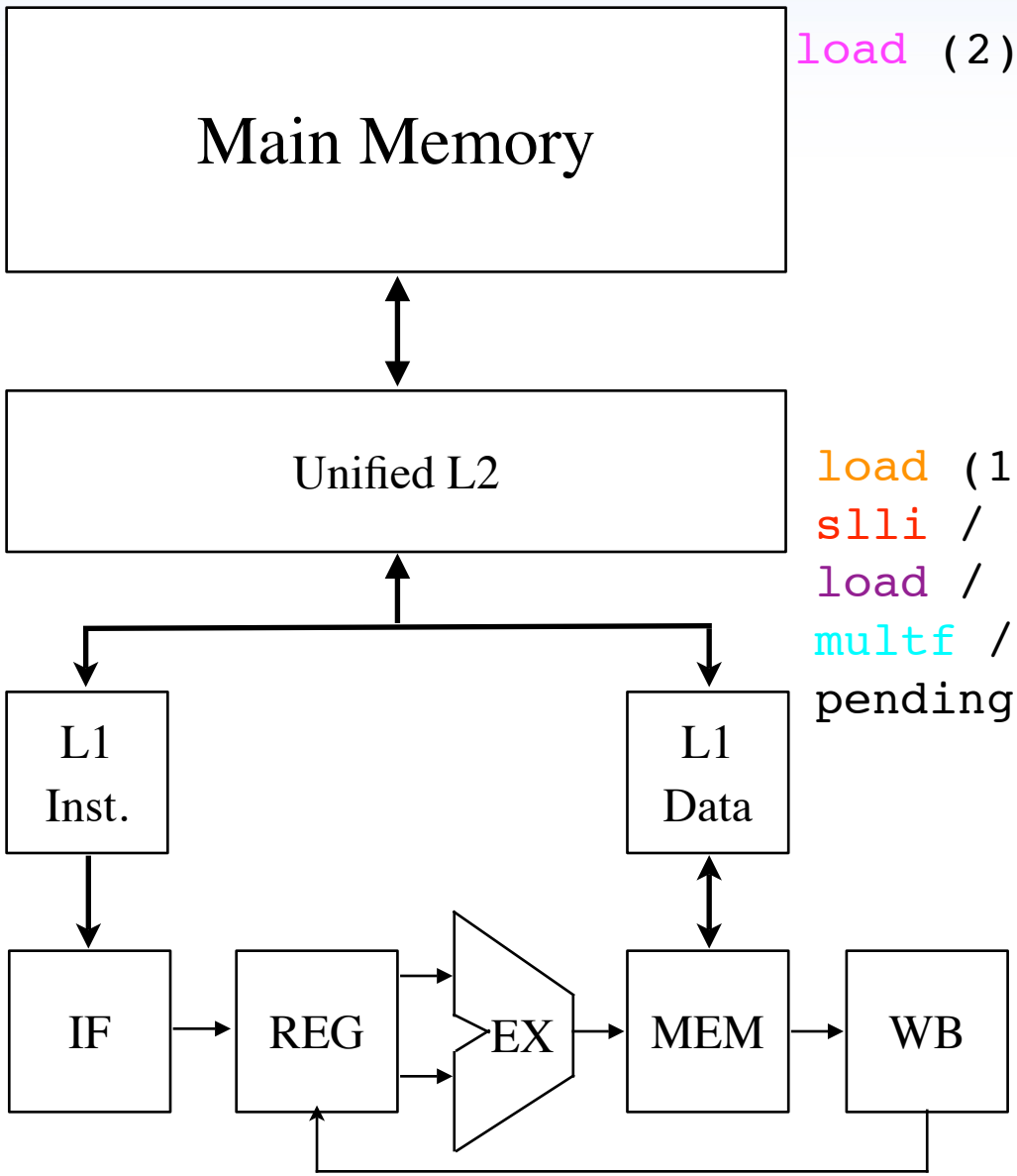
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



Finally, the first load misses to memory

Cycle 30

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```

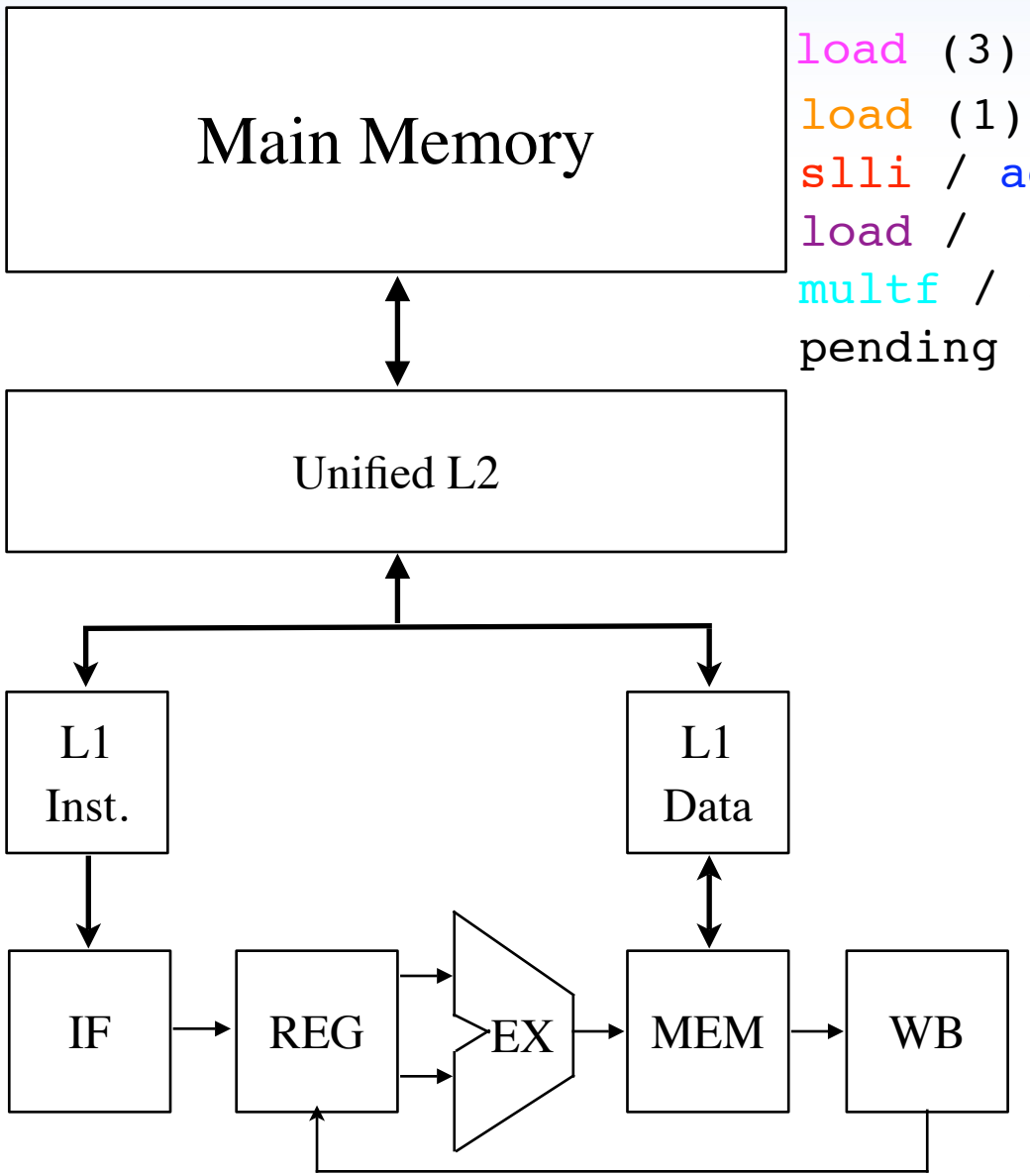



```
load (2)

load (19) /
slli / add /
load /
multf / addf
pending
```

Finally, the first load misses to memory

Cycle 31

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```

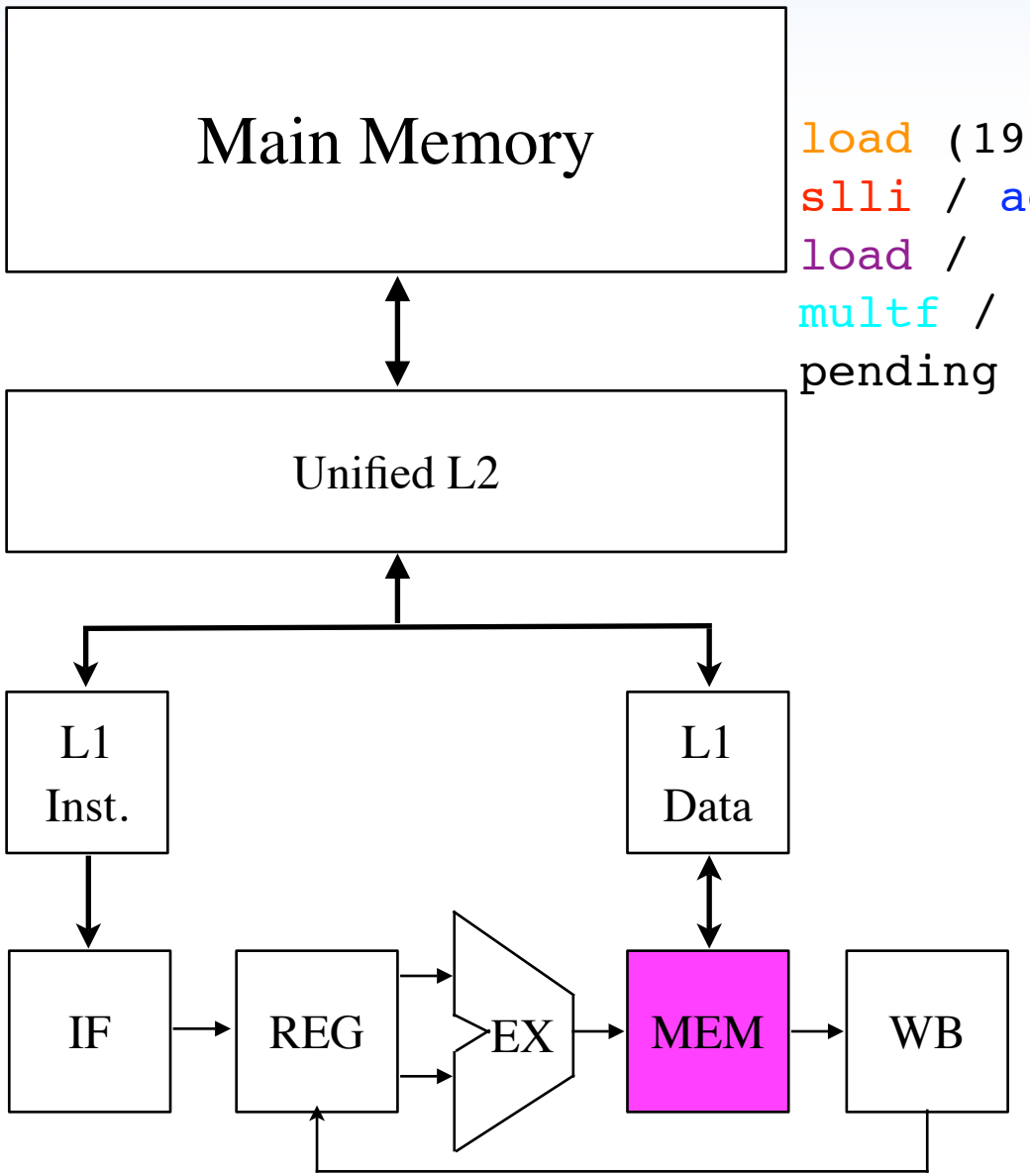



```
load (3)
load (1) /
slli / add /
load /
multf / addf
pending
```

