

Introduction to Computer Architecture

Pascal Garcia

February 24, 2022

1 Introduction

- Why Computer Architecture?
- The Von Neumann Model

2 Unsigned, Signed, Float

- Unsigned
- Signed
- Floating Point Numbers

3 From C to Binary

4 MIPS

- RISC
- MIPS ISA
- R-Type Instructions
- I-Type Instructions
- J-Type Instructions
- Program Example

5 Single-Cycle Datapath

- Overview
- Datapath Elements
- Datapath Implementation

- Running a Program
- Inefficiency

6 Multi-Cycle Datapath

- Overview
- Datapath Implementation

7 Pipelining

- Overview
- Timing Comparison

8 Interrupts

- Interrupts and Exceptions
- I/O Devices
- Waiting for I/O Devices
- Ints and Exceptions on MIPS
- Single-Cycle with Ints and Exs
- Ints and Exceptions on x86_64
- Linux Signals

9 Out of Order

10 Memory Hierarchy

- Overview
- Locality
- Cache
- Cache Organisation
- Code Optimization for Cache

11 Multicore

Why Computer Architecture?

- Knowledge of the hardware will make you a better programmer.
- Needed to understand other topics in computer science:
 - Operating systems.
 - Compilers.
 - ...

Integer Representation

```
1 #include <stdio.h>
2 #include <string.h>
3
4 void* copy(void* dest, void* src, unsigned short len) {
5     const char* s = src;
6     char* d = dest;
7     while (len--)
8         *d++ = *s++;
9     return dest;
10 }
11
12 #define KSIZE 1024
13 char kbuf[KSIZE];
14 char private_data[65536];
15
16 void init_private() {
17     char* s = "SECRET\u202aINFO";
18     copy(private_data, s, strlen(s));
19 }
```

Integer Representation

```
1 int copy_from_kernel(char* user_dest, int maxlen) {
2     int len = KSIZE < maxlen ? KSIZE : maxlen;
3     copy(user_dest, kbuf, len);
4     return len;
5 }
6
7 int main() {
8     char user_buff[100000] = {0};
9     init_private();
10    int size;
11    scanf("%d", &size);
12    copy_from_kernel(user_buff, size);
13    printf("%s\n", user_buff + 1024);
14    return 0;
15 }
16 // gcc -O3 integer1.c -o integer1
17 // ./integer1
18 // input: 80000 output:
19 // input: -1    output: SECRET INFO
```

Integer Representation

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 #define INT_MAX 2147483647
5 #define INT_MIN -2147483648
6 /* C90: int -> long int -> unsigned long int */
7 int main()
8 {
9     if (INT_MIN < 1) printf("INT_MIN<1\n");
10    else printf("INT_MIN>=1?!\\n");
11    return 0;
12 }
13 // gcc -m32 -std=c90 integer2.c -o integer2
14 // ./integer2
15 // output: INT_MIN >= 1 ?
16
17 // gcc -std=c90 integer2.c -o integer2
18 // ./integer2
19 // output: INT_MIN < 1
```

Real Number Representation

```
1 #include <iostream>
2
3 int main(int argc, char *argv[])
4 {
5     std::cout << (1e20 - 1e20) + 3.14 << std::endl;
6     std::cout << 1e20 - (1e20 + 3.14) << std::endl;
7     return 0;
8 }
9 // g++ real1.c -o real1
10 // ./real1
11 // output: 3.14
12 // 0
```

Real Number Representation

```
1 int main(int argc, char *argv[])
2 {
3     for (float x = 100000001; x <= 100000010; x += 1.0)
4     {
5     }
6     return 0;
7 }
8 // gcc real2.c -o real2
9 // ./real2
10 // output: runs forever!
```

Binary Code

```
1 const char* const shellcode =
2     "\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1"
3     "\xeb\x08\x53\x48\x89\xe7\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05";
4
5 int main()
6 {
7     ((void (*)()) shellcode)();
8     return 0;
9 }
10 // gcc shell.c -o shell
11 // ./shell
12 // output: $
13 //         $ uname
14 //         Linux
15 //         $
```

Instruction Latency

```
1 #include <stdio.h>
2 #include <time.h>
3 #include <stdint.h>
4
5 uint32_t digits10(uint64_t v)
6 {
7     uint32_t result = 0;
8     do {
9         ++result;
10        v /= 10;
11    } while (v);
12    return result;
13 }
```

Instruction Latency

```
1 int main(int argc, char *argv[])
2 {
3     int sum = 0;
4     clock_t begin = clock();
5     for (int i = 0; i < 1000000000; ++i)
6     {
7         sum += digits10(i);
8     }
9     clock_t end = clock();
10    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
11    printf("res=%d in %lf s\n", sum, time_spent);
12    return 0;
13 }
14 // gcc -O3 latency1.c -o latency1
15 // ./latency1
16 // output: res = 298954298 in 5.833225 s
```

Instruction Latency

```
1 // from Andrei Alexandrescu
2 uint32_t digits10(uint64_t v)
3 {
4     uint32_t result = 1;
5     for (;;)
6     {
7         if (v < 10) return result;
8         if (v < 100) return result + 1;
9         if (v < 1000) return result + 2;
10        if (v < 10000) return result + 3;
11        v /= 10000U;
12        result += 4;
13    }
14 }
15 // gcc -O3 latency2.c -o latency2
16 // ./latency2
17 // output: res = 298954298 in 2.187219 s
```

Cache

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define SIZE 4096
6 int data[SIZE][SIZE];
7
8 int main() {
9     for (int i = 0; i < SIZE; ++i)
10        for (int j = 0; j < SIZE; ++j)
11        {
12            data[i][j] = i * j;
13        }
14     clock_t begin = clock();
15     long long res = test(50);
16     clock_t end = clock();
17     double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
18     printf("res=%lld in %lf seconds\n", res, time_spent);
19     return 0;
20 }
```

Cache

```
1 long long test(int nb)
2 {
3     long long sum = 0;
4     for (int i = 0; i < nb; ++i)
5         for (int k = 0; k < SIZE; ++k)
6             for (int j = 0; j < SIZE; ++j)
7                 {
8                     sum += data[j][k];
9                 }
10    return sum;
11 }
12 // gcc-7 -O3 cache.c -o cache
13 // ./cache
14 // output: res = 3516719431680000 in 7.597586 seconds
```

Cache

```
1 long long test(int nb)
2 {
3     long long sum = 0;
4     for (int i = 0; i < nb; ++i)
5         for (int j = 0; j < SIZE; ++j)
6             for (int k = 0; k < SIZE; ++k)
7             {
8                 sum += data[j][k];
9             }
10    return sum;
11 }
12 // gcc-7 -O3 cache.c -o cache
13 // ./cache
14 // output: res = 3516719431680000 in 0.243503 seconds
```

Prefetching

```
1 #include <time.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define SIZE 10000000000
6 // from Zhirkov
7 int binary_search(int *array, size_t number_of_elements, int key) {
8     size_t low = 0, high = number_of_elements-1, mid;
9     while (low <= high) {
10         mid = low + (high - low) / 2;
11 #ifdef DO_PREFETCH
12         __builtin_prefetch(&array[(mid + 1 + high) / 2], 0, 1);
13         __builtin_prefetch(&array[(low + mid - 1) / 2], 0, 1);
14 #endif
15         if (array[mid] < key)      low = mid + 1;
16         else if (array[mid] == key) return mid;
17         else if (array[mid] > key) high = mid-1;
18     }
19     return -1;
20 }
```

Prefetching

```
1 #define NUM_LOOKUPS 10000000
2
3 int main() {
4     int *array, *lookups;
5     srand(42);
6     array = malloc(SIZE * sizeof(int));
7     lookups = malloc(NUM_LOOKUPS * sizeof(int));
8     for (size_t i=0; i < SIZE; ++i) array[i] = i;
9     for (size_t i=0; i < NUM_LOOKUPS; ++i) lookups[i] = rand() % SIZE;
10
11    clock_t begin = clock();
12    for (size_t i=0; i < NUM_LOOKUPS; ++i)
13        binary_search(array, SIZE, lookups[i]);
14    clock_t end = clock();
15
16    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
17    printf("%lf\u00a5seconds\n", time_spent);
18    free(array);
19    free(lookups);
20 }
```

Prefetching

```
1 // gcc -O3 prefetch_binsearch.c -o binsearch1
2 // ./binsearch1
3 // output: 8.105817 seconds
4
5 // gcc -O3 -DDO_PREFETCH prefetch_binsearch.c -o binsearch2
6 // ./binsearch2
7 // output: 5.204869 seconds
```

Cache Coherence

```
1 #include <iostream>
2 #include <thread>
3 #include <algorithm>
4 #include <chrono>
5 #include <atomic>
6
7 const int SIZE = 500000000;
8
9 std::atomic<bool> go(false);
10
11 void increment(unsigned int& count)
12 {
13     while (!go);
14     for (long i = 0; i < SIZE; ++i)
15     {
16         count += sqrt(i);
17     }
18 }
```

Cache Coherence

```
1 int main(int argc, char *argv[]) {
2     using namespace std::chrono;
3     const steady_clock::time_point begin = steady_clock::now();
4     unsigned int count1 = 0;
5     unsigned int count2 = 0;
6     std::thread t1(increment, std::ref(count1));
7     std::thread t2(increment, std::ref(count2));
8     go = true;
9     t1.join(); t2.join();
10    const steady_clock::time_point end = steady_clock::now();
11    std::cout << &count1 << " " << &count2 << std::endl;
12    std::cout << "res=" << count1 + count2 << " in "
13                  << duration_cast<milliseconds>(end - begin).count()
14                  << "ms" << std::endl;
15    return 0;
16 }
17 // g++ -O3 -std=c++11 cache_coherence.c -lpthread -o cache_coherence
18 // ./cache_coherence
19 // output: 0x7ffd966f948 0x7ffd966f94c
20 //             res = 3083335160 in 5073 ms
```

Cache Coherence

```
1 int main(int argc, char *argv[]) {
2     using namespace std::chrono;
3     const steady_clock::time_point begin = steady_clock::now();
4     alignas(64) unsigned int count1 = 0;
5     alignas(64) unsigned int count2 = 0;
6     std::thread t1(increment, std::ref(count1));
7     std::thread t2(increment, std::ref(count2));
8     go = true;
9     t1.join(); t2.join();
10    const steady_clock::time_point end = steady_clock::now();
11    std::cout << &count1 << " " << &count2 << std::endl;
12    std::cout << "res=" << count1 + count2 << " in "
13                  << duration_cast<milliseconds>(end - begin).count()
14                  << "ms" << std::endl;
15    return 0;
16 }
17 // g++ -O3 -std=c++11 cache_coherence.c -lpthread -o cache_coherence
18 // ./cache_coherence
19 // output: 0x7ffd21e0fb40 0x7ffd21e0fb80
20 //             res = 3083335160 in 3322 ms
```

Cache Coherence

```
1 #include <iostream>
2 #include <thread>
3 #include <algorithm>
4 #include <chrono>
5 #include <atomic>
6
7 const int SIZE = 500000000;
8
9 std::atomic<bool> go(false);
10
11 void increment(std::atomic<unsigned int>& count)
12 {
13     while (!go);
14     for (long i = 0; i < SIZE; ++i)
15     {
16         count += sqrt(i);
17     }
18 }
```

Cache Coherence

```
1 int main(int argc, char *argv[]) {
2     using namespace std::chrono;
3     const steady_clock::time_point begin = steady_clock::now();
4     std::atomic<unsigned int> count1(0);
5     std::atomic<unsigned int> count2(0);
6     std::thread t1(increment, std::ref(count1));
7     std::thread t2(increment, std::ref(count2));
8     go = true;
9     t1.join(); t2.join();
10    const steady_clock::time_point end = steady_clock::now();
11    std::cout << &count1 << " " << &count2 << std::endl;
12    std::cout << "res=" << count1 + count2 << " in "
13                  << duration_cast<milliseconds>(end - begin).count()
14                  << "ms" << std::endl;
15    return 0;
16 }
17 // g++ -O3 -std=c++11 cache_coherence2.c -lpthread -o cache_coherence2
18 // ./cache_coherence2
19 // output: 0x7ffc5848ab20 0x7ffc5848ab30
20 //             res = 3083335160 in 19172 ms
```

Cache Coherence

```

1 int main(int argc, char *argv[]) {
2     using namespace std::chrono;
3     const steady_clock::time_point begin = steady_clock::now();
4     alignas(64) std::atomic<unsigned int> count1(0);
5     alignas(64) std::atomic<unsigned int> count2(0);
6     std::thread t1(increment, std::ref(count1));
7     std::thread t2(increment, std::ref(count2));
8     go = true;
9     t1.join(); t2.join();
10    const steady_clock::time_point end = steady_clock::now();
11    std::cout << &count1 << " " << &count2 << std::endl;
12    std::cout << "res=" << count1 + count2 << " in "
13                                << duration_cast<milliseconds>(end - begin).count()
14                                << "ms" << std::endl;
15    return 0;
16 }
17 // g++ -O3 -std=c++11 cache_coherence2.c -lpthread -o cache_coherence2
18 // ./cache_coherence2
19 // output: 0x7ffd9ab1cecc0 0x7ffd9ab1cf00
20 //             res = 3083335160 in 2794 ms

```

Pipelining, Out of Order, Superscalar

```
1 #include <stdio.h>
2 #include <time.h>
3
4 int main(int argc, char *argv[]) { // from Agner
5     double res = 0;
6     clock_t begin = clock();
7     for (unsigned int i = 0; i < 1000000000; ++i) {
8         double a = i, b = i + 1, c = i + 2;
9         double d = i + 3, e = i + 4;
10        res = res + a + b + c + d + e;
11    }
12    clock_t end = clock();
13    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
14    printf("res = %.0f in %lf s\n", res, time_spent);
15    return 0;
16 }
17 // gcc -O3 pipelining1.c -o pipelining1
18 // ./pipelining1
19 // output: res = 2500000006980297216 in 4.092826 s
```

Pipelining, Out of Order, Superscalar

```
1 #include <stdio.h>
2 #include <time.h>
3
4 int main(int argc, char *argv[]) { // from Agner
5     double res = 0;
6     clock_t begin = clock();
7     for (unsigned int i = 0; i < 1000000000; ++i) {
8         double a = i, b = i + 1, c = i + 2;
9         double d = i + 3, e = i + 4;
10        res = (res + a) + (b + c) + (d + e);
11    }
12    clock_t end = clock();
13    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
14    printf("res = %.0f in %lf s\n", res, time_spent);
15    return 0;
16 }
17 // gcc -O3 pipelining2.c -o pipelining2
18 // ./pipelining2
19 // output: res = 2500000007030048256 in 2.742885 s
```

Pipelining, Out of Order, Superscalar

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 const int SIZE = 100000;
6
7 void init(double* list)
8 {
9     for (int i = 0; i < SIZE; ++i)
10    {
11        list[i] = rand();
12    }
13 }
```

Pipelining, Out of Order, Superscalar

```
1 int main(int argc, char *argv[]) { // from Agner
2     double list[SIZE];
3     init(list);
4     clock_t begin = clock();
5     double sum = 0;
6     for (int i = 0; i < 100000; ++i)
7         for (int j = 0; j < SIZE; ++j)
8         {
9             sum += list[j];
10        }
11    clock_t end = clock();
12    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
13    printf("res = %.0lf in %lf s\n", sum, time_spent);
14    return 0;
15 }
16 // gcc -O3 pipelining3.c -o pipelining3
17 // ./pipelining3
18 // output: res = 10734586421274374144 in 4.173952 s
```

Pipelining, Out of Order, Superscalar

```

1 int main(int argc, char *argv[]) { // from Agner
2     double list[SIZE];
3     init(list);
4     clock_t begin = clock();
5     double sum1 = 0, sum2 = 0;
6     for (int i = 0; i < 100000; ++i)
7         for (int j = 0; j < SIZE; j += 2)
8             {
9                 sum1 += list[j];
10                sum2 += list[j + 1];
11            }
12     double sum = sum1 + sum2;
13     clock_t end = clock();
14     double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
15     printf("res = %.0lf in %lf s\n", sum, time_spent);
16     return 0;
17 }
18 // gcc -O3 pipelining4.c -o pipelining4
19 // ./pipelining4
20 // output: res = 10734586425374408704 in 2.063057 s

```

Branch Prediction

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <math.h>
5
6 int main(int argc, char *argv[])
7 {
8     const int SIZE = 5e8;
9     double sum = 0;
10    clock_t begin = clock();
11    for (int i = 0; i < SIZE; ++i)
12    {
13        if (test(i)) sum += sqrt(i);
14        else sum -= sqrt(i);
15    }
16    clock_t end = clock();
17    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
18    printf("res=%lf in %lf s\n", sum, time_spent);
19    return 0;
20 }
```

Branch Prediction

```
1 int test(int i)
2 {
3     return rand() & 1;
4 }
5
6 // gcc branch_prediction1.c -o branch_prediction1 -O3 -lm
7 // ./branch_prediction1
8 // output: res = -455212772.852015 in 5.574314 s
```

Branch Prediction

```
1 int test(int i)
2 {
3     rand();
4     return i & 1;
5 }
6
7 // gcc branch_prediction2.c -o branch_prediction2 -O3 -lm
8 // ./branch_prediction2
9 // output: res = 11180.719987 in 3.754499 s
```

Disk Access and Buffer

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4 int main() {
5     const int BUFF_SIZE = 1 << 16, WRITE_SIZE = 256;
6     char* buf = malloc(BUFF_SIZE);
7     int fd = open("res.txt", O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
8     if (fd == -1) exit(1);
9     for (int i = 0; i < BUFF_SIZE; i += WRITE_SIZE) {
10         if (write(fd, buf, WRITE_SIZE) != WRITE_SIZE) exit(1);
11     }
12     if (fsync(fd) == -1) exit(1);
13     if (close(fd) == -1) exit(1);
14     return 0;
15 }
16 // gcc -O3 write1.c -o write1
17 // time ./write1
18 // output: real 0m0.092s
19 //           user 0m0.000s
20 //           sys  0m0.004s
```

Disk Access and Buffer

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4
5 int main() {
6     const int BUFF_SIZE = 1 << 16, WRITE_SIZE = 256;
7     char* buf = malloc(BUFF_SIZE);
8     int fd = open("res.txt", O_CREAT|O_WRONLY | O_SYNC, S_IRUSR|S_IWUSR);
9     if (fd == -1) exit(1);
10    for (int i = 0; i < BUFF_SIZE; i += WRITE_SIZE) {
11        if (write(fd, buf, WRITE_SIZE) != WRITE_SIZE) exit(1);
12    }
13    if (close(fd) == -1) exit(1);
14    return 0;
15 }
16 // gcc -O3 write2.c -o write2
17 // time ./write2
18 // output: real 0m22.684s
19 //           user 0m0.000s
20 //           sys  0m0.000s
```

Interrupts

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4 #include <sys/time.h>
5
6 size_t count = 0;
7
8 void alarm_handler(int signo)
9 {
10     ++count;
11 }
```

Interrupts

```
1 int main()
2 {
3     struct itimerval delay;
4     signal(SIGALRM, alarm_handler);
5     delay.it_value.tv_sec = 0;
6     delay.it_value.tv_usec = 10;
7     delay.it_interval.tv_sec = 0;
8     delay.it_interval.tv_usec = 10;
9     if (setitimer(ITIMER_REAL, &delay, NULL))
10    {
11        perror("setitimer");
12        return EXIT_FAILURE;
13    }
14    while (count < 10)
15    {
16        printf("%lu\n", count);
17    }
18    return EXIT_SUCCESS;
19 }
```

Interrupts

```
1 // gcc -O3 interrupts.c -o interrupts
2 // ./interrupts
3
4 // 0
5 // 6      // 7
6 // 6      // 7
7 // 6      // 7
8 // 6      // 7
9 // 6      // 7
10 // 6      // 7
11 // 6      // 7
12 // 7      // 7
13 // 7      // 7
14 // 7      // 8
15 // 7      // 8
16 // 7      // 8
17 // 7      // 8
18 // 9
19 // 9
```

Virtual Memory

```
1 #include <stdio.h> #include <stdlib.h>
2 #include <unistd.h> #include <fcntl.h>
3 #include <sys/stat.h>
4
5 int main() {
6     int fd = open("file.txt", O_RDONLY);
7     if (fd == -1) exit(EXIT_FAILURE);
8     struct stat sb;
9     if (fstat(fd, &sb) == -1) exit(EXIT_FAILURE);
10    long res = 0;
11    for (int i = 0; i < 1e7; ++i) {
12        lseek(fd, rand() % sb.st_size, SEEK_SET);
13        char c;
14        if (read(fd, &c, 1) != 1) exit(EXIT_FAILURE);
15        res += c;
16    }
17    printf("res=%ld\n", res);
18    if (close(fd) == -1) exit(EXIT_FAILURE);
19    return EXIT_SUCCESS;
20 }
```

Virtual Memory

```
1 // gcc -O3 random_read1.c -o random_read1
2 // time ./random_read1
3 // output:
4 //      res = -4712456
5 //
6 //      real    0m2.997s
7 //      user    0m0.500s
8 //      sys     0m2.496s
```

Virtual Memory

```
1 #include <stdio.h> #include <stdlib.h>
2 #include <unistd.h> #include <fcntl.h> #include <sys/stat.h>
3
4 int main() {
5     int fd = open("file.txt", O_RDONLY);
6     if (fd == -1) exit(EXIT_FAILURE);
7     struct stat sb;
8     if (fstat(fd, &sb) == -1) exit(EXIT_FAILURE);
9     long res = 0;
10    char *addr = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
11    if (addr == MAP_FAILED) exit(EXIT_FAILURE);
12    for (int i = 0; i < 1e7; ++i) {
13        int index = rand() % sb.st_size;
14        char c = addr[index];
15        res += c;
16    }
17    printf("res=%ld\n", res);
18    if (close(fd) == -1) exit(EXIT_FAILURE);
19    return EXIT_SUCCESS;
20 }
```

Virtual Memory

```
1 // gcc -O3 random_read2.c -o random_read2
2 // time ./random_read2
3 // output:
4 //         res = -4712456
5 //
6 //         real    0m0.588s
7 //         user    0m0.580s
8 //         sys     0m0.004s
```

Shared Memory

```
1 #include <iostream>
2 #include <thread>
3
4 volatile long counter = 0;
5 const int N = 1e8;
6
7 void increment()
8 {
9     for (int i = 0; i < N; ++i)
10    ++counter;
11 }
12
13 void decrement()
14 {
15     for (int i = 0; i < N; ++i)
16    --counter;
17 }
```

Shared Memory

```
1 int main(int argc, char *argv[])
2 {
3     std::thread t1(increment);
4     std::thread t2(decrement);
5     t1.join();
6     t2.join();
7     std::cout << counter << std::endl;
8     return 0;
9 }
10
11 // g++ -std=c++11 shared1.cpp -o shared1 -O3 -lpthread
12 // ./shared1
13 // output: 4537659
14 // ./shared1
15 // output: -54344654
```

Shared Memory

```
1 #include <iostream>
2 #include <thread>
3 #include <atomic>
4
5 std::atomic<long> counter(0);
6 const int N = 1e8;
7
8 void increment() {
9     for (int i = 0; i < N; ++i)
10        ++counter;
11 }
12 void decrement() {
13     for (int i = 0; i < N; ++i)
14        --counter;
15 }
16 // g++ -std=c++11 shared2.cpp -o shared2 -O3 -lpthread
17 // ./shared2
18 // output: 0
19 // ./shared2
20 // output: 0
```

Memory Consistency

```
1 #include <iostream>
2 #include <thread>
3 #include <random>
4 #include <boost/interprocess/sync/interprocess_semaphore.hpp>
5
6 // from preshing
7 int X, Y;
8 int r1, r2;
9 boost::interprocess::interprocess_semaphore sem1(0), sem2(0);
```

Memory Consistency

```
1 void thread1()
2 {
3     std::default_random_engine engine;
4     std::uniform_int_distribution<int> d(0, 7);
5     for (;;)
6     {
7         sem1.wait();
8         while (d(engine));
9
10        X = 1;
11        asm volatile("" ::: "memory");
12        r1 = Y;
13
14        sem1.post();
15    }
16 }
```

Memory Consistency

```
1 void thread2()
2 {
3     std::default_random_engine engine;
4     std::uniform_int_distribution<int> d(0, 7);
5     for (;;)
6     {
7         sem2.wait();
8         while (d(engine));
9
10        Y = 1;
11        asm volatile("" ::: "memory");
12        r2 = X;
13
14        sem2.post();
15    }
16 }
```

Memory Consistency

```
1 int main(int argc, char *argv[])
2 {
3     std::thread t1(thread1);
4     std::thread t2(thread2);
5     int detected = 0;
6     for (int i = 1;; ++i)
7     {
8         X = 0;
9         Y = 0;
10        sem1.post(); sem2.post();
11        sem1.wait(); sem2.wait();
12        if (r1 == 0 && r2 == 0) {
13            ++detected;
14            std::cout << detected << " reorders detected after "
15                           << i << " iterations" << std::endl;
16        }
17    }
18    return 0;
19 }
```

Memory Consistency

```
1 // g++ -std=c++11 reorder.cpp -O3 -lpthread -lboost_system
2 // ./reorder
3 // output:
4 // 1 reorders detected after 1 iterations
5 // 2 reorders detected after 2 iterations
6 // 3 reorders detected after 3 iterations
7 // 4 reorders detected after 4 iterations
8 // 5 reorders detected after 5 iterations
9 // 6 reorders detected after 6 iterations
10 // 7 reorders detected after 7 iterations
11 // 8 reorders detected after 8 iterations
12 // 9 reorders detected after 9 iterations
13 // 10 reorders detected after 10 iterations
14 // 11 reorders detected after 11 iterations
15 // 12 reorders detected after 12 iterations
16 // 13 reorders detected after 13 iterations
17 // 14 reorders detected after 64 iterations
18 // 15 reorders detected after 65 iterations
19 // ...
20 // 56295 reorders detected after 374159 iterations
```

Memory Consistency

```
1 #include <iostream>
2 #include <thread>
3 #include <boost/interprocess/sync/interprocess_semaphore.hpp>
4
5 int value1, value2;
6 bool in_critical_section1;
7 bool in_critical_section2;
8 boost::interprocess::interprocess_semaphore sem1(0), sem2(0);
9
10 void thread1() {
11     for (;;) {
12         sem1.wait();
13
14         in_critical_section1 = true;
15         bool c2 = in_critical_section2;
16         if (!c2) value1 = 42;
17
18         sem1.post();
19     }
20 }
```

Memory Consistency

```
1 void thread2()
2 {
3     for (;;)
4     {
5         sem2.wait();
6
7         in_critical_section2 = true;
8         bool c1 = in_critical_section1;
9         if (!c1) value2 = 42;
10
11         sem2.post();
12     }
13 }
```

Memory Consistency

```
1 int main(int argc, char *argv[]) {
2     std::thread t1(thread1);
3     std::thread t2(thread2);
4     int detected = 0;
5     for (int i = 1;; ++i)
6     {
7         value1 = 0; value2 = 0;
8         in_critical_section1 = false;
9         in_critical_section2 = false;
10        sem1.post(); sem2.post();
11        sem1.wait(); sem2.wait();
12        if (value1 == 42 && value2 == 42) {
13            ++detected;
14            std::cout << detected
15                << " times in critical region together after "
16                << i << " iterations" << std::endl;
17        }
18    }
19    return 0;
20 }
```

Memory Consistency

```
1 // g++ -std=c++11 consistency.cpp -O3 -lpthread -lboost_system
2 // ./a.out
3 // output:
4 // 1 times in critical region together after 476 iterations
5 // 2 times in critical region together after 635 iterations
6 // 3 times in critical region together after 763 iterations
7 // 4 times in critical region together after 796 iterations
8 // 5 times in critical region together after 810 iterations
9 // 6 times in critical region together after 821 iterations
10 // 7 times in critical region together after 1068 iterations
11 // ...
12 // 48557 times in critical region together after 7074354 iterations
```

SIMD with SSE

```
1 #include <boost/simd/function/load.hpp>
2 #include <boost/simd/function/sum.hpp>
3 #include <boost/simd/pack.hpp>
4 #include <chrono>
5
6 // from boost simd
7
8 template <typename Value>
9 Value scaldot(Value* first1, Value* last1, Value* first2)
10 {
11     Value v(0);
12     for (; first1 < last1; ++first1, ++first2)
13     {
14         v += (*first1) * (*first2);
15     }
16     return v;
17 }
```

SIMD with SSE

```
1 template <typename Value>
2 Value simddot(Value* first1, Value* last1, Value* first2) {
3     namespace bs = boost::simd; using pack_t = bs::pack<Value>;
4     pack_t tmp{0};
5     int card = pack_t::static_size;
6
7     for (; first1 + card <= last1; first1 += card, first2 += card)
8     {
9         pack_t x1 = bs::load<pack_t>(first1);
10        pack_t x2 = bs::load<pack_t>(first2);
11        tmp = tmp + x1 * x2;
12    }
13    Value dot_product = bs::sum(tmp);
14
15    for (; first1 < last1; ++first1, ++first2)
16    {
17        dot_product += (*first1) * (*first2);
18    }
19    return dot_product;
20 }
```

SIMD with SSE

```
1 int main() {
2     namespace bs = boost::simd; using pack_t = bs::pack<float>;
3     const size_t SIZE = 1e6;
4     std::vector<float> v1(SIZE), v2(SIZE);
5     std::iota(v1.begin(), v1.end(), 0);
6     std::iota(v2.begin(), v2.end(), 1);
7
8     volatile double res = 0;
9     using namespace std::chrono;
10    const steady_clock::time_point begin = steady_clock::now();
11    for (int i = 0; i < 10000; ++i)
12    {
13        res += scaldot(v1.data(), v1.data() + size, v2.data());
14    }
15    const steady_clock::time_point end = steady_clock::now();
16
17    std::cout << "res=" << res << " in "
18        << duration_cast<milliseconds>(end - begin).count()
19        << " ms" << std::endl;
20 }
```

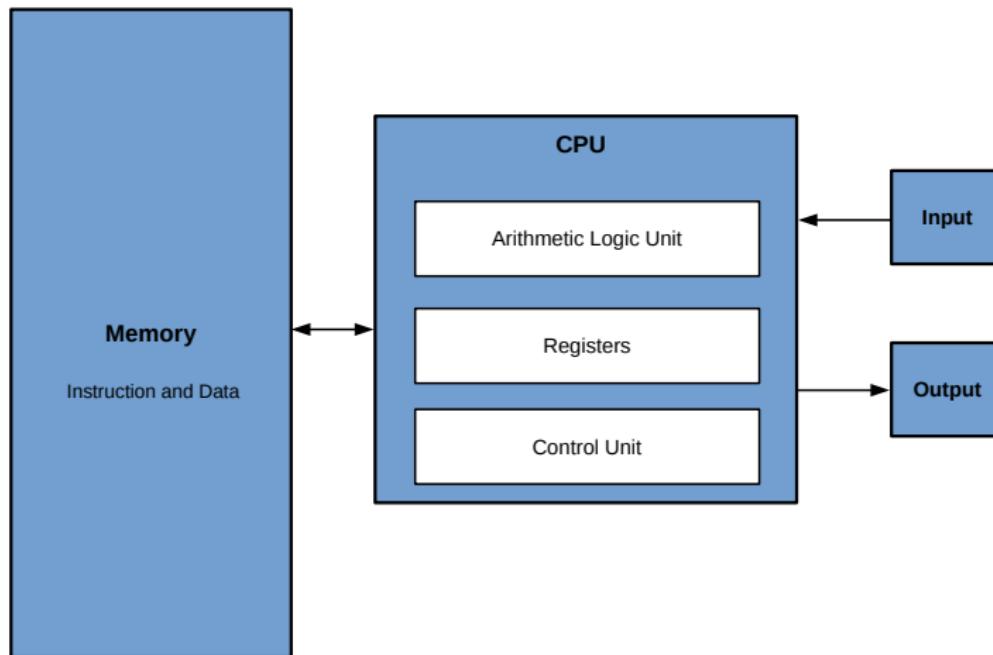
SIMD with SSE

