Mehrstufige Caches

Xuanqi Meng, Jeremias Rieser, Artem Bilovol

GRA SystemC Gruppe 192

2024

Inhalt I

Inhalt II

Problemstellung

- Moderne Prozessoren sind deutlich schneller als der Hauptspeicher.
- Von Neumann-Flaschenhals: Die beschränkte Bandbreite und langsamen Datenübertragungsraten zwischen Prozessor und Hauptspeicher führen zu einem Leistungsengpass..
- Ziel: Untersuchung der Auswirkungen von mehrstufigen Caches auf Laufzeit und Latenz.

Von-Neumann-Flaschenhals

- Prozessoren können Daten schneller verarbeiten als diese aus dem Hauptspeicher gelesen werden können.
- Der Hauptspeicher stellt einen Flaschenhals dar, der die Gesamtleistung des Systems limitiert.
- Lösung: Einsatz von Caches als Puffer zwischen Prozessor und Hauptspeicher.

Lösungsansätze und Optimierungen

- Implementierung einer Cache-Simulation in SystemC und C++.
- Simulation eines direkt-assoziativen Caches.
- Untersuchung von Parametern wie Cachegröße, Assoziativität, Cachezeilenanzahl und Zeilengröße und deren Auswirkung auf die Zugriffszeiten.

Cache-Architektur

- Verwendung von L1- und L2-Caches zur Reduktion der Zugriffszeit [4]
- L1-Cache: kleine Größe aber geringe Latenz (wenige Zyklen)
- L2-Cache: größer als L1-Cache (hier: inklusive) aber höhere Latenz
- Background-memory: Haupspeicher der Systems, höchste Latenz
- Inklusivität: alle Adressen, die in L1 speichern auch in L2 aber nicht umgekehrt.

Assoziativität

- Direkt abgebildeter Cache: Jeder Speicherblock verwaltet eine Speicheradresse (auch einfach assoziativ genannt)
- Alternativ: höhere Assoziativität erlaubt effizientere Cache-Nutzung aber höhere Kosten

Direkt abgebildeter Cache

- Jeder Speicherblock hat genau eine mögliche Position im Cache.
- Einfacher Aufbau und schnelle Zugriffszeiten.
- Nachteile: Höhere Konfliktwahrscheinlichkeit und niedrigere Trefferquote im Vergleich zu assoziativen Caches.

Write-Through Cache

- Write-Through: Daten werden gleichzeitig in den Cache und den Hauptspeicher geschrieben.
- Vorteile: Daten im Hauptspeicher sind immer aktuell.
- Nachteile: kumulative Latenz bei Schreiboperationen.

Daten- und Adressbus

- Datenbus und Adressbus sind 4 Byte breit.
- Der Hauptspeicher ist jedoch byte-adressiert.

Literaturrecherche

- Übliche Cachegrößen: L1 (32KB), L2 (256KB) [5]
- Latenzen: L1 (3 - 5 Zyklen), L2 (10 - 20 Zyklen) [1],[5]
- Vergleich der eigenen Implementierung mit den Ergebnissen

Typedefs

```c
typedef struct Request {
  uint32_t addr;
  uint32_t data;
  int we;  // 0 read 1 write
} Request;

typedef struct Result {
  size_t cycles;
  size_t misses;
  size_t hits;
  size_t primitiveGateCount;
} Result;

typedef struct CacheLine {
  int tag;
  uint8_t *bytes;
  int empty;
} CacheLine;
```

Listing 1: types.h

CACHEL1.h

```
SC_MODULE(CACHEL1){
    ...
    CacheLine *internal;

    sc_in<bool> clk;
    sc_in<bool> requestIncoming;
    sc_in<sc_bv<32>> inputData;
    sc_in<sc_bv<32>> address;
    sc_in<bool> rw;
    sc_out<bool> ready;
    sc_out<sc_bv<32>> outputData;
    sc_out<bool> isWriteThrough;
    ...
    void run();
    void write();
    void read();
    void writeThrough(int index, int address);
    int loadFromL2(int address);
    ...
};
```

Listing 2: CACHEL1.h - Cache Level 1 Header

CACHEL1.h

```
SC_CTOR(CACHEL1);
CACHEL1(sc_module_name name, int latency, int cacheLines, int cacheLineSize)
    : sc_module(name) {
  /*...*/
  offsetLength = (int)(log(cacheLineSize) / log(2));
  indexLength = (int)(log(cacheLines) / log(2));
  tagbits = 32 - offsetLength - indexLength;
  tagOffset = 32 - tagbits;
  indexOffset = 32 - tagbits - indexLength;

  internal = new CacheLine[cacheLines];
  for (int i = 0; i < cacheLines; i++) {
    internal[i].bytes = (uint8_t*)calloc(cacheLineSize, sizeof(uint8_t));
    internal[i].empty = 1;
  }

  SC_THREAD(run);
  sensitive << clk.pos() << requestIncoming;
}
```