Finally, the first load misses to memory

Cycle 229

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```

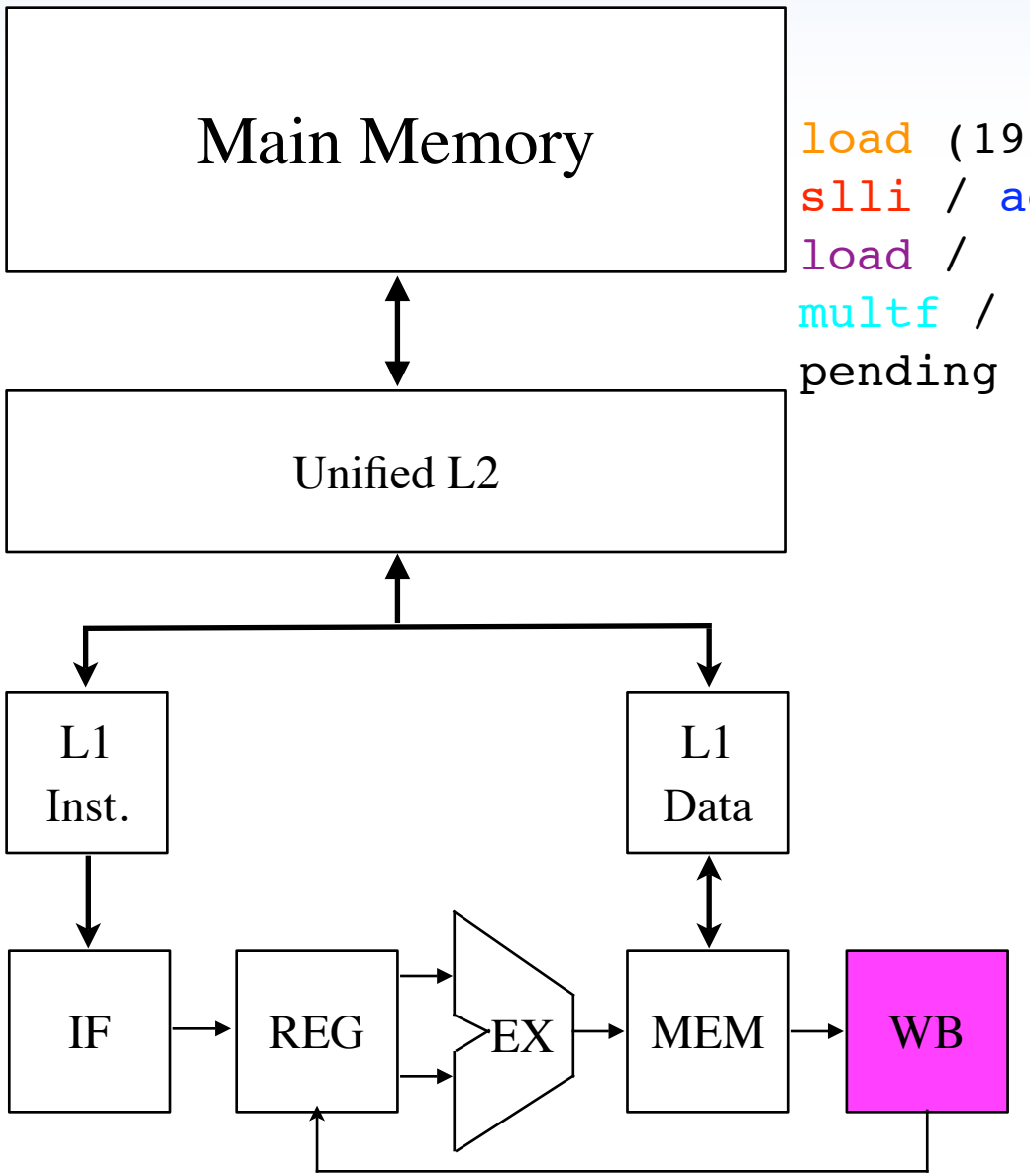


load (197) / slli / add / load / multf / addf pending

The first load returns to finish executing

Cycle 230

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```

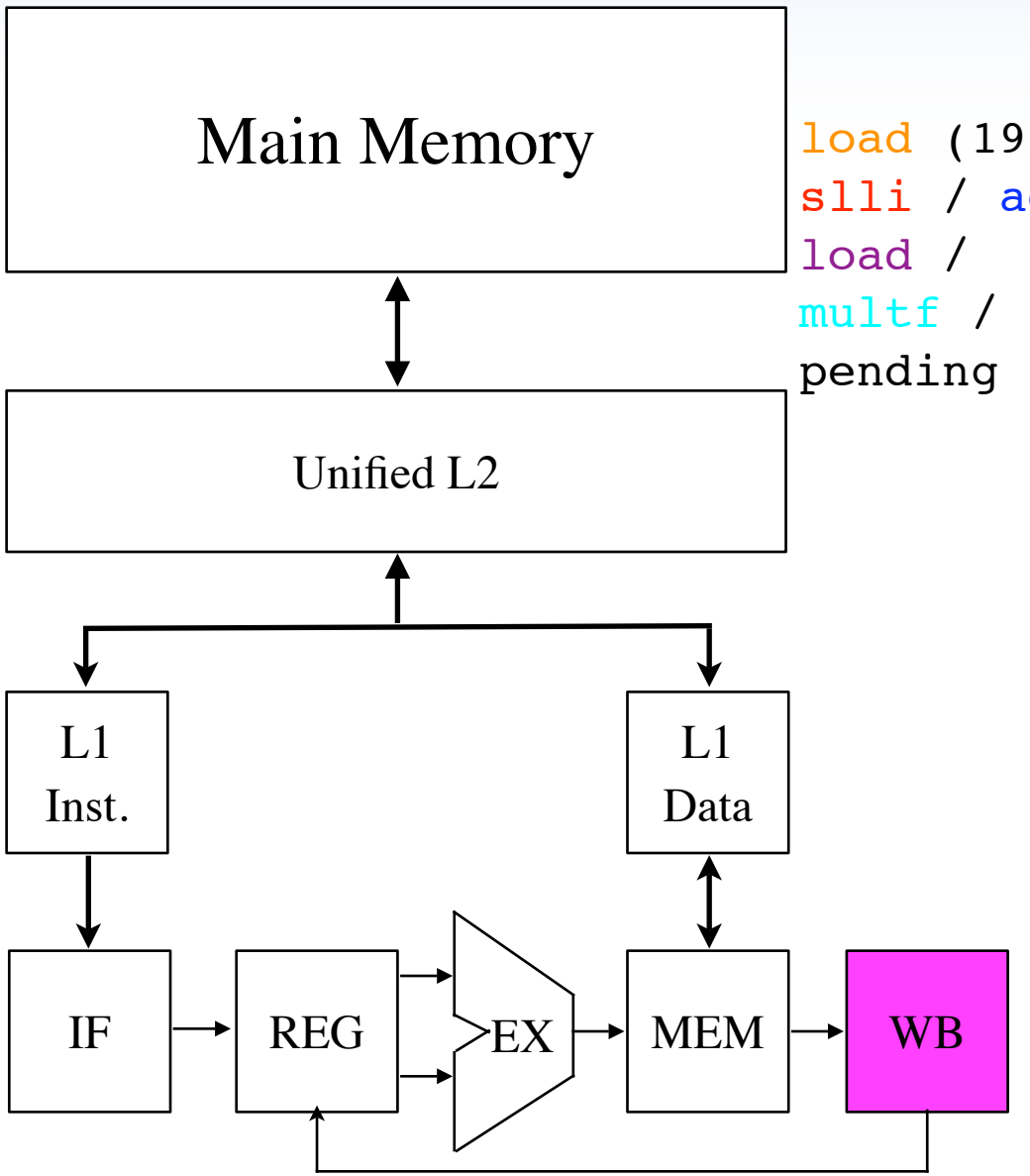


load (198) / slli / add / load / multf / addf pending

The first load returns to finish executing

Cycle 231

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```

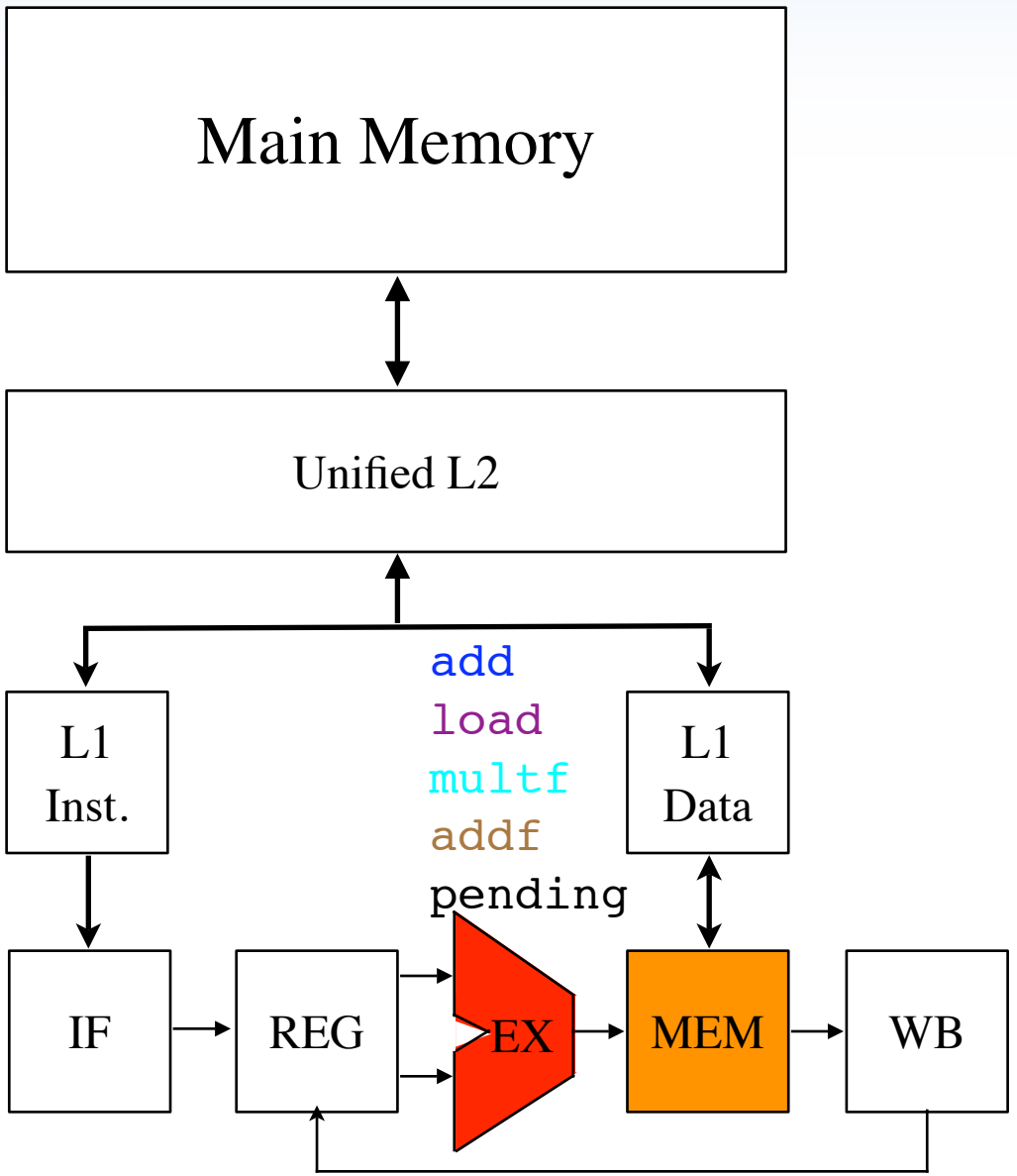


load (199) / slli / add / load / multf / addf pending

The first load returns to finish executing

Cycle 232

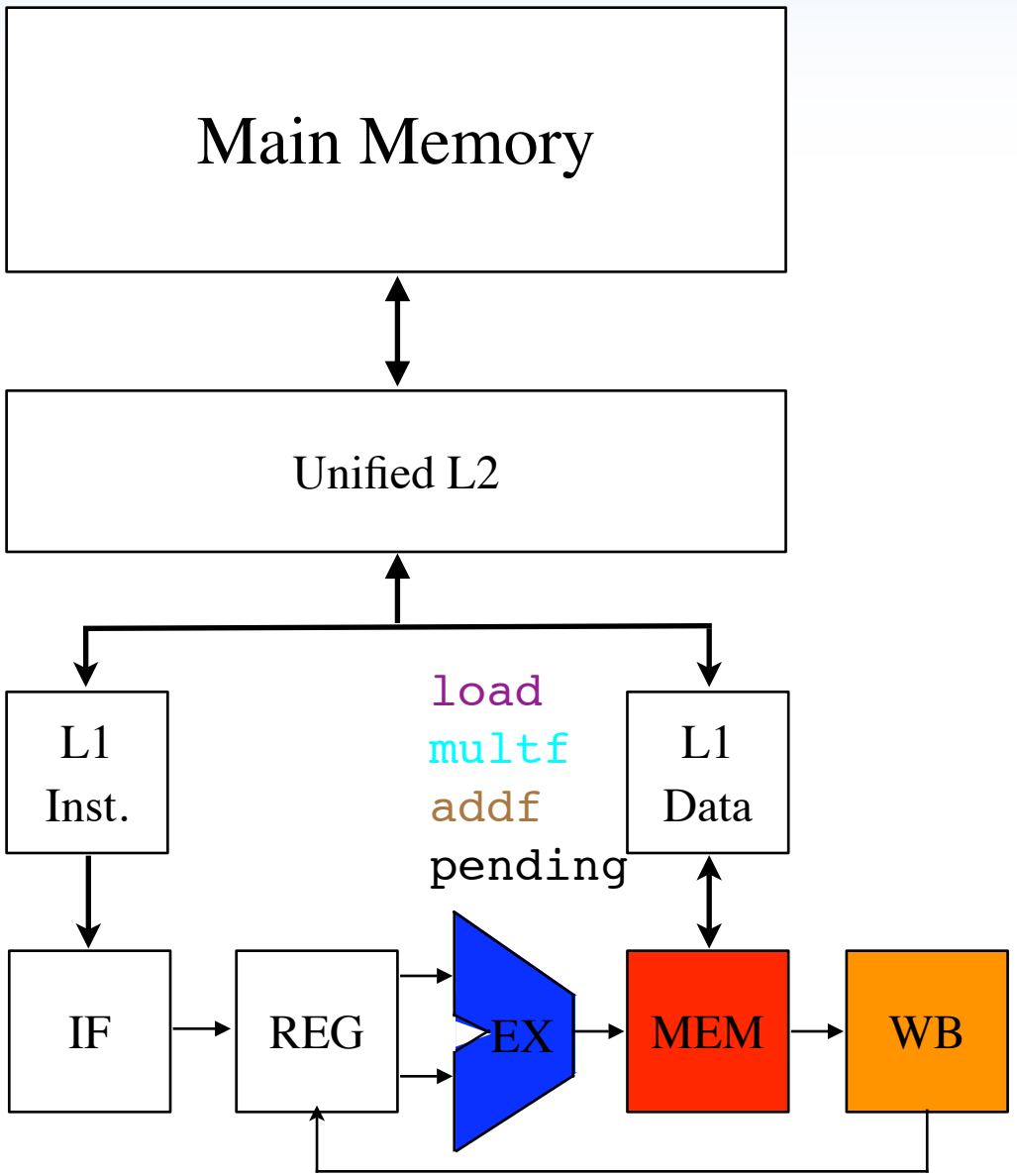
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



The second load completes

Cycle 233

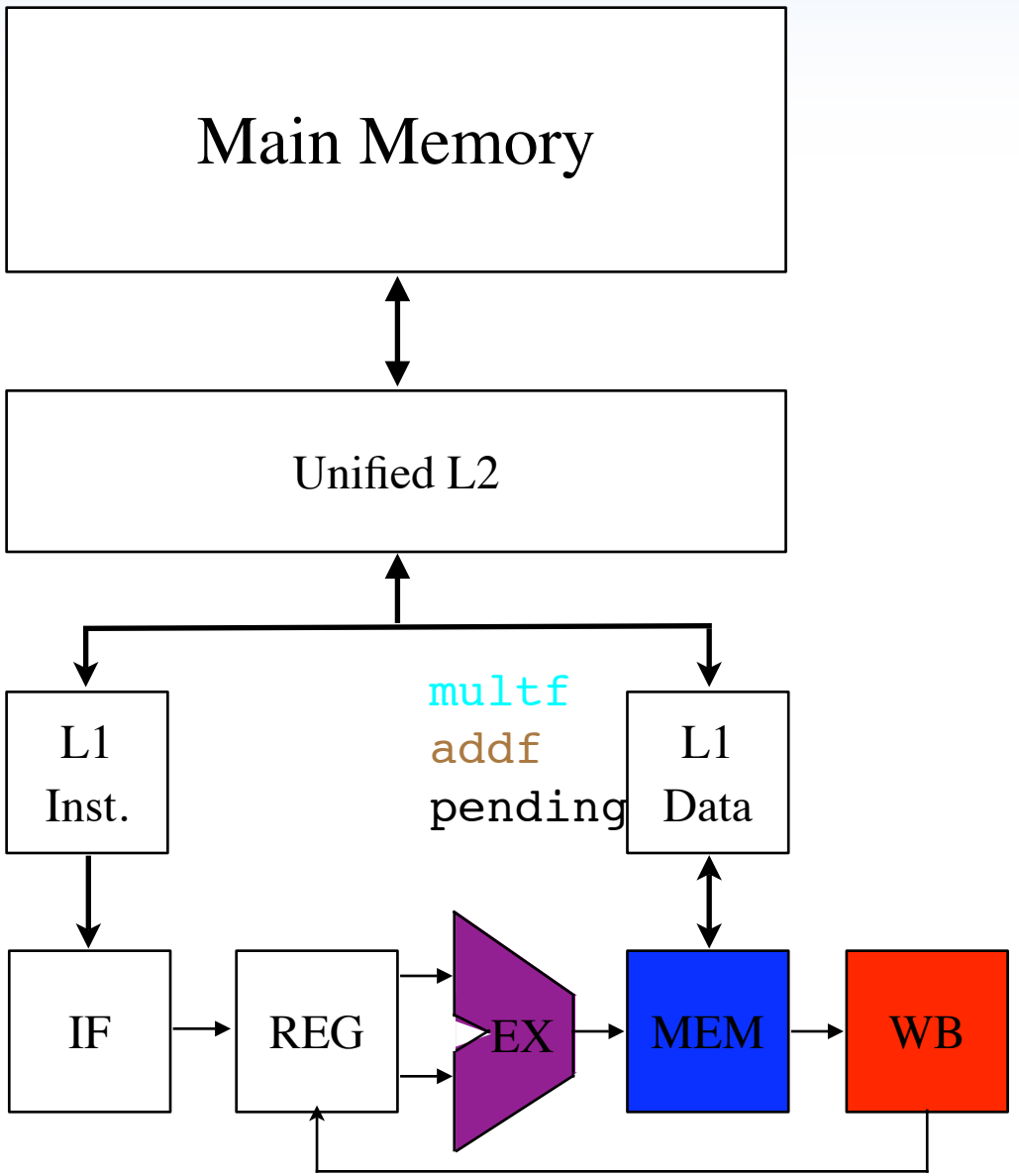
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



Finally, the first load is retired!

Cycle 234

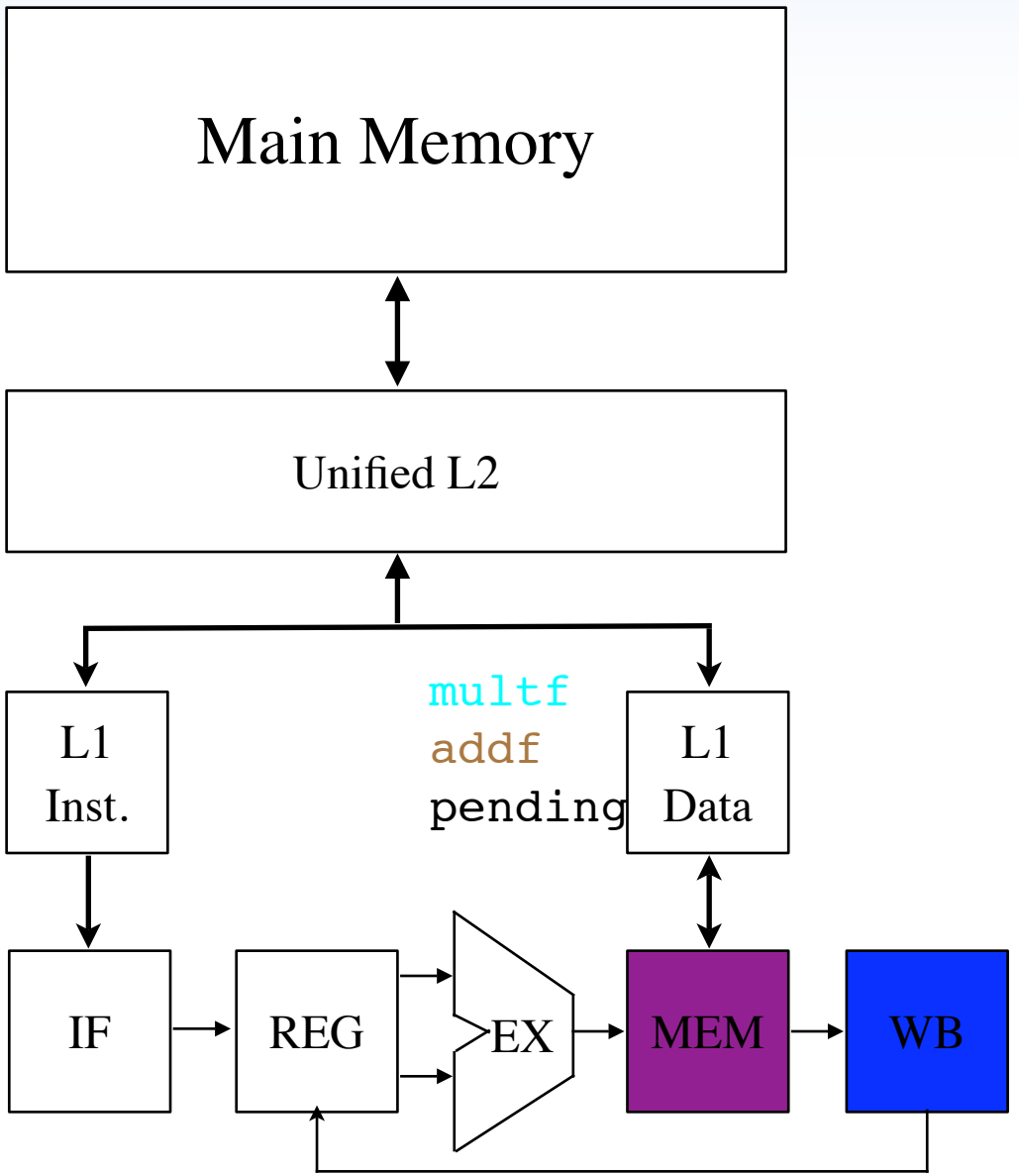
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



The second load retires...