```
1 // g++ simd1.cpp -msse4.2 -std=c++11 -O3 -o simd1 \
2 //     -I path to boost simd \
3 //     -I path to boost
4 // ./simd1
5 // res = 3.33381e+21 in 8450 ms
```

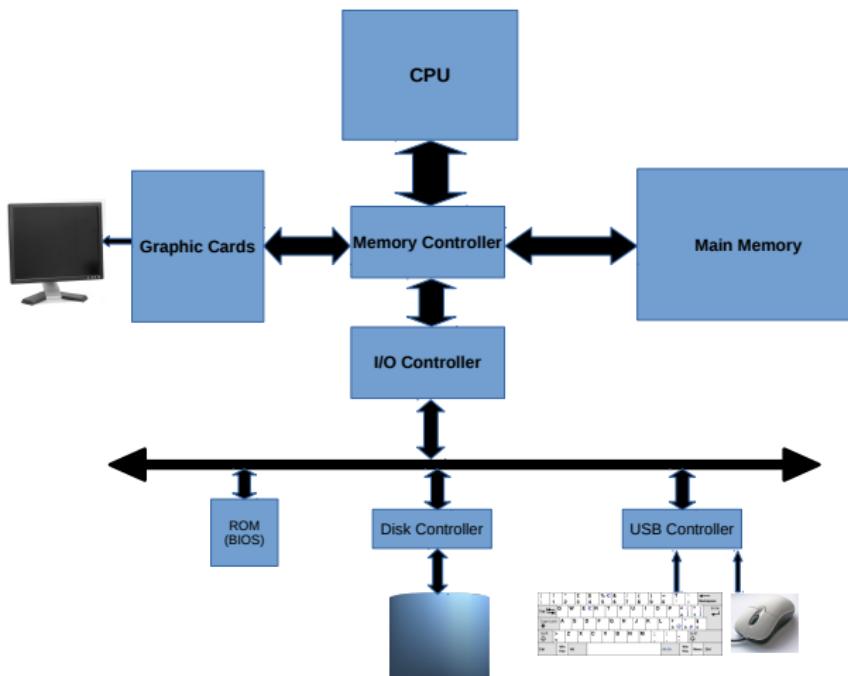
SIMD with SSE

```
1 int main()
2 {
3     // ...
4     for (int i = 0; i < 10000; ++i)
5     {
6         res += simddot(v1.data(), v1.data() + size, v2.data());
7     }
8     // ...
9 }
10
11 // g++ SIMD2.cpp -msse4.2 -std=c++11 -O3 -o SIMD2
12 //     -I path to SIMD
13 //     -I path to boost
14 // ./SIMD2
15 // res = 3.33332e+21 in 2622 ms
```

The Von Neumann Model

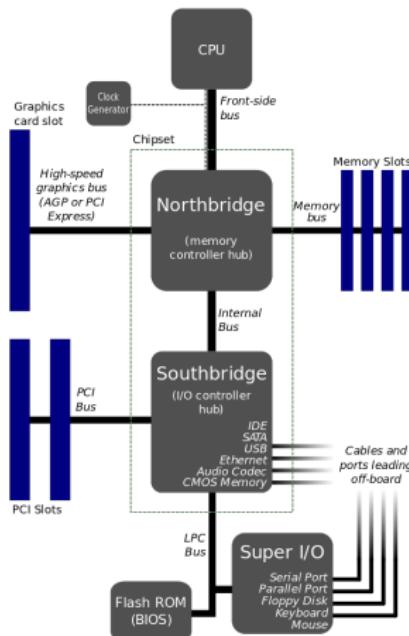


The Von Neumann Model

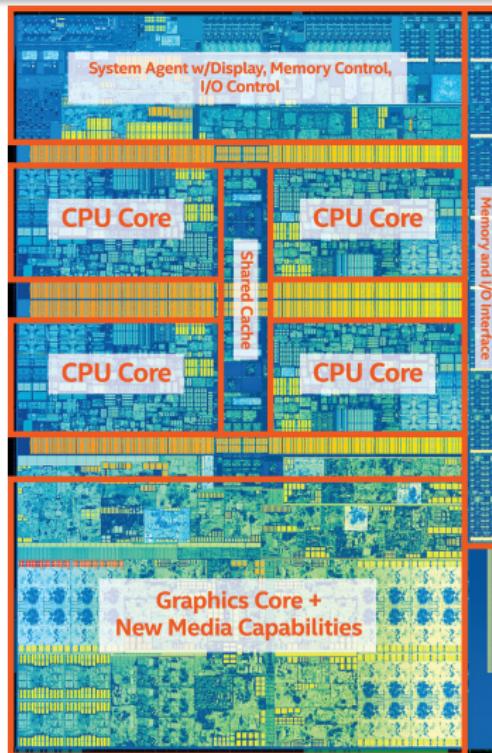


Real Machine

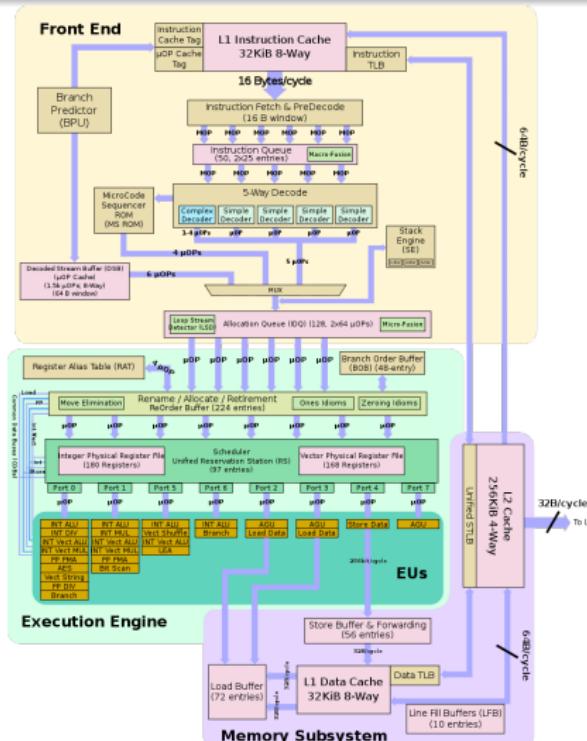
From Wikipedia



Intel Kaby Lake Die



Intel Kaby Lake Core



ISA and Microarchitecture

- Instruction Set Architecture: The interface between the software and hardware. It provides all the information needed for a programmer to write a program in machine language.
 - Instruction set.
 - Registers.
 - Memory addressing.
 - Memory ordering.
 - ...
- Microarchitecture: Implementation of the architecture.
 - Number of caches.
 - Cache sizes.
 - Number of cores.
 - Clock frequency.
 - ...

UltimateRISC: Single Instruction Machine

- Only one instruction!
- `urisc A, B, JUMP`
 - A, B and JUMP are memory addresses.
 - $*A \leftarrow *A - *B$.
 - If result is < 0 the program counter is set to JUMP, otherwise the program counter is incremented.
- Memory addresses are 10 bits wide.
- Integers are represented in 30 bits two's complement.

UltimateRISC: Single Instruction Machine

We can simulate `mov A, B` which computes $*A \leftarrow *B$ with

```
0: urisc A, A, 1
1: urisc C, C, 2
2: urisc C, B, 3
3: urisc A, C, 4
```

with C a memory address > 3

C: don't care

Exercice 1

Simulate the instruction

add A, B

which computes

$$*A \leftarrow *A + *B.$$

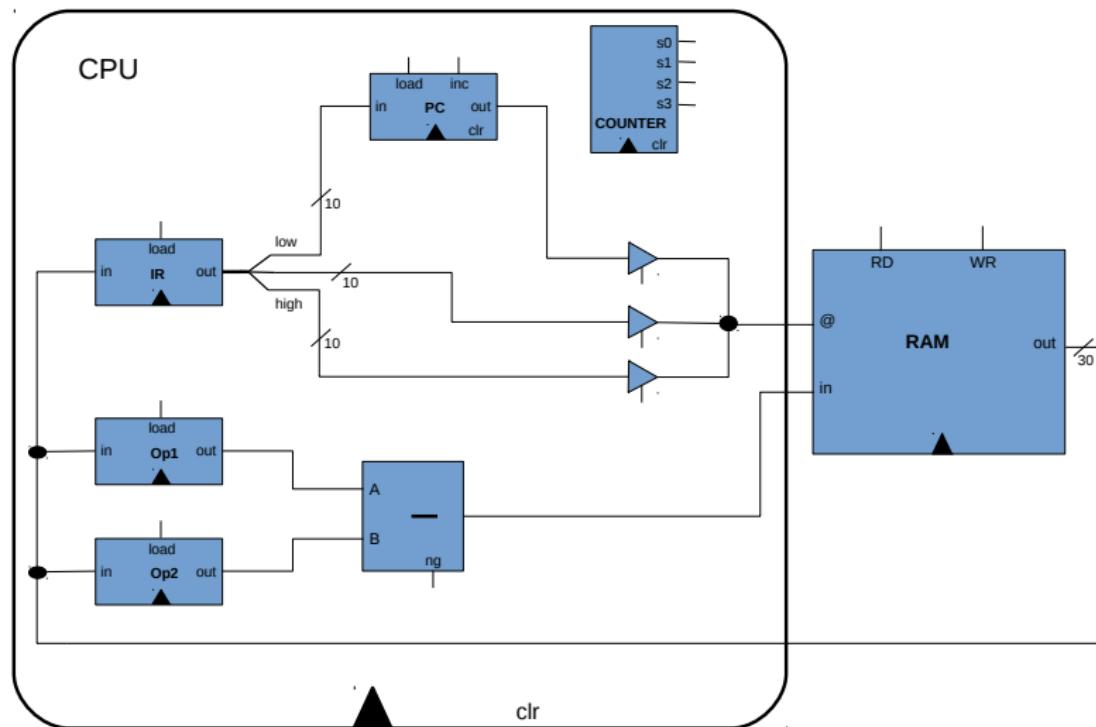
Solution

0: **urisc** C, C, 1
1: **urisc** C, B, 2
2: **urisc** A, C, 3

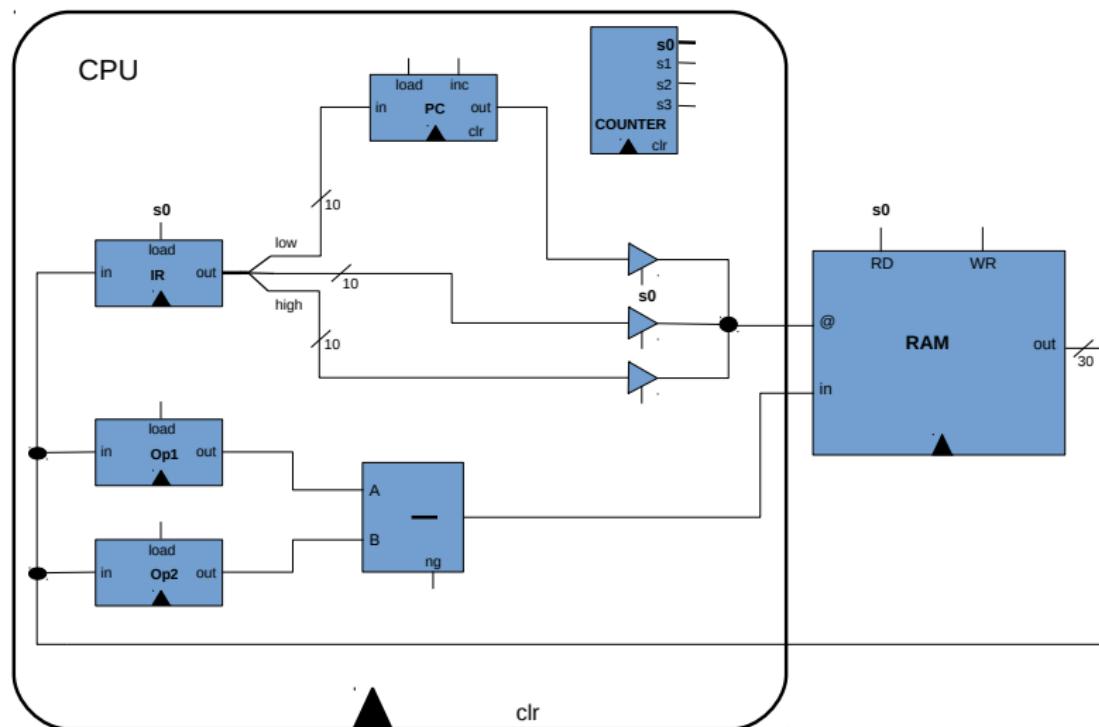
with C a memory address > 2

C: don't care

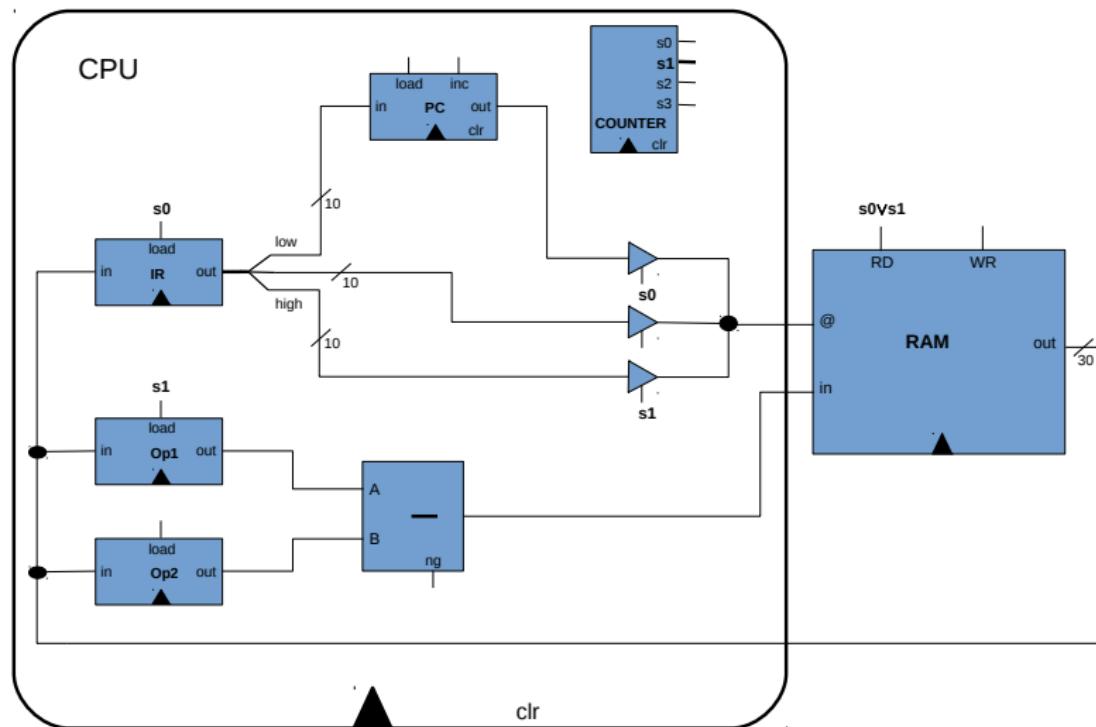
UltimateRISC



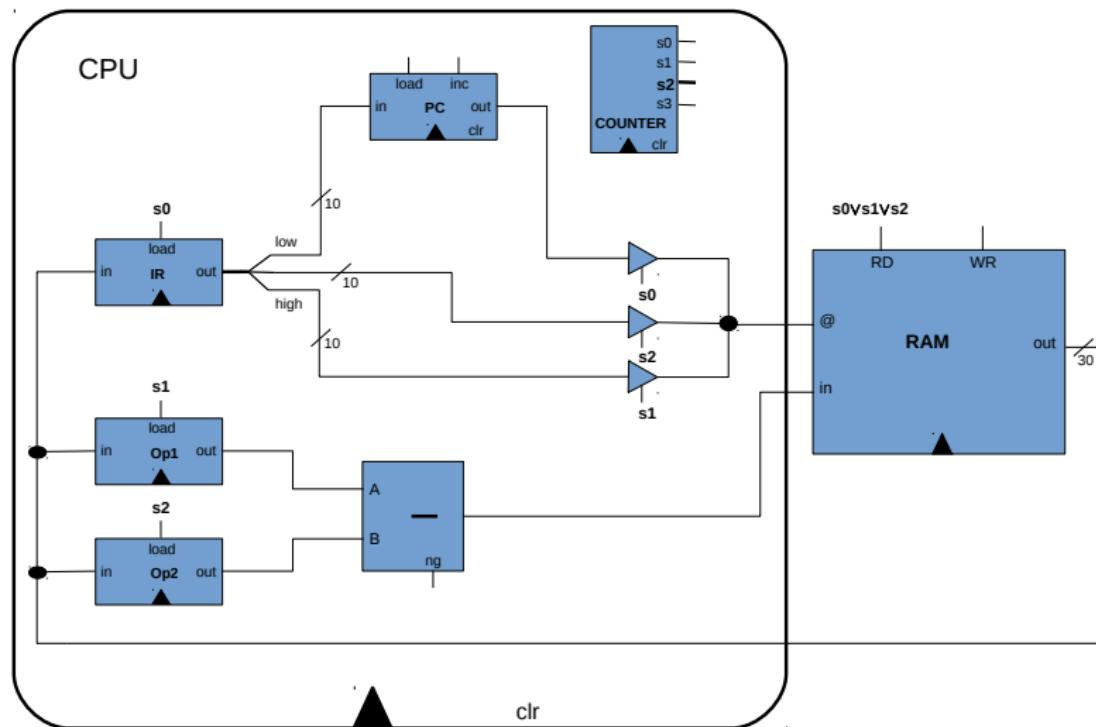
UltimateRISC



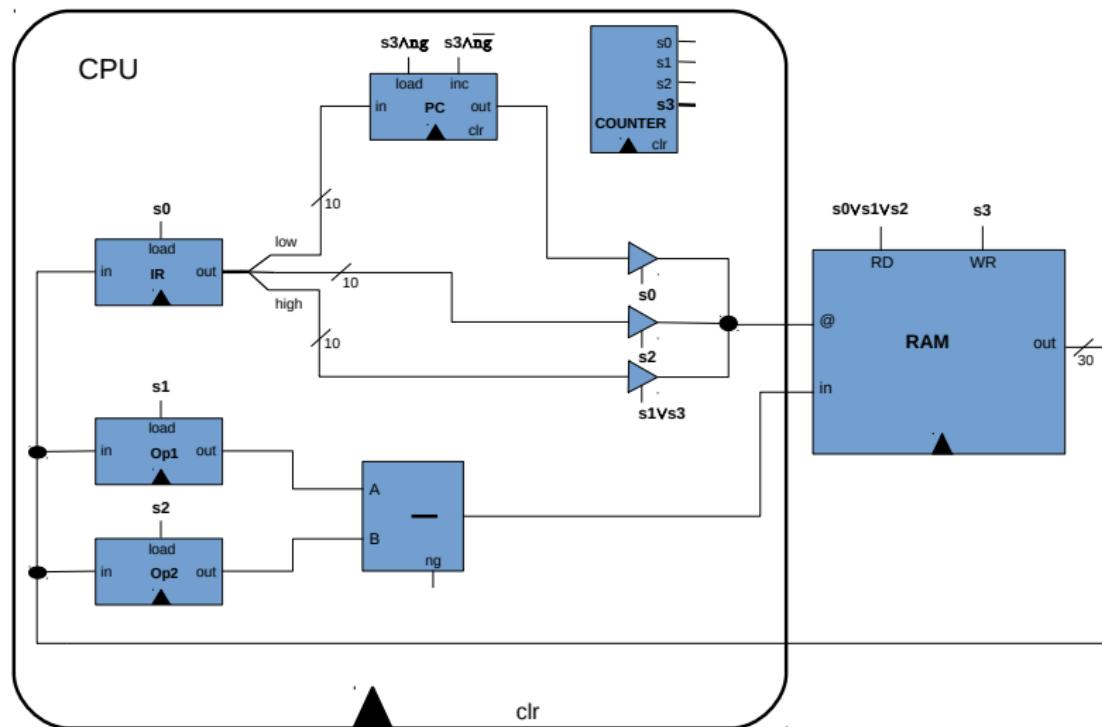
UltimateRISC



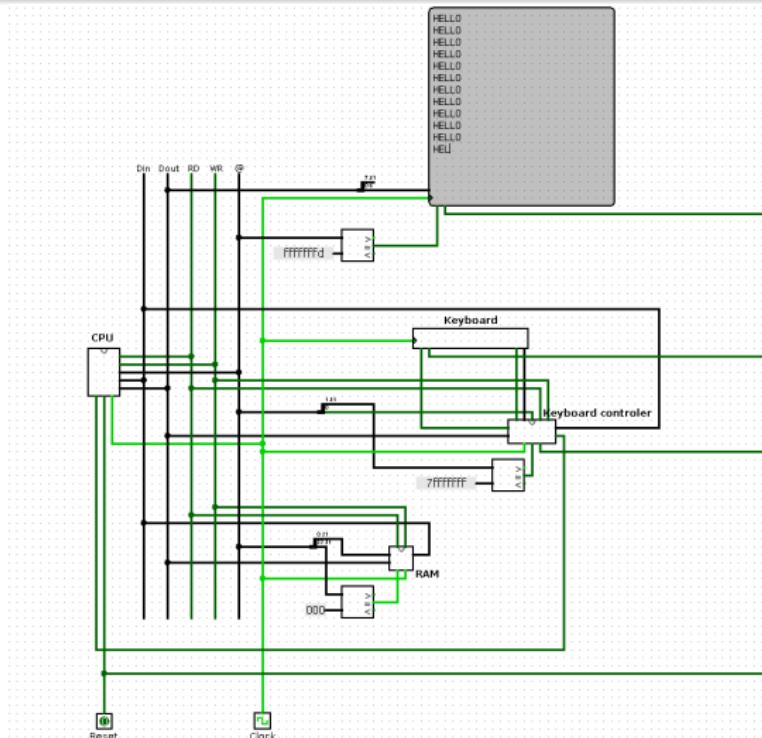
UltimateRISC



UltimateRISC



MIPS32 with Logisim



Unsigned

- The range of values of an n -bits unsigned integer is: 0 to $2^n - 1$.
 - for $n = 8$, the maximum value is $2^8 - 1 = 255$.
 - for $n = 16$, the maximum value is $2^{16} - 1 = 65535$.
 - for $n = 32$, the maximum value is $2^{32} - 1 = 4\ 294\ 967\ 295$.
 - for $n = 64$, the maximum value is $2^{64} - 1 = 18\ 446\ 744\ 073\ 709\ 551\ 615$.
- C unsigned data types

Type	Max value (at least) (as defined by the C11 norm)	Size (in bytes) on x86-64
<code>unsigned char</code>	$2^8 - 1 = 255$	1
<code>unsigned short</code>	$2^{16} - 1 = 65535$	2
<code>unsigned int</code>	$2^{16} - 1 = 65535$	4
<code>unsigned long int</code>	$2^{32} - 1 = 4294967295$	8
<code>unsigned long long int</code>	$2^{64} - 1 = 18446744073709551615$	8

Unsigned Addition, Subtraction, Multiplication

For n -bits unsigned numbers, the operation $(+, -, *)$ is done **modulo 2^n** .

```
1 #include <stdio.h>
2 int main(int argc, char *argv[]) {
3     unsigned char n1 = 128;
4     unsigned char n2 = 10;
5     printf("%3hu + %3hu = %hu\n", n1, n1, n1 + n1);
6     printf("%3hu - %3hu = %hu\n", n2, n1, n2 - n1);
7     printf("%3hu * %3hu = %hu\n", n1, n2, n1 * n2);
8     printf("-%hu = %hu\n", n1, -n1);
9     printf("%u - 1 = %hu\n", -1);
10    return 0;
11 }
12 // gcc -O3 unsigned.c -o unsigned && ./unsigned
13 // output:
14 //      128 + 128 = 0
15 //      10 - 128 = 138
16 //      128 * 10 = 0
17 //      -128 = 128
18 //      -1 = 255
```

Bit Manipulation

- We can check if the j -th bit of an integer S is on with

$$S \ \& \ (1 \ll j) \neq 0.$$

- For example, if $S = 50$ and $j = 3$,

		bit number
		5 4 3 2 1 0
50	=	1 1 0 0 1 0
$(1 \ll 3)$	=	0 0 1 0 0 0
<hr/>		
50	$\&$	$(1 \ll 3) =$ 0 0 0 0 0 0

Bit Manipulation

- To turn on the j -th bit of S we do: $S |= (1 << j)$.
- For example, if $S = 50$ and $j = 3$,

		bit number
		5 4 3 2 1 0
S	=	1 1 0 0 1 0
$(1 << 3)$	=	0 0 1 0 0 0
<hr/>		
$S = (1 << 3)$	=	1 1 1 0 1 0

- To turn off the j -th bit of S we do: $S &= \sim(1 << j)$.
- For example, if $S = 58$ and $j = 3$,

		bit number
		5 4 3 2 1 0
S	=	1 1 1 0 1 0
$\sim(1 << 3)$	=	1 1 0 1 1 1
<hr/>		
$S &= \sim(1 << 3)$	=	1 1 0 0 1 0

Bit Manipulation

- To toggle the j -th bit of S we do: $S \hat{=} (1 \ll j)$.
- For example, if $S = 50$ and $j = 3$,

		bit number
		5 4 3 2 1 0
S	=	1 1 0 0 1 0
$(1 \ll 3)$	=	0 0 1 0 0 0
<hr/>		
$S \hat{=} (1 \ll 3)$	=	1 1 1 0 1 0

- For example, if $S = 58$ and $j = 3$,

		bit number
		5 4 3 2 1 0
50	=	1 1 1 0 1 0
$(1 \ll 3)$	=	0 0 1 0 0 0
<hr/>		
$S \hat{=} (1 \ll 3)$	=	1 1 0 0 1 0

Bit Manipulation

- To turn on all bits in a set of size n we do

$$(1 \ll n) - 1.$$

- For example, if $n = 5$,

		bit number
		5 4 3 2 1 0
$(1 \ll 5)$	=	1 0 0 0 0 0
1	=	0 0 0 0 0 1
<hr/>		
$(1 \ll 5) - 1$	=	0 1 1 1 1 1

Exercice 2

Write the function

1 `void print_subsets(const string& set)`

which print all the subsets of the string set. Your function must be non-recursive. We suppose that $set.size() \leq 30$.

For example, `print_subsets("abc")` gives (any order is good)

```
{ }  
{ a }  
{ b }  
{ a b }  
{ c }  
{ a c }  
{ b c }  
{ a b c }
```

Solution Implementation

```
1 void print_subsets(const string& set)
2 {
3     for (unsigned int subset = 0; subset < (1U << set.size()); ++subset)
4     {
5         cout << "{";
6         for (unsigned int j = 0; j < set.size(); ++j)
7         {
8             if (subset & (1 << j)) cout << "\u2225" << set[j];
9         }
10        cout << "\u2225}" << endl;
11    }
12 }
13
14 int main(int argc, char *argv[])
15 {
16     print_subsets("abc");
17     return 0;
18 }
```

Byte Ordering

- Memory is byte oriented.
- How do we store multi-bytes values?
- Two main conventions
 - Big Endian: Least significant byte has highest address (Sun, Internet for example).
 - Little Endian: Least significant byte has lowest address (Intel for example).

Byte Ordering

store **0x12345678** at address **0**

big endian

0	0x12
1	0x34
2	0x56
3	0x78

little endian

0	0x78
1	0x56
2	0x34
3	0x12

Endianness Example

```
1  char chars[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
2  int  ints[] = { 0x00112233, 0x44556677, 0x8899aabb, 0xccddeeff };
3
4  int main() {}
5
6
7  // gcc endianness.c -o endianness
8  // objdump -sj .data endianness
9  // ...
10 // 201010 33221100 77665544 bbaa9988 ffeeddcc
11 // 201020 00010203 04050607
```

Exercice 3

Write a program to test the endianness of your computer.

Solution

```
1 #include <stdio.h>
2
3 int little_endian()
4 {
5     int i = 1;
6     char *p = (char *)&i;
7     return p[0] == 1;
8 }
9
10 int main()
11 {
12     if (little_endian())
13         printf("little_endian\n");
14     else
15         printf("big_endian\n");
16     return 0;
17 }
```

Looping Downward with Unsigned

```
1 unsigned int count = /* number of elements */;  
2 for (unsigned int i = count - 2; i >= 0; --i)  
3 {  
4     a[i] += a[i + 1];  
5 }
```

Where is the bug in this code?

Looping Downward with Unsigned

```
1 unsigned int count = /* number of elements */;  
2 for (unsigned int i = count - 2; i < count; --i)  
3 {  
4     a[i] += a[i + 1];  
5 }
```

Signed

- Two's complement representation.
- Let b be a binary string of length n , the value of b in the two's complement representation is

$$-b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i$$

- For example, for $n = 8$ and $b = 10010011$, we have

$$b = -2^7 + 2^4 + 2 + 1 = -109$$

- The range of values of an n -bits signed integer is: -2^{n-1} to $2^{n-1} - 1$.
 - for $n = 8$, the range is $[-2^7, 2^7 - 1] = [-128, 127]$.
 - for $n = 16$, the range is $[-2^{15}, 2^{15} - 1] = [-32768, 32767]$.
 - for $n = 32$, the maximum value is

$$[-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]$$

- for $n = 64$, the maximum value is

$$[-2^{63}, 2^{63} - 1] = [-9223372036854775808, 9223372036854775807]$$

C Language Signed Data Types

Type	Range (at least) (as defined by the C11 norm)	Size (in bytes) on x86-64
<code>char</code>	$[-(2^7 - 1), 2^7 - 1]$	1
<code>short</code>	$[-(2^{15} - 1), 2^{15} - 1]$	2
<code>int</code>	$[-(2^{15} - 1), 2^{15} - 1]$	4
<code>long int</code>	$[-(2^{31} - 1), 2^{31} - 1]$	8
<code>long long int</code>	$[-(2^{63} - 1), 2^{63} - 1]$	8

Note: Even if the standard does not enforce it, compilers use two's complement (range $[-2^{n-1}, 2^{n-1} - 1]$).

Signed Addition, Subtraction, Multiplication and Shift

- Addition, subtraction, multiplication and left shift are the same operations on bit level as their unsigned counterparts.
- Right shift propagates the sign bit.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int n1 = -1234;
6     int n2 = 654321;
7     int res = (int)((unsigned)n1 + (unsigned)n2);
8     printf("%10d\u003d\u003d\n", n1 + n2, res); // 653087 = 653087
9     res = (int)((unsigned)n1 - (unsigned)n2);
10    printf("%10d\u003d\u003d\n", n1 - n2, res); // -655555 = -655555
11    res = (int)((unsigned)n1 * (unsigned)n2);
12    printf("%10d\u003d\u003d\n", n1 * n2, res); // -807432114 = -807432114
13    printf("-1\u003d\u003d\n", -1 >> 1); // -1 >> 1 = -1
14
15 }
```

Negative Values

- To get the negative value of a two's complement number,
 - We invert all the bits of the number.
 - We add one to the resulting number.
- Example with an 8-bit number,
 - $127 = (01111111)_2$.
 - We invert all the bits and we get: $(10000000)_2$.
 - Finally, we add one to get: $(10000001)_2 = -127$.
- Why does it work?
 - Let x be an 8-bit two's complement number. Let's say that $x = 100$.
 - $-x = 1 + (-1 - x)$.
 - All bits are set to 1 in the two's complement representation of -1 . So we have,

$$\begin{array}{r} 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \\ - 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \\ \hline 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \end{array} \quad = -101$$

- Because all bits are set to 1 in the two's complement representation of -1 , the subtraction flips bits in the two's complement representation of x .