Memory.h

```
SC_MODULE(MEMORY){
    int latency;
    sc_in<bool> clk;

    sc_in<bool> requestIncoming;
    sc_in<bool> rw;
    sc_in<sc_bv<32>> addr;
    sc_in<sc_bv<32>> rData; // the incomming data from L2

    sc_out<bool> ready;
    sc_out<sc_bv<32>> wData; // output data to L2

    std::unordered_map<uint32_t,uint8_t>  internal;

  SC_CTOR(MEMORY);
  MEMORY(sc_module_name name, int latency) : sc_module(name) {
    this->latency = latency;
    SC_THREAD(run);
    sensitive << clk.pos() << requestIncoming;
  }
  void run();
```

Überblick über die Simulationsfunktion

```
Result run_simulation(int cycles, unsigned l1CacheLines,
                      unsigned l2CacheLines,
                      unsigned cacheLineSize, unsigned l1CacheLatency,
                      unsigned l2CacheLatency, unsigned memoryLatency,
                      size_t numRequests, struct Request requests[],
                      const char *tracefile) {

  MEMORY memory("mem",memoryLatency);
  CACHEL2 l2Cache("l2",l2CacheLatency,l2CacheLines,cacheLineSize);
  CACHEL1 l1Cache("l1",l1CacheLatency,l1CacheLines,cacheLineSize);

  l1Cache.l2 = l2Cache.internal;
  l2Cache.l1 = l1Cache.internal;

};
```

Weiter in der Simulationsfunktion

```
// one tick for initialization
  clk.write(true);
  sc_start(1,SC_NS);
  clk.write(false);
  sc_start(1,SC_NS);

  bool lastR = false;
  int i;
  for(i = 0; i < cycles ; i++){
    /*...*/ // pr fe ob alle Module bereit sind und schicke die Anfrage
    clk.write(true);
    sc_start(1,SC_NS);
    requestToL1.write(false);
    clk.write(false);
    sc_start(1,SC_NS);
  }
```

Korrektheit und Genauigkeit

- Korrektheit bei Schreiben und Lesen: Es soll kein Unterschied zwischen Zugriffen mit und ohne Cache bestehen.
  - Beispiel für erfolgreiche Speicherung von Daten im Speicher und deren korrekte Darstellung beim Lesen.
  - Automatischer Vergleich von Daten aus Cache und ohne Cache bei 1000 zufällig generierten Anfragen (Kommandozeilenargumente werden sowohl manuell als auch automatisch erzeugt).
- Anzahl von Hits und Misses:
  - Codeanalyse
  - Beispiel
- Anzahl von Cycles:
  - Bestimmung der Anzahl von Zyklen beim Lesen und Schreiben
  - Analyse von Beispielen

Korrektheit bei Schreiben und Lesen

Ein Beispiel: CachelineSize = 64
Parameter:
`--l1-lines 2 --l2-lines 2 e.csv`
Eingabe:

```
W,0x0000003e,0x12345678
W,0x000000be,0x87654321
R,0x0000003d,
R,0x000000bd,
R,0x0000003c,
```

```
Request 0:
    addr: 0x0000003E
    data: 0x12345678
    we: 1
Request 1:
    addr: 0x000000BE
    data: 0x87654321
    we: 1
Request 2:
    addr: 0x0000003D
    data: 0x00123456
    we: 0
Request 3:
    addr: 0x000000BD
    data: 0x00876543
    we: 0
# ...
```

Korrektheit bei Schreiben und Lesen (Fort.)

Ein Beispiel: CachelineSize = 64
Parameter: `--l1-lines 2 --l2-lines 2 e.csv`
Eingabe:

```
W,0x0000003e,0x12345678
W,0x000000be,0x87654321
R,0x0000003d,
R,0x000000bd,
R,0x0000003c,
```

Ausgabe:

```
# ...
Request 3:
    addr: 0x000000BD
    data: 0x00876543
    we: 0
Request 4:
    addr: 0x0000003C
    data: 0x00001234
    we: 0
Result:
    cycles: 58570
    misses: 14
    hits: 0
    primitiveGateCount: 42649
```

Korrektheit bei Schreiben und Lesen (Fort.)

- Erste Stufe nur mit dem Memory-modul.
- Kein Overhead bzw. keine Fehleranfälligkeit durch mehrstufiges Cachen
- Als input zufällig erzeugte .csv (1000 Zeilen) und vergleich der Ergibnisse
- Test ist natürlich nur sinnvoll, wenn sich die Adressen wiederholen bzw. nahe anernander liegen (überschneidung)
- gesichert durch simulation von 1000 Adressen aus einem Adresspool von Größe 500
- eine Testeingabe mit parameter –l1-lines 2 –l2-lines 2:

```
W,0x00000581,0x66666666
W,0x00000440,0x33333333
R,0x00000070,
R,0x00001CE1,
#...
W,0x00000901,0x22222222
W,0x00001B13,0x55555555
R,0x00001C60,
```

Korrektheit bei Schreiben und Lesen(Fortsezung)

- Die Ergebnisse werden dann durch eine .sh verglichen

```
for INPUTFILE in *.csv; do
  if [ -f "$INPUTFILE" ]; then
    ./project -args ... > to_test.txt
    ./test/project -args ... > direct_memory.txt #  this is only memory
    if diff -q to_test.txt direct_memory.txt > /dev/null; then
        echo  "No difference between the output with and without cache."
    else
        echo  "There is a difference, something is wrong."
    fi
  fi
done
```