Cycle 235

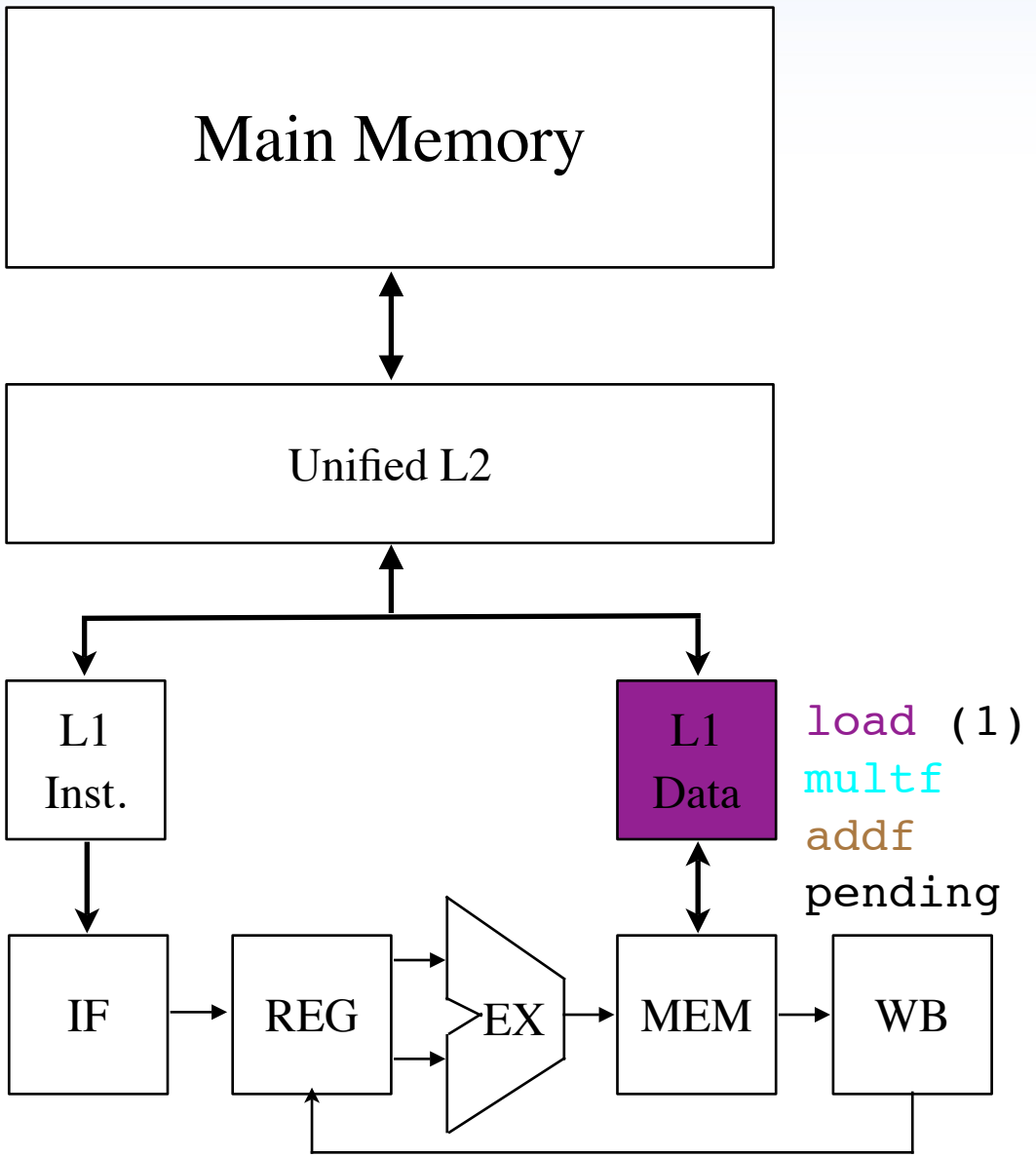
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



Now we issue the data dependent load

Cycle 236

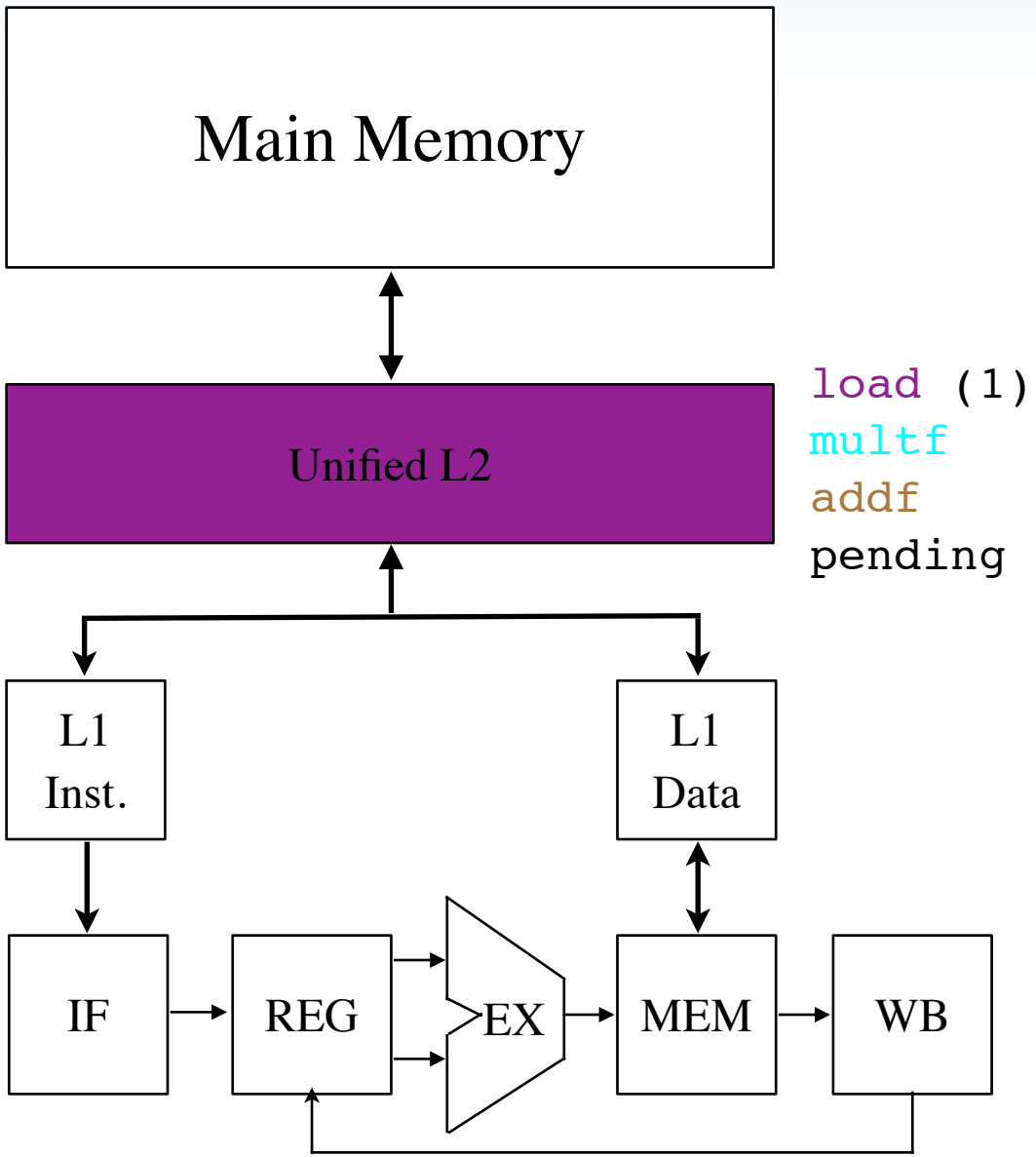
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



And wait...

Cycle 239

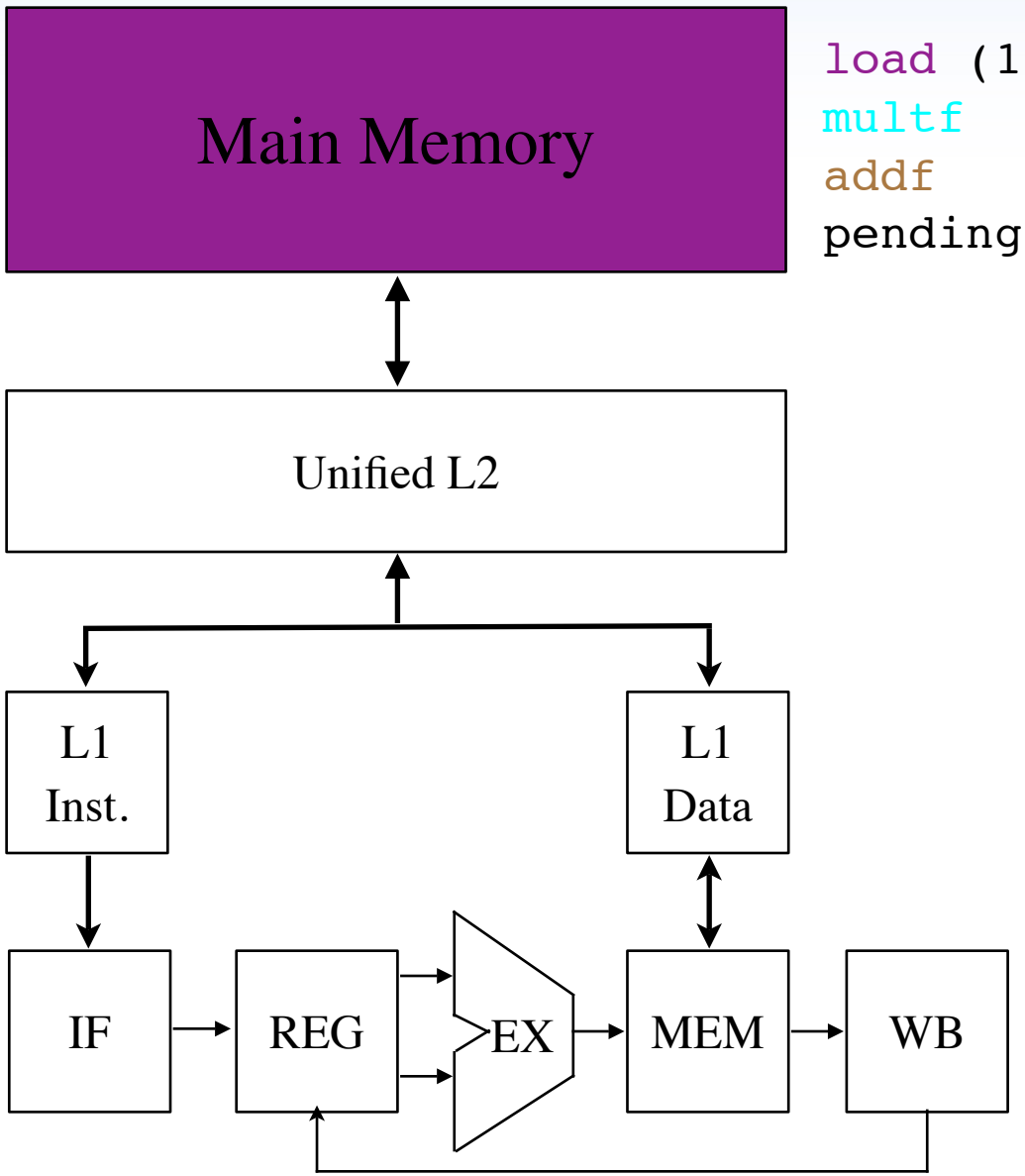
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



And wait... and wait...

Cycle 258

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```

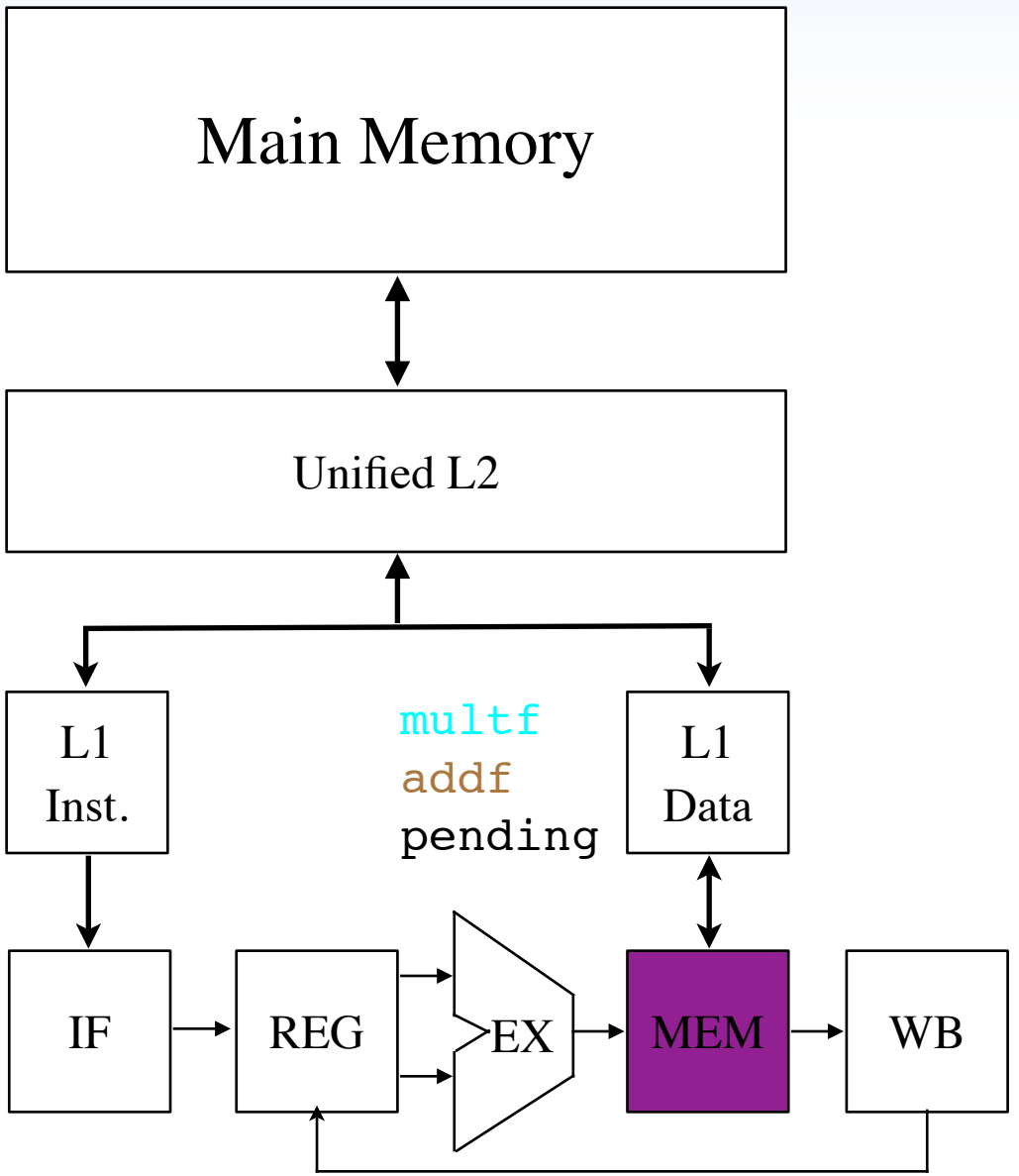



```
load (1)
multf
addf
pending
```

And wait... and wait... and wait...