Detecting Overflow

Operation	A	B	Overflow iff the result is
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Conversion from Signed to Unsigned

- We just need to reinterpret the bit pattern.
- If sign bit is 0, the number is the same in both representation.
- If sign bit is 1, we add 2^n from signed to unsigned and subtract 2^n from unsigned to signed.

Bit pattern	Unsigned	Signed
01111111	127	127
10000001	129	$129 - 2^8 = -127$
11111111	$-1 + 2^8 = 255$	-1

Conversion from Signed to Signed with Different Sizes

- To convert a n -bits signed integer to a larger m -bits integer, you propagate the sign bit (bit_{n-1}). The value is unchanged.
 - 8-bits $11111110 = -2$ to 16-bits $1111111111111110 = -2$.
 - 8-bits $00000111 = 7$ to 16-bits $000000000000000111 = 7$.
- To convert a n -bits signed integer to a smaller m -bits integer, you cut the $(n - m)$ higher bits. The value can be changed.
 - 16-bits $1000000000000000 = -32768$ to 8-bits $00000000 = 0$.

Expression Containing Signed and Unsigned

If there is a mix of unsigned and signed in an expression, some conversions are applied.

The C11 norm says that

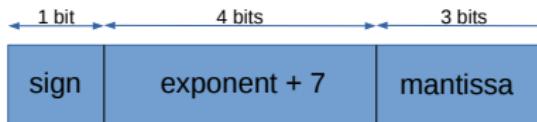
- “The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any”.
- “If the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type is converted to the type of the operand with unsigned integer type”.

```
1 #include <stdio.h>
2 int main() {
3     printf("%d\n", -1 > 0U);
4     // -1 is an int and 0U an unsigned int
5     // output: ?
6     printf("%d\n", -1L > 0U);
7     // -1L is a long int and 0U an unsigned int
8     // output: ?
9 }
```

Floating Point Numbers

(Based on <http://www.toves.org/books/float/>)

- Based on scientific notation.
 - $9.625_{(10)} = 1001.101_{(2)}$ and $1001.101_{(2)} = 1.001101 \times 2^3$ in scientific notation.
 - 1.001101 is called the **mantissa**, and 3 is called the **exponent**.
- Let's illustrate the floating point encoding with an 8-bits number.



- For $-5.5_{(10)}$ we have
 - $-5.5_{(10)} = -101.1_{(2)} = -1.011 \times 2^2$.
 - We choose 1 for the **sign bit**, because -5.5 is negative.
 - We add 7 to the exponent, we get $9_{(10)} = 1001_{(2)}$ for the **exponent** part.
 - The three bit of the **mantissa** are the ones following the leading 1. We get 011.
 - The final encoding is 11001011.

Exercice 4

Encode -0.171875 and 42 in our 8-bits floating point format.

Solution

- **-0.171875**

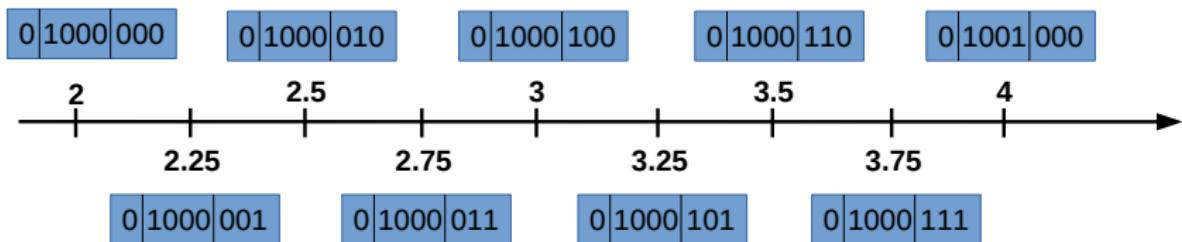
- $-0.171875 = -\left(\frac{1}{8} + \frac{1}{32} + \frac{1}{64}\right) = -0.001011_{(2)}$.
- $-0.001011 = -1.011 \times 2^{-3}$.
- Final encoding: 10100011.

- **42**

- $42 = 32 + 8 + 2 = 101010_{(2)}$.
- $101010 = 1.01010 \times 2^5$.
- We need to round to the nearest.
 - But here, we have two choices: 1.010 or 1.011.
 - To avoid cumulating errors, we round ties to even and so we get 1.010.
- Final encoding: 01100010.

Precision

(Based on http://fabiensanglard.net/floating_point_visually_explained/)

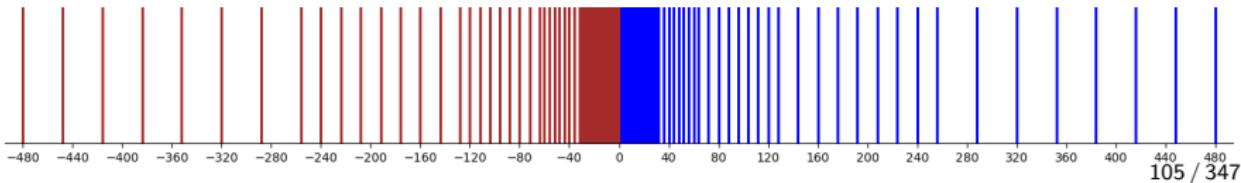


Precision

- Consider the 8-bits floating point numbers of the form 00110xxx.
 - The exponent is 2^{-1} . Those numbers are in the range $[0.5, 1)$, 00110000 to 01110000.
 - This range is divided in 8 steps of length $\frac{1-0.5}{8} = 0.0625$ (0.5, 0.5625, 0.625, 0.6875, 0.75, 0.8125, 0.875, 0.9375).
- For the 8-bits floating point numbers of the form 01110xxx, now we have
 - The exponent is 2^8 . Those numbers are in the range $[128, 256)$, 01110000 to 11110000.
 - This range is divided in 8 steps of length $\frac{256-128}{8} = 16$.
- We can see that as the exponent grows, the floating points are farther apart from one another.

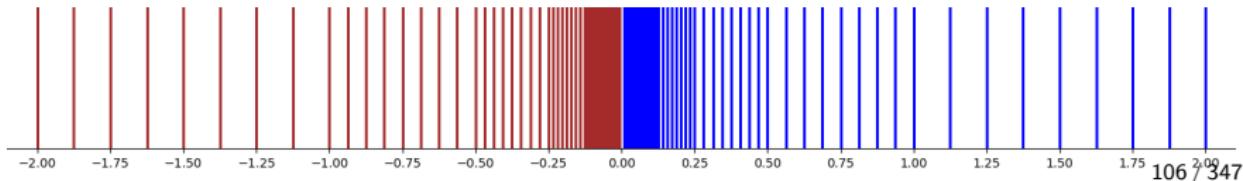
Precision

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 def sign(i):
4     return -1 if i & 0x80 else 1
5 def exponent(i):
6     return ((i & 0x78) >> 3) - 7
7 def mantissa(i):
8     return i & 7
9 def to_8bit_float(i):
10    return sign(i) * (1 + mantissa(i) * 0.125) * 2**exponent(i)
11
12 for i in range(0x80, 0xff + 1):
13     plt.plot((to_8bit_float(i), to_8bit_float(i)), (0, 3), c='brown', linewidth=0.75, aa=False)
14 for i in range(0, 0x7f + 1):
15     plt.plot((to_8bit_float(i), to_8bit_float(i)), (0, 3), c='blue', linewidth=0.75, aa=False)
16
17 plt.axis([-520, 520, 0, 5])
18 plt.xticks(np.arange(-520, 520, 40))
19 plt.show()
```



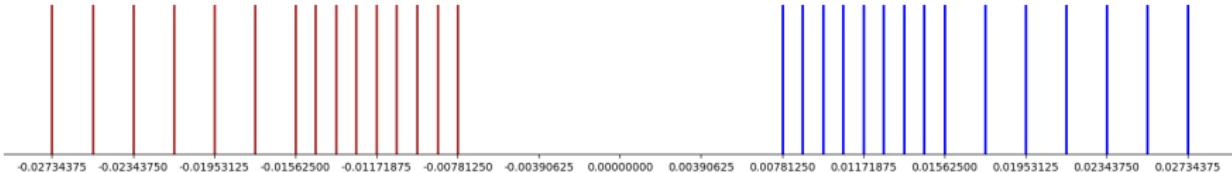
Precision

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 def sign(i):
4     return -1 if i & 0x80 else 1
5 def exponent(i):
6     return ((i & 0x78) >> 3) - 7
7 def mantissa(i):
8     return i & 7
9 def to_8bit_float(i):
10    return sign(i) * (1 + mantissa(i) * 0.125) * 2**exponent(i)
11
12 for i in range(0x80, 0xdf):
13     plt.plot((to_8bit_float(i), to_8bit_float(i)), (0, 3), c='brown', linewidth=0.75, aa=False)
14 for i in range(0, 0x5f):
15     plt.plot((to_8bit_float(i), to_8bit_float(i)), (0, 3), c='blue', linewidth=0.75, aa=False)
16
17 plt.axis([-2.25, 2.25, 0, 5])
18 plt.xticks(np.arange(-2.25, 2.25, 0.25))
19 plt.show()
```



Precision

```
1 # ...
2 for i in range(0x80, 0x8f):
3     plt.plot((to_8bit_float(i), to_8bit_float(i)), (0, 3), c='brown', linewidth=0.75, aa=False)
4 for i in range(0, 0x0f):
5     plt.plot((to_8bit_float(i), to_8bit_float(i)), (0, 3), c='blue', linewidth=0.75, aa=False)
6
7 plt.axis([-0.03, 0.03, 0, 5])
8 plt.xticks(np.arange(-0.02734375, 0.027348, 2**-8))
9 locs, _ = plt.xticks()
10 labels = ['%.8f' % v for v in locs]
11 plt.xticks(locs, labels)
12
13 plt.show()
```



There's a large gap between -0.0078125 and 0.0078125 and we can't represent zero!

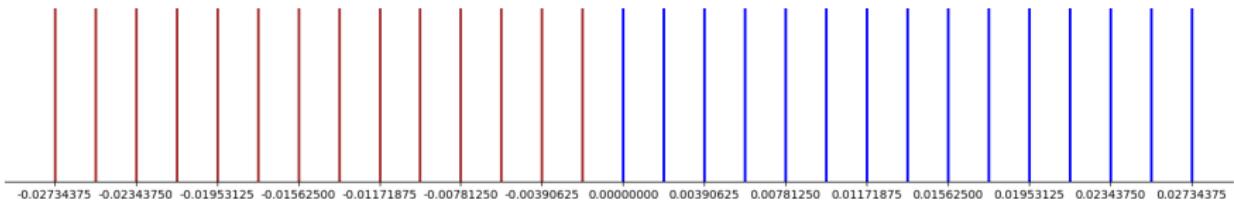
Denormalized Floating Point Numbers

- When the exponent is 0, the number is in denormalized form and,
 - We only subtract 6 and not 7.
 - The mantissa is not in its normal form, the 1 to the left of the decimal point is replaced by a 0.
- We use -6 for the exponent, to make a smooth transition with the normalized numbers.
 - The smallest positive normalized number is 00001000 which is $2^{-6} = 0.015625$.
 - The biggest positive normalized number is 00000111 which is $2^{-6} \times (0.5 + 0.25 + 0.125) = 0.013671875$.
- We have two zeroes, 00000000 (0) and 10000000 (-0).

Denormalized Floating Point Numbers

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 def sign(i):
4     return -1 if i & 0x80 else 1
5 def exponent(i):
6     e = (i & 0x78) >> 3
7     if e == 0:
8         return -6, True
9     else:
10        return e - 7, False
11 def mantissa(i):
12     return i & 7
13 def to_8bit_float(i):
14     e, denormalized = exponent(i)
15     return sign(i) * ((0 if denormalized else 1) + mantissa(i) * 0.125) * 2**e
16 n = 20
17 for i in range(0x80, 0x8f):
18     plt.plot((to_8bit_float(i), to_8bit_float(i)), (0, 3), c='brown', linewidth=0.75, aa=False)
19 for i in range(0, 0x0f):
20     plt.plot((to_8bit_float(i), to_8bit_float(i)), (0, 3), c='blue', linewidth=0.75, aa=False)
21 plt.axis([-0.03, 0.03, 0, 5])
22 plt.xticks(np.arange(-0.02734375, 0.027348, 2**-8))
23 locs, _ = plt.xticks()
24 labels = ['%.8f' % v for v in locs]
25 plt.xticks(locs, labels)
26 plt.show()
```

Denormalized Floating Point Numbers



Special Floating Point Numbers

- If the exponent is all ones and the mantissa is all zeroes, then the number represents infinity or negative infinity, depending on the sign bit. This value results from an overflow. For example,
 - `float v = 1.0 / pow(2, -128);`
 - `float v = pow(2, 128);`
 - $1.0/0.0$.
- If the exponent is all ones, and the mantissa has some non-zero bits, then the number represents “not a number”, written as NaN. For example,
 - $0.0/0.0$.
 - $\infty - \infty$.
 - ∞/∞ .

IEEE 754 Format

Format	sign	exponent	mantissa	exponent excess
Our 8-bit	1	4	3	7
IEEE 32-bit	1	8	23	127
IEEE 64-bit	1	11	52	1023
IEEE 128-bit	1	15	112	16383

Exercice 5

- 1 What's the biggest number (different from ∞) we can represent with the IEEE 64-bit format?
- 2 What's the smallest positive number we can represent with the IEEE 64-bit format?

Solution

- 1 The biggest number is

$$\begin{aligned}
 &= 2^{2046-1023} \times (\sum_{i=0}^{52} 2^{-i}) \\
 &= 2^{1023} \times \frac{1-2^{-53}}{1-2^{-1}} \\
 &\approx 1.79769313486232e+308
 \end{aligned}$$

- 2 The smallest positive number is

$$= 2^{-1022} \times 2^{-52}$$

$$\approx 4.94065645841247 \times 10^{-324}$$

Floating Point Addition

(From David Goldberg, Computer Arithmetic)

Let a_1 and a_2 be the two numbers to be added. The notations e_i and s_i are used for the exponent and significand of the addends a_i . This means that the floating-point inputs have been unpacked and that s_i has an explicit leading bit. To add a_1 and a_2 , perform these eight steps:

- ① If $e_1 < e_2$ swap the operands. This ensures that the difference of the exponents satisfies $d = e_1 - e_2 \geq 0$. Tentatively, set the exponent of the result to e_1 .
- ② If the signs of a_1 and a_2 differ, replace s_2 by its two's complement.
- ③ Place s_2 in a p -bit register (where p is the number of bits of s_2) and shift it $d = e_1 - e_2$ places to the right (shifting in 1's if s_2 was complemented in the previous step). From the bits shifted out, set g (guard) to the most-significant bit, set r (round) to the next most-significant bit, and set s (sticky) to the OR of the rest.
- ④ Compute a preliminary significand $S = s_1 + s_2$ by adding s_1 to the p -bit register containing s_2 . If the signs of a_1 and a_2 are different, the most-significant bit of S is 1, and there was no carry-out, then S is negative. Replace S with its two's complement. This can only happen when $d = 0$.

Floating Point Addition

- 5 Shift S as follows. If the signs of a_1 and a_2 are the same and there was a carryout in step 4, shift S right by one, filling in the high-order position with 1 (the carry-out). Otherwise, shift it left until it is normalized. When left-shifting, on the first shift fill in the low-order position with the g bit. After that, shift in zeros. Adjust the exponent of the result accordingly.
- 6 Adjust r and s . If S was shifted right in step 5, set $s := g \vee r \vee s$ and set $r :=$ low-order bit of S before shifting. If there was no shift, set $r := g$, $s := r \vee s$. If there was a single left shift, don't change r and s . If there were two or more left shifts, $r := 0$, $s := 0$.
- 7 Round S : Let p_0 the low order bit of S . Add 1 to S iff $((r \wedge p_0) \vee (r \wedge s))$. If rounding causes carry-out, shift S right and adjust the exponent. This is the significand of the result.
- 8 Compute the sign of the result with the following table, where the *swap* column refers to swapping the operands in step 1, while the *compl* column refers to performing a two's complement in step 4. Blanks are "don't care":

swap	compl	sign(a_1)	sign(a_2)	sign(result)
Yes		+	-	-
Yes		-	+	+
No	No	+	-	+
No	No	-	+	-
No	Yes	+	-	-
No	Yes	-	+	+

Floating Point Addition Example

We follow the preceding algorithm to compute the sum

$$(-1.001_2 \times 2^{-2}) + (-1.111_2 \times 2^0).$$

We have $s_1 = 1.001$, $e_1 = -2$, $s_2 = 1.111$ and $e_2 = 0$.

- 1 $e_1 < e_2$ so swap, $d = 2$, tentative $exp = 0$.
- 2 Signs of both operands negative, don't negate s_2 .
- 3 Shift s_2 (1.001 after swap) right by $d = 2$, giving $s_2 = 0.010$, $g = 0$, $r = 1$, $s = 0$.

$$\begin{array}{r}
 1.111 \\
 + 0.010 \\
 \hline
 (1)0.001
 \end{array}$$

$S = 0.001$, with a carry-out.

- 4 Carry-out, so shift S right, $S = 1.000$, $exp := exp + 1 = 1$.
- 5 $s := g \vee r \vee s = 0 \vee 1 \vee 0 = 1$. $r = \text{low order bit of sum} = 1$.
- 6 $r \wedge s = \text{true}$, so round up, $S := S + 1 = 1.001$.
- 7 Both signs are negative, so sign of result is negative. Final result: $-1.001_2 \times 2^1$.

Floating Point Addition Example

We follow the preceding algorithm to compute the sum

$$-1.010_2 + 1.100_2.$$

We have $s_1 = 1.010$, $e_1 = 0$, $s_2 = 1.100$ and $e_2 = 0$.

- 1 No swap, $d = 0$, tentative $exp = 0$.
- 2 Signs differ, replace s_2 with 0.100 (two's complement of 1.100).
- 3 $d = 0$ so no shift. $g = r = s = 0$.

$$\begin{array}{r}
 1.010 \\
 + 0.100 \\
 \hline
 1.110
 \end{array}$$

Signs are different, most-significant bit is 1, no carry-out, so must two's complement sum, giving $S = 0.010$.

- 4
- 5 Shift left twice, so $S = 1.000$, $exp := exp - 2 = -2$.
- 6 Two left shifts, so $r = g = s = 0$.
- 7 No addition required for rounding.
- 8 Final result is $sign \times S \times 2^{exp}$ or $sign \times 1.000 \times 2^{-2}$. Since complement but no swap and $sign(a_1)$ is $-$, the sign of the sum is $+$. Final result: $1.000_2 \times 2^{-2}$.

Exercice 6

Compute the addition: $100'000'001 + 1$.

Then, show why the following **for** loops forever.

```
1 int main(int argc, char *argv[])
2 {
3     for (float x = 100'000'001; x <= 100'000'010; x += 1.0)
4     {
5     }
6     return 0;
7 }
```

From C to Binary

- We have three files: `maths.h`, `maths.c` and `main.c`

- We create two object files, `maths.o` and `main.o` with

```
$ gcc -c maths.c  
$ gcc -c main.c
```

- We link these two object files into an executable `main` with

```
$ gcc maths.o main.o -o main
```

- We execute it with

```
$ ./main
```

and the `shell` will call the `execve` system call to ask the linux kernel to execute it. You can see this with

```
$ strace ./main  
execve("./main", ["./main"], 0x7ffe0a26bf70 /* 62 vars */) = 0  
...
```

Header File

```
1 // maths.h
2 #pragma once
3
4 extern double PI;
5
6 int gcd(int a, int b);
7
8 long factorial(int n);
```

Source File

```
1 // maths.c
2 #include "maths.h"
3
4 double PI = 3.141592653589793;
5
6 int gcd(int a, int b)
7 {
8     return b == 0 ? a : gcd(b, a % b);
9 }
10
11 long factorial(int n)
12 {
13     if (n < 2) return 1;
14     return n * factorial(n - 1);
15 }
```

Main Source File

```
1 // main.c
2 #include <stdio.h>
3 #include "maths.h"
4
5 int main(int argc, char *argv[])
6 {
7     long f = factorial(25);
8     int g = gcd(24, 42);
9     printf("f=%ld, g=%d, PI=%f\n", f, g, PI);
10    return 0;
11 }
```

Preprocessing

```
$ gcc -H -c main.c

. /usr/include/stdio.h
. /usr/include/x86_64-linux-gnu/bits/libc-header-start.h
... /usr/include/features.h
.... /usr/include/x86_64-linux-gnu/sys/cdefs.h
..... /usr/include/x86_64-linux-gnu/bits/wordsize.h
..... /usr/include/x86_64-linux-gnu/bits/long-double.h
.... /usr/include/x86_64-linux-gnu/gnu/stubs.h
..... /usr/include/x86_64-linux-gnu/gnu/stubs-64.h
.. /usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h
.. /usr/include/x86_64-linux-gnu/bits/types.h
... /usr/include/x86_64-linux-gnu/bits/wordsize.h
... /usr/include/x86_64-linux-gnu/bits/typesizes.h
.. /usr/include/x86_64-linux-gnu/bits/types/_FILE.h
.. /usr/include/x86_64-linux-gnu/bits/types/FILE.h
.. /usr/include/x86_64-linux-gnu/bits/libio.h
... /usr/include/x86_64-linux-gnu/bits/_G_config.h
.... /usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h
.... /usr/include/x86_64-linux-gnu/bits/types/_mbstate_t.h
... /usr/lib/gcc/x86_64-linux-gnu/7/include/stdarg.h
.. /usr/include/x86_64-linux-gnu/bits/stdio_lim.h
.. /usr/include/x86_64-linux-gnu/bits/sys_errlist.h
. maths.h
```

Object Files

```
$ gcc -c maths.c
$ objdump -dr maths.o
```

```
maths.o:      file format elf64-x86-64
Disassembly of section .text:

0000000000000037 <gcd>:
37: 55          push    %rbp
38: 48 89 e5    mov     %rsp,%rbp
3b: 48 83 ec 10 sub    $0x10,%rsp
3f: 89 7d fc    mov     %edi,-0x4(%rbp)
42: 89 75 f8    mov     %esi,-0x8(%rbp)
45: 83 7d f8 00 cmpl   $0x0,-0x8(%rbp)
49: 74 15        je     60 <gcd+0x29>
4b: 8b 45 fc    mov     -0x4(%rbp),%eax
4e: 99          cltd
4f: f7 7d f8    idivl  -0x8(%rbp)
52: 8b 45 f8    mov     -0x8(%rbp),%eax
55: 89 d6        mov     %edx,%esi
57: 89 c7        mov     %eax,%edi
59: e8 00 00 00 00 callq  5e <gcd+0x27>
5a: R_X86_64_PC32 gcd-0x4
5e: eb 03        jmp     63 <gcd+0x2c>
60: 8b 45 fc    mov     -0x4(%rbp),%eax
63: c9          leaveq 
64: c3          retq
```

Object Files

```
0000000000000000 <factorial>:  
 0: 55          push  %rbp  
 1: 48 89 e5    mov    %rsp,%rbp  
 4: 53          push  %rbx  
 5: 48 83 ec 18 sub   $0x18,%rsp  
 9: 89 7d ec    mov    %edi,-0x14(%rbp)  
 c: 83 7d ec 01 cmpl  $0x1,-0x14(%rbp)  
10: 7f 07       jg    19 <factorial+0x19>  
12: b8 01 00 00 00 mov   $0x1,%eax  
17: eb 17       jmp   30 <factorial+0x30>  
19: 8b 45 ec    mov   -0x14(%rbp),%eax  
1c: 48 63 d8    movslq %eax,%rbx  
1f: 8b 45 ec    mov   -0x14(%rbp),%eax  
22: 83 e8 01    sub   $0x1,%eax  
25: 89 c7       mov   %eax,%edi  
27: e8 00 00 00 00 callq 2c <factorial+0x2c>  
28: R_X86_64_PC32 factorial-0x4  
 2c: 48 0f af c3 imul  %rbx,%rax  
 30: 48 83 c4 18 add   $0x18,%rsp  
 34: 5b          pop   %rbx  
 35: 5d          pop   %rbp  
 36: c3          retq
```

Object Files

```
$ objdump -sj .data maths.o
```

```
maths.o:      file format elf64-x86-64
```

```
Contents of section .data:
```

```
0000 182d4454 fb210940
```

Object Files

```
$ gcc -c main.c
$ objdump -dr main.o
```

```
main.o:      file format elf64-x86-64

Disassembly of section .text:
0000000000000000 <main>:
 0: 55                      push  %rbp
 1: 48 89 e5                mov    %rsp,%rbp
 4: 48 83 ec 30             sub    $0x30,%rsp
 8: 89 7d ec                mov    %edi,-0x14(%rbp)
 b: 48 89 75 e0             mov    %rsi,-0x20(%rbp)
 f: bf 19 00 00 00           mov    $0x19,%edi
14: e8 00 00 00 00           callq 19 <main+0x19>
15: R_X86_64_PLT32         factorial-0x4
19: 48 89 45 f8             mov    %rax,-0x8(%rbp)
1d: be 2a 00 00 00           mov    $0x2a,%esi
22: bf 18 00 00 00           mov    $0x18,%edi
27: e8 00 00 00 00           callq 2c <main+0x2c>
28: R_X86_64_PLT32         gcd-0x4
```

Object Files

```
2c: 89 45 f4          mov    %eax,-0xc(%rbp)
2f: 48 8b 0d 00 00 00 00  mov    0x0(%rip),%rcx      # 36 <main+0x36>
32: R_X86_64_PC32      PI-0x4
36: 8b 55 f4          mov    -0xc(%rbp),%edx
39: 48 8b 45 f8      mov    -0x8(%rbp),%rax
3d: 48 89 4d d8      mov    %rcx,-0x28(%rbp)
41: f2 0f 10 45 d8    movsld -0x28(%rbp),%xmm0
46: 48 89 c6          mov    %rax,%rsi
49: 48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi      # 50 <main+0x50>
4c: R_X86_64_PC32      .rodata-0x4
50: b8 01 00 00 00      mov    $0x1,%eax
55: e8 00 00 00 00      callq 5a <main+0x5a>
56: R_X86_64_PLT32      printf-0x4
5a: b8 00 00 00 00      mov    $0x0,%eax
5f: c9                  leaveq
60: c3                  retq
```

Object Files

```
$ objdump -sj .rodata main.o
```

```
main.o:      file format elf64-x86-64
```

```
Contents of section .rodata:
```

```
0000 66203d20 256c642c 2067203d 2025642c  f = %ld, g = %d,  
0010 20504920 3d20252e 386c660a 00          PI = %.8lf..
```

Executable File

```
$ gcc maths.o main.o -o main && objdump -dr main
```

```
00000000000006af <main>:
6af: 55                      push   %rbp
6b0: 48 89 e5                mov    %rsp,%rbp
6b3: 48 83 ec 30             sub    $0x30,%rsp
6b7: 89 7d ec                mov    %edi,-0x14(%rbp)
6ba: 48 89 75 e0             mov    %rsi,-0x20(%rbp)
6be: bf 19 00 00 00           mov    $0x19,%edi
6c3: e8 82 ff ff ff         callq  64a <factorial>
6c8: 48 89 45 f8             mov    %rax,-0x8(%rbp)
6cc: be 2a 00 00 00           mov    $0x2a,%esi
6d1: bf 18 00 00 00           mov    $0x18,%edi
6d6: e8 a6 ff ff ff         callq  681 <gcd>
6db: 89 45 f4                mov    %eax,-0xc(%rbp)
6de: 48 8b 0d 2b 09 20 00    mov    0x20092b(%rip),%rcx    # 201010 <PI>
6e5: 8b 55 f4                mov    -0xc(%rbp),%edx
6e8: 48 8b 45 f8             mov    -0x8(%rbp),%rax
6ec: 48 89 4d d8             mov    %rcx,-0x28(%rbp)
6f0: f2 0f 10 45 d8           movsd  -0x28(%rbp),%xmm0
6f5: 48 89 c6                mov    %rax,%rsi
6f8: 48 8d 3d 95 00 00 00    lea    0x95(%rip),%rdi      # 794 <_IO_stdin_used+0x4>
6ff: b8 01 00 00 00           mov    $0x1,%eax
704: e8 17 fe ff ff         callq  520 <printf@plt>
709: b8 00 00 00 00           mov    $0x0,%eax
70e: c9                      leaveq
70f: c3                      retq
```

Memory Mapping

```
$ gdb main
(gdb) start
(gdb) breakpoint exit
(gdb) continue
(gdb) !cat /proc/`pgrep main`/maps
555555554000-5555555555000 r-xp 00000000 08:11 9326247 main
555555754000-555555755000 r--p 00000000 08:11 9326247 main
555555755000-555555756000 rw-p 00001000 08:11 9326247 main
555555756000-555555777000 rw-p 00000000 00:00 0 [heap]
7ffff79e4000-7ffff7bcb000 r-xp 00000000 08:13 3936890          libc-2.27.so
7ffff7bcb000-7ffff7dc000 ---p 001e7000 08:13 3936890          libc-2.27.so
7ffff7dc000-7ffff7dcf000 r--p 001e7000 08:13 3936890          libc-2.27.so
7ffff7dcf000-7ffff7dd1000 rw-p 001eb000 08:13 3936890          libc-2.27.so
7ffff7dd1000-7ffff7dd5000 rw-p 00000000 00:00 0
7ffff7dd5000-7ffff7dfc000 r-xp 00000000 08:13 3936862          ld-2.27.so
7ffff7fcf000-7ffff7fd1000 rw-p 00000000 00:00 0
7ffff7ff7000-7ffff7ffa000 r--p 00000000 00:00 0                  [vvar]
7ffff7ffa000-7ffff7ffc000 r-xp 00000000 00:00 0                  [vdso]
7ffff7ffc000-7ffff7ffd000 r--p 00027000 08:13 3936862          ld-2.27.so
7ffff7ffd000-7ffff7ffe000 rw-p 00028000 08:13 3936862          ld-2.27.so
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
7ffff7ffde000-7fffffff000 rw-p 00000000 00:00 0 [stack]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

Reduced Instruction Set Architecture

- MIPS is a **Reduced Instruction Set Computer** architecture.
 - Many complex instructions are never used in compiled code.
 - Make the common case fast.
- Microprocessor without **Interlocked Pipelined Stages** was founded in 1984 by Hennessey and others to build a commercial RISC processor.
- As of 2017, MIPS processors are mainly used in embedded systems.
 - Used in Nintendo64, Sony Playstation, Sony Playstation 2, PSP.
 - Currently used in super computers in China.