- Ausgabe von .sh zeigt die Korrektheit

```
Checking cache_accesses.csv ...
        No difference between the output with and without cache.
The test is over.
```

Korrektheit bei Schreiben und Lesen(Fortsezung)

- zusätzliches Testen durch .sh mit zufällig generierten Kommandozeilenargumenten

```
...

CYCLES=$(( (RANDOM * RANDOM) % 100000000 ))

LINESIZE=$(( 2 ** (2 + RANDOM % 13) ))

L2LINES=$(( 2 ** (2 + RANDOM % 13) ))

L1LINES=$(( 2 ** (2 + RANDOM % 13) ))

...
```

- Ausgabe von .sh zeigt die Korrektheit

```
The test is over.
No problems found.
```

Anzahl von Hits und Misses

- Codeanalyse (Mechanismus von Hit und Miss)

```
index = ifExist(t_tmp,i_tmp);
if(index==-1){ // in the cache there are no such information
    miss++;
    hit = false;
    index = loadFromL2(address.read().to_int());
}
if(hit){
    hits++;
}
```

Listing 3: L1Cache.cpp

- `ifExist(tag,index)` sucht die zugreifende Zeile
- Ein Ähnlicher Mechanismus kommt bei L2 zum einsatz
- Bei einem zeileübergreifenden Zugriff gilt es nur als Hit, wenn beide Zeilen gefunden werden können

Berechnung von Hits und Misses (Fortsetzung)

- Berechnung von Hit und Miss
  - Die Hits und Misses von beiden L1 und L2 werden zusammen gezählt
  - Beim Read (Zugriff in 1 Zeile):
    L1 Hit = 1 Hit
    L1 Miss + L2 Hit = 1 Miss + 1 Hit
    L1 Miss + L2 Miss = 2 Miss
    (Zugriff in 2 Zeilen)
    L1 Hit = 1 Hit
    L1 Miss(1 Zeile) + L2 Hit = 1 Miss + 1 Hit
    L1 Miss(1 Zeile) + L2 Miss = 2 Miss
    L1 Miss(2 Zeile) + L2 Hit + L2 Hit = 1 Miss + 2 Hits
    L1 Miss(2 Zeile) + L2 Miss + L2 Hit = 2 Miss + 1 Hit (2 mal Anfrage zur L2)
    L1 Miss(2 Zeile) + L2 Miss + L2 Miss = 3 Miss
  - Beim Write ist der einzige Unterschied: L1 Hit + L2 Hit = 2 Hit
  - Schreiben findet in allen Modulen statt

Berechnung von Hits und Misses (Beispiel)

- Berechnung von Hit und Miss

Ein Beispiel: CachelineSize = 4

Parameter: `--l1-lines 2 --l2-lines 4`

```
1  W,0x0000003e,0x12345678  #Unaligned L1 miss + L2 miss + L2 miss
2                  #2 Zeile in L1 und L2 geladt
3  W,0x0000003f,0x87654321  #Unaligned L1 hit + L2 hit
4                  #2 Zeile existert in L1 und L2
5  W,0x0000003c,0x56789123  #aligned   L1 hit + L2 hit
6                  #1 Zeile existiert in L1 und L2
7  W,0x00000037,0x12345678  #unaligned L1 miss + L2 miss + L2 miss
8                  #2 Zeile in L1 und L2 geladt
9  W,0x0000003d,0x12345678  #unaligned L1 miss + L2 hit + L2 hit
10                 #2 Zeile in L1 geladt, ist aber in L2 vorhanden
11 R,0x000000f4,            #aligned   L1 miss + L2 miss
12                 #1 Zeile wird in beide L1 und L2 geladt
13 R,0x0000003d,            #unaligned L1 miss(1 Zeile) + L2 hit
14                 #1 Zeile existiert nicht in L2, wird von Memory geladt
```

Berechnung von Hits und Misses (Hit Log)

- Compile mit Flag HIT_LOG:

```
l1 received <write> with addr:0000003e and data:12345678
l1 miss by writing first line with addr: 0000003e detected, sending
    signal to next level
l2 miss by reading with addr: 0000003e detected, sending signal to
    next level
l1 miss by writing second line with addr: 00000041 detected, sending
    signal to next level
l2 miss by reading with addr: 00000041 detected, sending signal to
    next level
l1 miss by writing by 2 lines with addr:0000003e
# ...
l1 received <read> with addr:000000f4
l1 miss by reading with addr: 000000f4 detected, sending signal to
    next level
l2 miss by reading with addr: 000000f4 detected, sending signal to
    next level
l1 received <read> with addr:0000003d
l1 miss by reading first line with addr: 0000003d detected, sending
    signal to next level
```

Berechnung von Hits und Misses (Ergebnis)

- Ausgegebene Anzahl an Hits und Misses

```
Result:
        cycles: 177
        misses: 10
        hits: 7
        primitiveGateCount: 4369
```

Auswirkungen von Parametern

- Zu kleine Cachelines nutzen die lokalität der Inputs nicht richtig aus, also mehr misses, gerade bei unaligned-access [6]