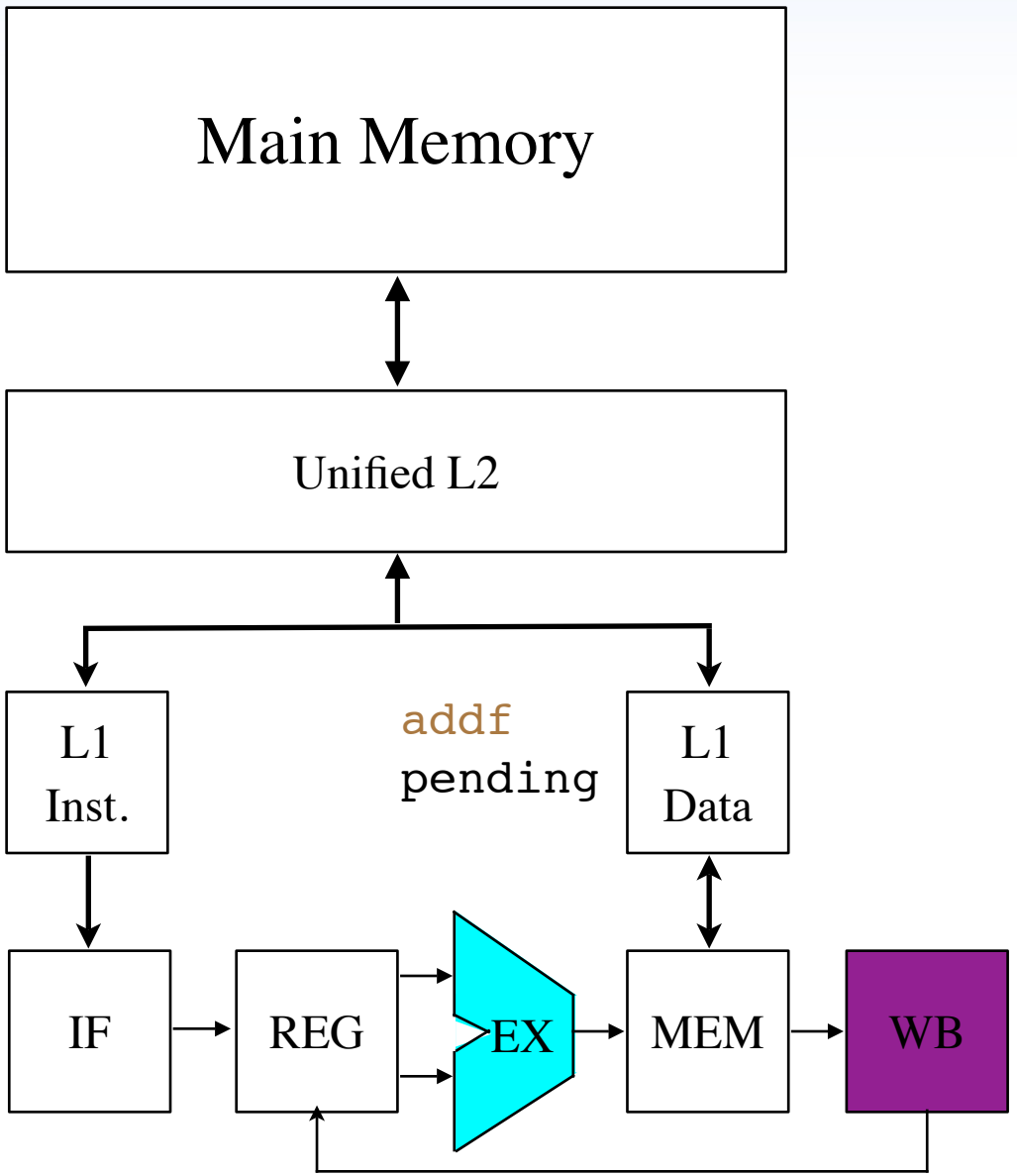
Cycle 457

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



Cycle 458

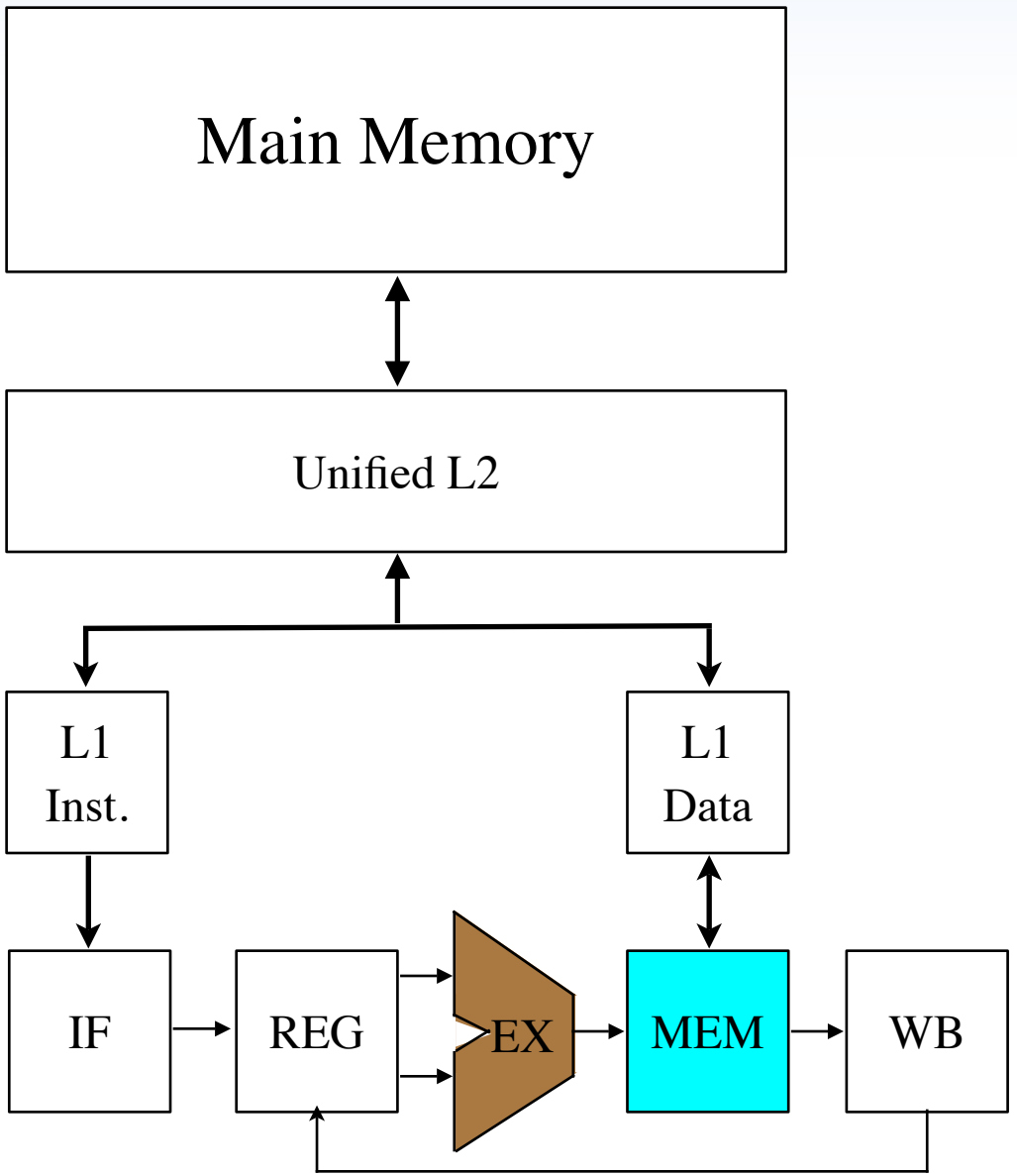
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



The multif is ~4 cycles

Cycle 461

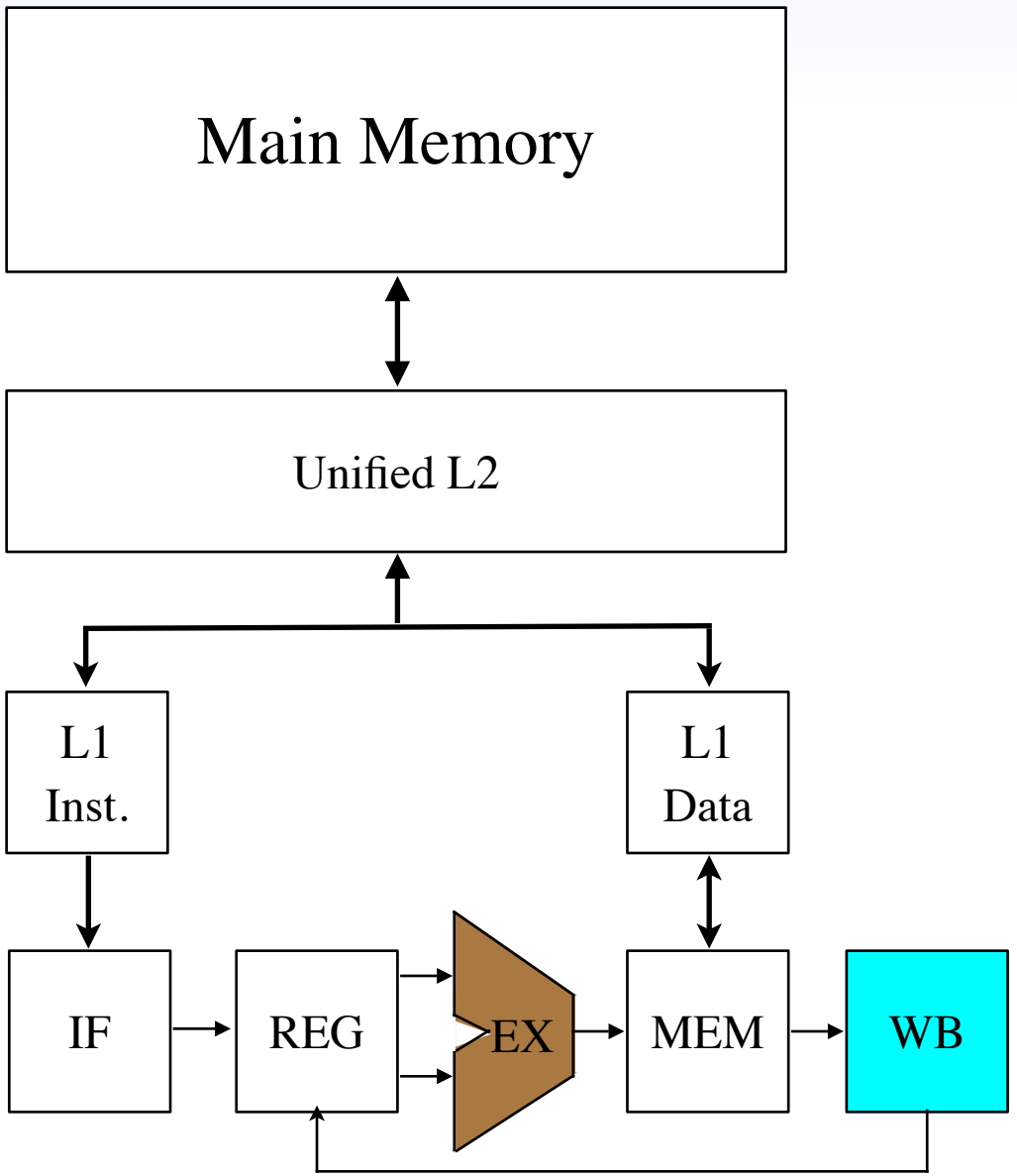
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



The addf is ~2 cycles

Cycle 462

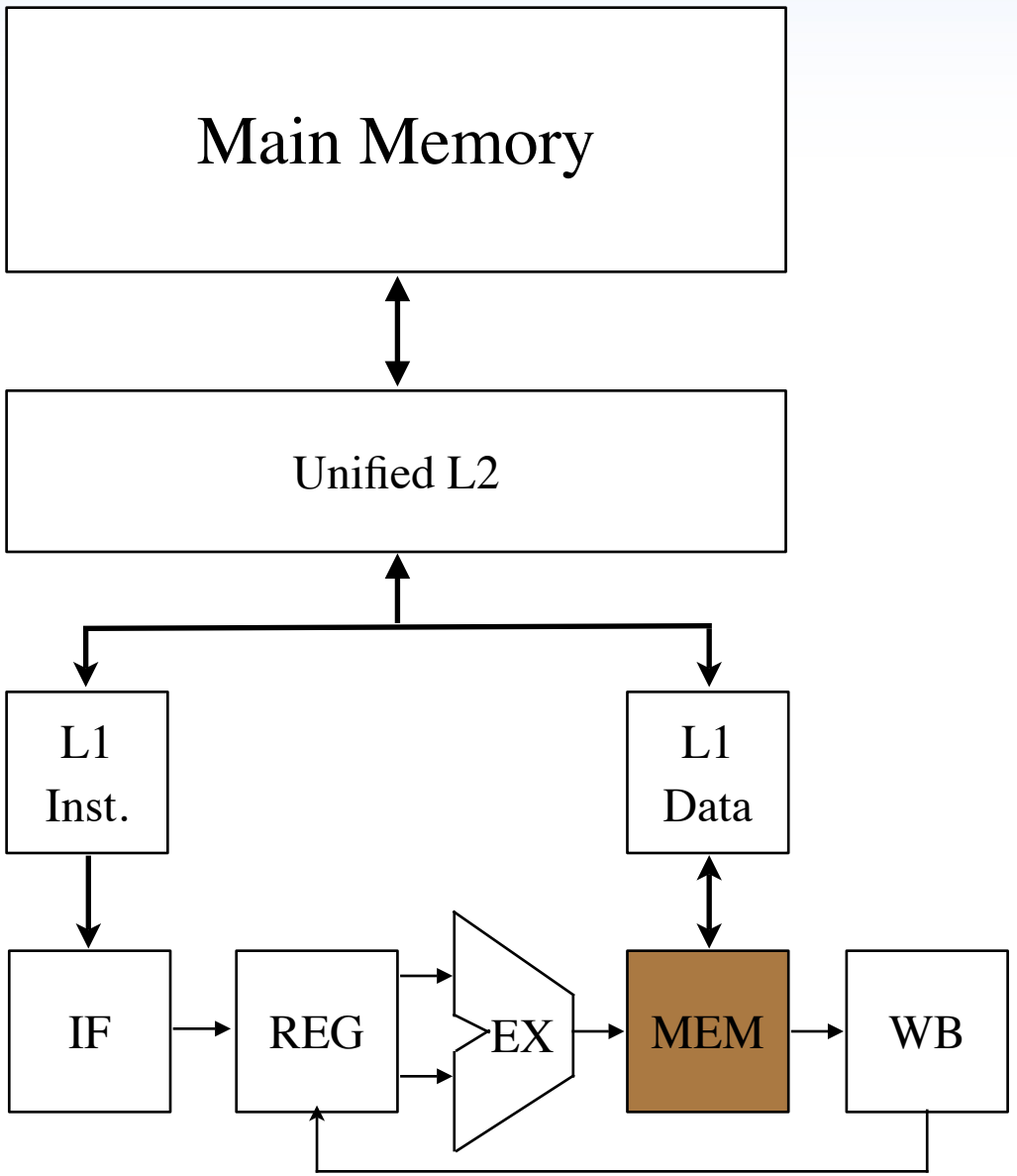
```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



The addf is ~2 cycles

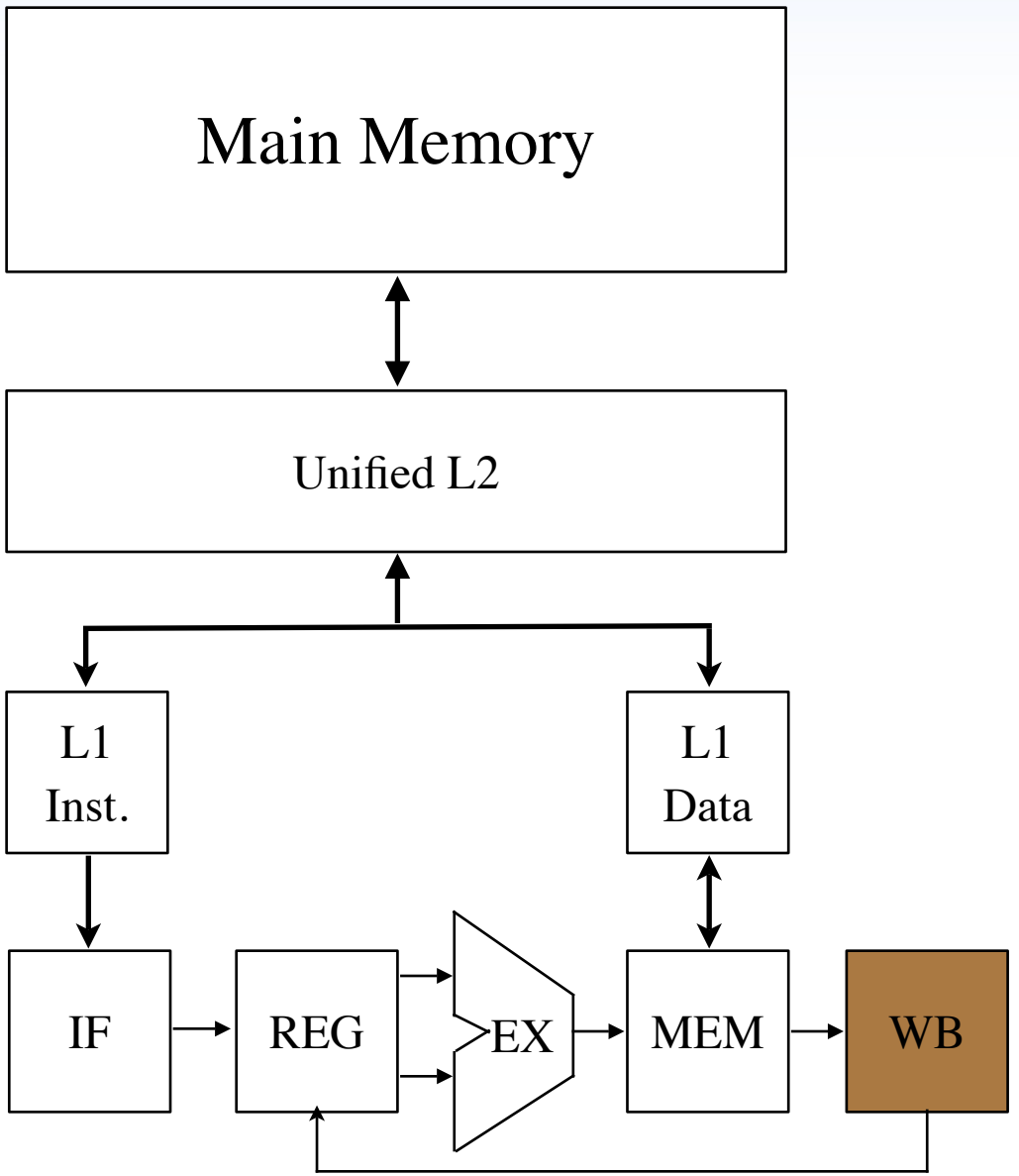
Cycle 463

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



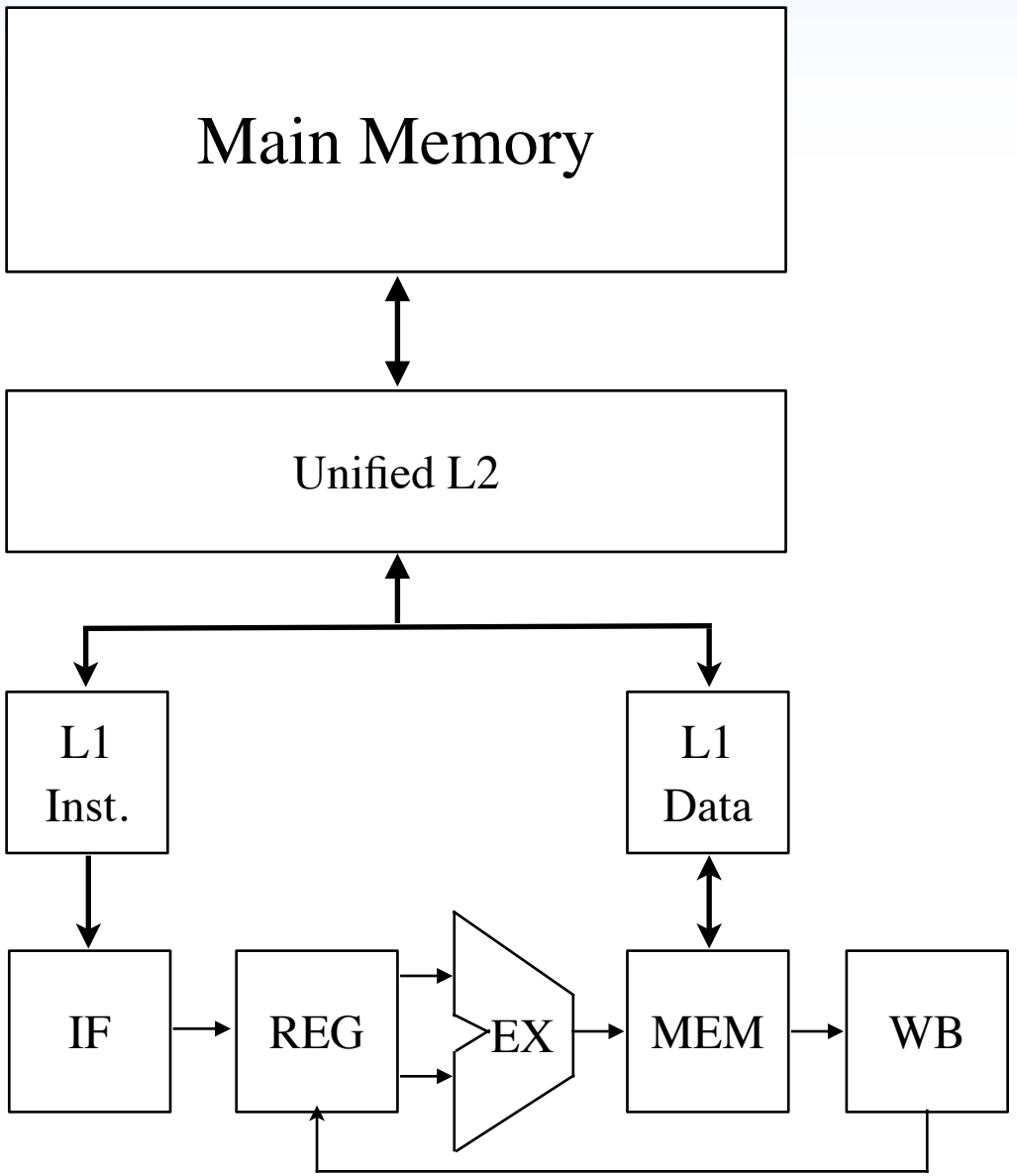
Cycle 464

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```



Cycle 465