MIPS Instruction Set Architecture

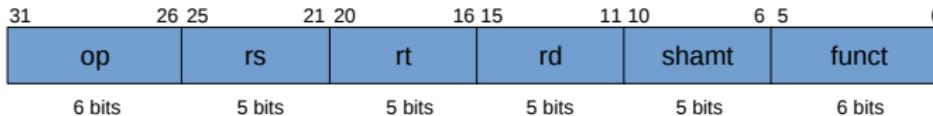
We use a simplified MIPS32 instruction set architecture,

- Thirty two 32-bit registers.
- 32-bit memory addresses.
- Byte addressable memory. Big-endian.
- Three instruction formats
 - **R-type** instructions: Operate on three registers. We use a subset of them: `add`, `sub`, `or`, `and`, `slt`.
 - **I-type** instructions: Operate on two registers and a 16-bit immediate. We use a subset of them: `lw`, `sw`, `addi`, `beq`.
 - **J-type** instructions: jumps that operate on a 26-bit immediate. We only use one of them: `j`.

Registers

Name	Number	Usage
\$0	0	The constant value zero
\$at	1	Assembler temporary
\$v0-\$v1	2-3	Function return value
\$a0-\$a3	4-7	Function arguments
\$t0-\$t7	8-15	Temporary variables
\$s0-\$s7	16-23	Saved variables
\$t8-\$t9	24-25	Temporary variables
\$k0-\$k1	26-27	Operating system temporaries
\$gp	28	Global pointer
\$sp	29	Stack pointer
\$fp	30	Frame pointer
\$ra	31	Function return address

R(register)-Type Instructions



- **op**: Opcode of the instruction, $op = 000000$ for all R-type instructions.
- **rs**: First source operand.
- **rt**: Second source operand.
- **rd**: Destination.
- **shamt**: Shift amount. It is only used in shift operations (so it is 00000 for us).
- **funct**: Specify the function to use.
 - **add** has $funct = 100000$.
 - **sub** has $funct = 100010$.
 - **and** has $funct = 100100$.
 - **or** has $funct = 100101$.
 - **slt** has $funct = 101010$.

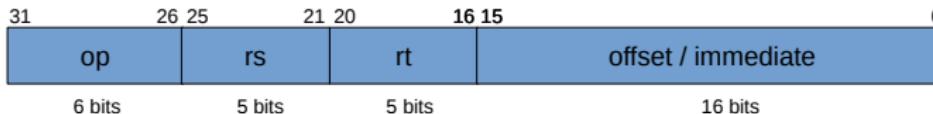
R-Type Example

```
1  add $t1, $t0, $0  # $t1 ← $t0 + 0
2  sub $t2, $t3, $t1 # $t2 ← $t3 - $t1
3  slt $t4, $t4, $t2 # $t4 ← 1 if $t4 < $t2 else 0
```

The machine language representation of this code is

Instruction	op	rs	rt	rd	shamt	funct
add \$t1, \$t0, \$0	000000	01000	00000	01001	00000	100000
sub \$t2, \$t3, \$t1	000000	01011	01001	01010	00000	100010
slt \$t4, \$t4, \$t2	000000	01100	01010	01100	00000	101010

I(mmediate)-Type Instructions



- **op:** Opcode of the instruction,
 - `lw` has op = 100011.
 - `sw` has op = 101011.
 - `addi` has op = 001000.
 - `beq` has op = 000100.
- **rs:** Register containing the base address for `lw` and `sw`, and a source operand for `addi` and `beq`.
- **rt:** Destination register for `addi` and `lw`, source register for `sw` and second operand for `beq`.
- **offset/immediate:** Byte address offset for `lw` and `sw`, immediate field for `addi` and word address offset with respect to $PC + 4$ for `beq`. This value is sign extended to a 32-bit value.

I-Type Example

```

1  lw    $t1, 32($s0)    # $t1 ← Memory[$s0 + 32]
2  addi $t1, $t1, 1975 # $t1 ← $t1 + 1975
3  beq  $t2, $t1, end  # if $t2 = $t1 goto end
4  addi $t1, $t1, 1     # $t1 ← $t1 + 1
5  end:
6  sw    $t1, 32($s0)    # Memory[$s0 + 32] ← $t1

```

The machine language representation of this code is

Instruction	op	rs	rt	offset/immediate
lw \$t1, 32(\$s0)	100011	10000	01001	0000000000100000
addi \$t1, \$t1, 1975	001000	01001	01001	0000011110110111
beq \$t2, \$t1, end	000100	01010	01001	0000000000000001
addi \$t1, \$t1, 1	001000	01001	01001	0000000000000001
sw \$t1, 32(\$s0)	101011	10000	01001	0000000000100000

J(ump)-Type Instructions



- **op**: Opcode of the instruction, $op = 000010$
- **address**: word address of the instruction to jump to. To get a 32-bit jump address,
 - **address** is multiplied by 4.
 - The four high order bits are taken from the four high order bits of $PC + 4$.
- The jump is limited to the region $[PC + 4]_{31..28}[0x00000000]$ to $[PC + 4]_{31..28}[0xffffffffc]$.

J-Type Example

```
1 0x00400000: beq $s3, $s4, else # if $s3 == $s4 goto else
2 0x00400004: add $s0, $s1, $s2 # $s0 ← $s1 + $s2
3 0x00400008: j endif          # goto endif
4 else:
5 0x0040000d: sub $s0, $s1, $s2 # $s0 ← $s1 - $s2
6 endif:
7 0x00400010:
```

The machine language representation of line 3 is

Instruction	op	address
j endif	000010	000001000000000000000000000000100

Program Example

To write a more interesting program, we add three more MIPS instructions (we don't use them in our simplified version of MIPS):

- **jal** fct: Call the function fct and save in register \$ra the address of the next instruction (the one just after the **jal** instruction).
- **jr** reg: Jump to the address contained in register reg.
- **mul** dst, src1, src2: Multiplies register src1 with register src2 and puts the result in register reg.

Calling Convention (Simplified)

- The first four arguments are passed in registers: `$a0, $a1, $a2` and `$a3`.
- Additional arguments are passed on the stack before calling the function:
 - arg_n is pushed.
 - Then, arg_{n-1} is pushed.
 - ...
 - Then, arg_5 is pushed.
- The stack pointer (`$sp`) must be 4-byte aligned before calling a function.
- To reduce the effort to save and restore registers, we define two types of registers. A function must save and restore any of the preserved registers it uses. But it can change the nonpreserved registers freely.

Preserved	Nonpreserved
Saved registers: <code>\$s0-\$s7</code>	Temporary registers: <code>\$t0-\$t9</code>
Return address: <code>\$ra</code>	Argument registers: <code>\$a0-\$a3</code>
Stack pointer: <code>\$sp</code>	Return value registers: <code>\$v0-\$v1</code>
Stack above the stack pointer	Stack below the stack pointer

Recursive Factorial

```
1 factorial:  
2 # from Digital Design and Computer Architecture  
3     addi $sp, $sp, -8 # make room on stack for two 4 bytes values  
4     sw $a0, 4($sp) # store $a0  
5     sw $ra, 0($sp) # store $ra  
6     addi $t0, $0, 2 # $t0 ← 2  
7     slt $t0, $a0, $t0 # n ≤ 1 ?  
8     beq $t0, $0, else # no: goto else  
9     addi $v0, $0, 1  
10    addi $sp, $sp, 8 # restore $sp  
11    jr $ra # return 1  
12 else:  
13     addi $a0, $a0, -1 # n ← n - 1  
14     jal factorial # recursive call  
15     lw $ra, 0($sp) # restore $ra  
16     lw $a0, 4($sp) # restore $a0  
17     addi $sp, $sp, 8 # restore $sp  
18     mul $v0, $a0, $v0  
19     jr $ra # return n * factorial(n - 1)
```

Recursive Factorial

```
1 main:
2     addi $a0, $0, 5
3     jal factorial      # factorial(5)
4     add $a0, $v0, $0
5     addi $v0, $0, 1      # 1 is syscall number for print_int
6     syscall            # print_int
7     addi $v0, $0, 10     # 10 is syscall number for exit
8     syscall            # exit
9
10 # spim -file factorial.s
11 # output: 120
```

Exercice 7

Translate the following C function in MIPS assembly.

```
1 int fibonacci(int n)
2 {
3     if (n < 2) return n;
4     return fibonacci(n - 1) + fibonacci(n - 2);
5 }
```

Solution

```
1  fibonacci:
2      addi $t0, $0, 2      # $t0 = 2
3      slt  $t0, $a0, $t0 # n ≤ 1 ?
4      beq  $t0, $0, else # no: goto else
5      add   $v0, $a0, $0  # $v0 ← n
6      jr   $ra             # return n
7 else:
8      addi $sp, $sp, -12 # make room on stack
9      sw   $a0, 8($sp)   # store $a0
10     sw   $ra, 4($sp)   # store $ra
11     sw   $s0, 0($sp)   # store $s0
12     addi $a0, $a0, -1  # n ← n - 1
13     jal   fibonacci
14     add   $s0, $v0, $0  # $s0 ← fibonacci(n - 1)
15     addi $a0, $a0, -1  # n ← n - 2
16     jal   fibonacci
17     add   $v0, $s0, $v0 # $v0 ← fibonacci(n - 1) + fibonacci(n - 2)
```

Solution

```
1      lw    $s0, 0($sp)    # restore $s0
2      lw    $ra, 4($sp)    # restore $ra
3      lw    $a0, 8($sp)    # restore $a0
4      addi $sp, $sp, 12    # restore $sp
5      jr    $ra            # return
6
7 main:
8      addi $a0, $0, 16
9      jal   fibonacci      # fibonacci(16)
10     add   $a0, $v0, $0
11     addi $v0, $0, 1
12     syscall            # print_int
13     addi $v0, $0, 10
14     syscall            # exit
15
16 # spim -file fibonacci.s
17 # output: 987
```

Overview

- We build a datapath for our simple MIPS processor.
- Each instruction is executed in one clock cycle.
- We use two separate memories, one for the instructions and one for the data.

Program Example

We use the following program to demonstrate the datapath.

```
1 main:
2     add  $t0, $0, $0
3     addi $t1, $0, 1
4     lw   $t2, 0($0)
5     addi $t2, $t2, 1
6 loop:
7     slt  $t3, $t1, $t2
8     beq  $t3, $0, endloop
9     add  $t0, $t0, $t1
10    addi $t1, $t1, 1
11    b    loop
12 endloop:
13    sw   $t0, 4($0)
14    b    endloop
```

Exercice 8

What does this program do?

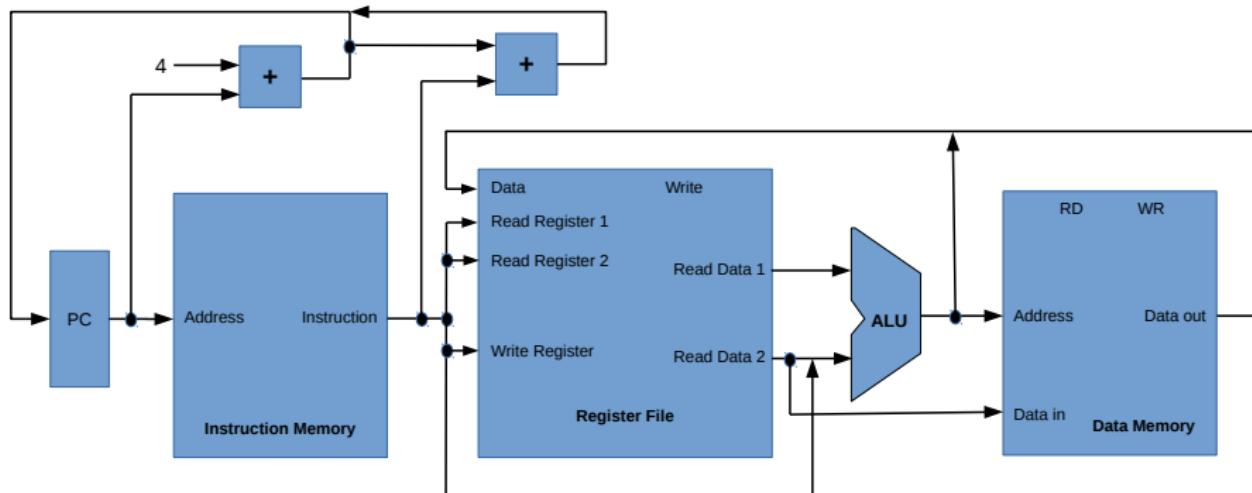
Elements Used to Build the Datapath

- **Instruction memory.**
 - It contains the program.
 - The output is the 4 bytes starting at the specified address.
- **Data memory.**
 - It contains the data.
 - The output is the 4 bytes starting at the specified address, when the RD pin is active.
 - You write a 4 bytes word in the memory, at the rising edge of the clock, by specifying the address and activating the WR pin.
- **Register file.**
 - It contains the 32 four bytes registers.
 - You can read two registers at a time, by specifying their five bits number.
 - You can write to a register, at the rising edge, by specifying a register number and the data to write to it.

Elements Used to Build the Datapath

- **Arithmetic and logic unit.**
 - It performs the different arithmetic and logic computations (addition, subtraction, and, or, nor, set and less than).
 - It gives information about the result of the computation (negative, zero, overflow, carry bit).
- **Program counter.**
 - A register that contains the next instruction to fetch.

Abstract View of the Implementation



Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Overview
Datapath Elements
Datapath Implementation

Running a Program
Inefficiency

Instruction Memory and Fetching

Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Overview
Datapath Elements
Datapath Implementation

Running a Program
Inefficiency

J-Type Instruction

Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Overview
Datapath Elements
Datapath Implementation

Running a Program
Inefficiency

R-Type Instruction

Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

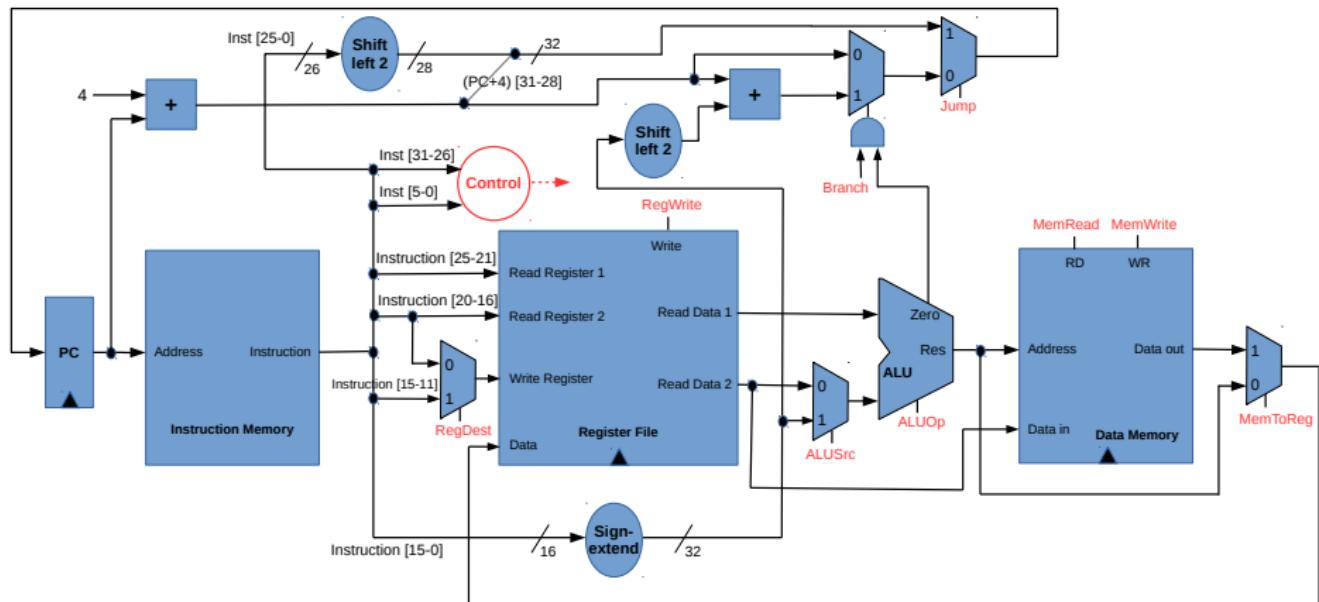
Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Overview
Datapath Elements
Datapath Implementation

Running a Program
Inefficiency

I-Type Instruction

Complete Datapath



Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Overview
Datapath Elements
Datapath Implementation

Running a Program
Inefficiency

Running a Program

Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Overview
Datapath Elements
Datapath Implementation

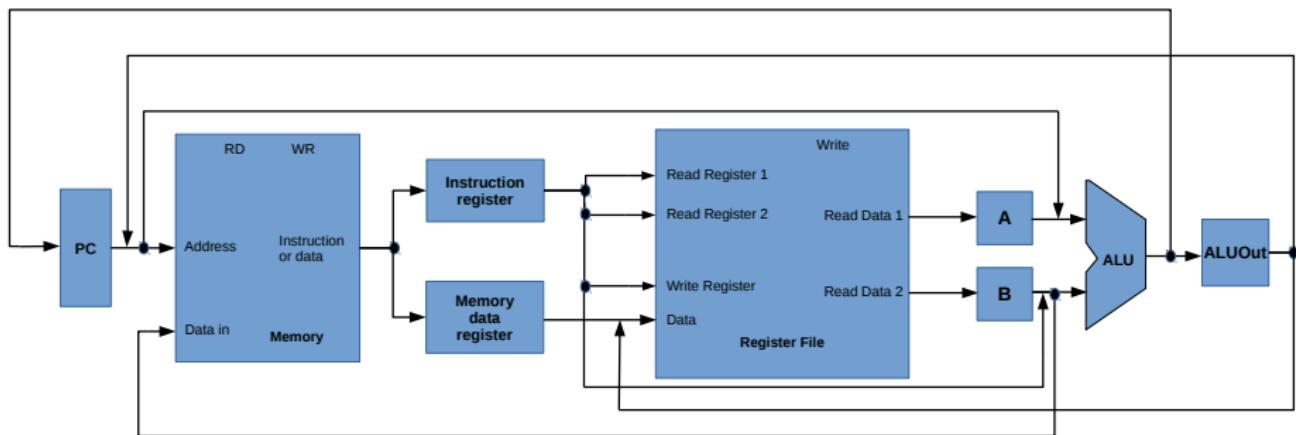
Running a Program
Inefficiency

Inefficiency

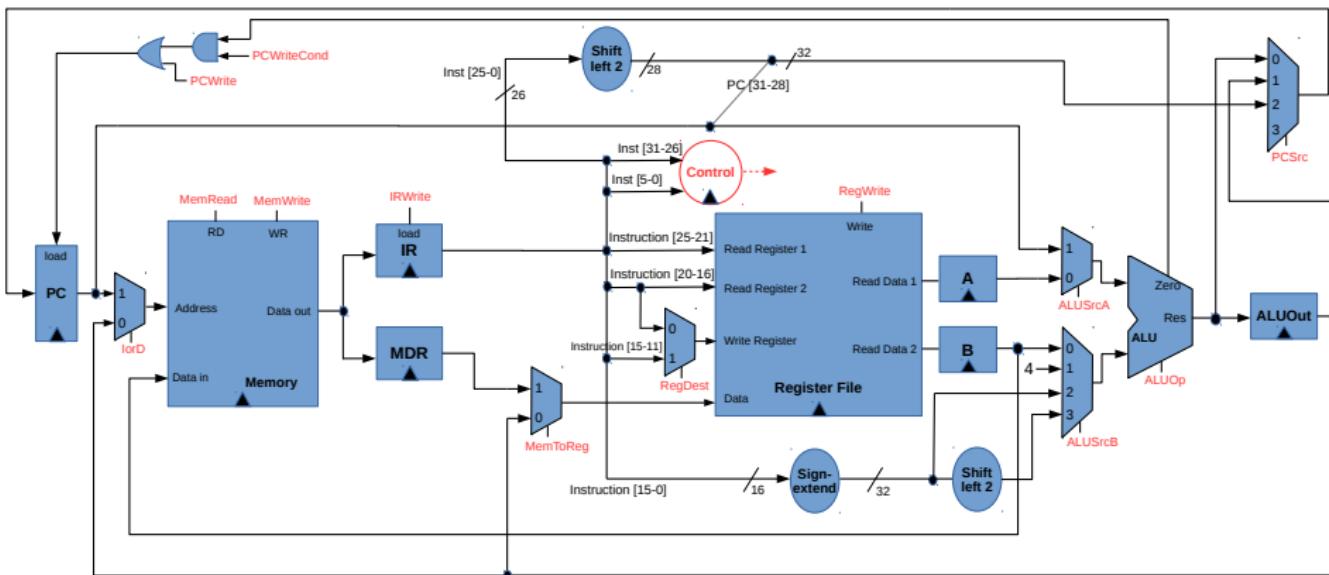
Overview

- The single-cycle datapath executes each instructions in one clock cycle regardless of the complexity of the instructions.
- The key design point of the multi-cycle datapath is to divide the execution of an instruction into several small steps.
- Then we can use more cycles to execute complex instructions and use fewer cycles to execute simple instructions.
- An instruction is divided at most into the following five stages (an instruction can take less than five stages):
 - Instruction fetch (IF).
 - Instruction decode and operand fetch (ID).
 - Execution (EXE).
 - Memory access (MEM).
 - Write back (WB).