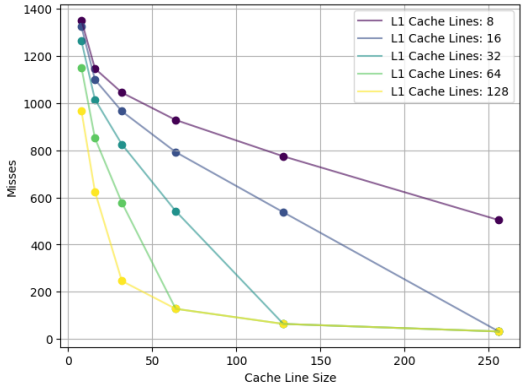
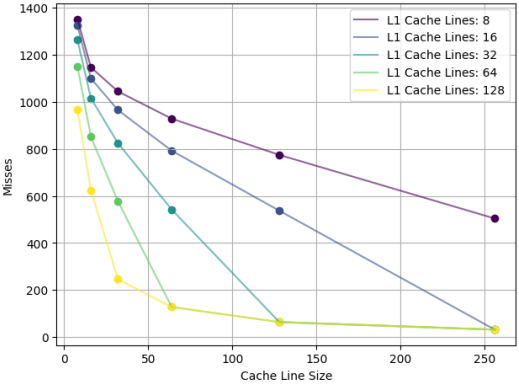


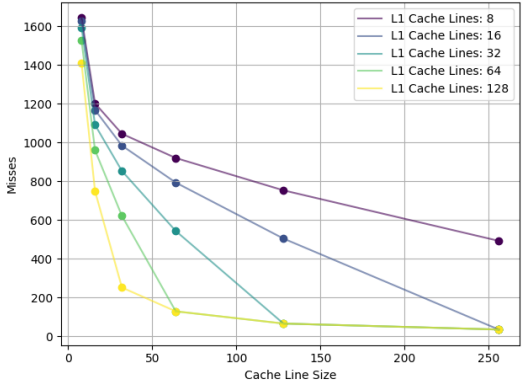
Figure: Test with 1000 requests (aligned)

Auswirkungen von Parametern (Forts.)

- Aligned acesses haben durchschnittlich mehr misses als aligned accesses, vorallem bei kleinen Cachelines.



Test with 1000 requests (aligned)



Test with 1000 requests (unaligned)

Anzahl von Cycles

- Anzahl von Cycles sind deterministisch je nach Verhalten des Inputs
- Für einen Datenzugriff nehmen wir die Latenzen in den verschiedenen Modulen
- In modernen Caches ist es deutlich schneller, nur den Tag anstatt von gesamten Daten zu verändern.

Anzahl von Cycles

- Anzahl von Cycles bei verschiedenen Verhalten
- Bei Read
  L1 Hit = L1 Latency
  L1 Miss + L2 Hit = L1 Latency + L2 Latency
  L1 Miss + L2 Miss = L1 Latency + L2 Latency + cachelineSize/4* Memory Latency
  (Zugriff in 2 Zeilen)
  L1 Hit + L1 Hit= 2*L1 Latency
  L1 Miss + L1 Hit + L2 Hit = 2* L1 Latency + L2 Latency
  L1 Miss + L1 Hit + L2 Miss = 2*L1 Latency + L2 Latency + cachelineSize/4* Memory Latency
  L1 Miss + L1 Miss + L2 Hit + L2 Hit = 2*L1 Latency + 2* L2 Latency
  L1 Miss + L1 Miss + L2 Miss + L2 Hit = 2*L1 Latency + 2* L2 Latency + cachelineSize/4* Memory Latency
  L1 Miss + L1 Miss + L2 Miss + L2 Miss = 2*L1 Latency + 2* L2 Latency + 2*(cachelineSize/4* Memory Latency)

Anzahl von Cycles(Fortsetzung)

- Anzahl von Cycles bei verschiedenen Verhalten
- Bei Write(Through) — Write-Allocate wird angepasst
  L1 Hit = Memory Latency
  L1 Miss + L2 Hit = L2 Latency + Memory Latency
  L1 Miss + L2 Miss = L2 Latency + cachelineSize/4* Memory Latency + Memory Latency
  ...
- Alle L1 Latenzen werden in einer Memory Latenz mitgezählt, solange
  Memory Latency >2*L2 Latency >2*L1 Latency
  Ansonsten: Memory Latency wir durch die größte der oberen drei ersetzt.