```
; compute j*4 for pointer math
slli $j_tmp, $j, 2 ; j*4

; $t0 <= A_values[j]
add $a0, $A_values, $j_tmp
load $t0, 0($a0)

; $t1 <= A_indices[j]
add $a1, $A_indices, $j_tmp
load $t1, 0($a1)
slli $t1, $t1, 2 ; $t1*4

; t2 <= x[$t1]
add $a2, $x, $t1
load $t2, 0($a2)

; t3 <= $t0 * $t2
mulf $t3, $t0, $t2

; add to sum
addf $sum, $sum, $t3
```

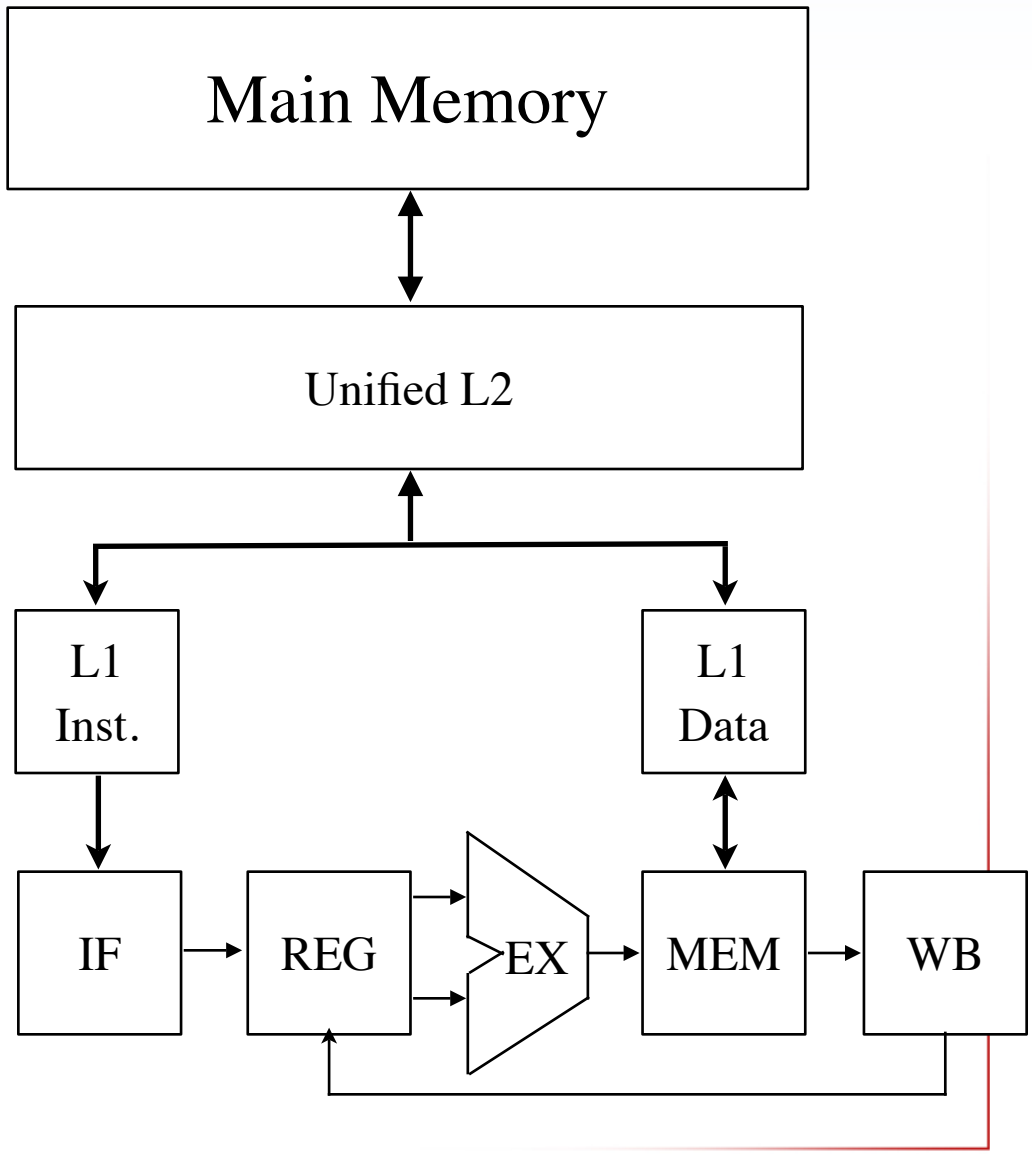


DONE!

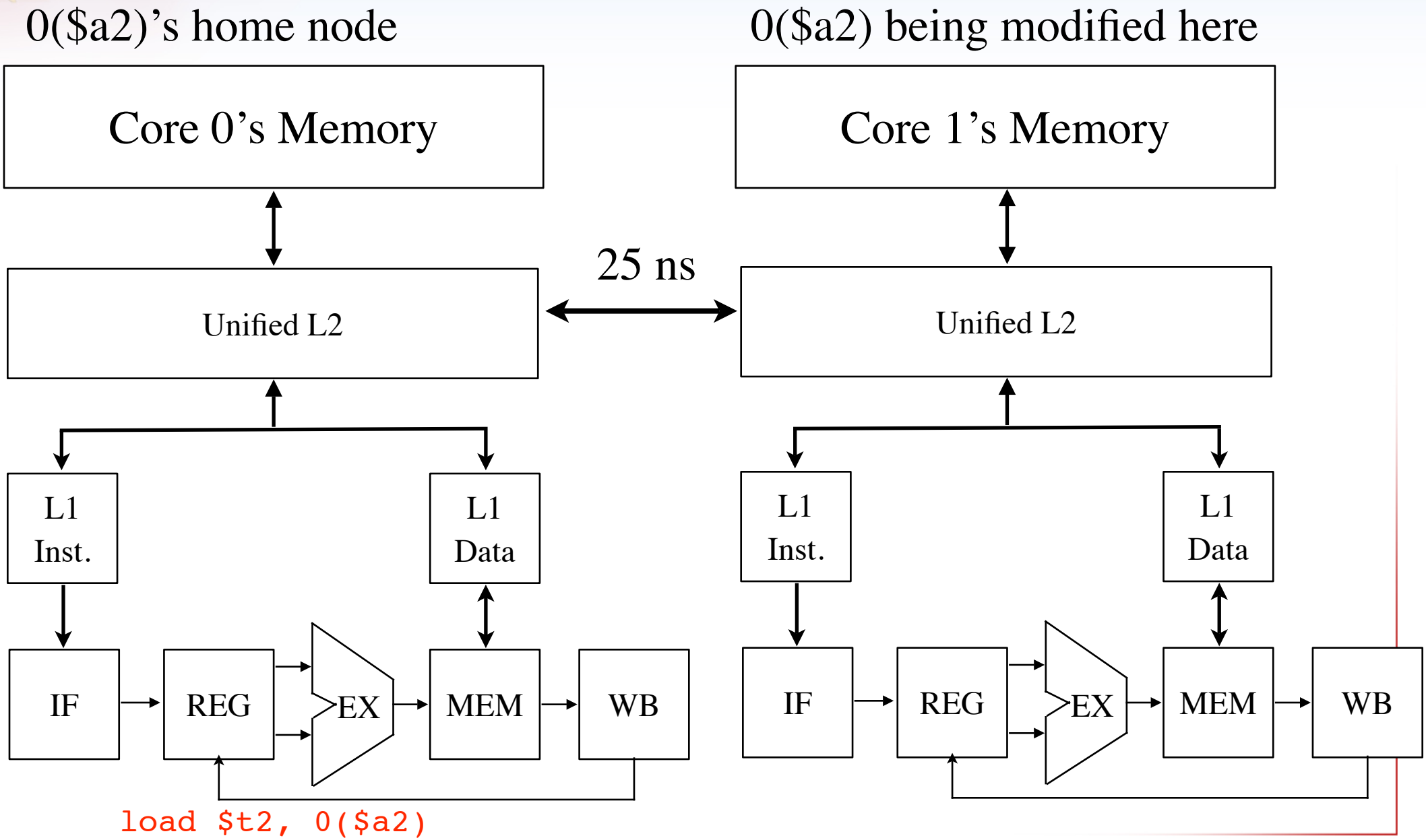
Observations

- Completing 10 instructions took 465 cycles!
  - Nearly 1/4 microseconds of real time at 2 GHz
- The FP multiplication only took 2ns
- Yes, I can un-role the loop and pipeline multiple iterations
  - However, only so many loads can be outstanding at a time
- I haven't talked about multicore/cache coherency yet

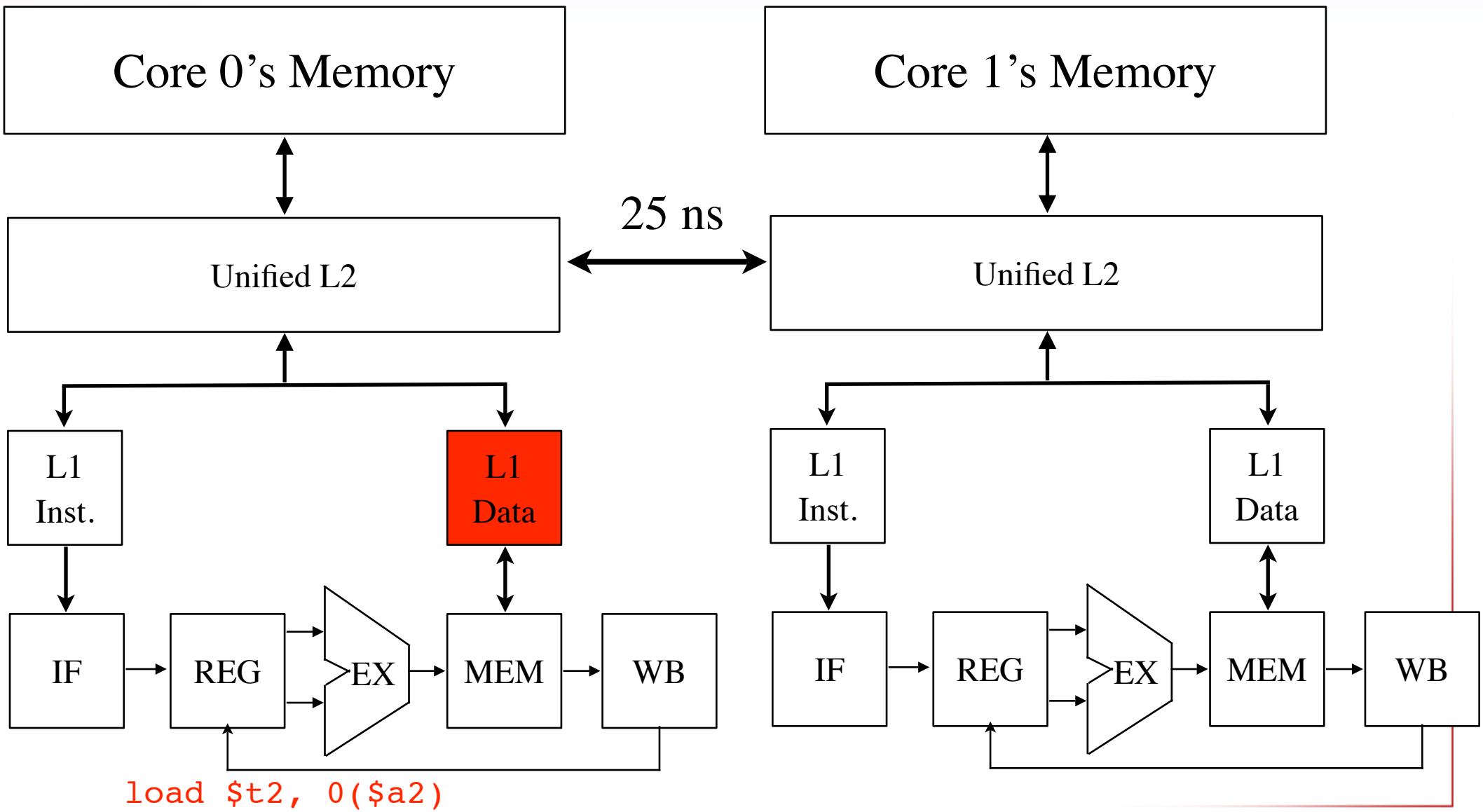
What about a cache coherent model?



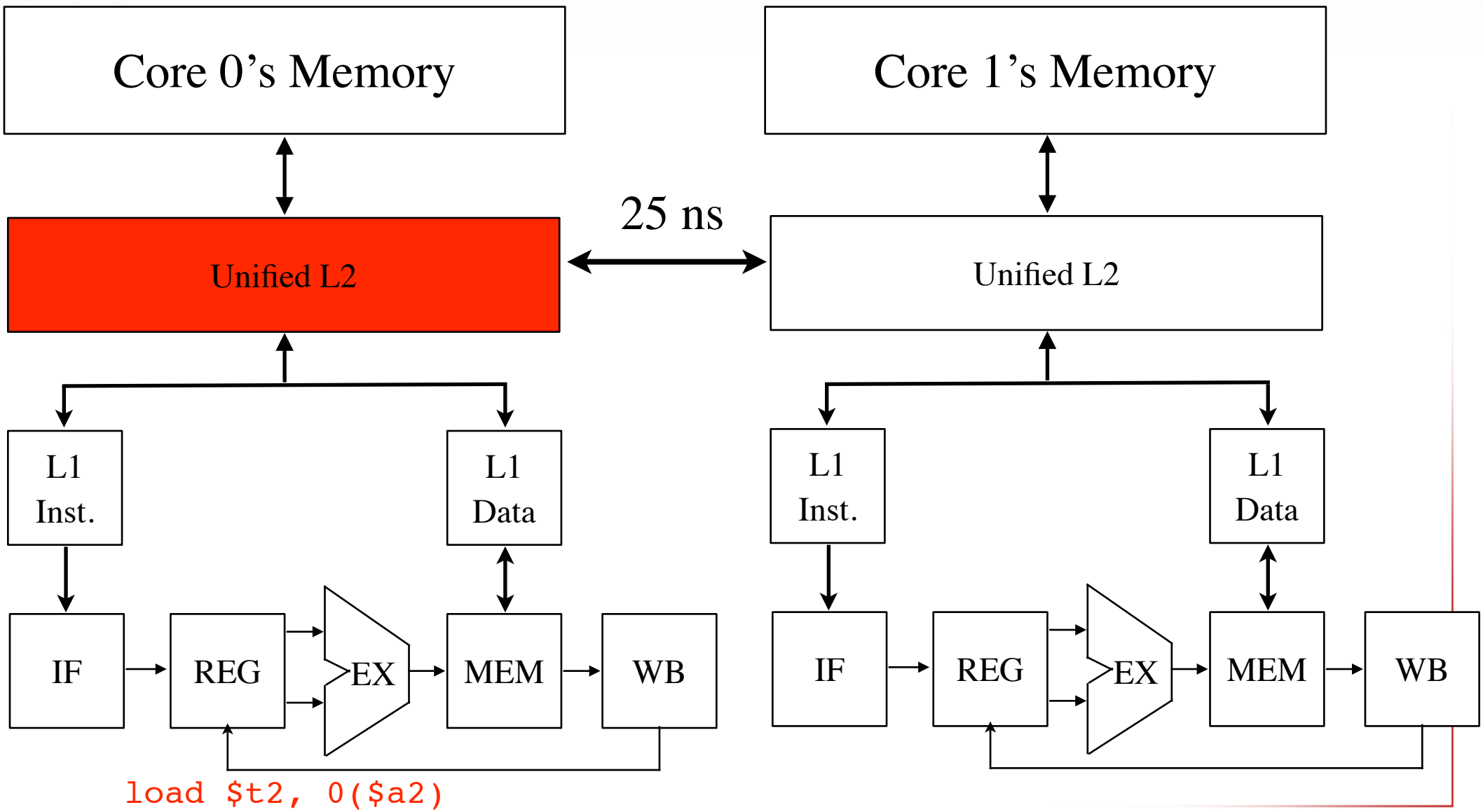
What about a cache coherent model?



Cycle 5

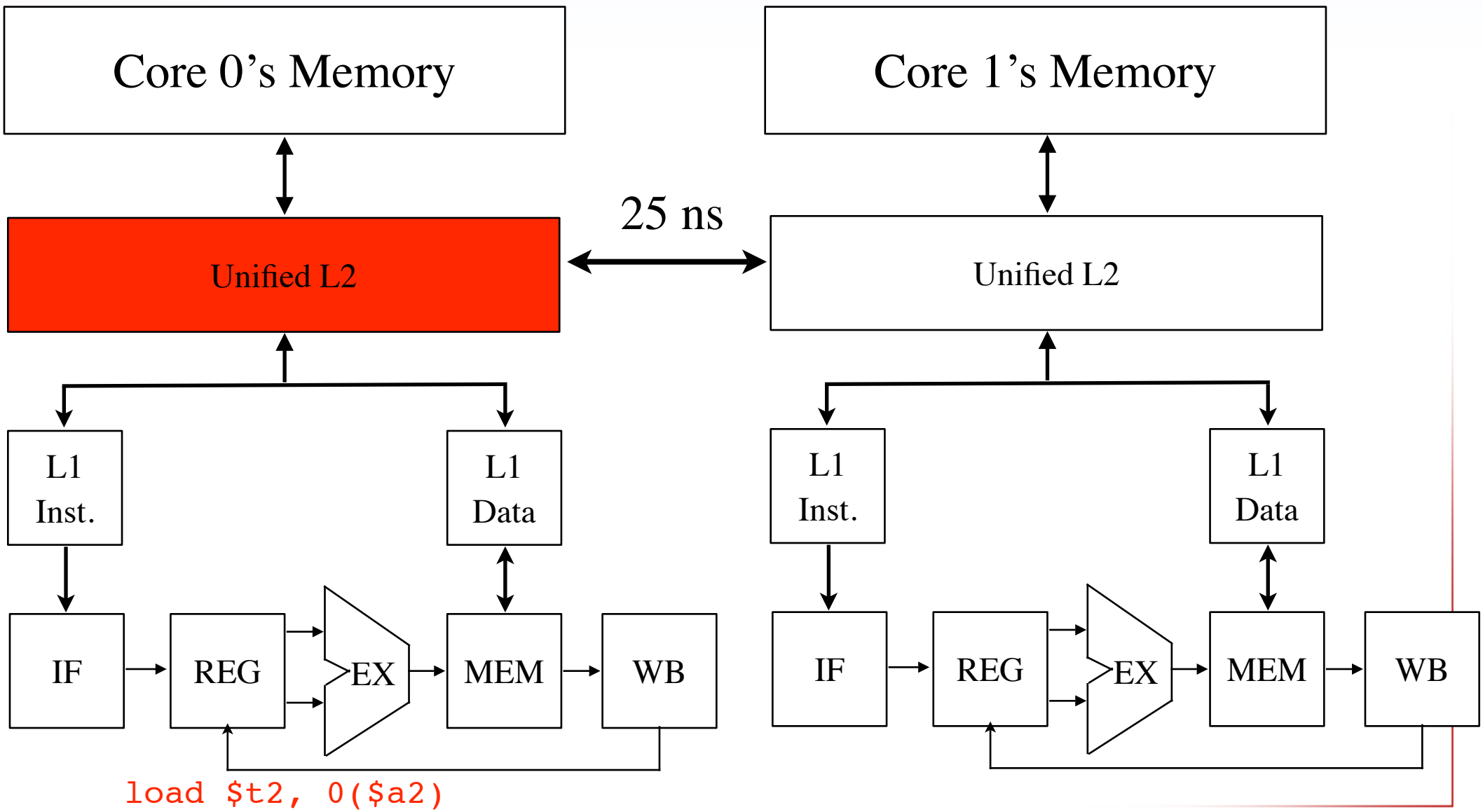

Core 0's Memory
Core 1's Memory
25 ns
Unified L2
Unified L2
L1
Inst.
L1
Data
L1
Inst.
L1
Data
IF
REG
EX
MEM
WB
IF
REG
EX
MEM
WB
load $t2, 0($a2)

Cycle 6


Core 0's Memory
Core 1's Memory
25 ns
Unified L2
Unified L2
L1 Inst.
L1 Data
L1 Inst.
L1 Data
IF
REG
EX
MEM
WB
IF
REG
EX
MEM
WB
load $t2, 0($a2)

Cycle 25

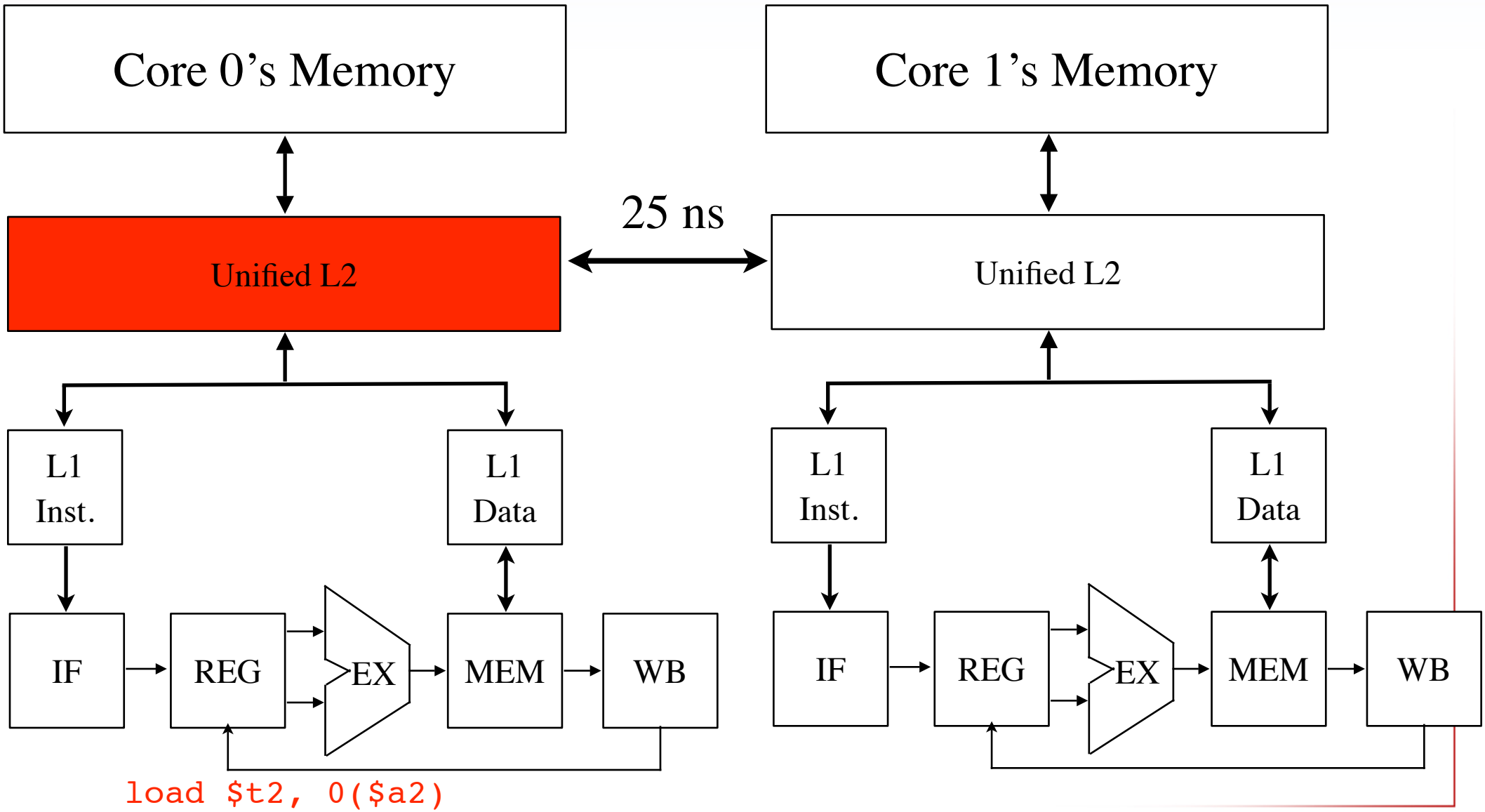
0($a2) is owned by Core 0 but ... Core 1 is modifying it