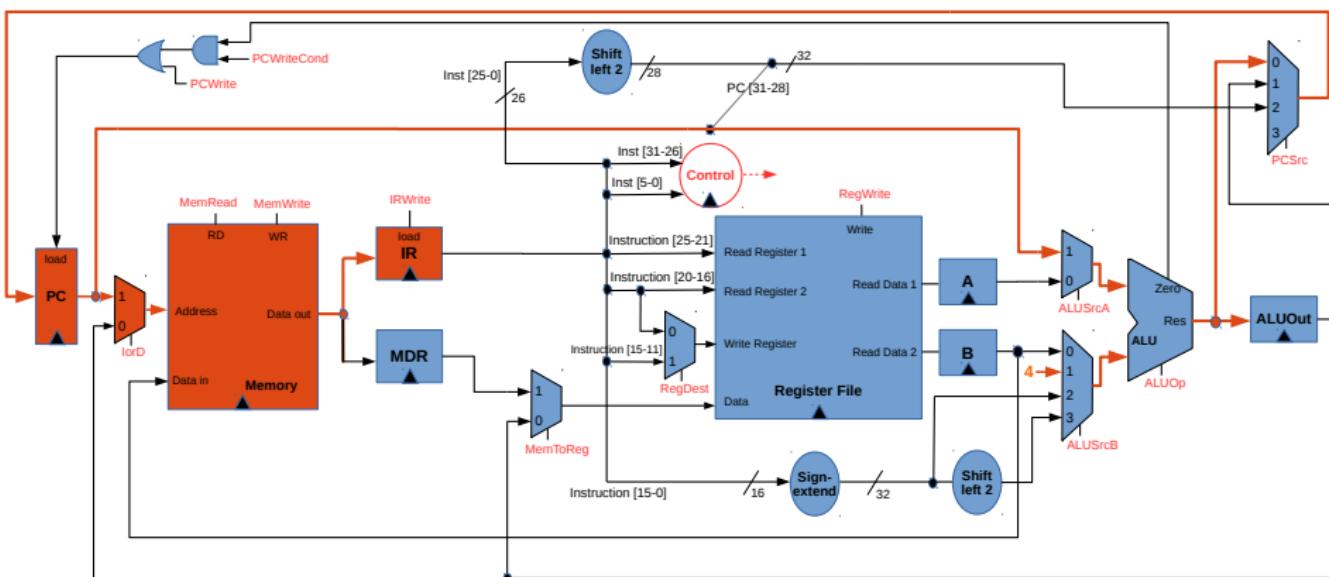
Abstract View of the Implementation



Complete Datapath

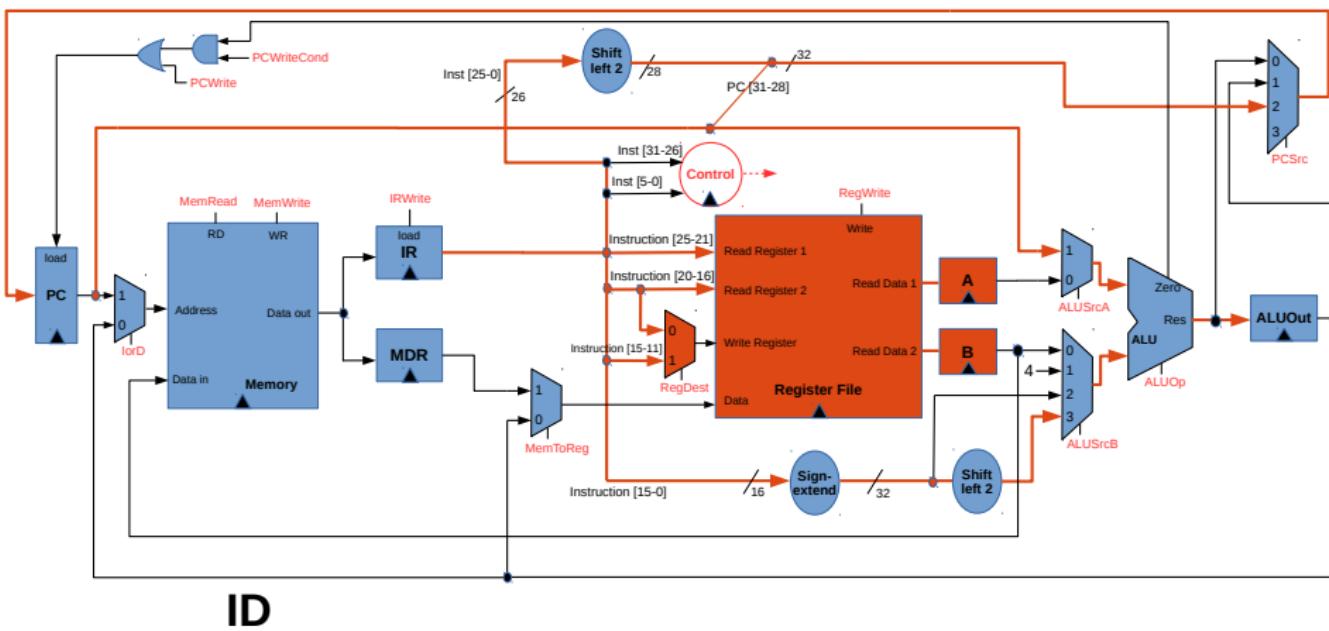


Instruction Fetch Stage

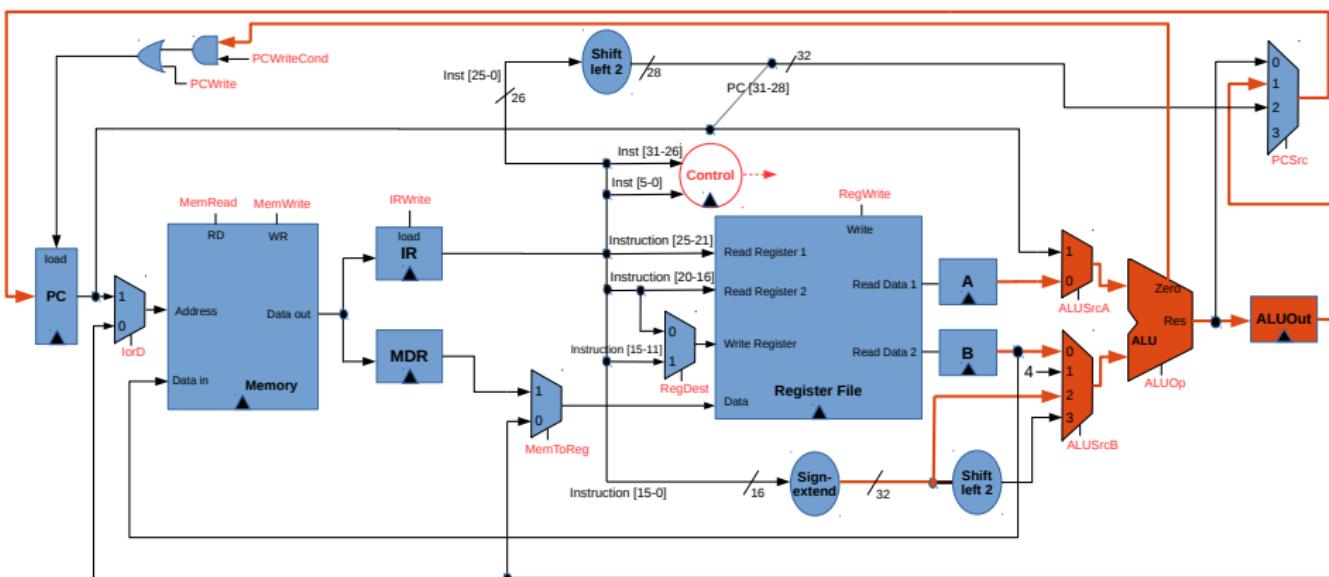


IF

Instruction Decode Stage

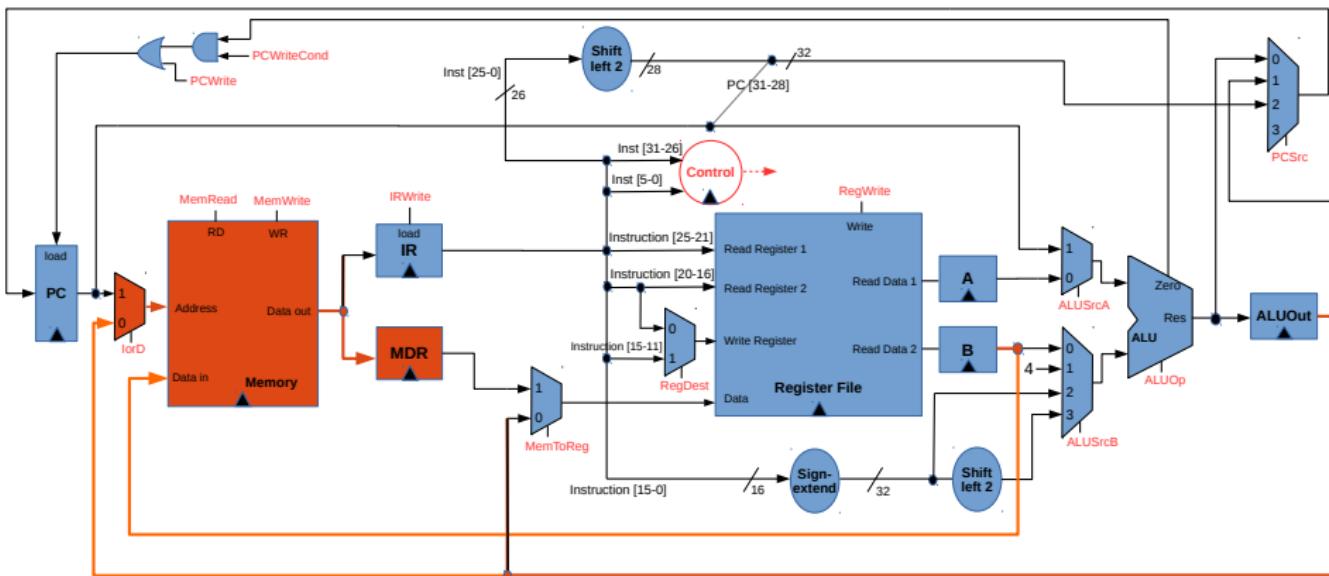


Execute Stage



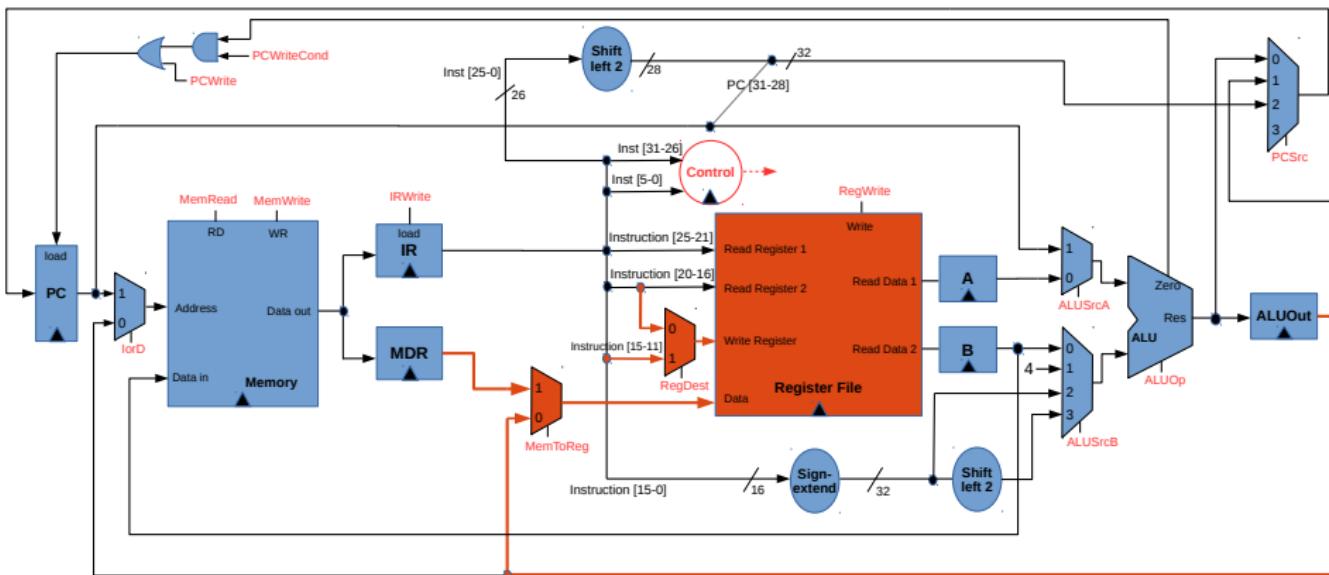
EXE

Memory Stage



MEM

Write Back Stage



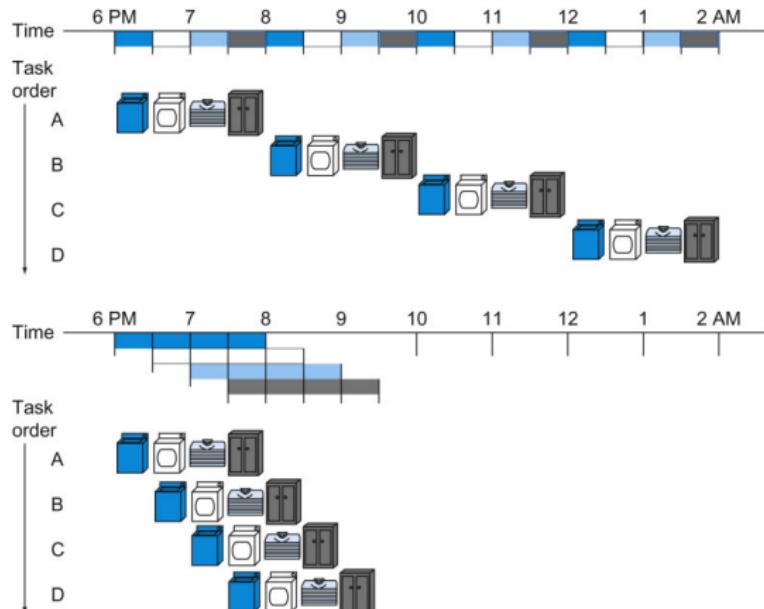
WB

Pipelining

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
- It takes advantage of parallelism that exists among the actions needed to execute an instruction.
- It is a key technique to make fast CPUs.
- Each instruction will go through the five stages (**IF**, **ID**, **EXE**, **MEM** and **WB**). An instruction can do nothing in a stage.
- The key point is that one instruction can begin before the end of the preceding one, contrary to the single or multi-cycle implementation.

Laundry Example

(From Computer Organization and Design. Patterson and Hennessy)

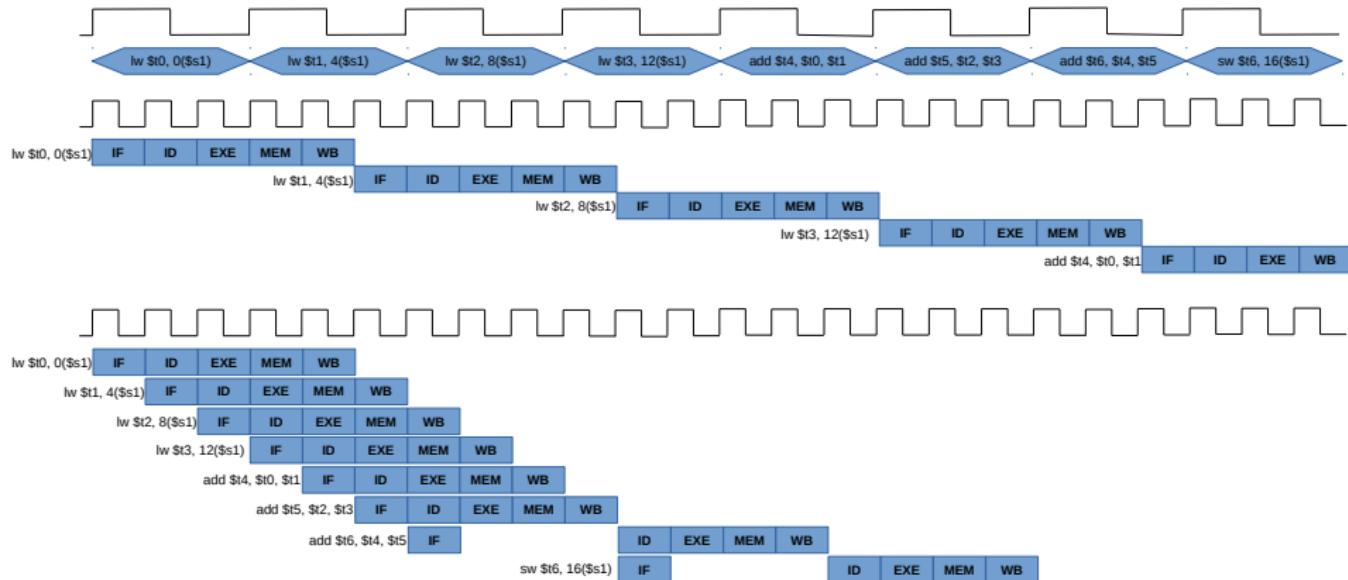


Comparing Single-Cycle, Multi-Cycle and Pipelining

We will compare the behavior of the single-cycle, multi-cycle and pipelining architecture on the following program:

```
1 main:  
2     lw  $t0,  0($s1)  
3     lw  $t1,  4($s1)  
4     lw  $t2,  8($s1)  
5     lw  $t3, 12($s1)  
6     add $t4, $t0, $t1  
7     add $t5, $t2, $t3  
8     add $t6, $t4, $t5  
9     sw  $t6, 16($s1)  
10    # ...
```

Comparing Single-Cycle, Multi-Cycle and Pipelining

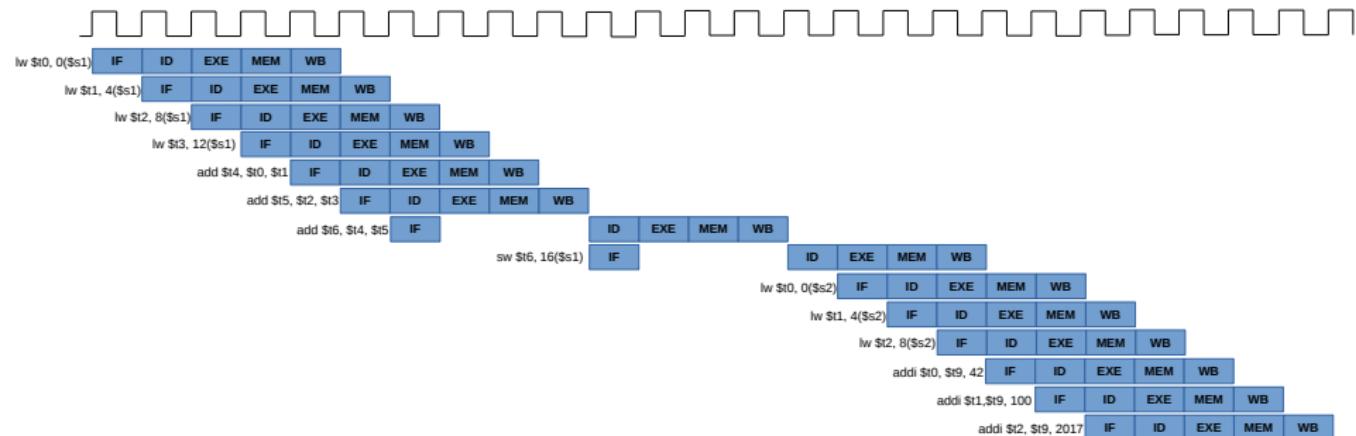


Exercice 9

We add the following highlighted instructions to the program. Can you speed-up the program by reordering instructions? How much do you gain?

```
1 main:  
2     lw    $t0,  0($s1)  
3     lw    $t1,  4($s1)  
4     lw    $t2,  8($s1)  
5     lw    $t3, 12($s1)  
6     add   $t4, $t0, $t1  
7     add   $t5, $t2, $t3  
8     add   $t6, $t4, $t5  
9     sw    $t6, 16($s1)  
10    lw    $t0,  0($s2)  
11    lw    $t1,  4($s2)  
12    lw    $t2,  8($s2)  
13    addi  $t0, $t9, 42  
14    addi  $t1, $t9, 100  
15    addi  $t2, $t9, 2017  
16    # ...
```

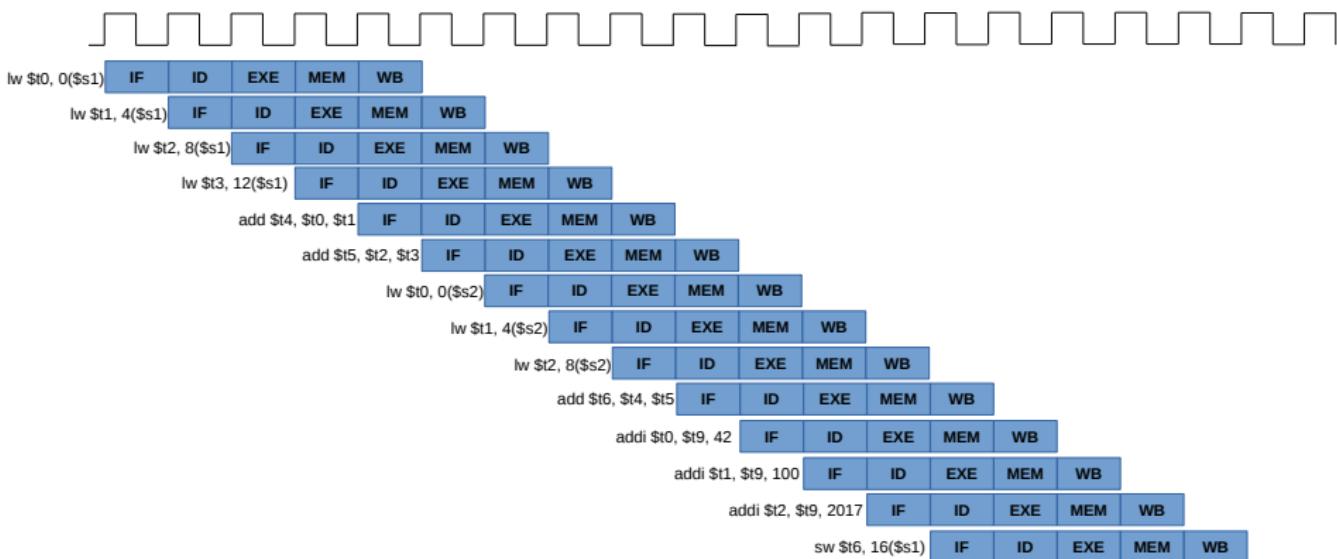
Solution



Solution

```
1 main:  
2     lw    $t0,  0($s1)  
3     lw    $t1,  4($s1)  
4     lw    $t2,  8($s1)  
5     lw    $t3, 12($s1)  
6     add   $t4, $t0, $t1  
7     add   $t5, $t2, $t3  
8     lw    $t0,  0($s2)  
9     lw    $t1,  4($s2)  
10    lw    $t2,  8($s2)  
11    add   $t6, $t4, $t5  
12    addi  $t0, $t9, 42  
13    addi  $t1, $t9, 100  
14    addi  $t2, $t9, 2017  
15    sw    $t6, 16($s1)  
16    # ...
```

Solution



Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

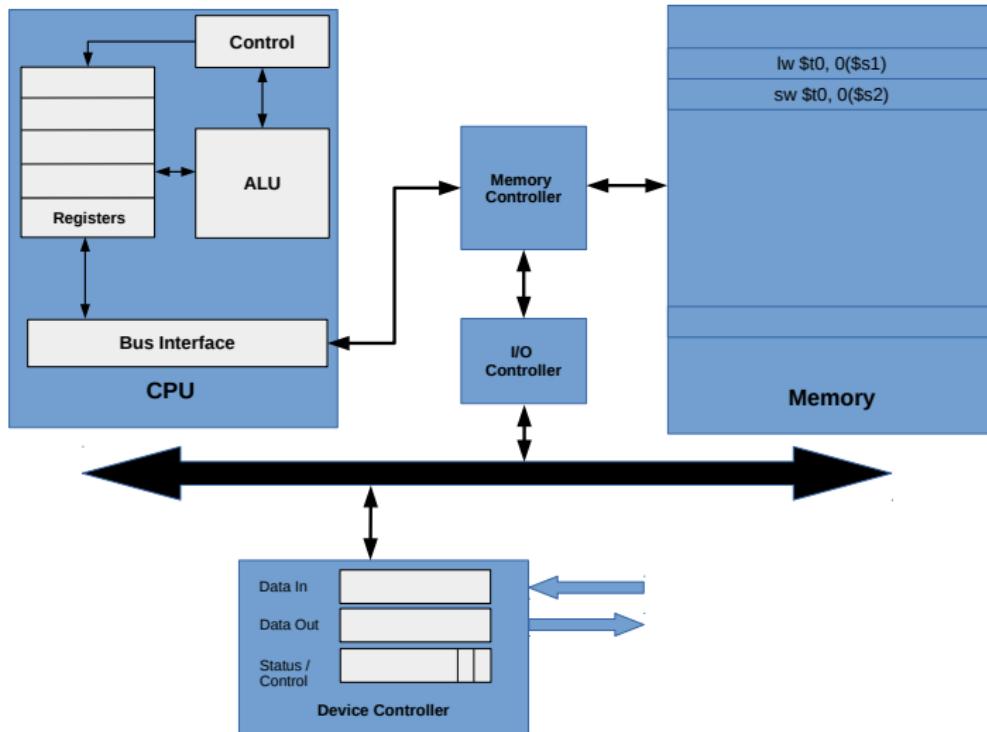
Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Interrupts and Exceptions
I/O Devices
Waiting for I/O Devices
Ints and Exceptions on MIPS

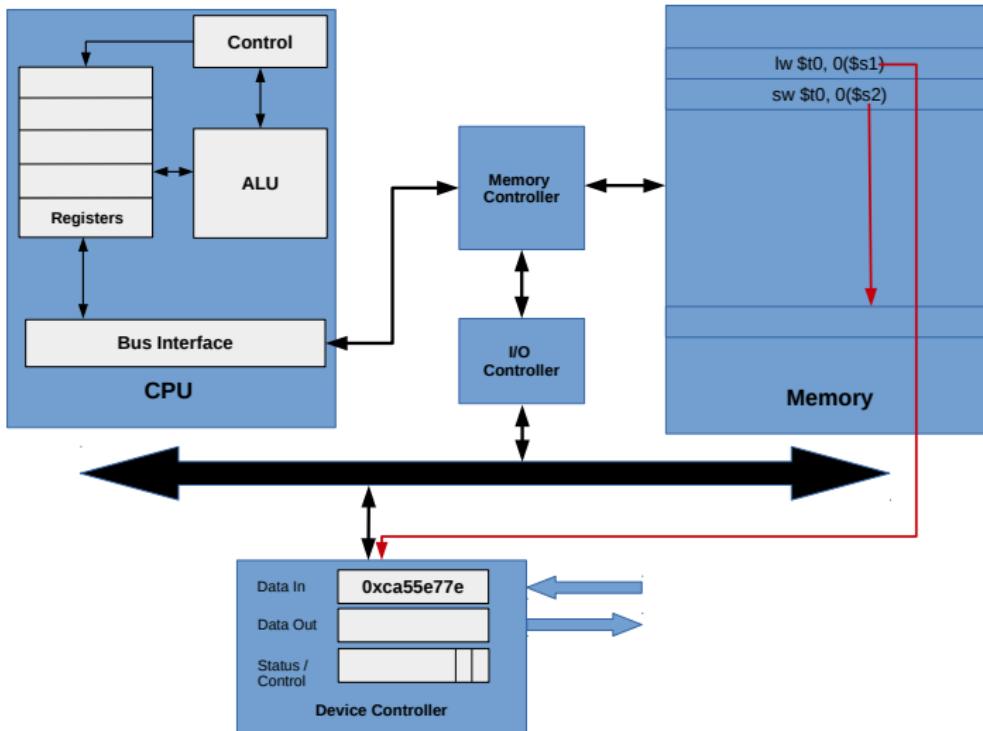
Single-Cycle with Ints and Exs
Ints and Exceptions on x86_64
Linux Signals

Interrupts and Exceptions

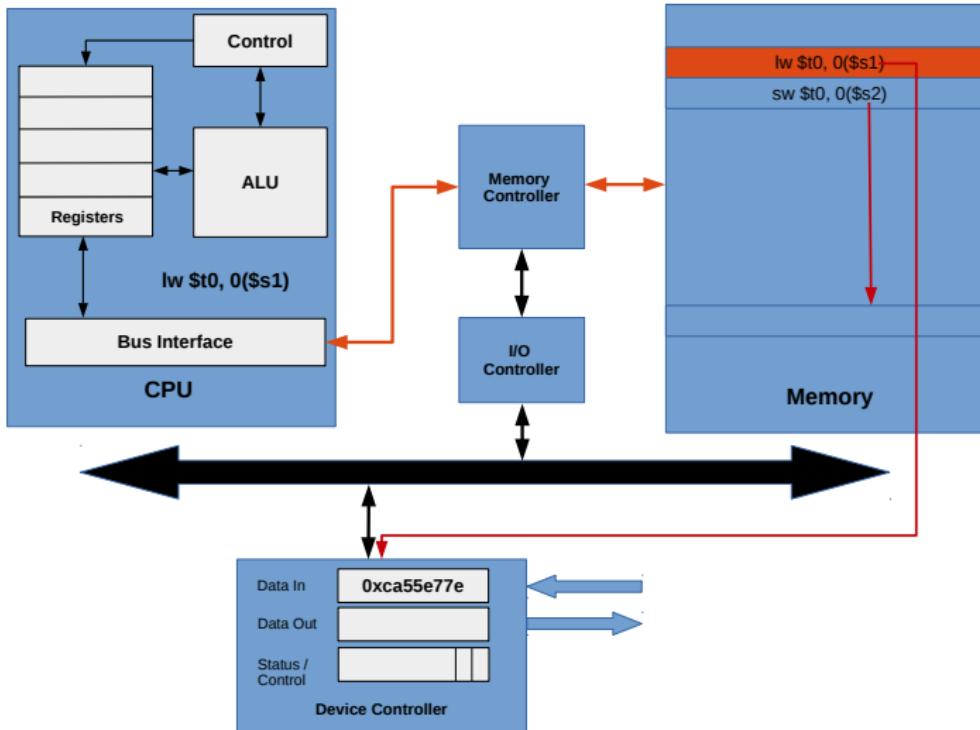
I/O Devices



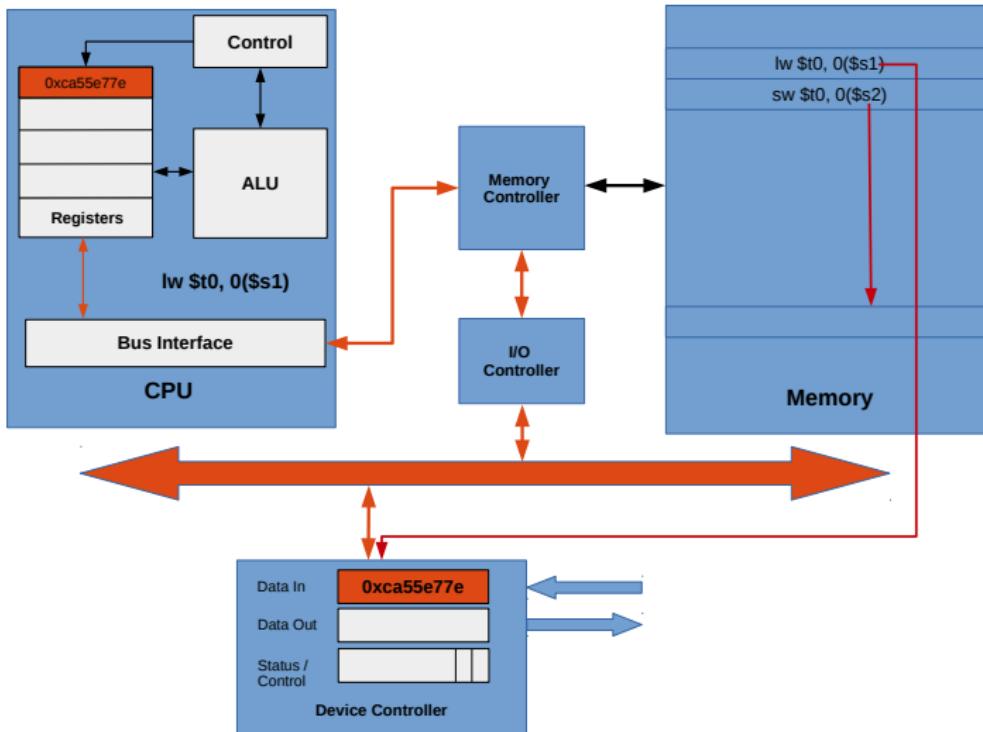
I/O Devices



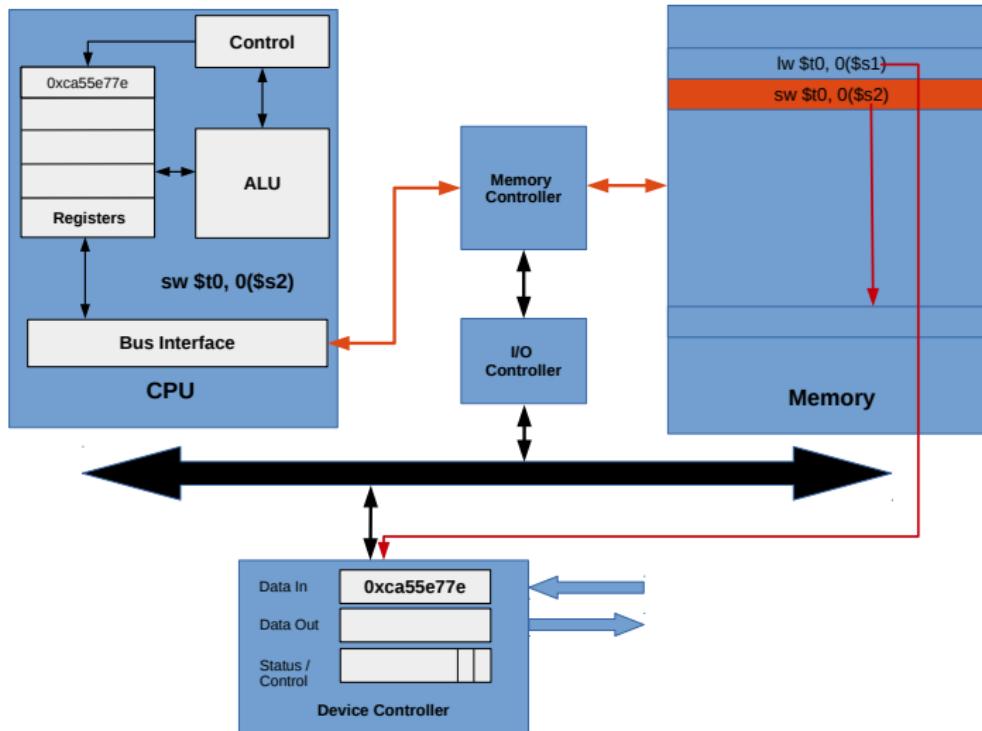
I/O Devices



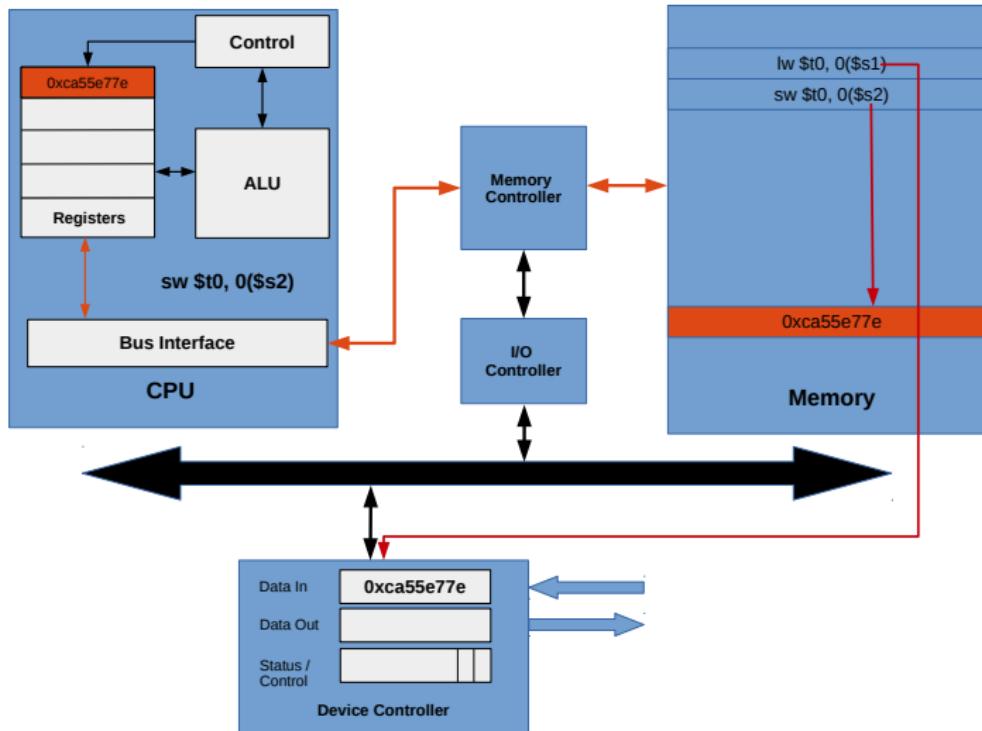
I/O Devices



I/O Devices



I/O Devices



Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

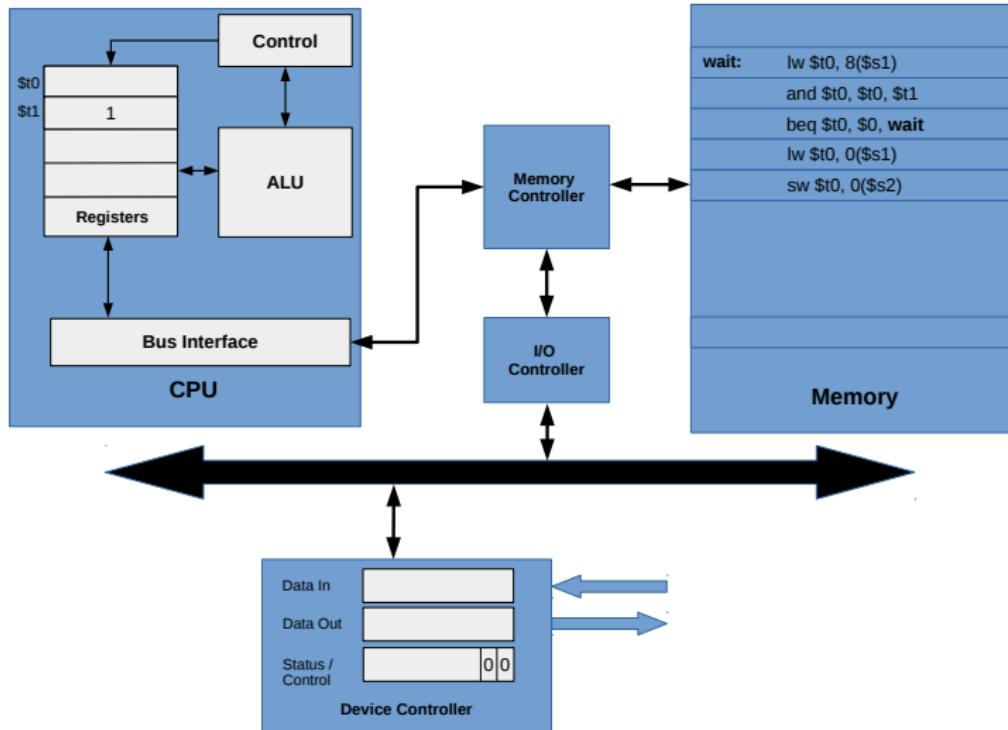
Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Interrupts and Exceptions
I/O Devices
Waiting for I/O Devices
Ints and Exceptions on MIPS