Anzahl von Cycles (Beispiel)

- Berechnung von Hit und Miss

Ein Beispiel: CachelineSize = 8

Parameter: `--l1-lines 2 --l2-lines 4 --l1-latency 3 --l2-latency 5 --memory-latency 12`

```
1  W,0x0000003e,0x12345678  #Unaligned L1 miss + L1 miss + L2 miss + L2 miss
2                   # 2*5 + 2*(8/4)*12 + 12 = 70
3  W,0x0000003f,0x87654321  #Unaligned L1 hit + L1 hit + L2 hit + L2 hit
4                   # 12
5  W,0x00000038,0x11111111  #aligned   L1 hit + L2 hit
6                   # 12
7  W,0x0000004f,0x12345566  #unaligned L1 miss + L1 miss + L2 miss + L2 miss
8                   # 2*5 + 2*(8/4)*12 + 12 = 70
9  W,0x0000003d,0xdddddddd  #unaligned L1 miss + L1 miss + L2 hit + L2 hit
10                  # 2*5 + 12 = 22
11 #...
```

Anzahl von Cycles (Beispiel—Fortsetzung)

- Berechnung von Hit und Miss

Ein Beispiel: CachelineSize = 8

Parameter: `--l1-lines 2 --l2-lines 4 --l1-latency 3 --l2-latency 5 --memory-latency 12`

```
1 #...
2 R,0x000000ec,              #aligned   L1 miss + L2 miss
3                  # 3 + 5 + (8/4)*12 = 32
4 R,0x0000003d,             #unaligned L1 miss + L1 hit + L2 hit
5                  # 2*3 + 5 = 11
6                  # insgesamt : 229
```

Anzahl von Cycles (Ergebnis)

- Ergebnis

```
Result:
        cycles: 237
        misses: 10
        hits: 7
        primitiveGateCount: 6991
```

- Overhead

  1. Bei jedem Ready von L1 wird noch ein Cycle gewartet bis zur nächsten Anfrage also 7 Anfragen + 229 = 236
  2. Nach dem Ready von allen Module wird noch ein Cycle gewartet bis zum Ende des Programms
  also 236 + 1 = 237

- die Anzahl an Cycles sind also deterministisch gegenüber der Operation

Aligned vs Unaligned access

- Zugriffe sollen auch mit Offset möglich sein
- Nachteil bei der Performanz [3]:
- Generell brauchen unaligned-accesses in manchen Fällen (anhängig von Größe) Zugriff auf 2 Cachelines - Damit verdoppelt sich die Zugriffslatenz bei Misses
- Dies ist im Test korrekt wiedergegeben