Cycle 25 (continued)

0($a2) is owned by Core 0 but ... Core 1 is modifying it

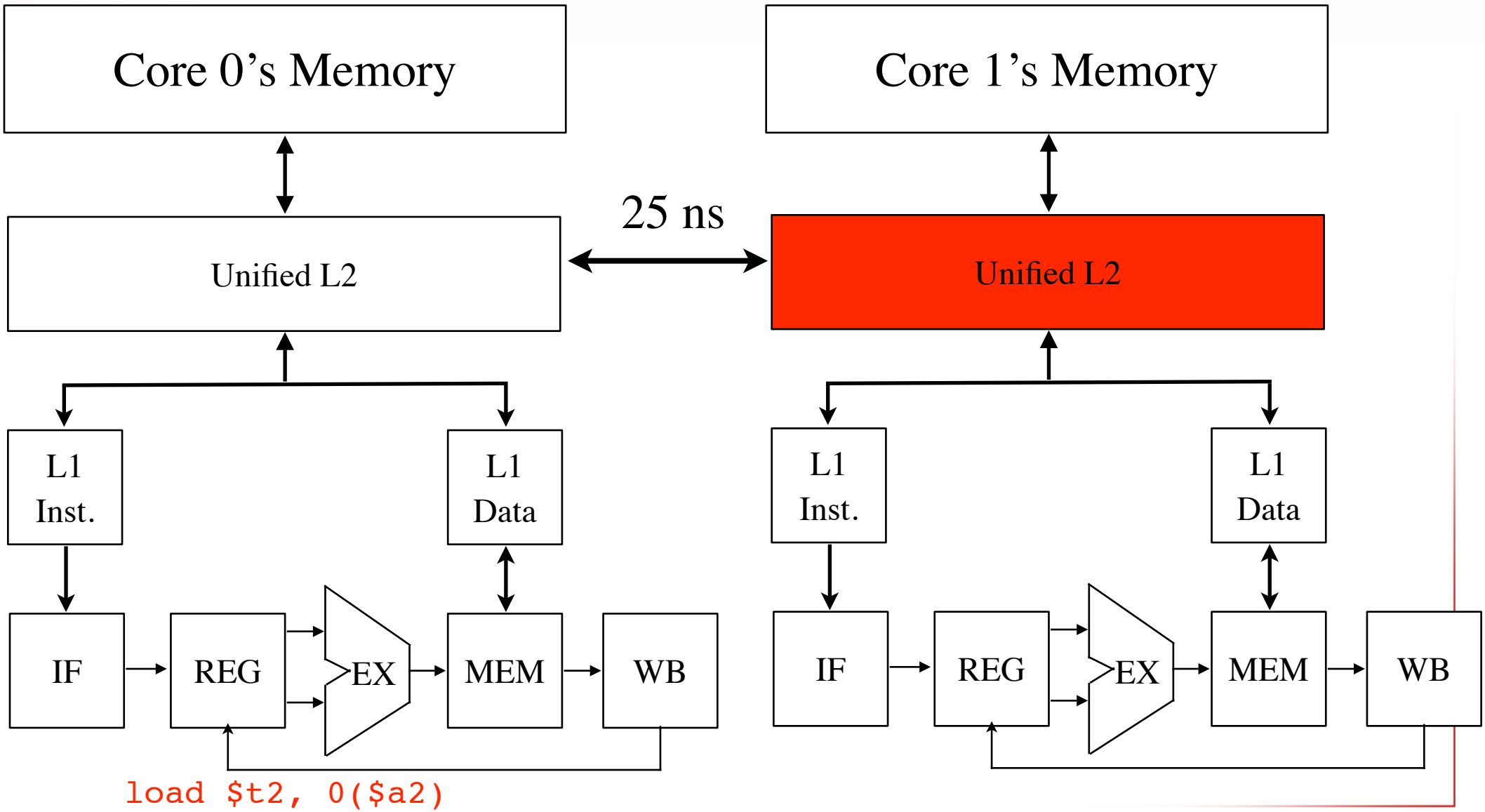
Send Invalidate Request



Cycle 75

0($a2) is owned by Core 0 but ... Core 1 is modifying it

20 clock cycle L2 access



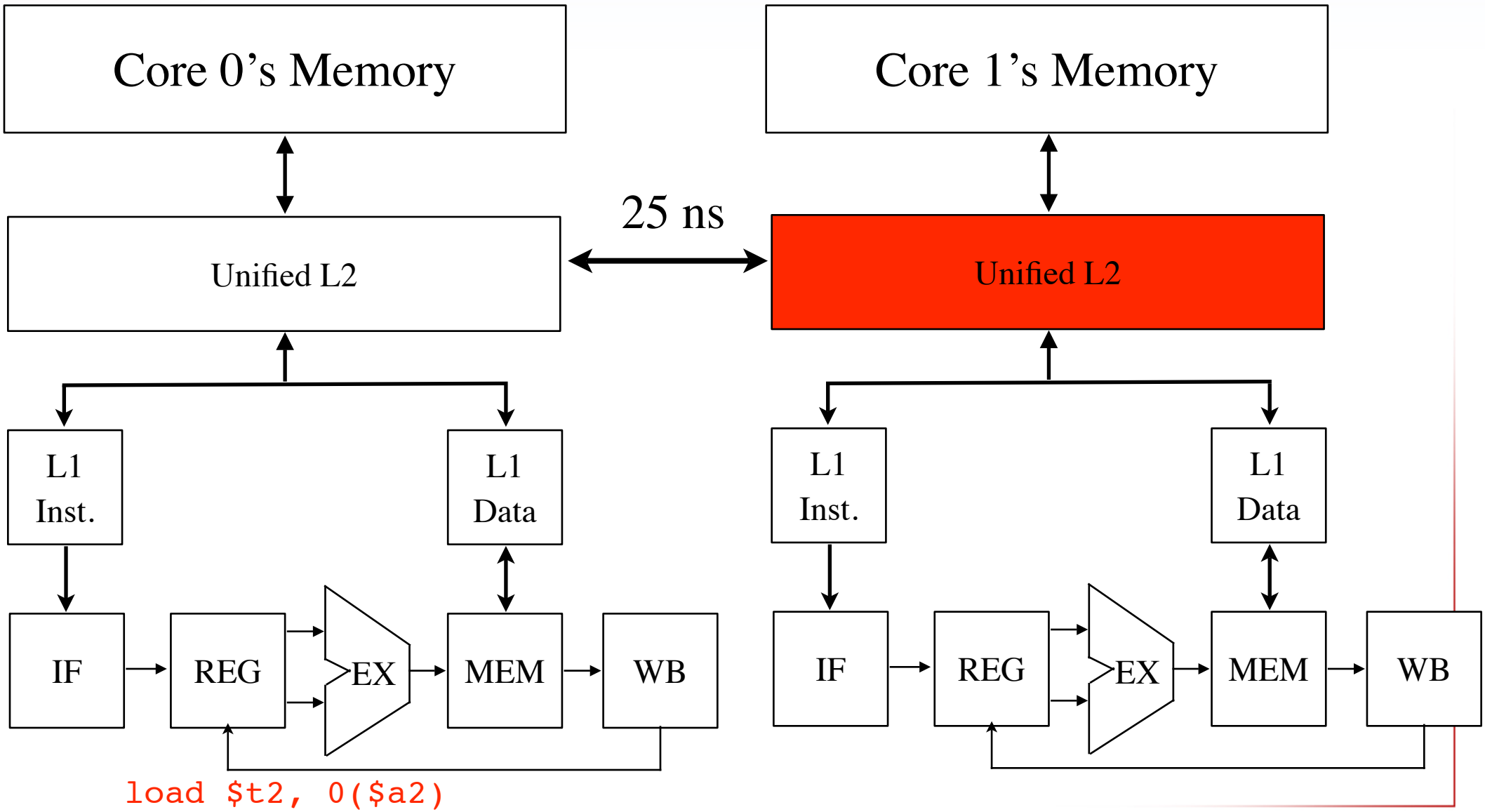
National Nuclear Security Administration

Sandia National Laboratories

Cycle 95

0($a2) is owned by Core 0 but ... Core 1 is modifying it

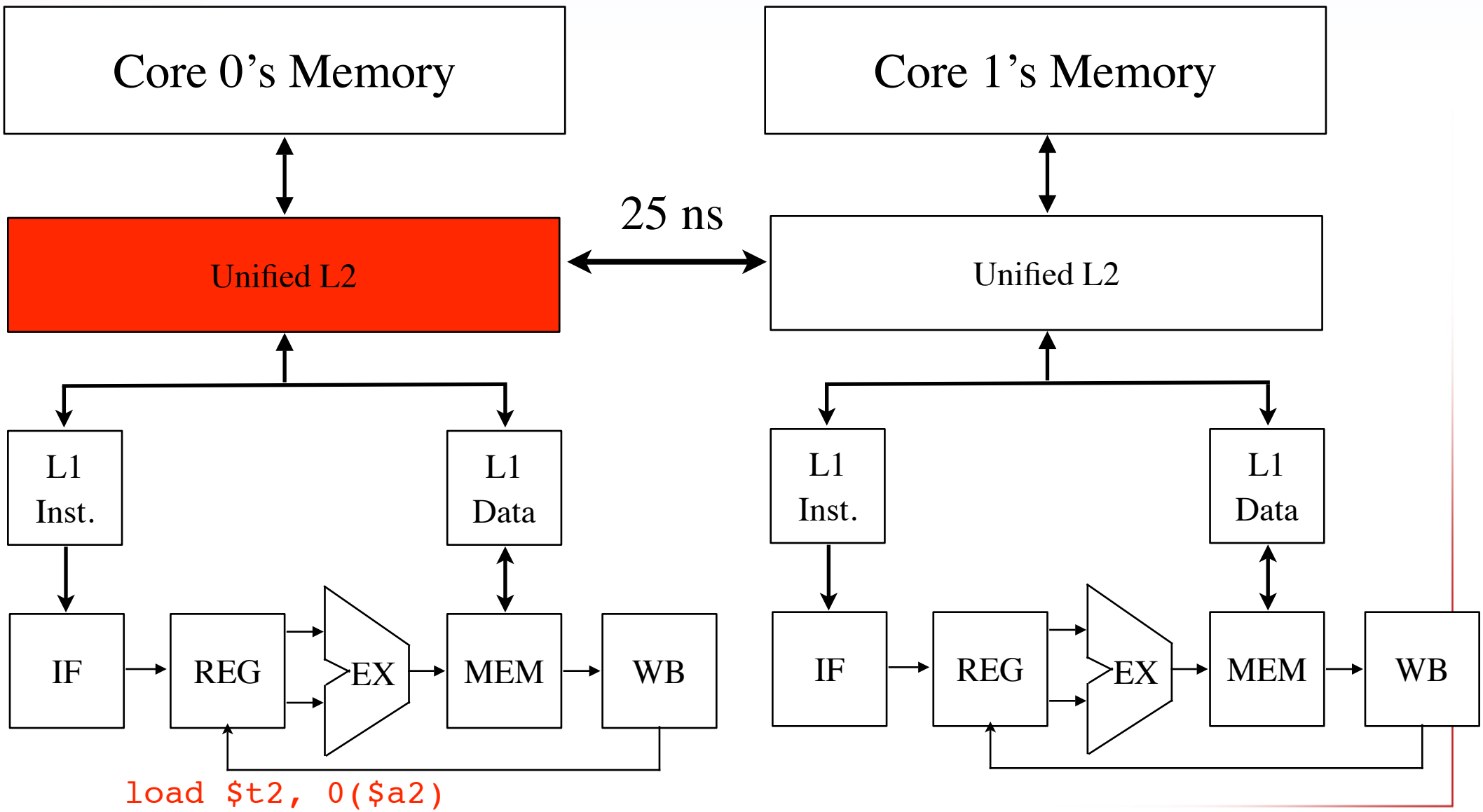
Flush the modified line back to the owner



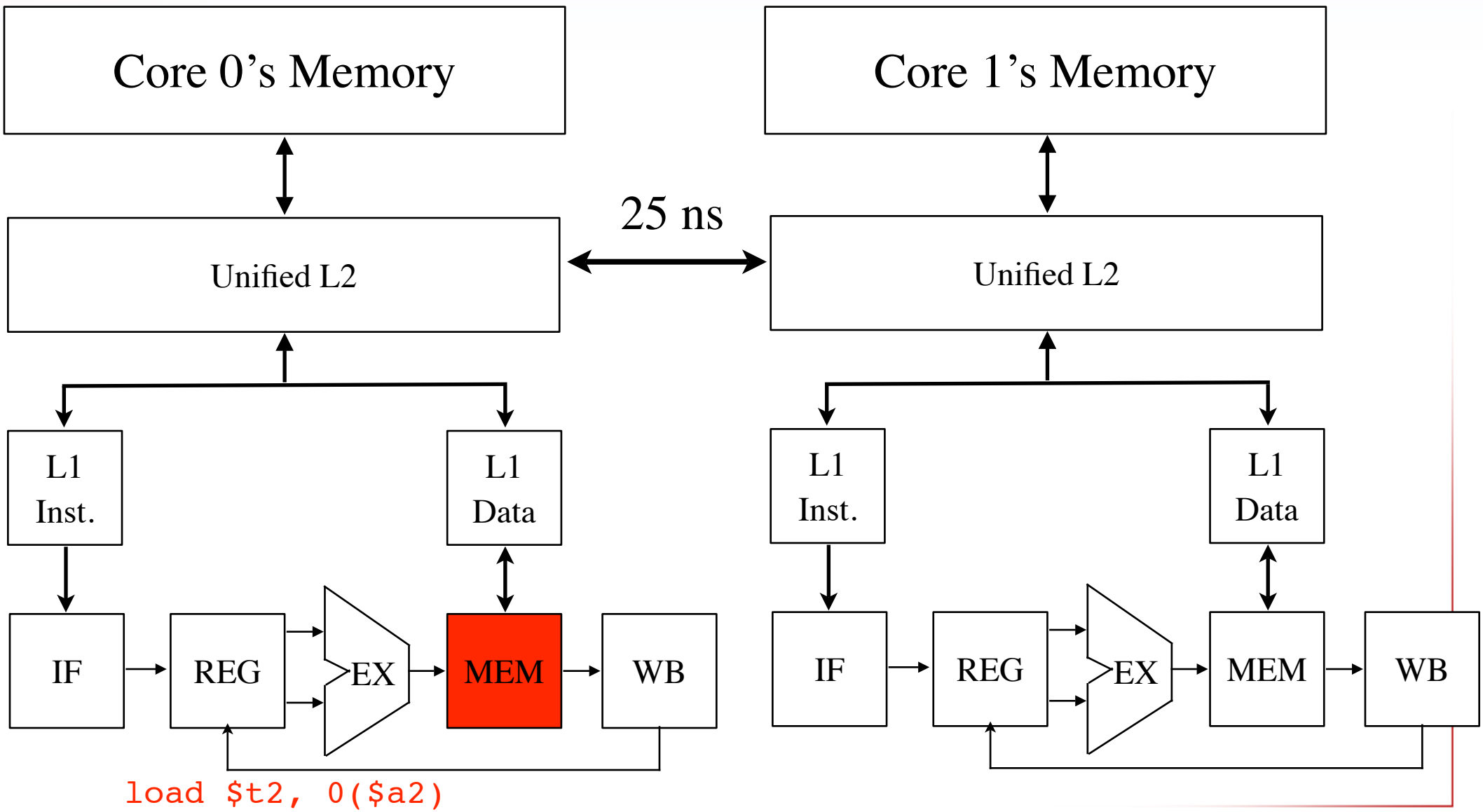
Cycle 145

0($a2) is owned by Core 0 but ... Core 1 is modifying it

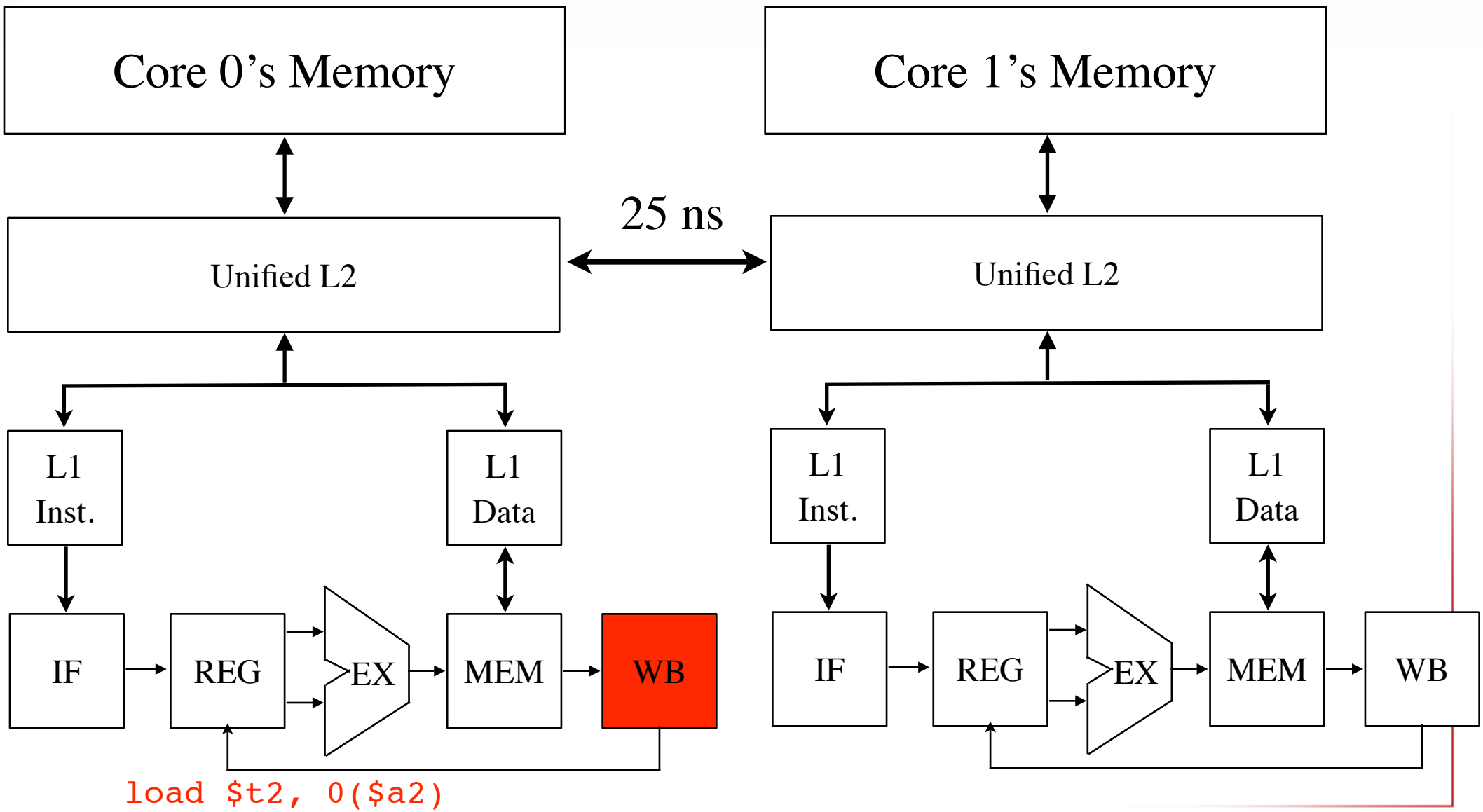
Flush the modified line back to the owner



Cycle 146


Core 0's Memory
Core 1's Memory
25 ns
Unified L2
Unified L2
L1 Inst.
L1 Data
L1 Inst.
L1 Data
IF
REG
EX
MEM
WB
IF
REG
EX
MEM
WB
load $t2, 0($a2)

Cycle 147


Core 0's Memory
Core 1's Memory
25 ns
Unified L2
Unified L2
L1 Inst.
L1 Data
L1 Inst.
L1 Data
IF
REG
EX
MEM
WB
IF
REG
EX
MEM
WB
load $t2, 0($a2)

Observations

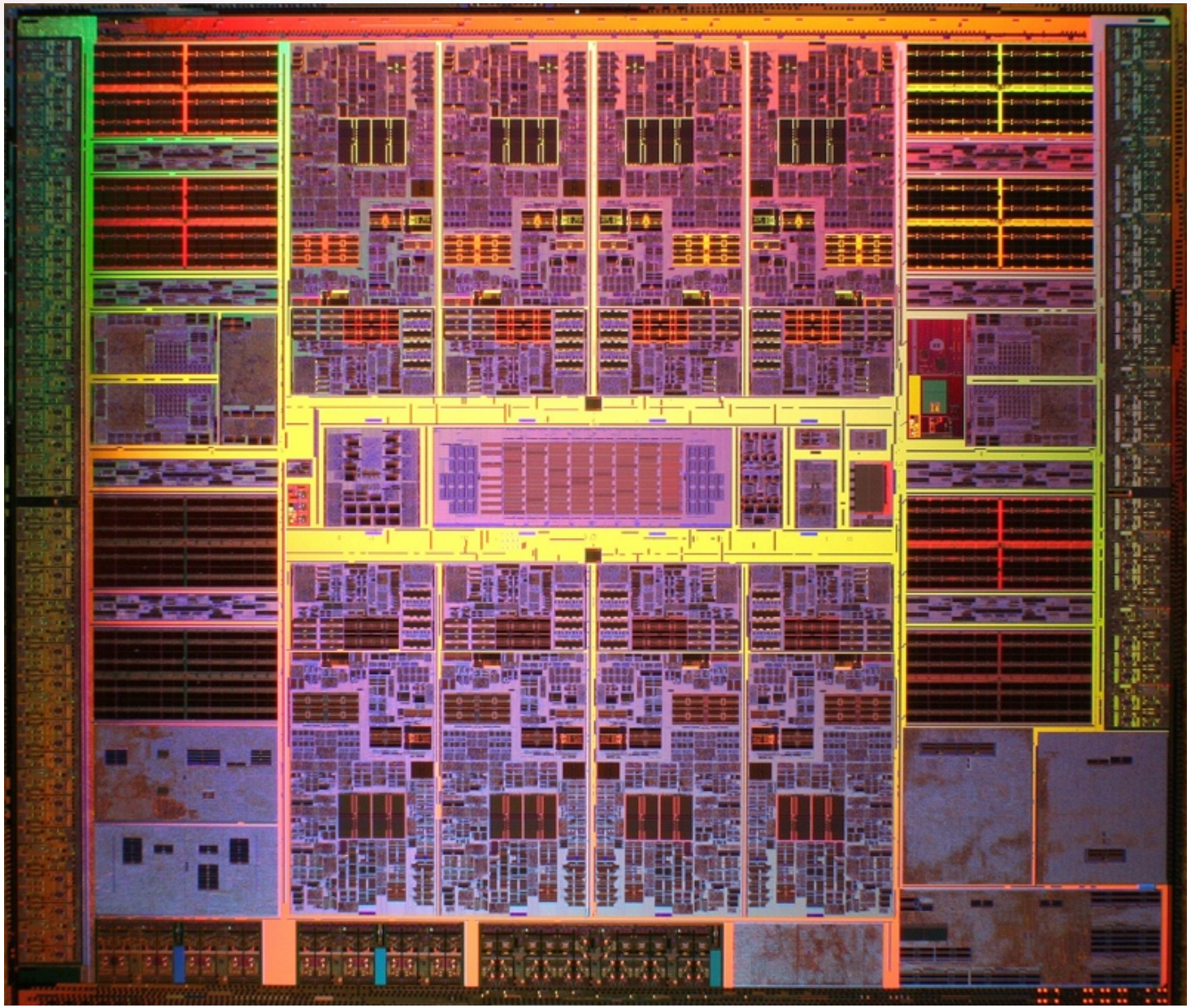
- The “memory” latency in this example is 140 clock cycles
  - “Closer” than DRAM
  - Optimistic because it was serviced by L2 cache
- Latency is “additive”
  - Each level of hierarchy serves “hits” faster but misses are slower
  - If I do a “read” owned by another node