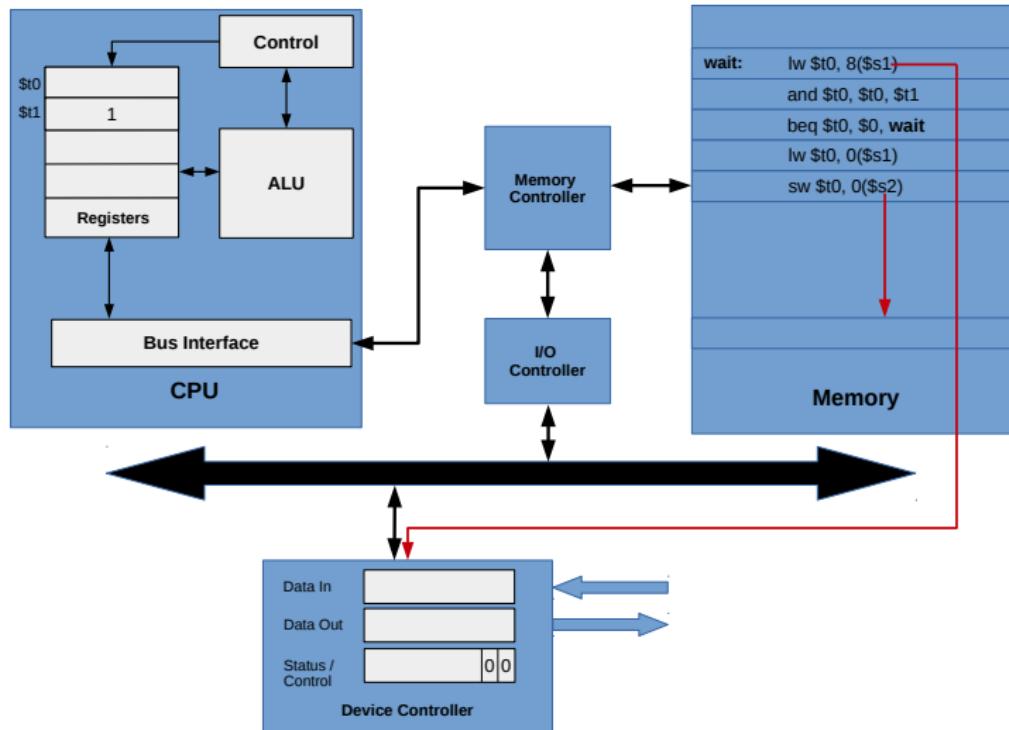
Single-Cycle with Ints and Exs
Ints and Exceptions on x86_64
Linux Signals