Schaltkreis

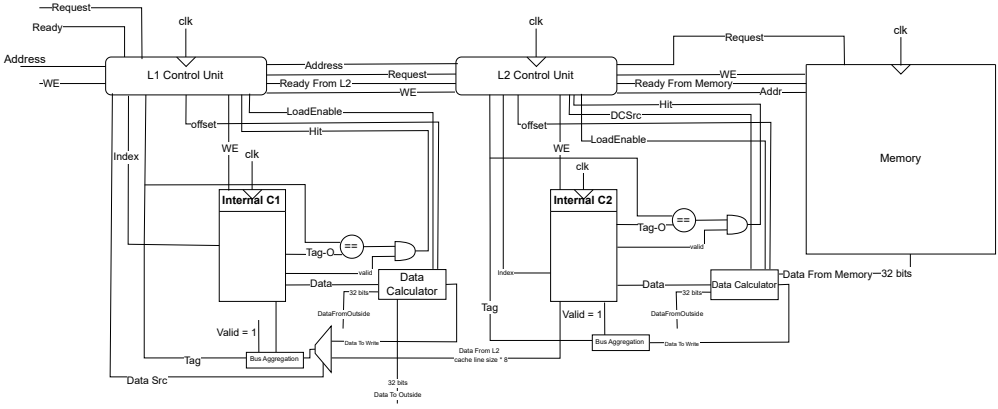


Figure: Schaltkreis - Simulation

L1 Control Unit

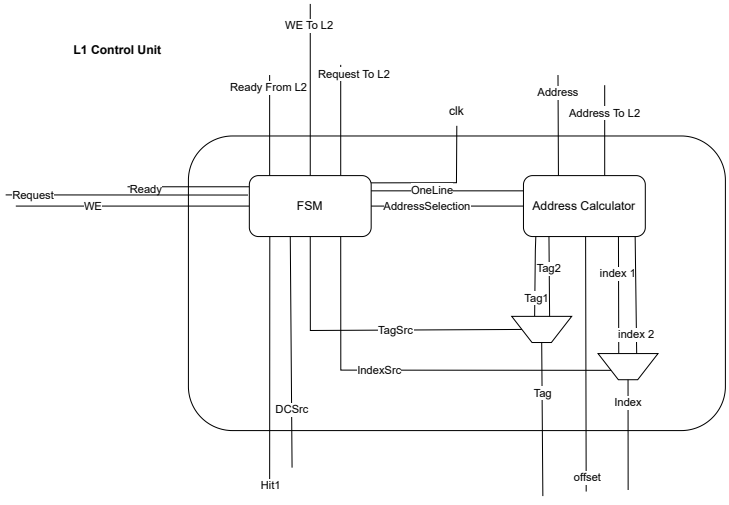


Figure: L1 Control Unit

L1 Control Unit - FSM

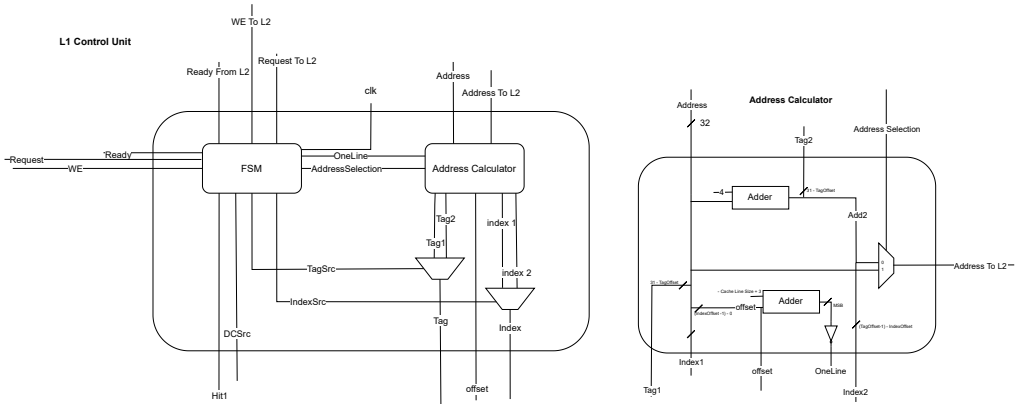


Figure: L1 Control unit and Adress calculator

L1 Control Unit - FSM

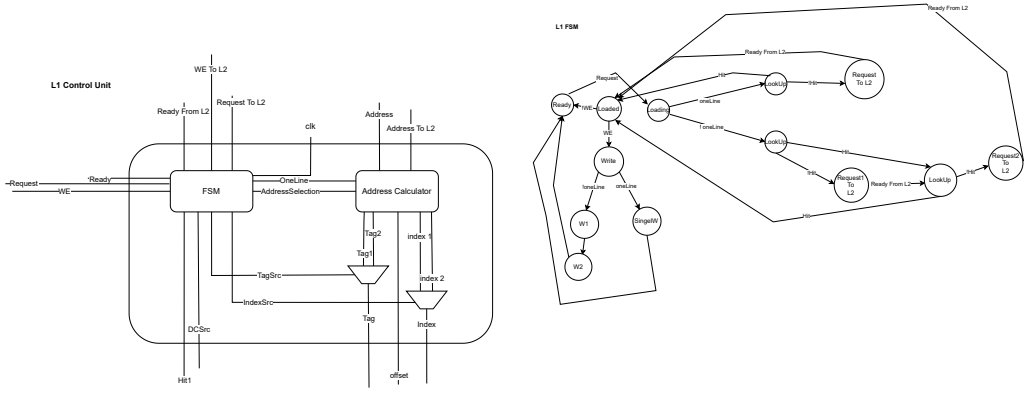


Figure: L1 Control unit and FSM

L2 Control Unit

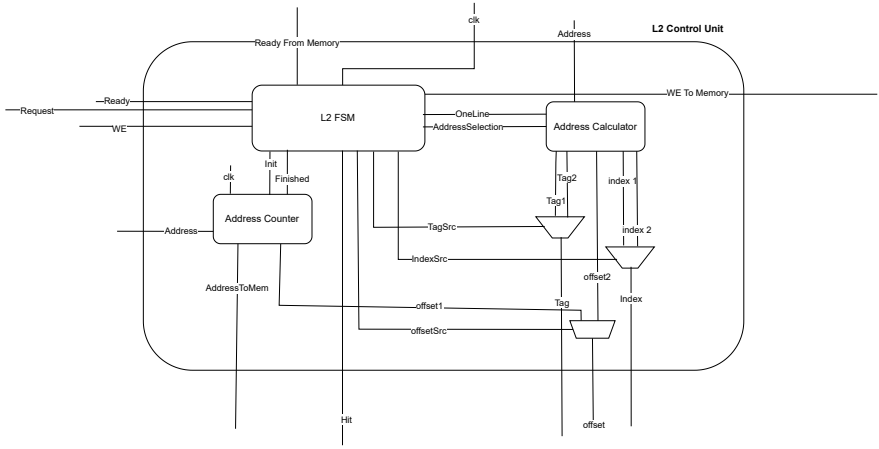
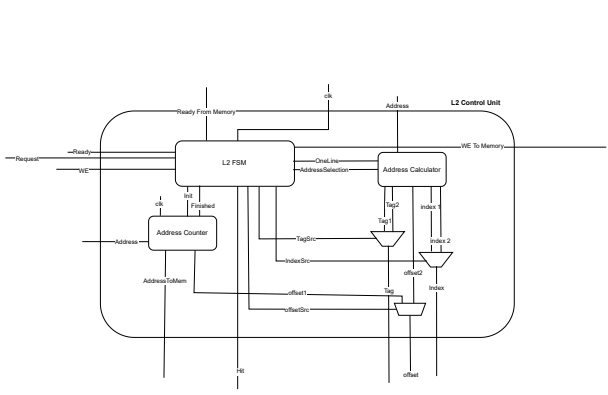