    - 10ns L2 cache miss
    - 25ns request over the on-chip network
    - 10ns L2 cache miss by the owner
    - 100ns memory read by the owner
    - 25ns response over the on-chip network
    - TOTAL: 170ns!
- On-chip network latency will increase as the number of cores increase (not linearly, but...)

What do I do about it?

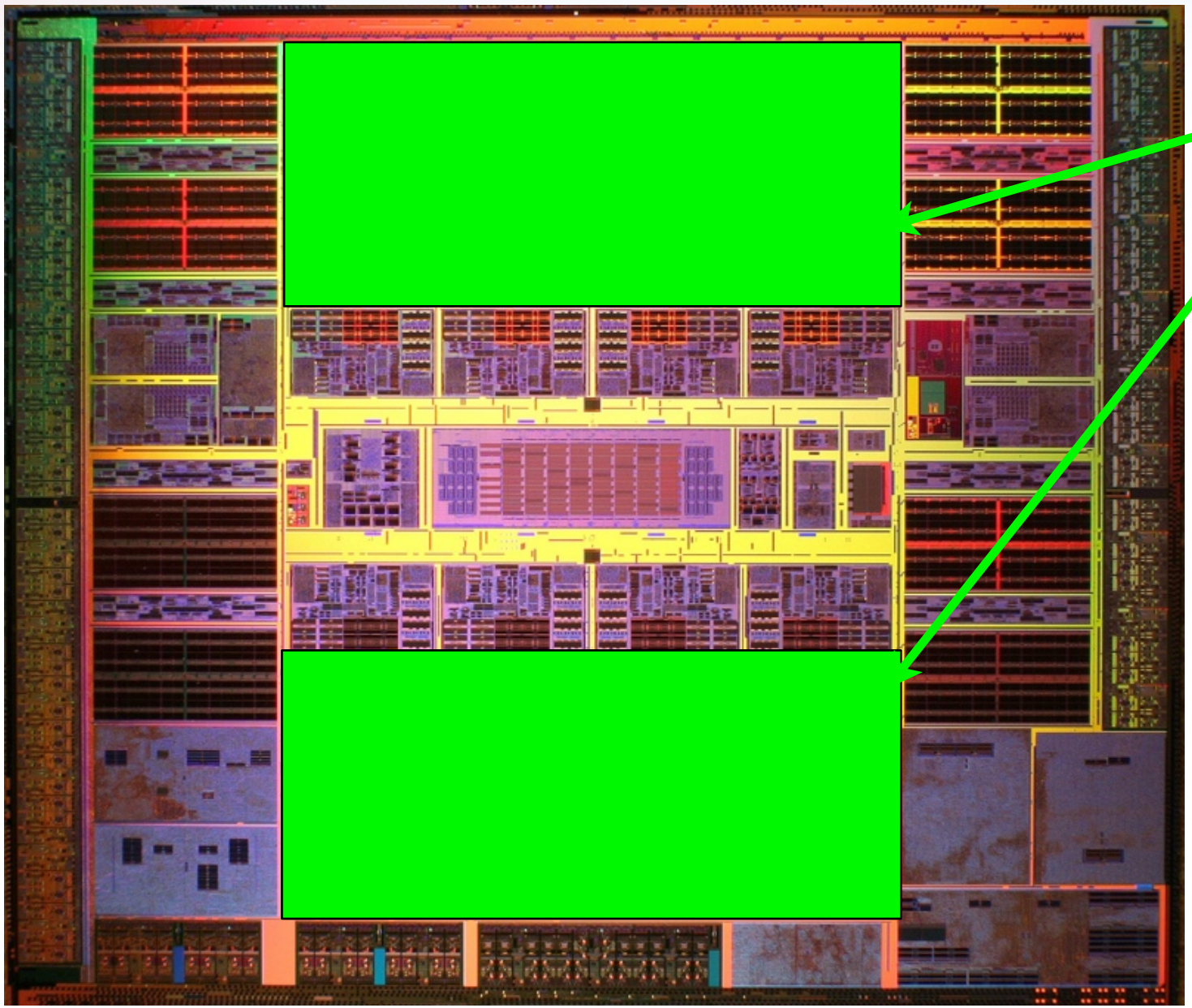
Little's Law: $\frac{\text{Concurrency}}{\text{Latency}} = \text{Throughput}$

- **Concurrency:**
  - More cores may make the memory bus busier
  - Vendors are putting fewer channels/memory controllers per core in each generation
  - Decreasing on a per-core basis!
- **Latency**
  - Relatively increasing or flat
- **How do you fill those cycles?**
  - Traditionally: Out-of-Order Execution
  - Today: More Threads

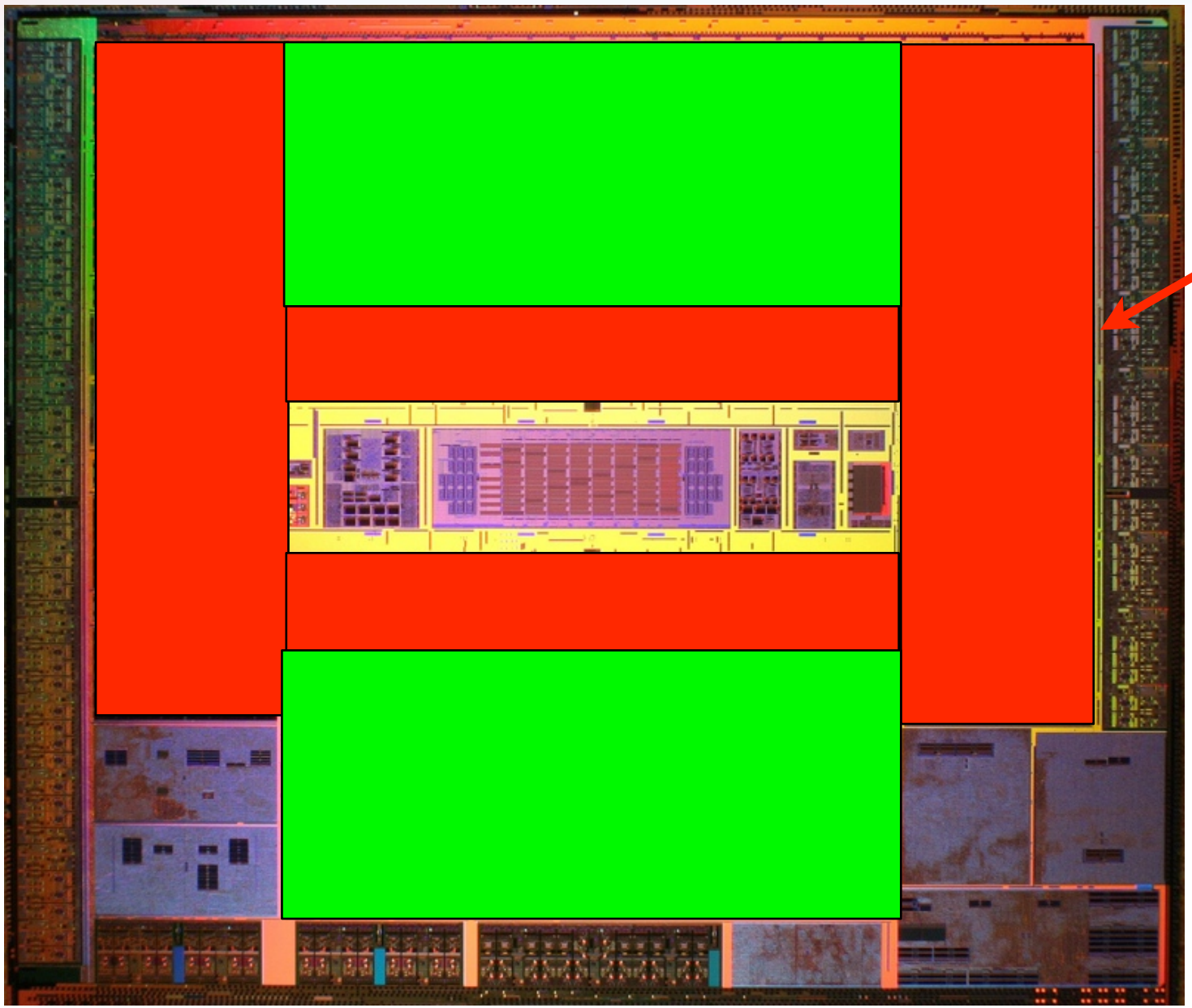
VLSI Designer's View of the World



VLSI Designer's View of the World



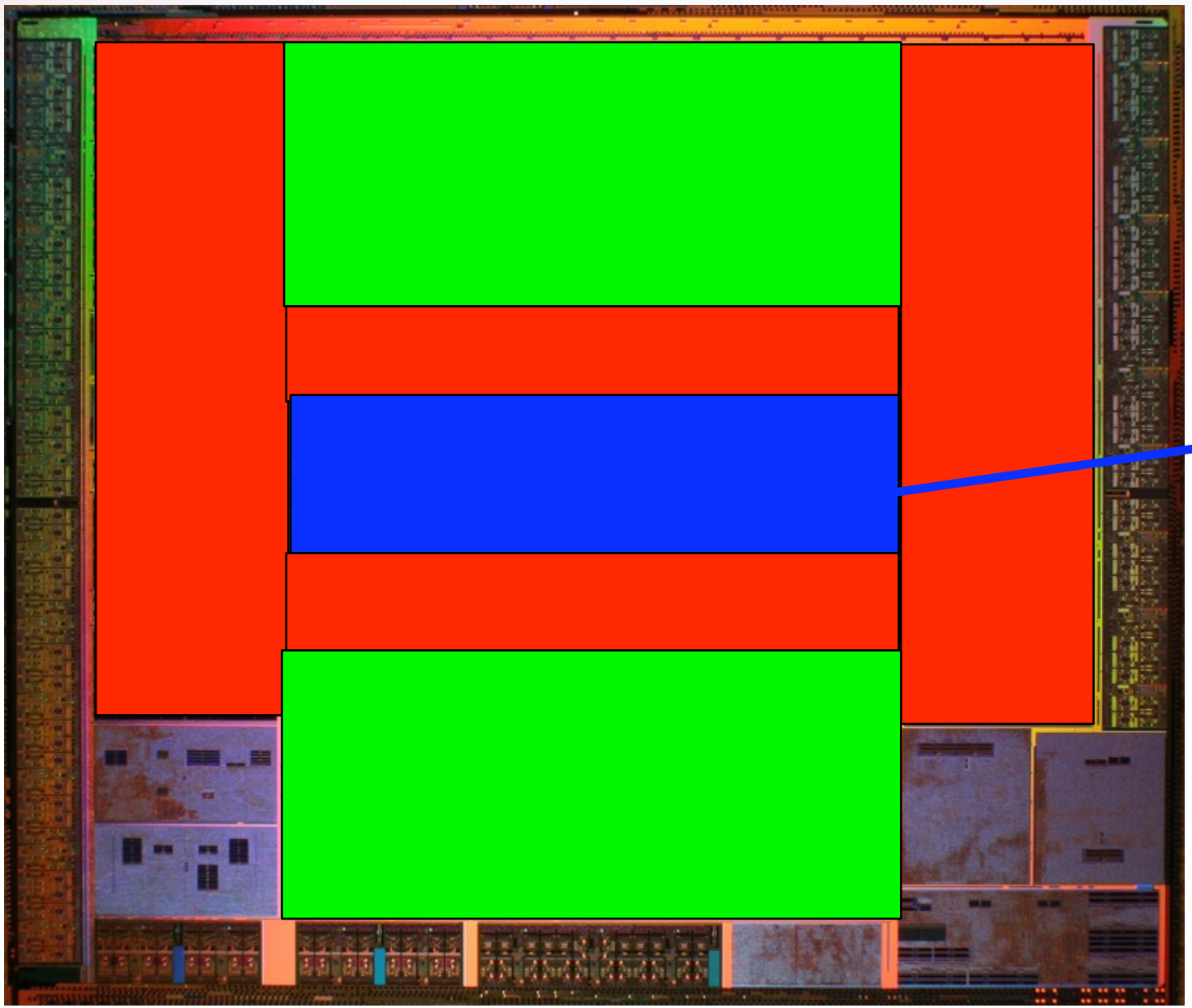
VLSI Designer's View of the World



Cores

Cache

VLSI Designer's View of the World

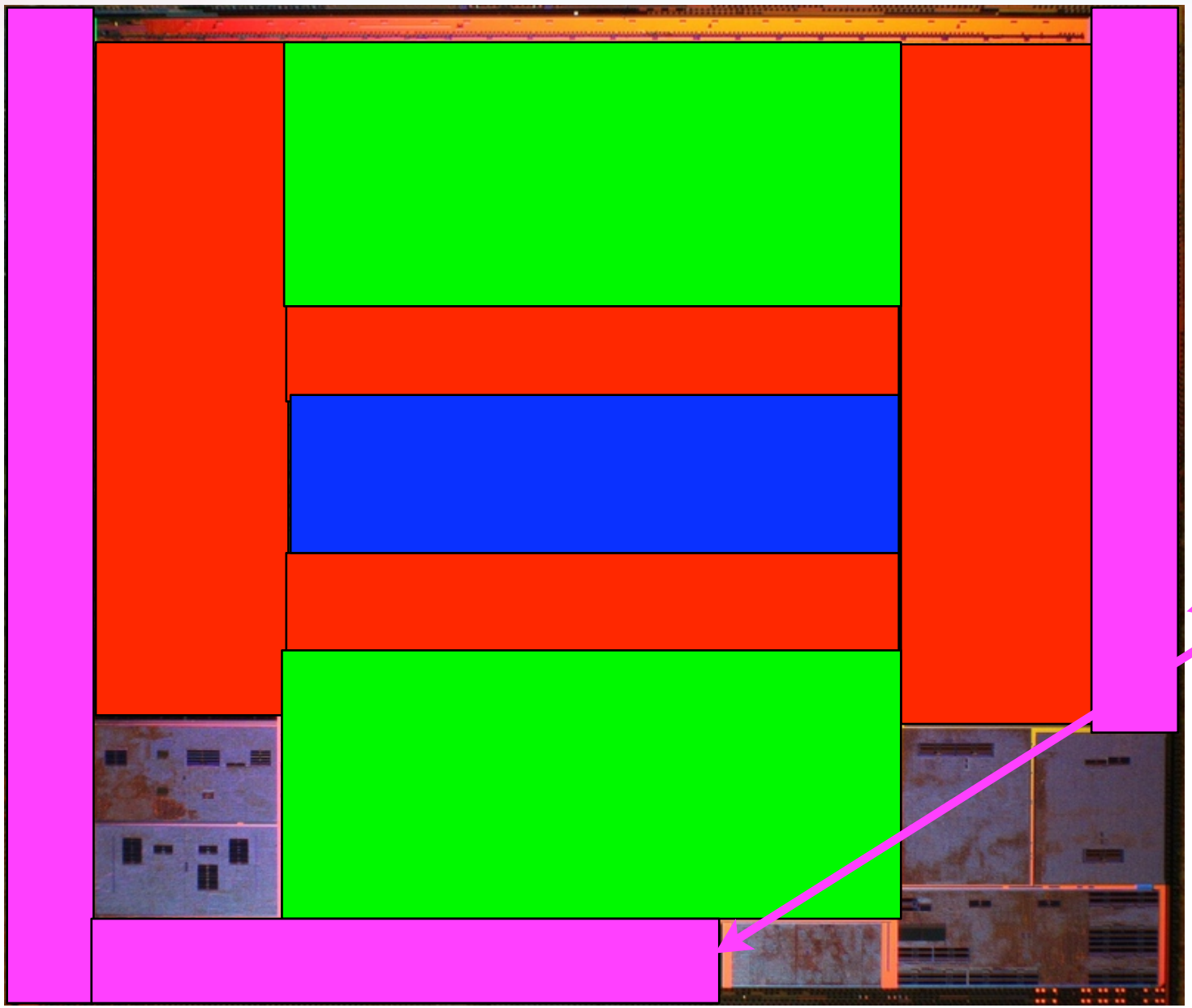


Cores

Cache

Core-to-Core Communication

VLSI Designer's View of the World



Cores

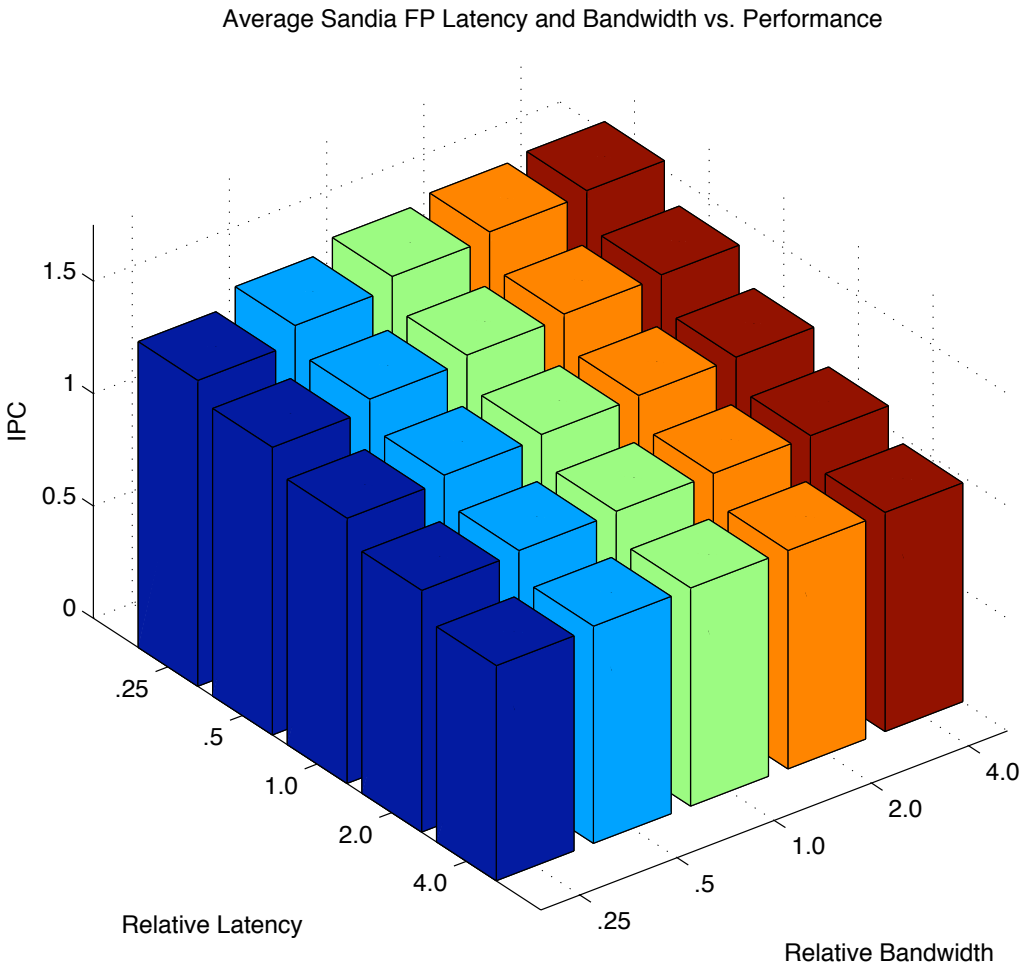
Cache

Core-to-Core Communication

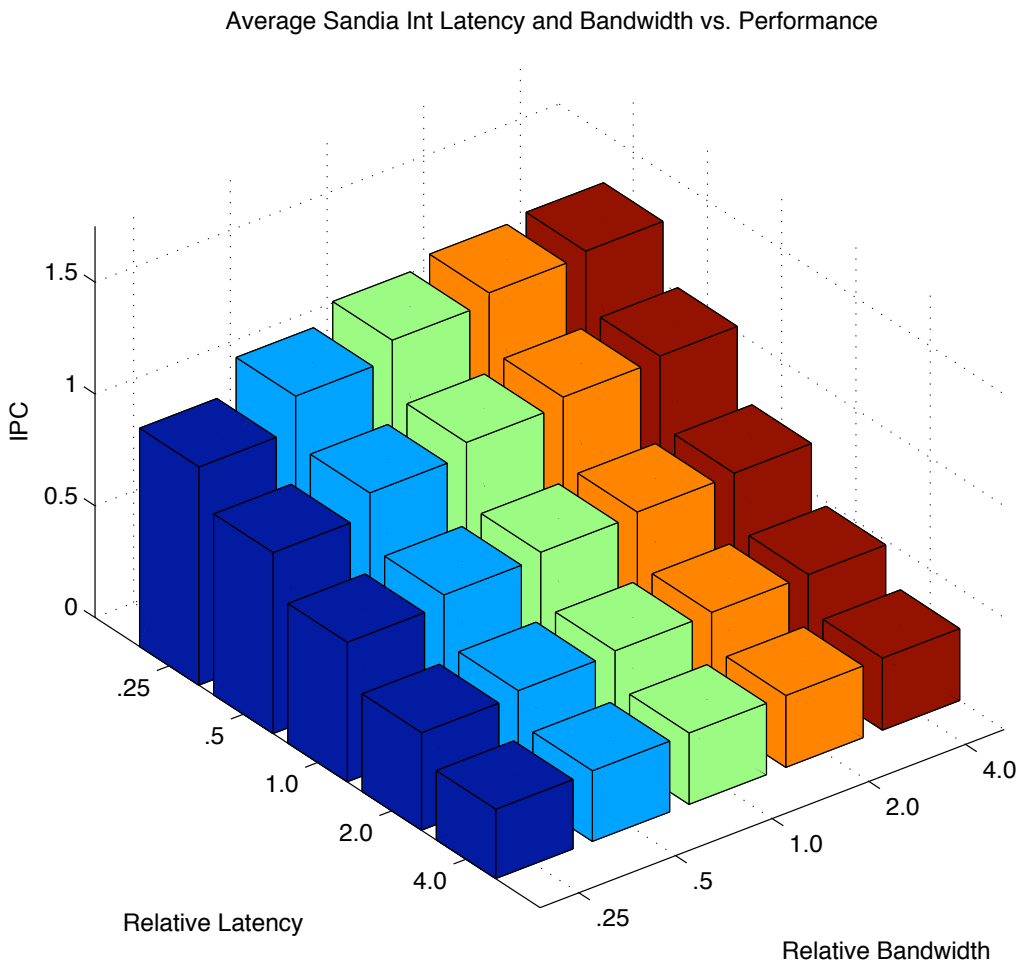
Off-chip I/O

Application Impact

Physics Applications



Informatics Applications