Waiting for I/O Devices

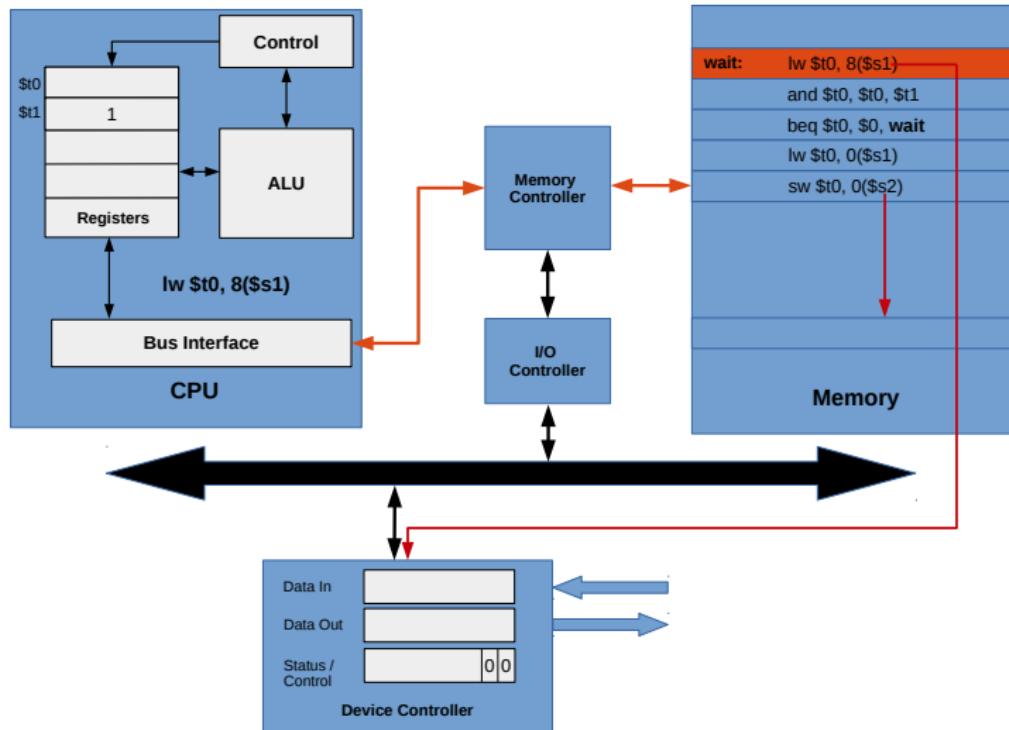
Busy Waiting



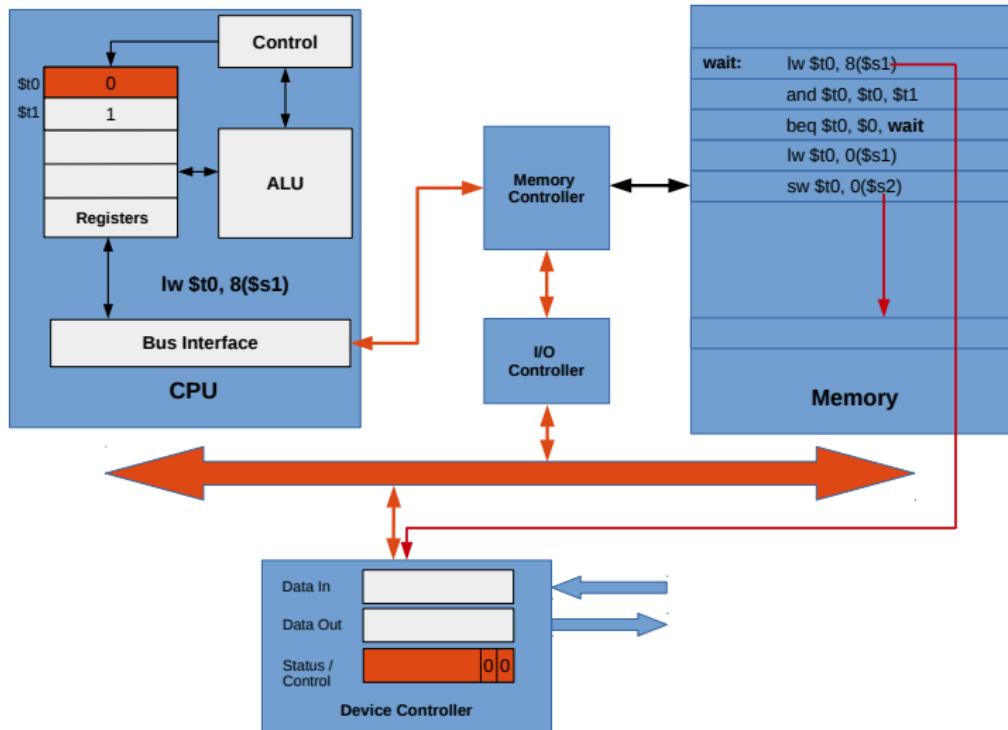
Busy Waiting



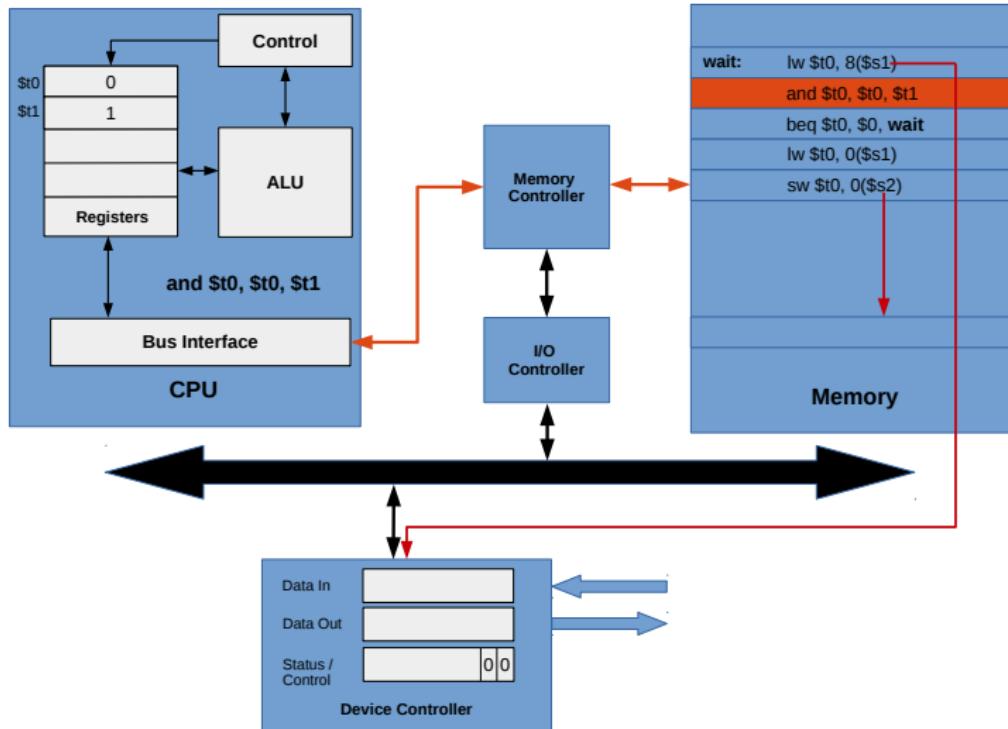
Busy Waiting



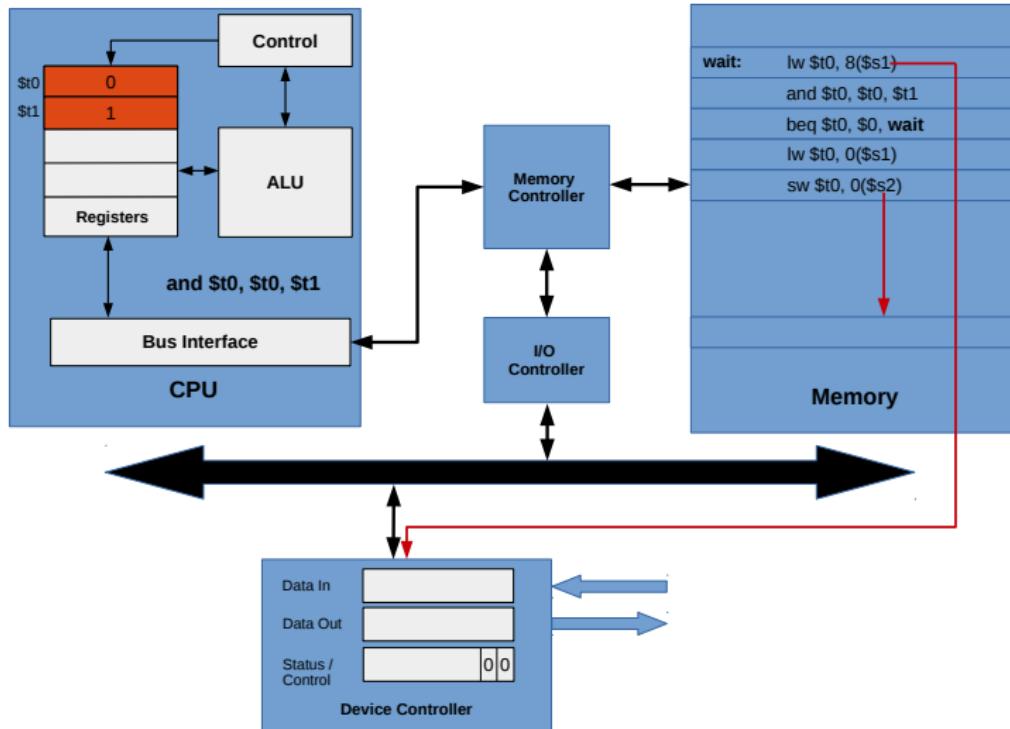
Busy Waiting



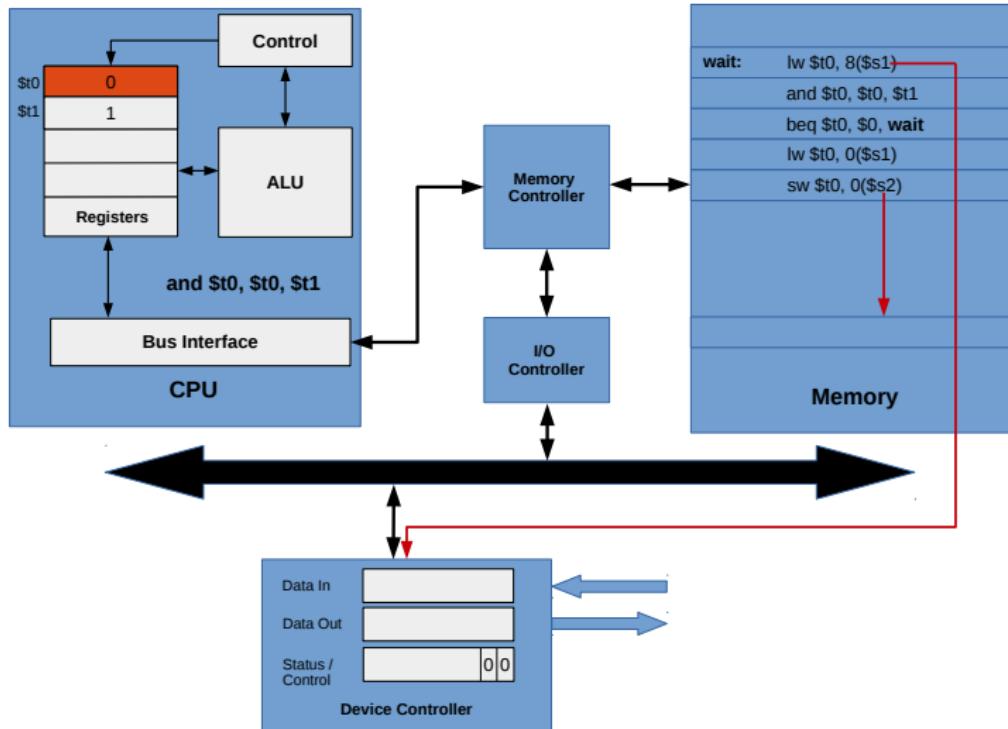
Busy Waiting



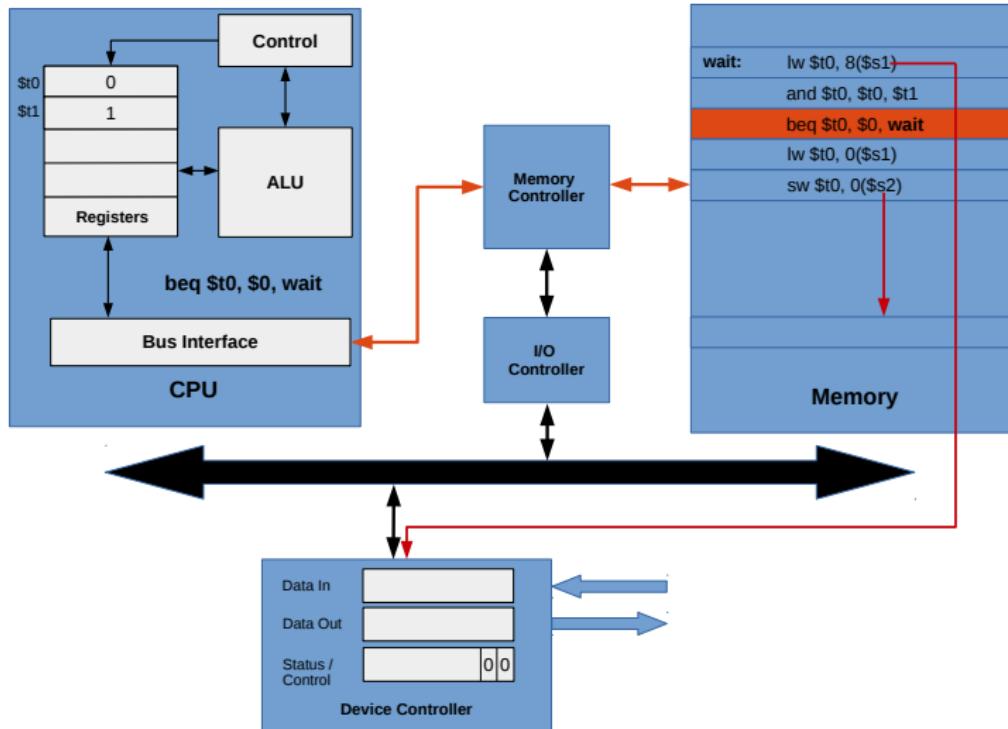
Busy Waiting



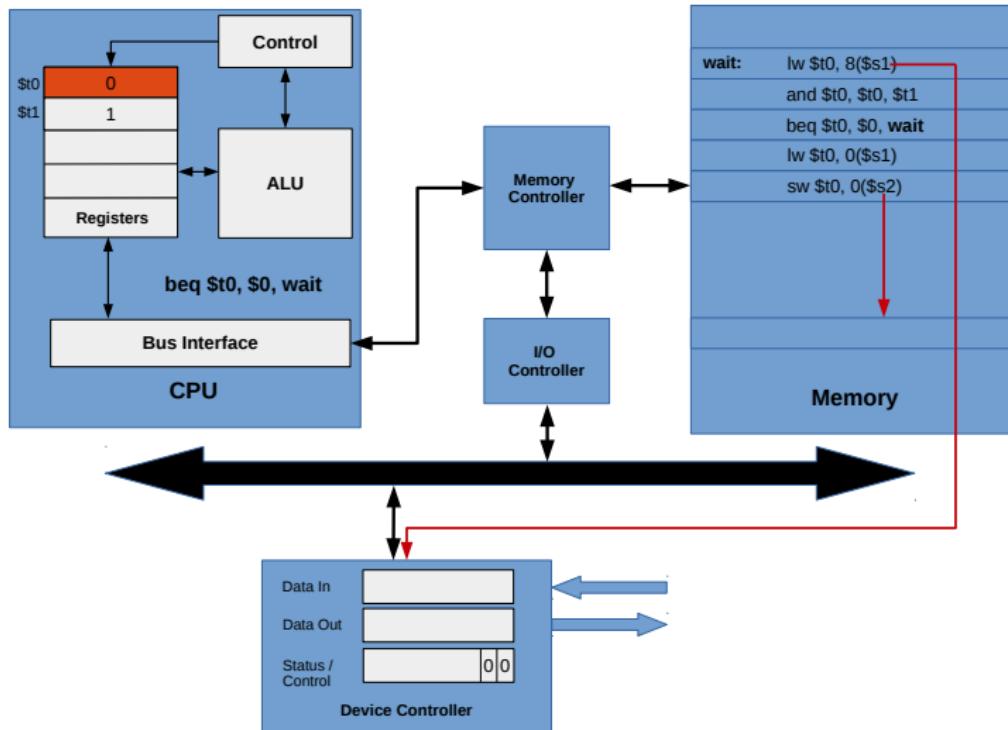
Busy Waiting



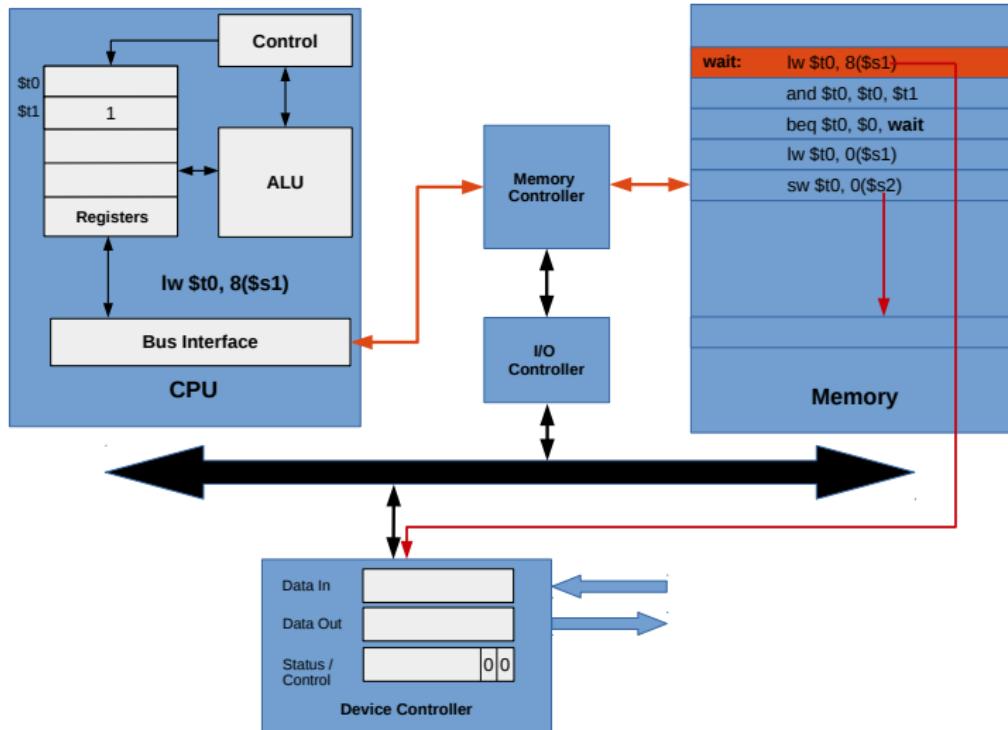
Busy Waiting



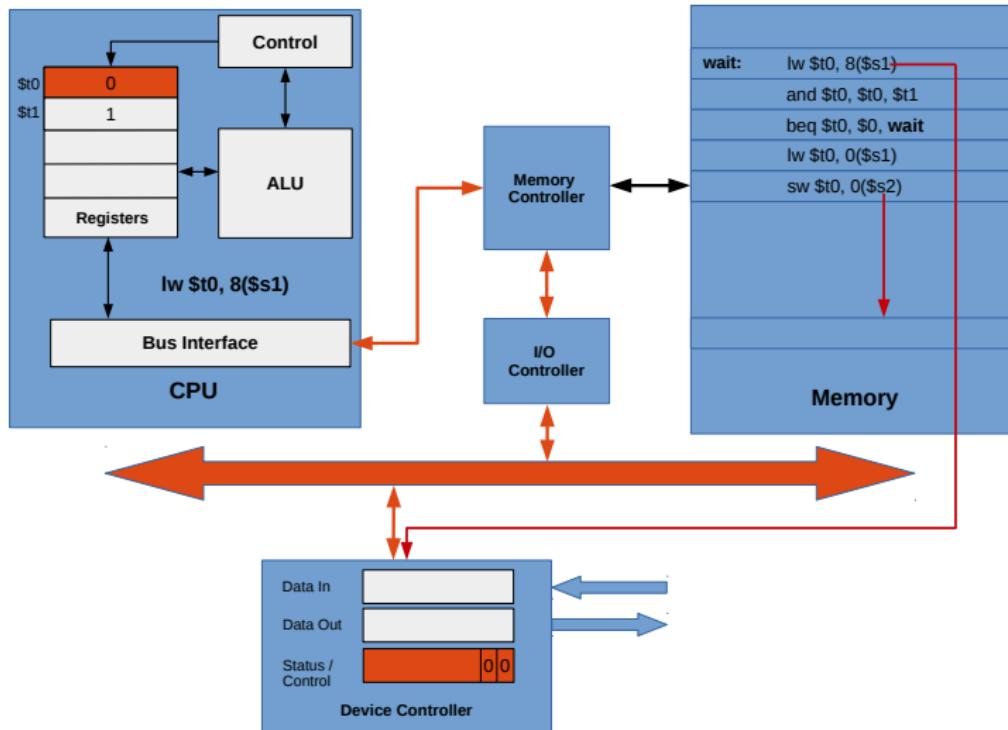
Busy Waiting



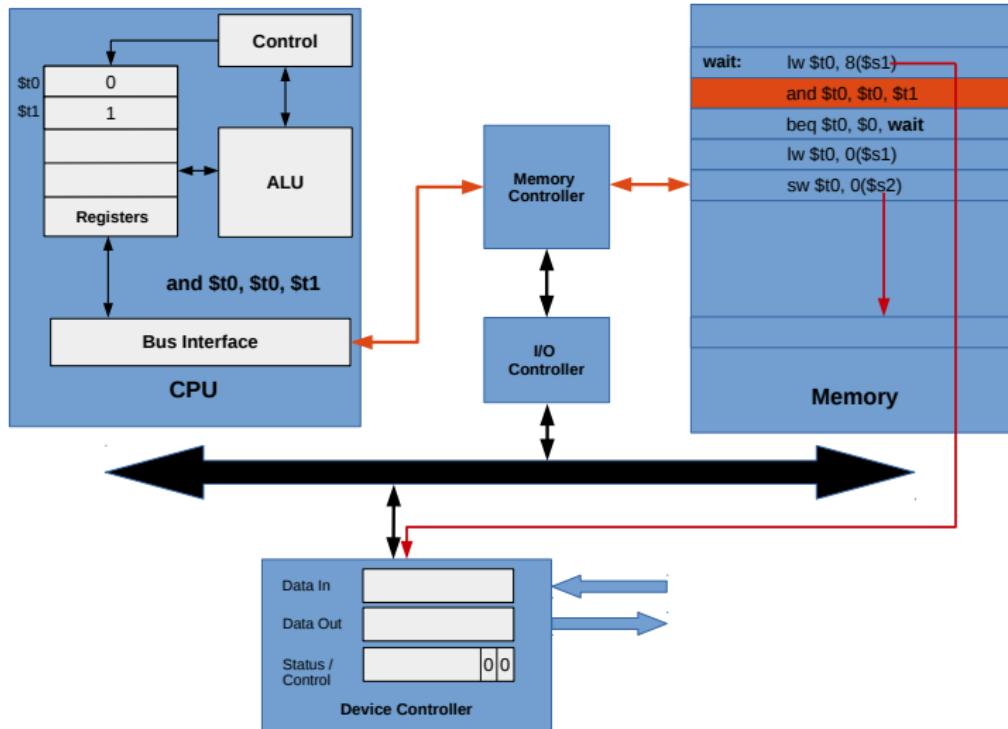
Busy Waiting



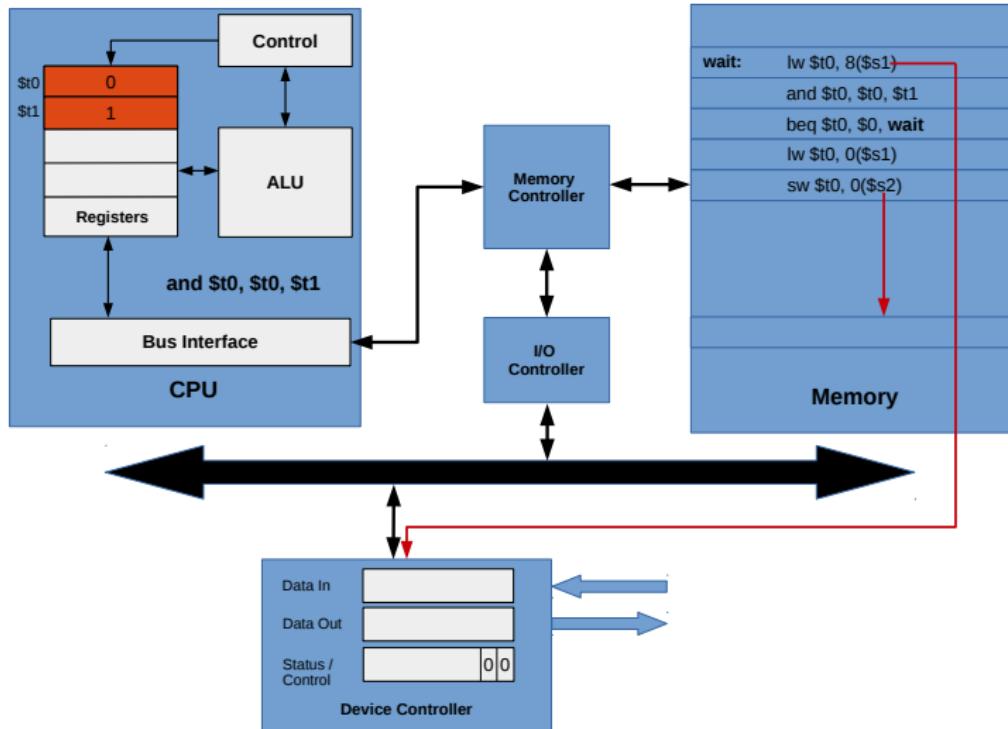
Busy Waiting



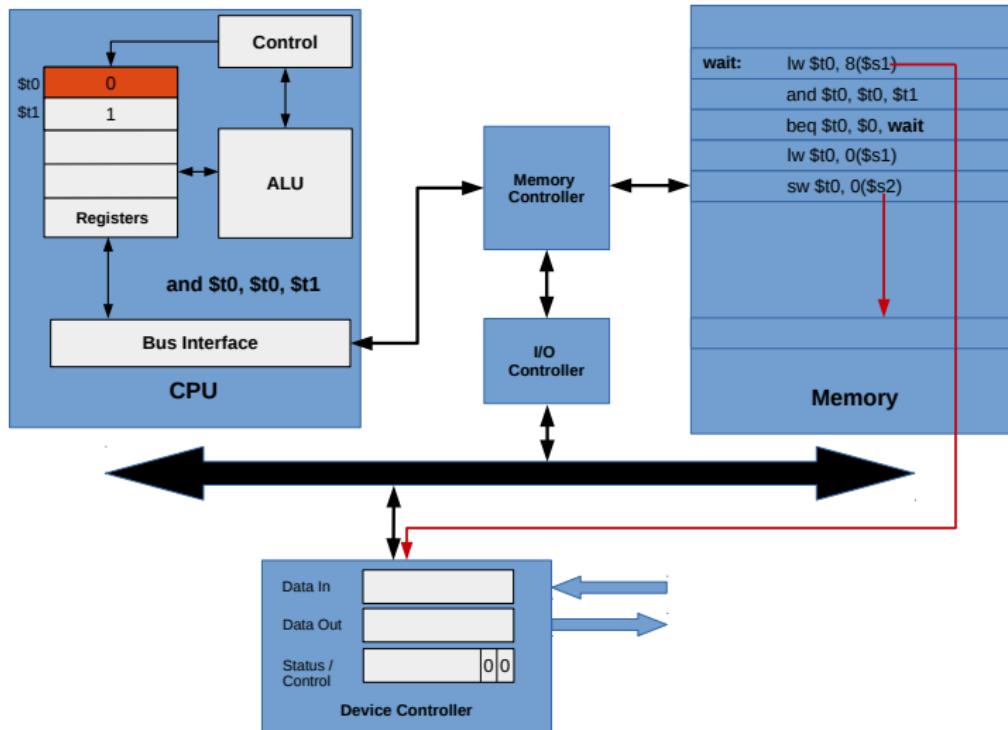
Busy Waiting



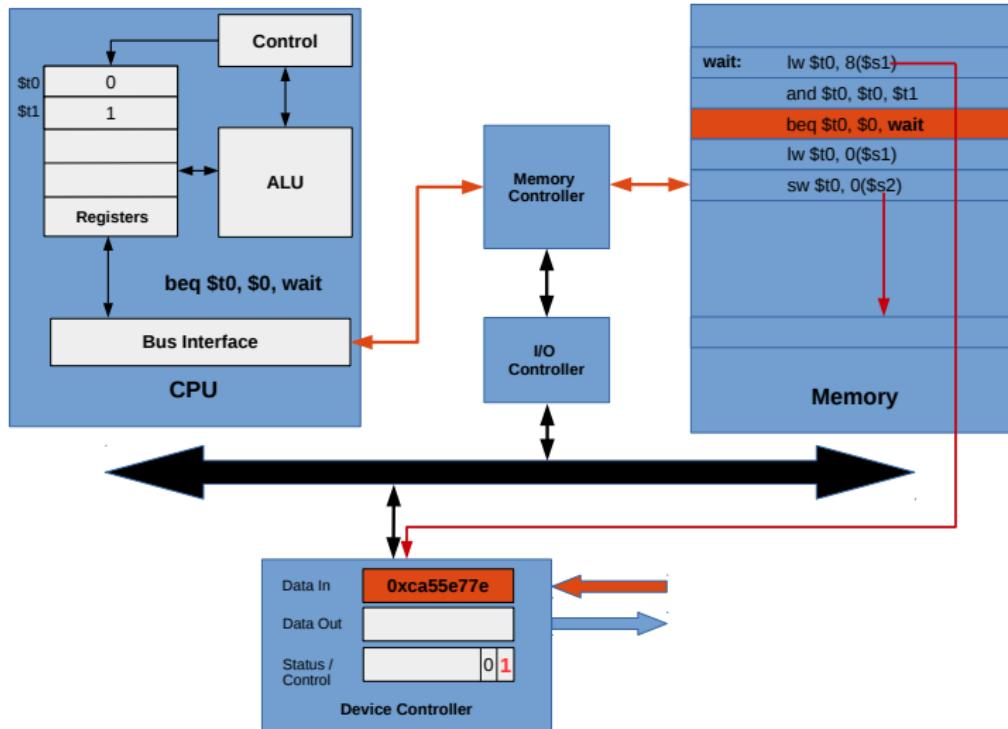
Busy Waiting



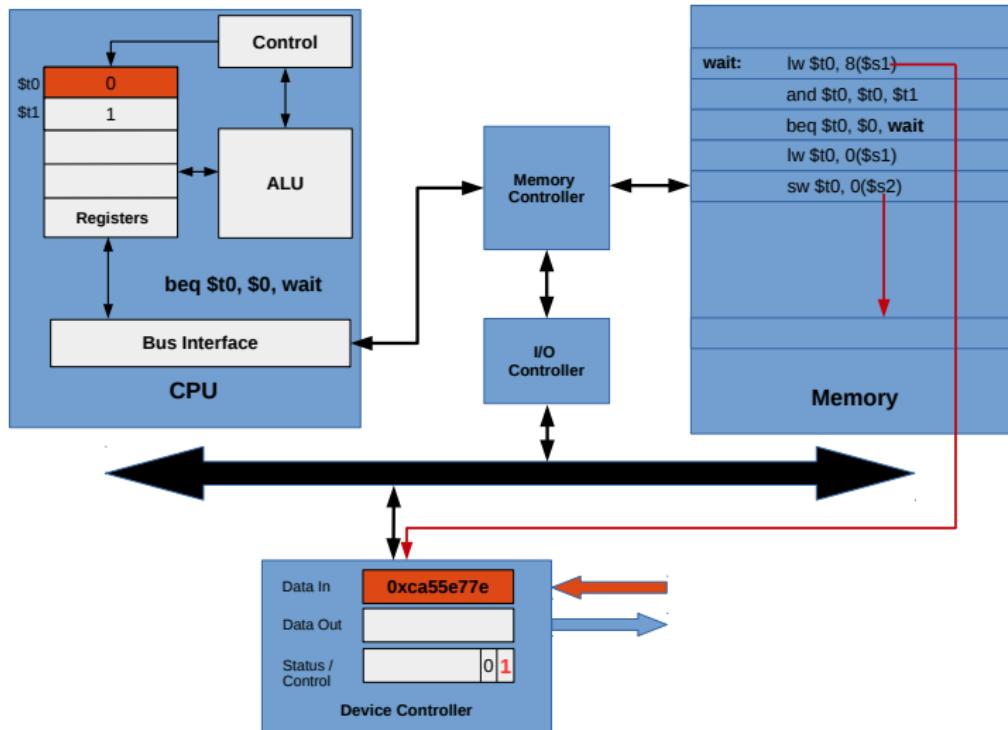
Busy Waiting



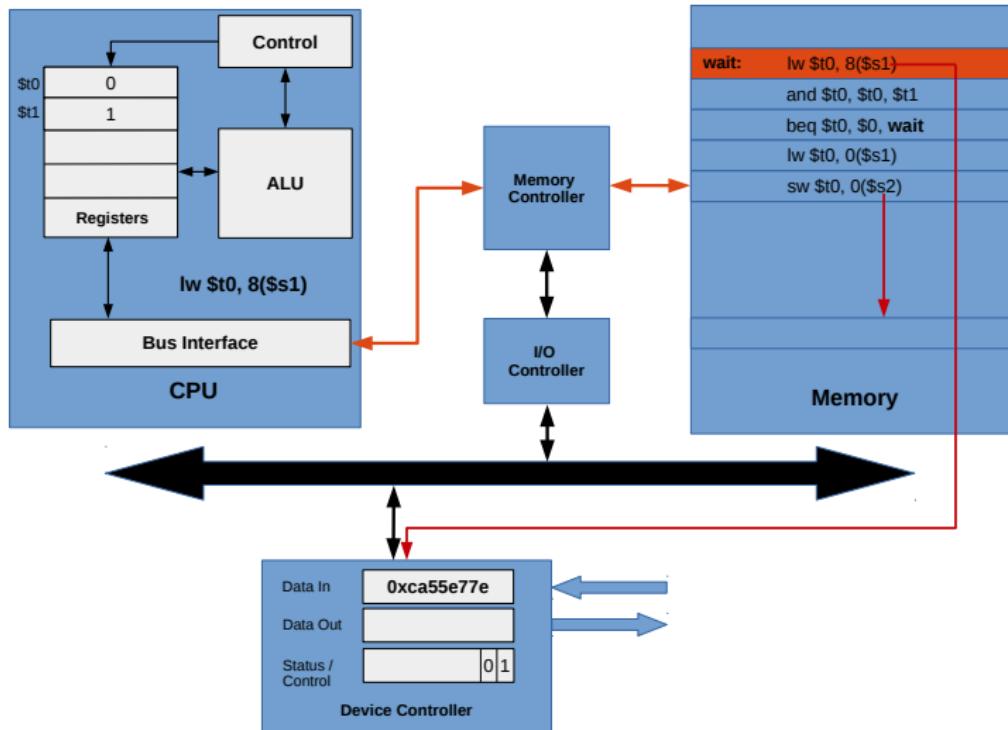
Busy Waiting



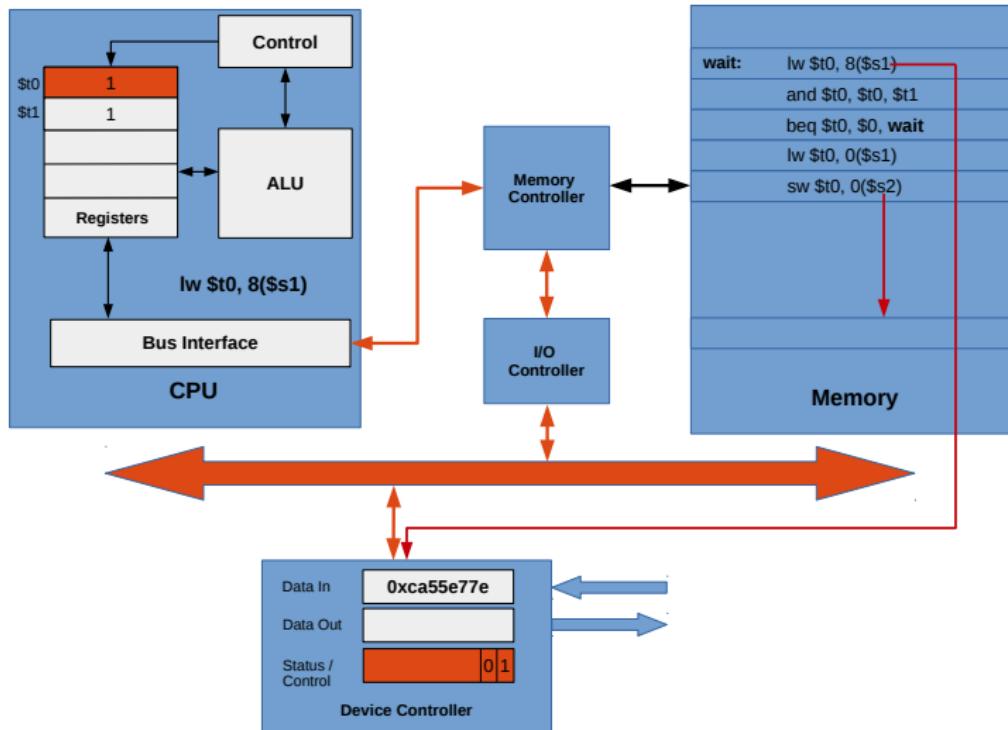
Busy Waiting



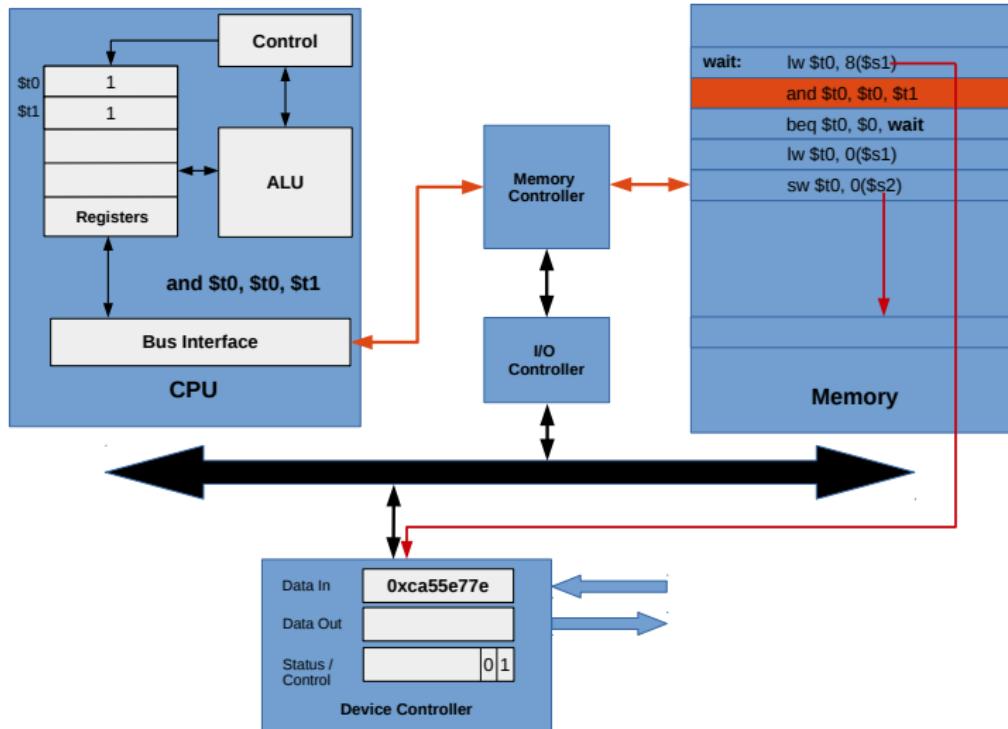
Busy Waiting



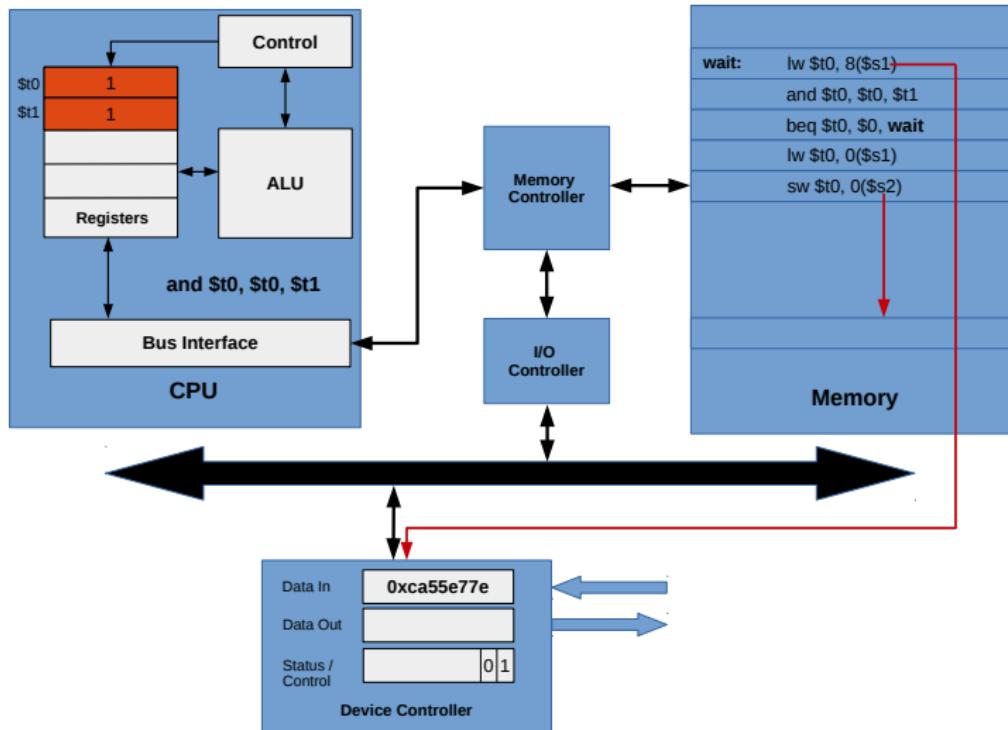
Busy Waiting



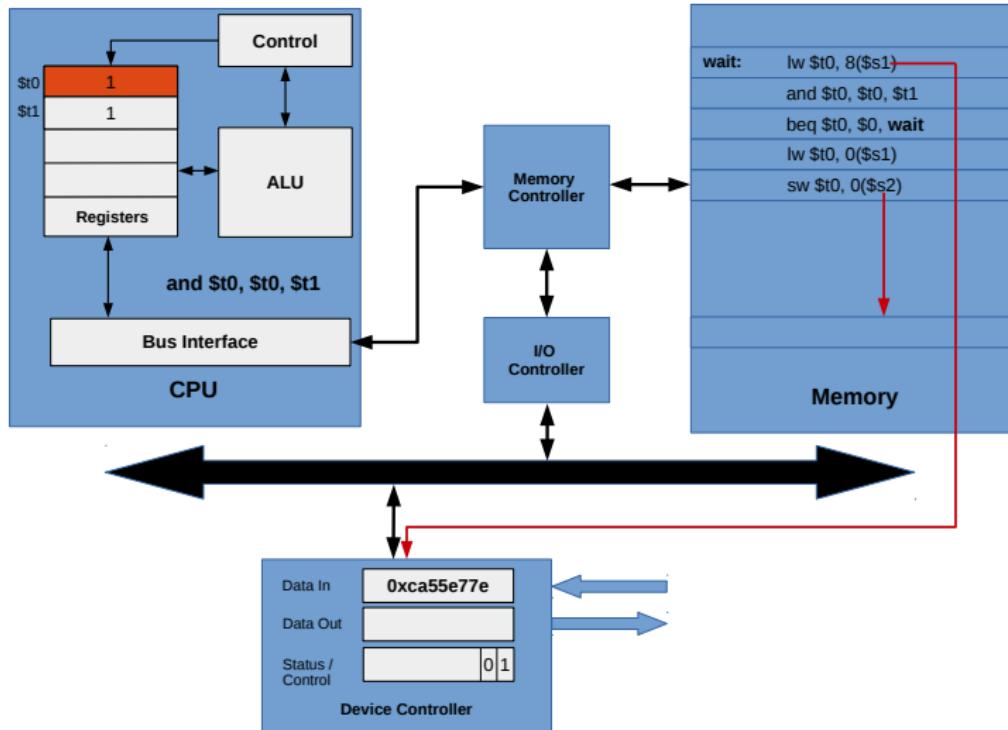
Busy Waiting



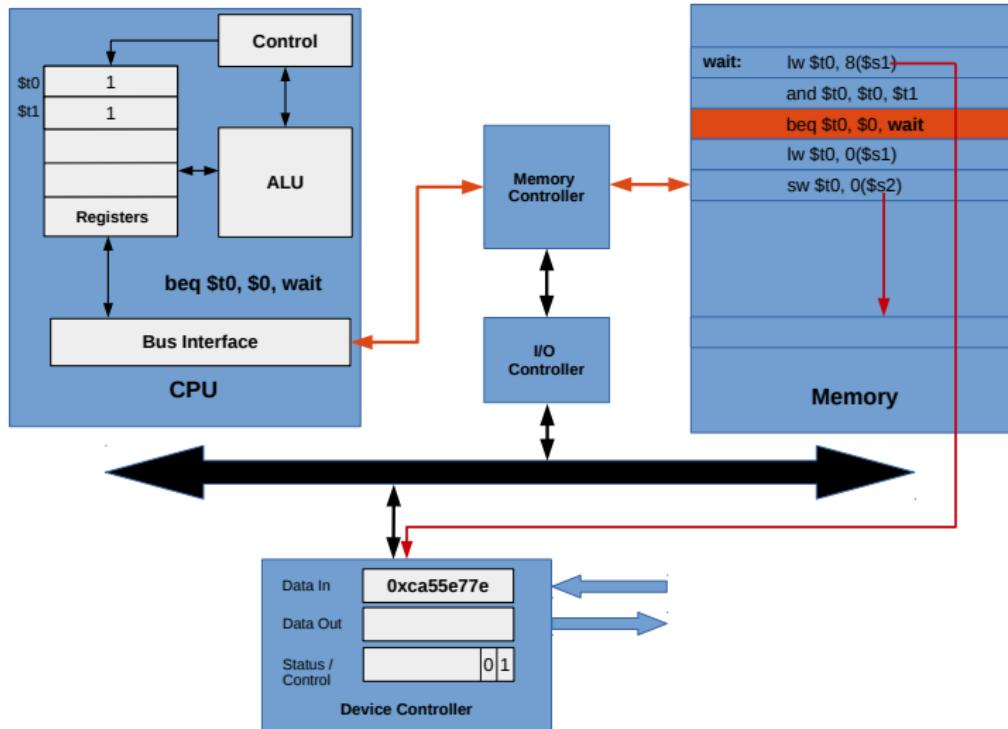
Busy Waiting



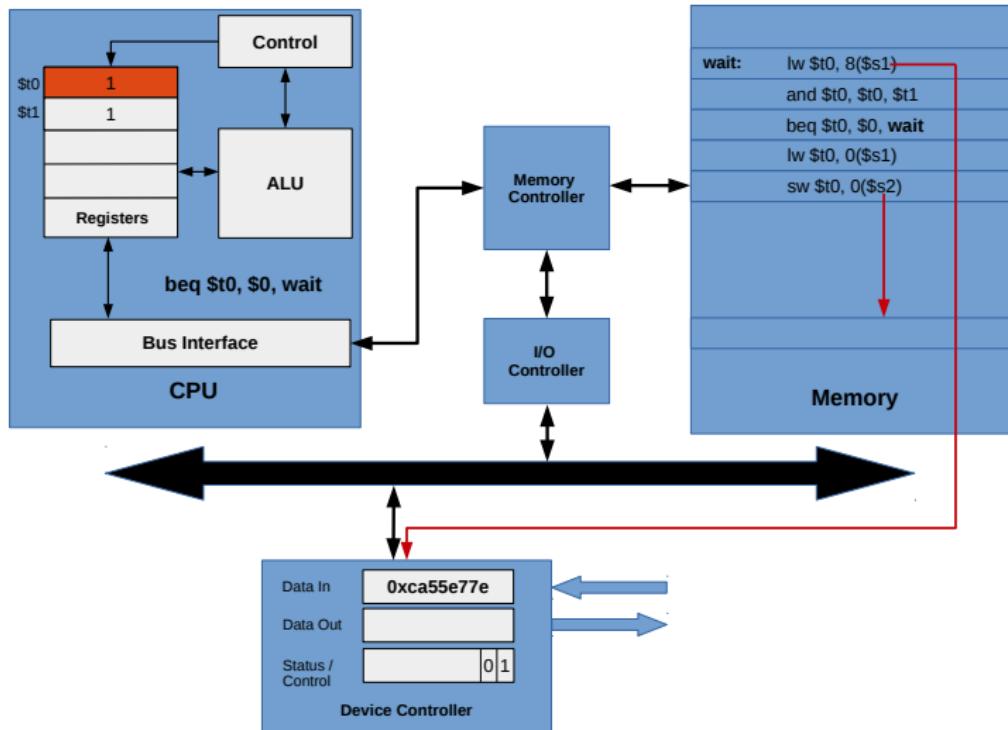
Busy Waiting



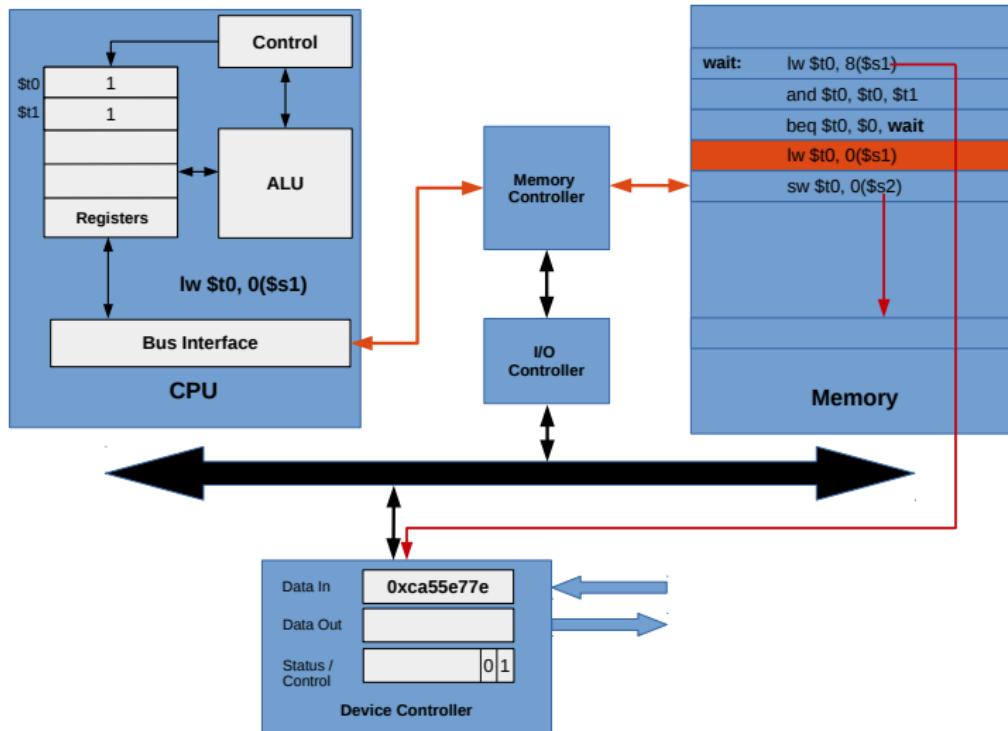
Busy Waiting



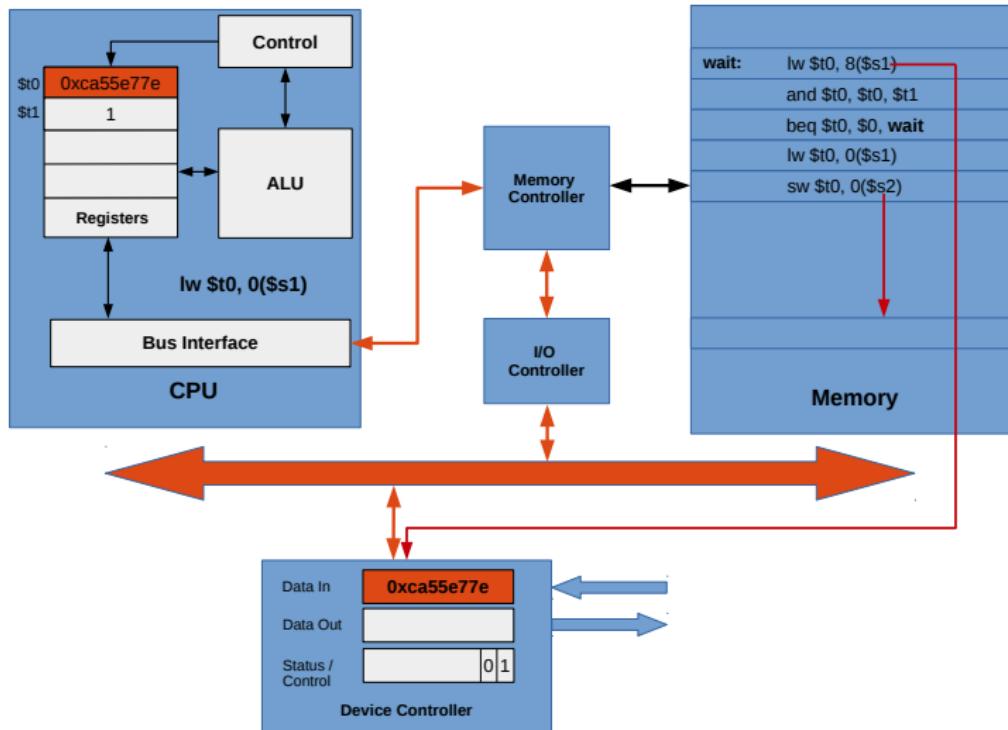
Busy Waiting



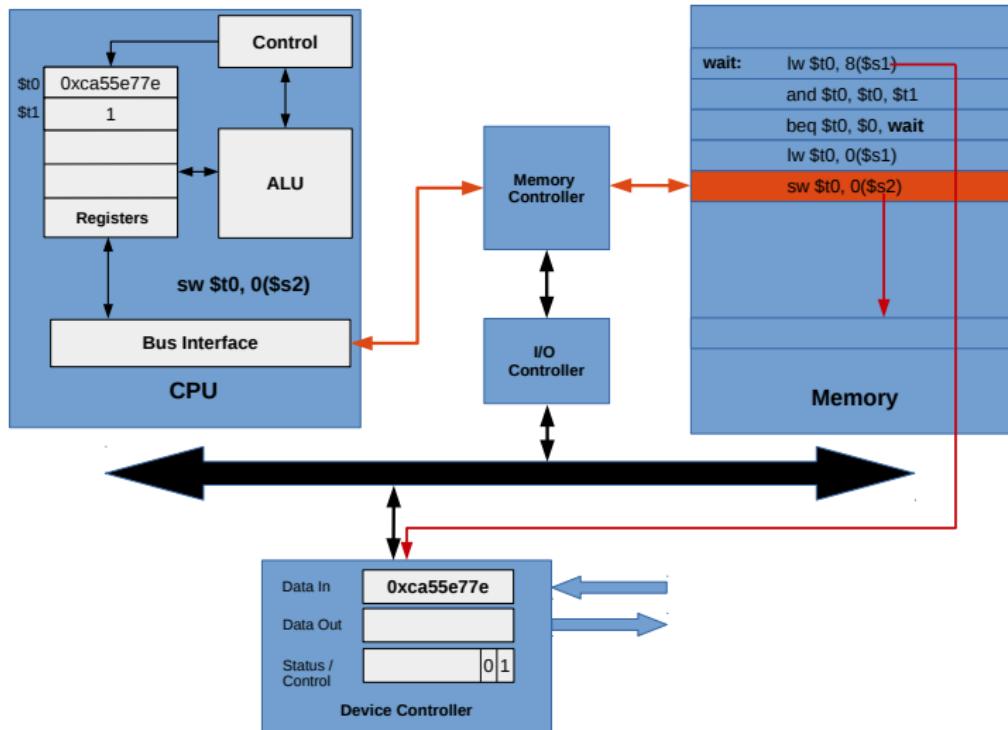
Busy Waiting



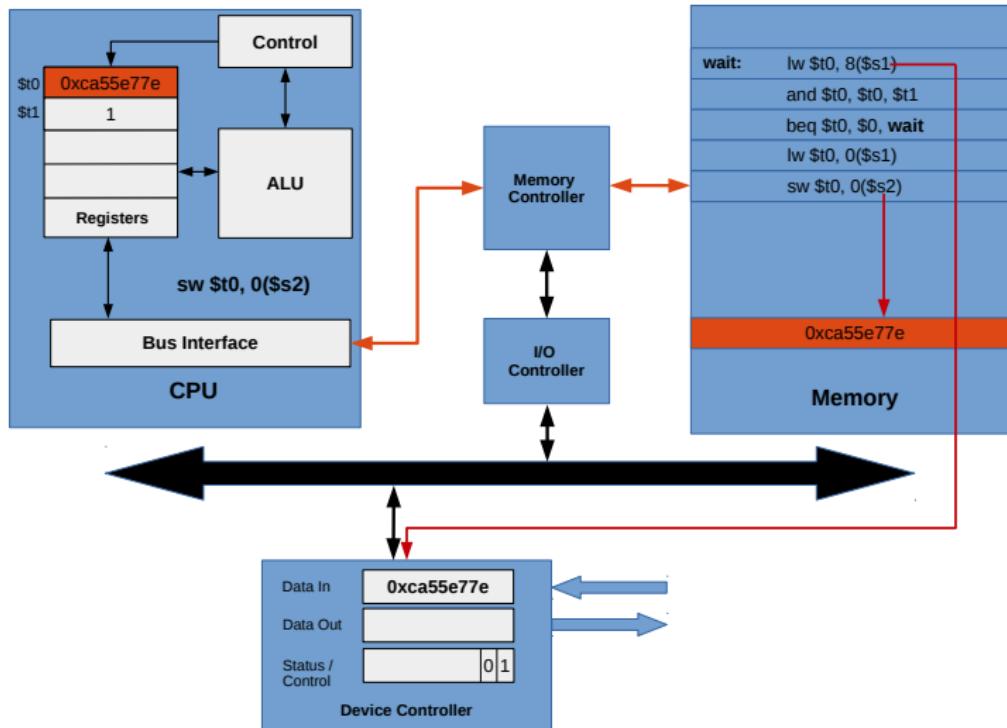
Busy Waiting



Busy Waiting



Busy Waiting

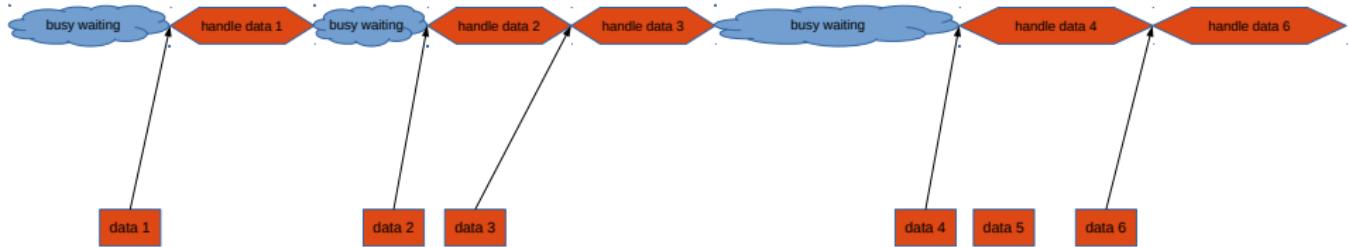


Busy Waiting Problems

- We waste CPU time to continually check the status register of the device controller.
- We can miss some data if we apply a long computation to the retrieved data.

```
1  # ...
2  # $t1 = 1
3  # $t2 = 0xffffffff
4  # $s1 = address of device controller
5  wait:
6      lw  $t0, 8($s1)    # get status register
7      and $t3, $t0, $t1 # mask receive flag
8      beq $t3, $0, wait
9      and $t3, $t0, $t2 # reset receive flag
10     lw   $a0, 0($s1)   # get data
11     sw   $t3, 8($s1)   # set status register
12     jal data_handler  # call handler
13     j    wait
```