Figure: L2 Control Unit

L2 Control Unit - FSM



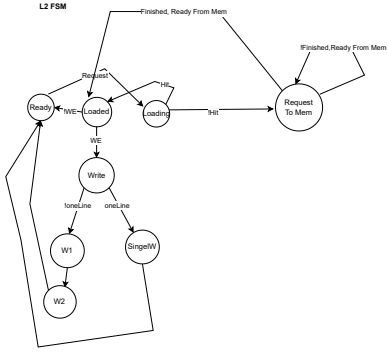


Figure: L2 Control unit and FSM

L2 Control Unit - Address Counter

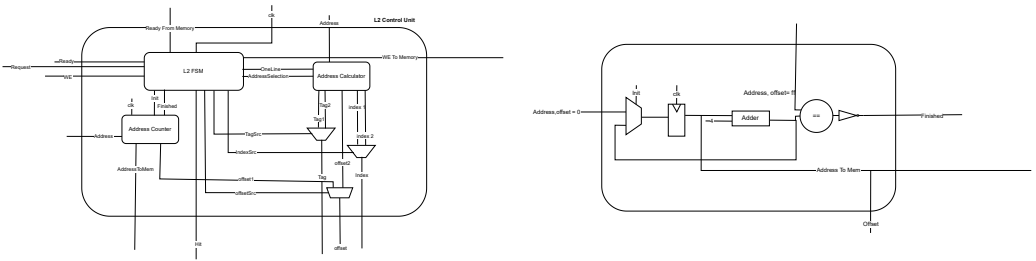


Figure: L2 control unit and Address Counter

Gatterberechnung L1

- L1:
  Speicherung: (Tagbits + valid + CachelineSize*8)*Cachlines
  Tag Comparator: TagLength + TagLength -1 (XOR und OR) + 1 (AND)
  CachelineSize*8-bit-2-to-1 MUX : CachelineSize *8 * 4
  Control Unit:
  2-to-1 MUX für Tag: TagLength * 4
  2-to-1 Mux für Index: IndexLength * 4
  FSM:
  Zustandspeicherung: log2(13) = 4 (bits) D-Flip-Flop, Gatter = 4 * 5 = 20
  Zustandübergang: (5 Eingabe + 4 bits) * 4 bits = 36
  Ausgabe: 4 bits * 7 Ausgabe = 28

Gatterberechnung L1 (Forts.)

- L1:
  Address Calculator:
  32-bit Adder : (2 And + 1 Or + 2 XOR) * 32 = 160
  offset Adder : (2 And + 1 Or + 2 XOR) * offsetLength
  32-bit-2-to-1 MUX: 32 * 4 = 128
  Data Calculator:
  Speicherung: (2*CachelineSize*8) D-Latch(jeweils 5 Gatter)
  1-to-2 Decoder: 1 Not
  2 * 1-bit-2-to-1 MUX: 2 * 4
  CachelineSize*8-bit-2-to-1 MUX: CachelineSize*8*4
  CachelineSize * 32-bit-2-to-1 MUX : CachelineSize*32*4
  CachelineSize * 32-bit-2-to-1 DEMUX : CachelineSize*32*3

Gatterberechnung L2

- L2:
  Speicherung: (Tagbits + valid + CachelineSize*8)*Cachlines
  Tag Comparator: TagLength + TagLength -1 (XOR und OR) + 1 (AND)
  Control Unit:
  2-to-1 MUX für Tag: TagLength * 4
  2-to-1 MUX für Index: IndexLength * 4
  2-to-1 MUX für Offset : OffsetLength * 4
  FSM:
  Zustandspeicherung: log2(8) = 3 (bits) D-Flip-Flop, Gatter = 3 * 5 = 15
  Zustandübergang: (6 Eingabe + 3 bits) * 3 bits = 27
  Ausgabe: 3 bits * 7 Ausgabe = 21

Gatterberechnung L2 (Forts.)

- L2:

  Address Counter:

  32-bit-2-to-1 MUX : 32 * 4 = 128

  32-bit D-Flip-Flop = 32 * 5 = 160

  32-bit Adder = 160

  32-bit Comparator = 32 XOR + 31 OR = 63

  1 Inverter = 1

  Address Calculator:

  gleich wie bei L1

  Data Calculator:

  Speicherung: (2*CachelineSize*8) D-Latch(jeweils 5 Gatter)

  1-to-2 Decoder: 1 Not

  2 * 1-bit-2-to-1 MUX: 2 * 4

  CachelineSize*8-bit-2-to-1 MUX: CachelineSize*8*4

  CachelineSize * 32-bit-2-to-1 MUX : CachelineSize*32*4

  Kein Demux nötig, da wir keine 32-bit Daten auslesen sollen

Vergleich mit realen Systemen

- Bei vielen SRAM-Chips ist neben der Speicherung häufig auch Funktionalität wie FSM, Adressverarbeitung und Datenverarbeitung eingebaut, die mit unseren Komponenten vergleichbar ist.
- z.B CY7C1380D von CYPRESS [2]

Zusammenfassung

- Das Ergebnis dieses Projekts ist eine Implementierung eines zweistufigen Caches, der als Werkzeug zur Messung des Nutzens von Cache-Speicher dient.
- Im Verlauf des Projekts wurden mehrere Standardprobleme analysiert und gelöst, insbesondere:
  - Abstraktion von Hardwarekomponenten in der Simulation
  - Synchronisation zwischen Modulen
  - Datenausrichtung und Bytereihenfolge
  - Softwaretests
- Ein weiterer wichtiger Teil des Projekts war die Strukturierung und das Kompilieren, insbesondere unter Berücksichtigung der Umgebung, die nicht vom Team kontrolliert wird.