Informatics Apps ~3X more sensitive to latency than Physics

National Nuclear Security Administration

Sandia National Laboratories

Application Observations

- The commodity path has produced power-inefficient architectures
  - We observe IPCs of ~.3 at 2.5 GHz
  - With an IPC of 1, the same system could be clocked at 750 MHz
    - Doing so would decrease power requirements superlinearly
- Address generation is a bigger problem than FLOPS
- We can decrease latency or increase *effective* bandwidth
  - Many apps throw away 7/8ths of a cache line
  - Increase effective bandwidth by: Scatter/Gather to improve spatial locality
  - Decrease latency by: tighter integration, advanced packaging, etc.

What I didn't talk about

- Memory capacity/core
  - Currently declining, probably not a good thing
  - Fewer pins/core devoted to memory
  - Slow, bus interface (point-to-point serial solves the problem)
  - Inefficient protocols (e.g., FBDIMM)
  - No virtualization in the memory system
  - Horrible materials (F4) for boards

Conclusions

What we talk about in a procurement	What we should talk about
FLOPS	Anything but FLOPS: • Memory Latency Dominates • Integer Operations More Frequent • Generally, “data movement”
Memory Bandwidth	Effective Bandwidth Latency Concurrency in the memory system
Power	Data Movement Efficiency
“Peaks”	Sustained