Ausblick

Basierend auf den Ergebnissen des Projekts werden folgende Bereiche für mögliche Verbesserungen in Betracht gezogen:

- Hinzufügen von L3-Cache und externem Cache zur Simulation
- Multiprocessing und unterschiedliche Implementationen von L1-Cache (insbesondere Cache-Fairness)
- I-Cache und D-Cache
- Testen aligned vs unaligned Offset-Zugriffe
- Simulation von anderen Arten von Cache (exklusive, andere Assozivität usw.)

[1] colin scott colin. *Numbers Every Programmer Should Know By Year*. 2021. URL: https://colin-scott.github.io/personal_website/research/interactive_latency.html (visited on 07/21/2024).

[2] Cypress Semiconductor Corporation. *18-Mbit (512K × 36/1M × 18) Pipelined SRAM*. 2016. URL: https://www.mouser.com/datasheet/2/100/CYPR_S_A0002300802_1-2540696.pdf.

[3] Daniel Drake. *Unaligned Memory Accesses — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/latest/core-api/unaligned-memory-access.html (visited on 07/21/2024).

[4] Christoph Ersfeld. "Organisation von Caches". de. In: (2002).

[5] "Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1". en. In: (2024).

[6] Viraj Kumar. *CS232: Computer Architecture II*. 2010. URL: https://courses.grainger.illinois.edu/cs232/sp2010/lectures/L17.pdf (visited on 07/21/2024).

Daten Berechner (Einlesen und Auslesen mit Cache)

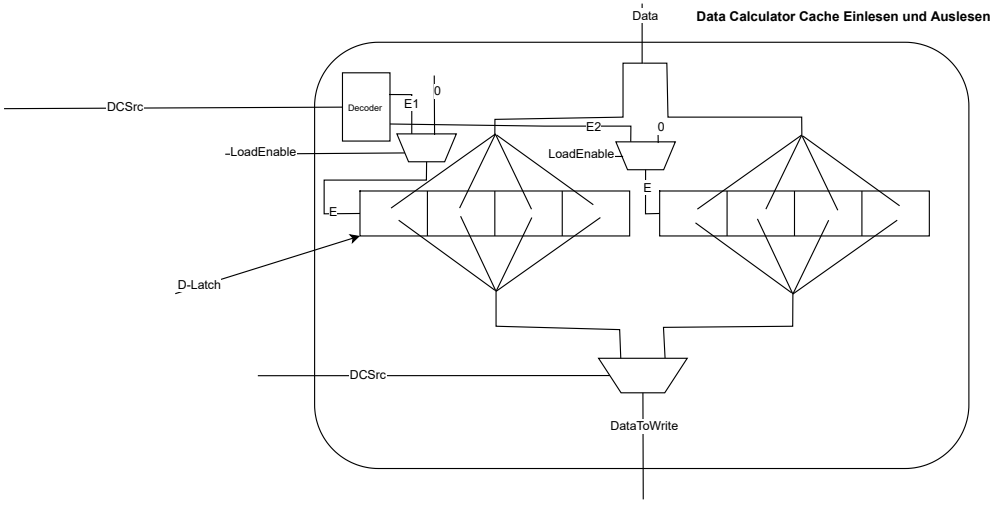


Figure: Einlesen und Auslesen mit Cache

Daten Berechner (Einlesen mit 32 bits Daten)

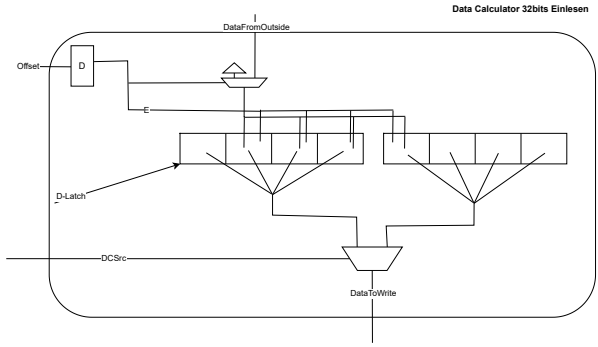


Figure: Einlesen mit 32 bits Daten

Daten Berechner (Auslesen mit 32 bits Daten)

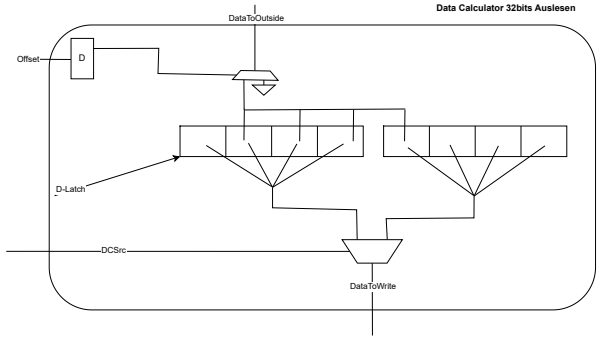


Figure: Auslesen mit 32 bits Daten