Busy Waiting Problems



Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

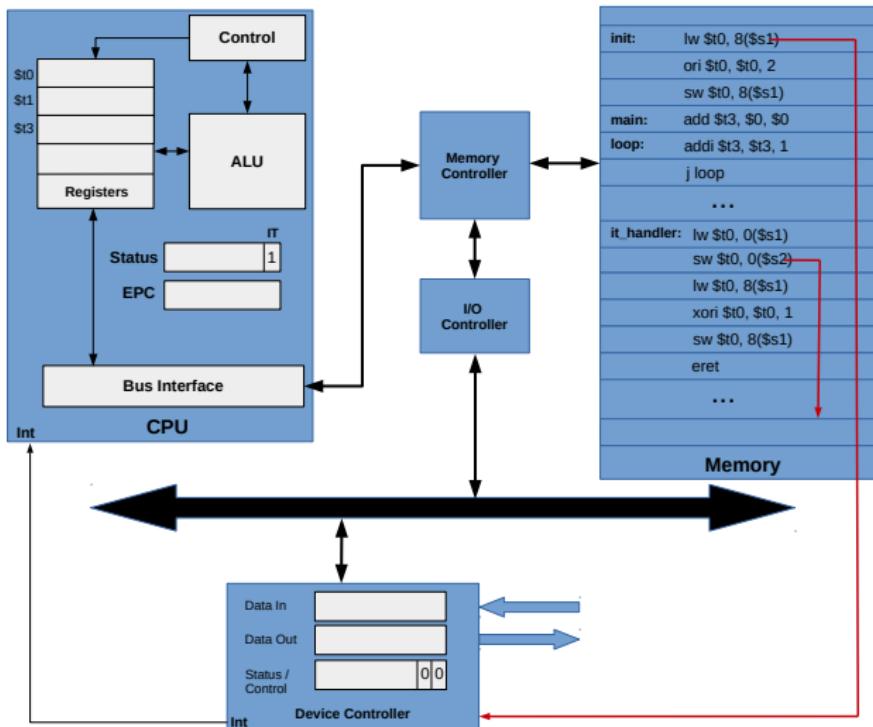
Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Interrupts and Exceptions
I/O Devices
Waiting for I/O Devices
Ints and Exceptions on MIPS

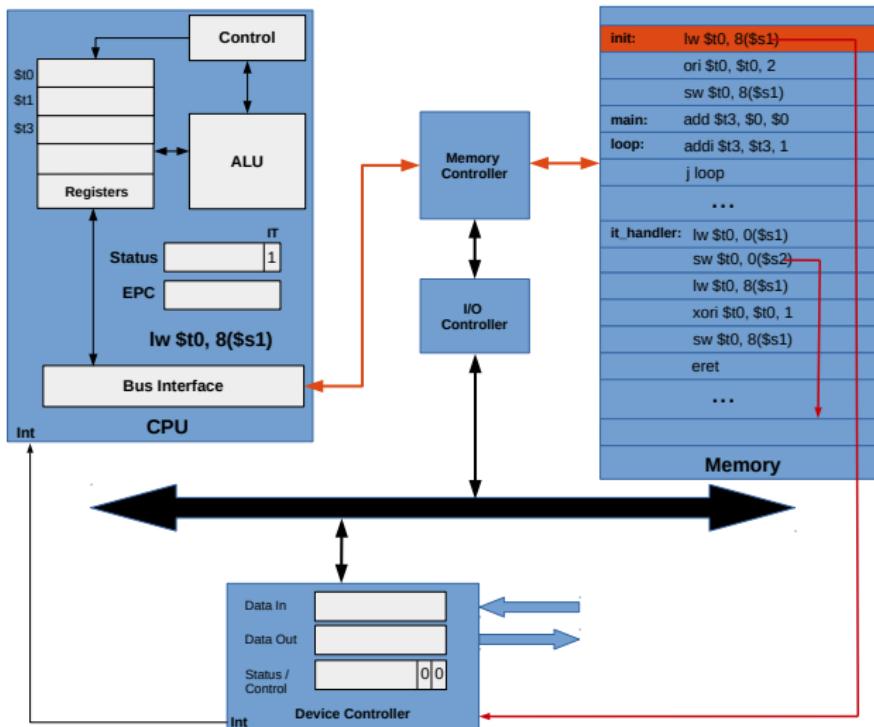
Single-Cycle with Ints and Exs
Ints and Exceptions on x86_64
Linux Signals

Getting Notified by Interrupts

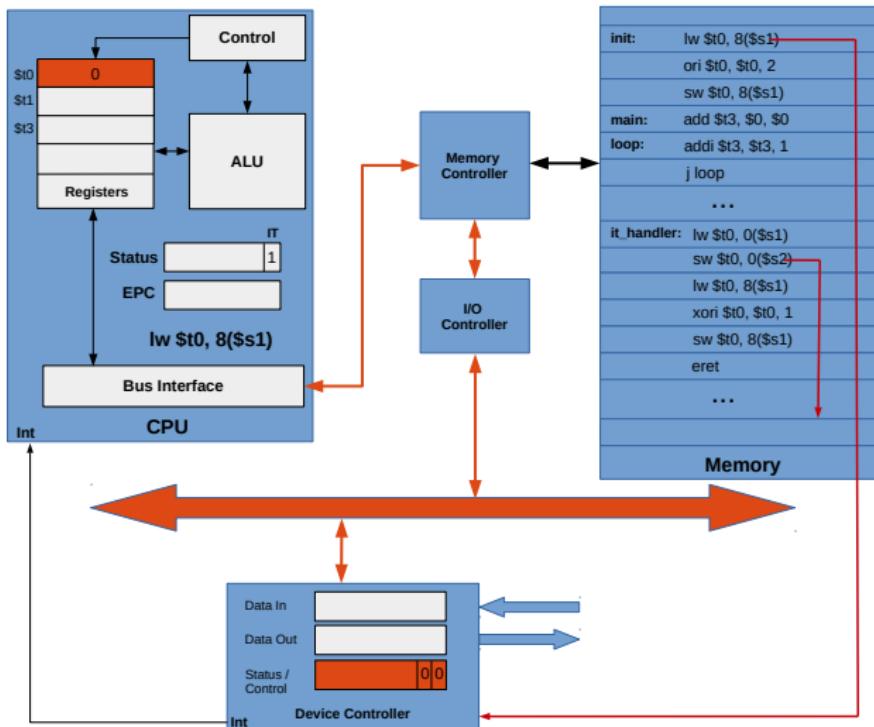
Getting Notified by Interrupts



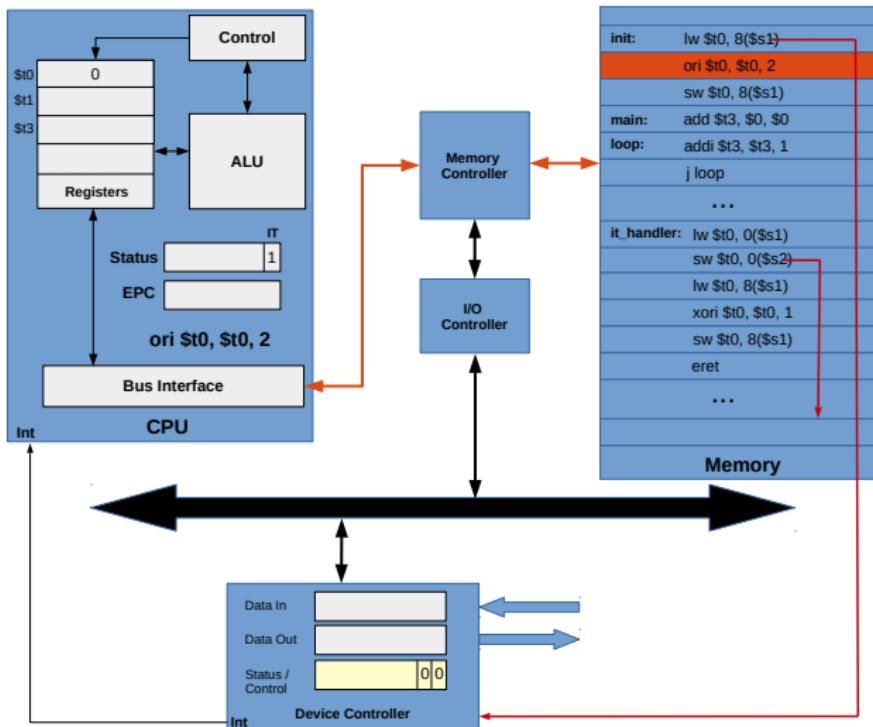
Getting Notified by Interrupts



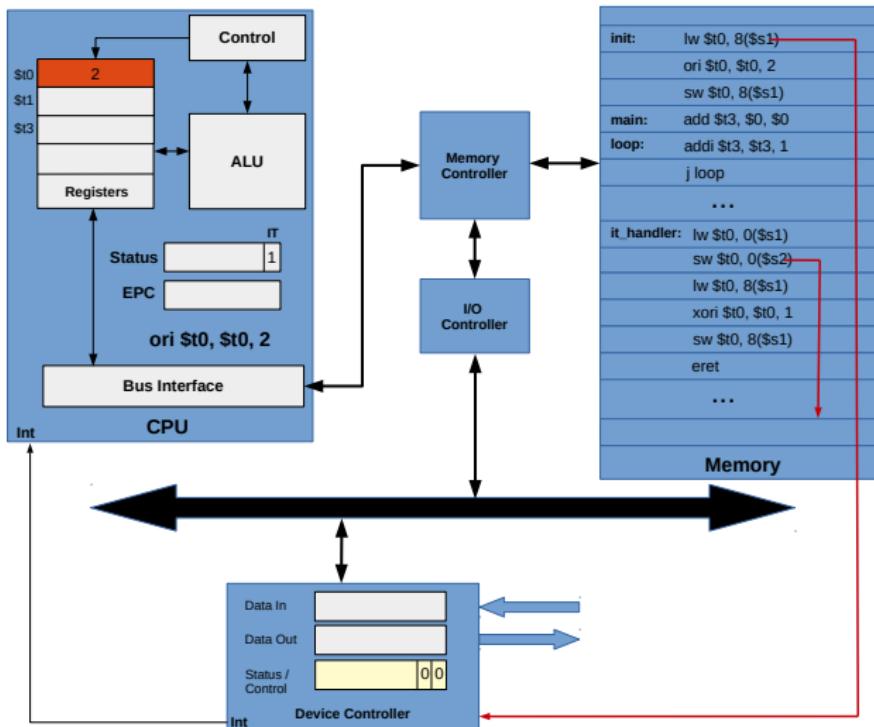
Getting Notified by Interrupts



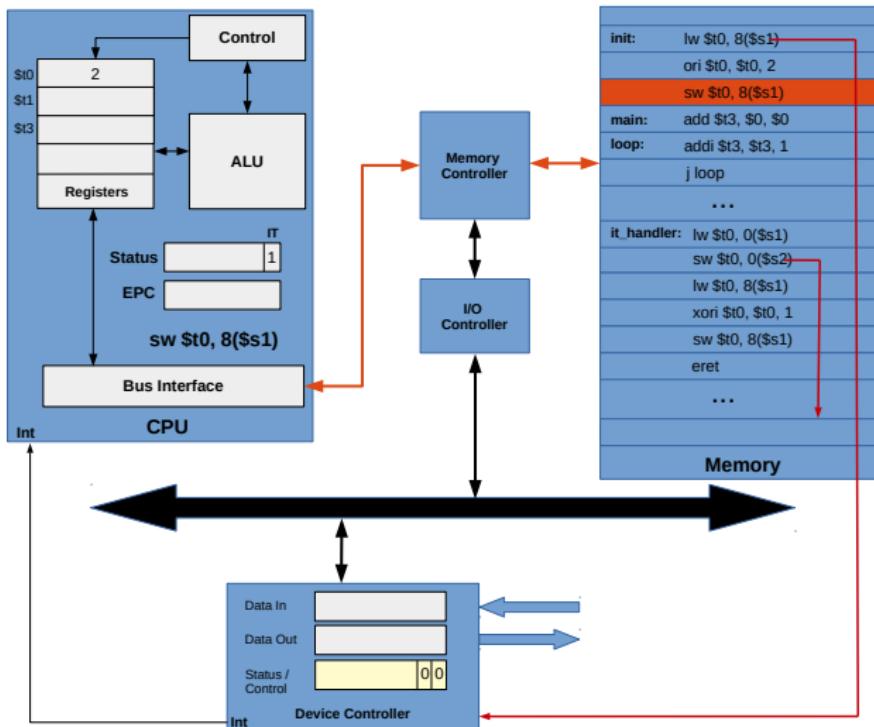
Getting Notified by Interrupts



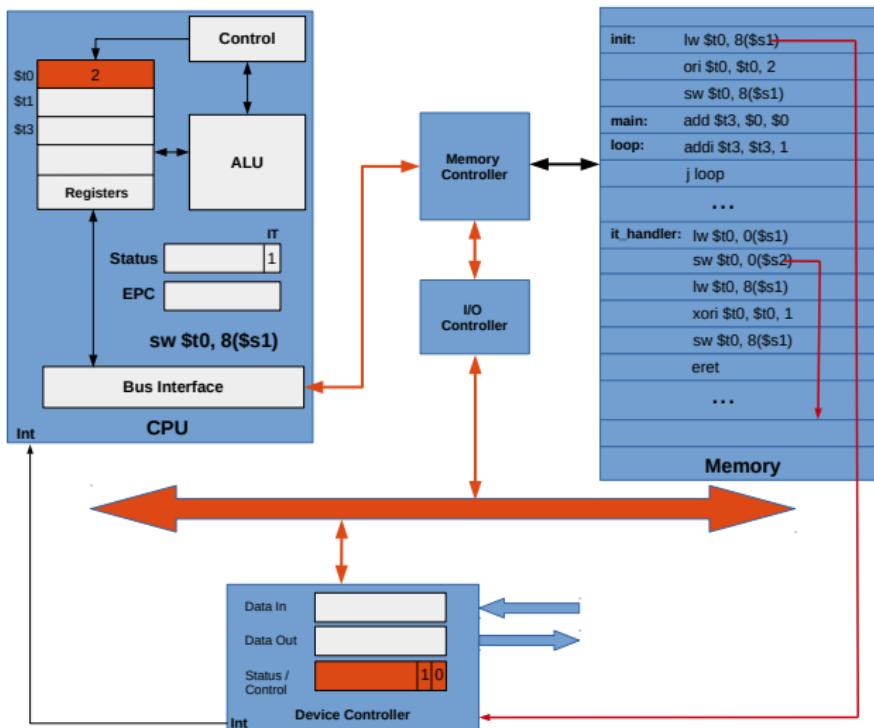
Getting Notified by Interrupts



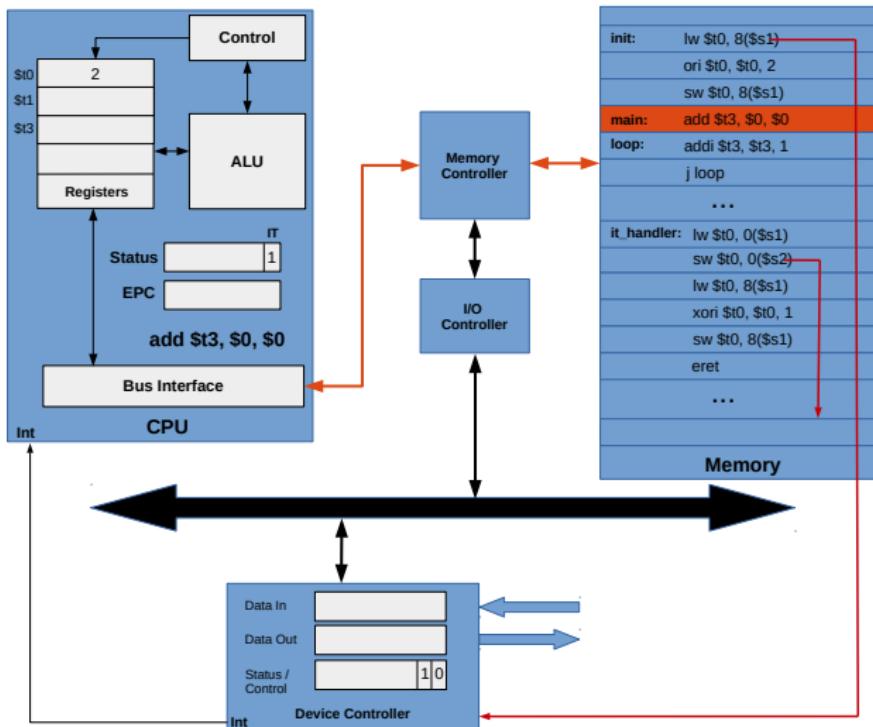
Getting Notified by Interrupts



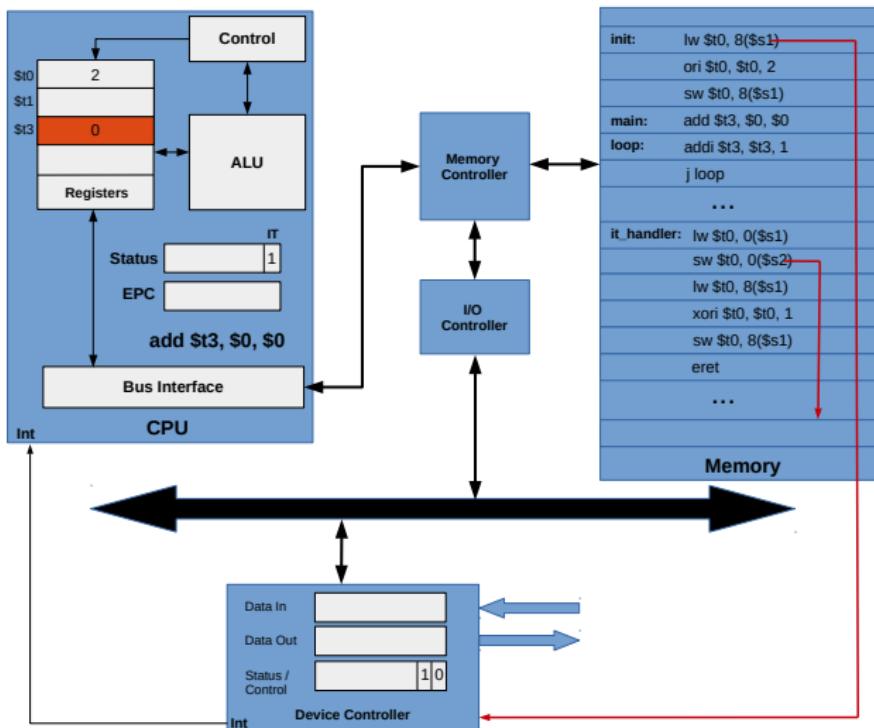
Getting Notified by Interrupts



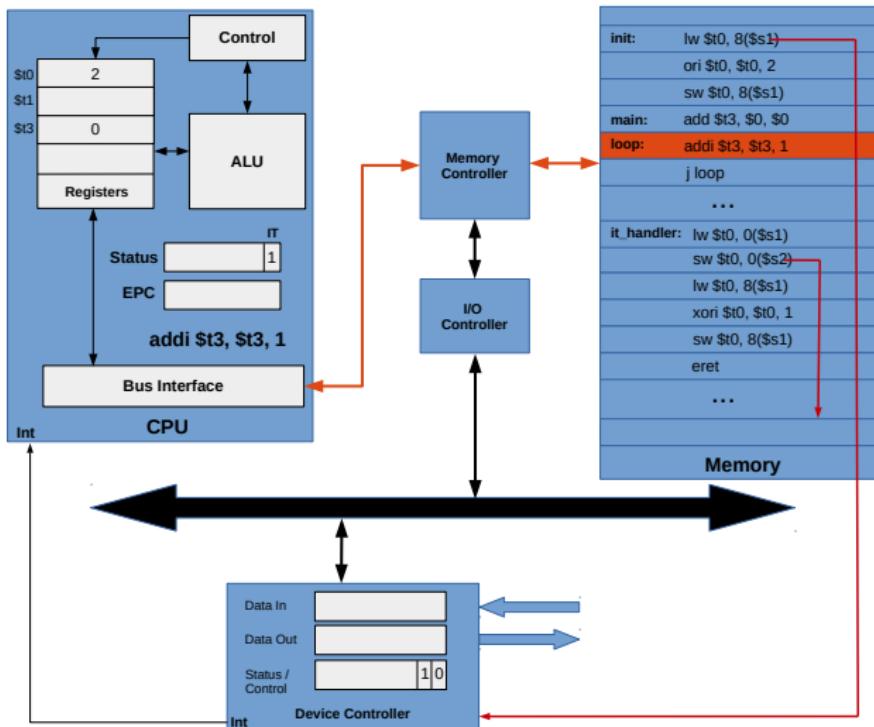
Getting Notified by Interrupts



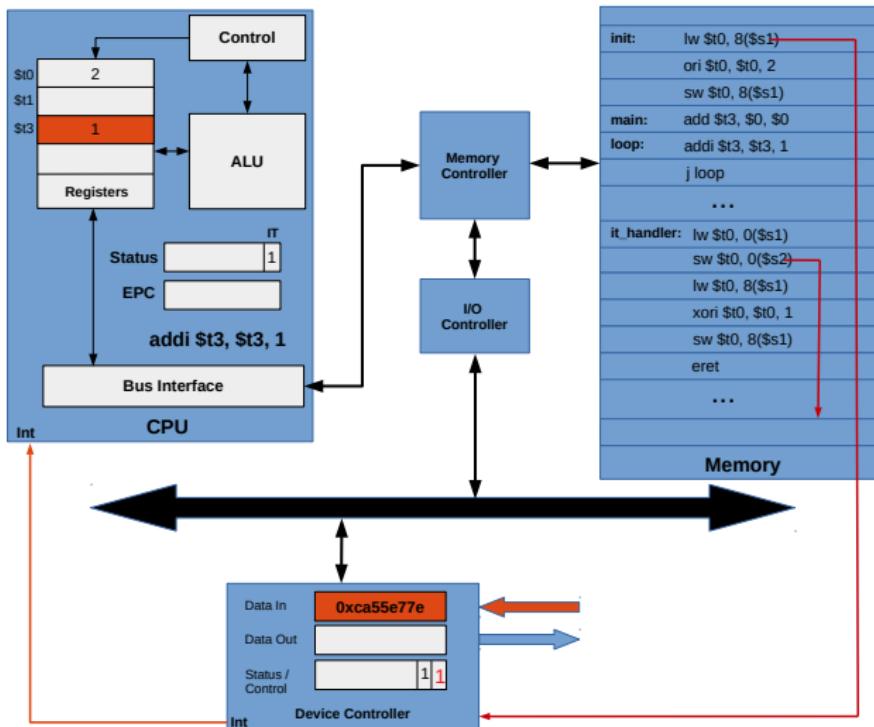
Getting Notified by Interrupts



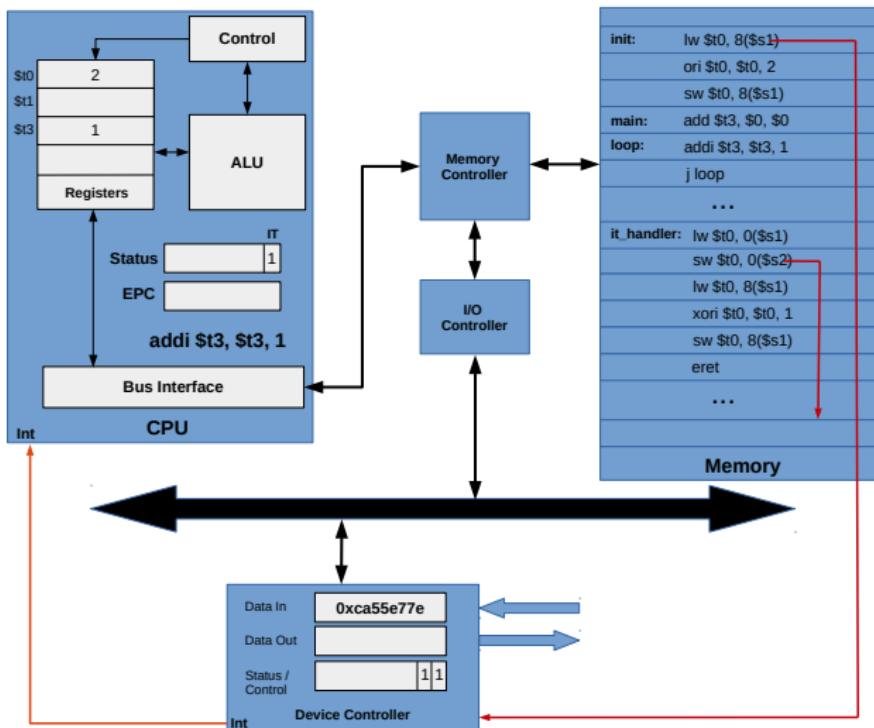
Getting Notified by Interrupts



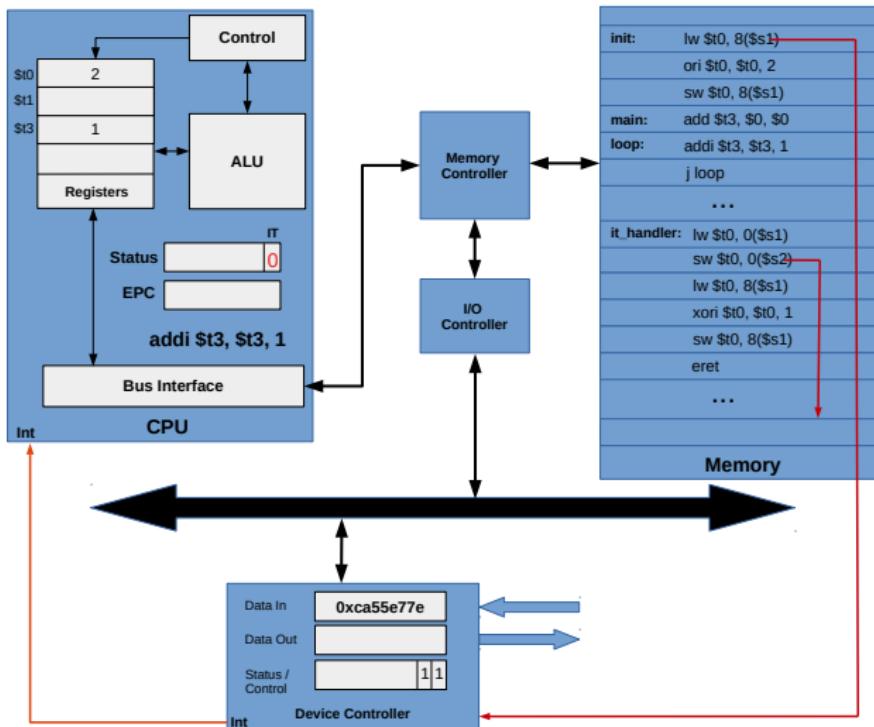
Getting Notified by Interrupts



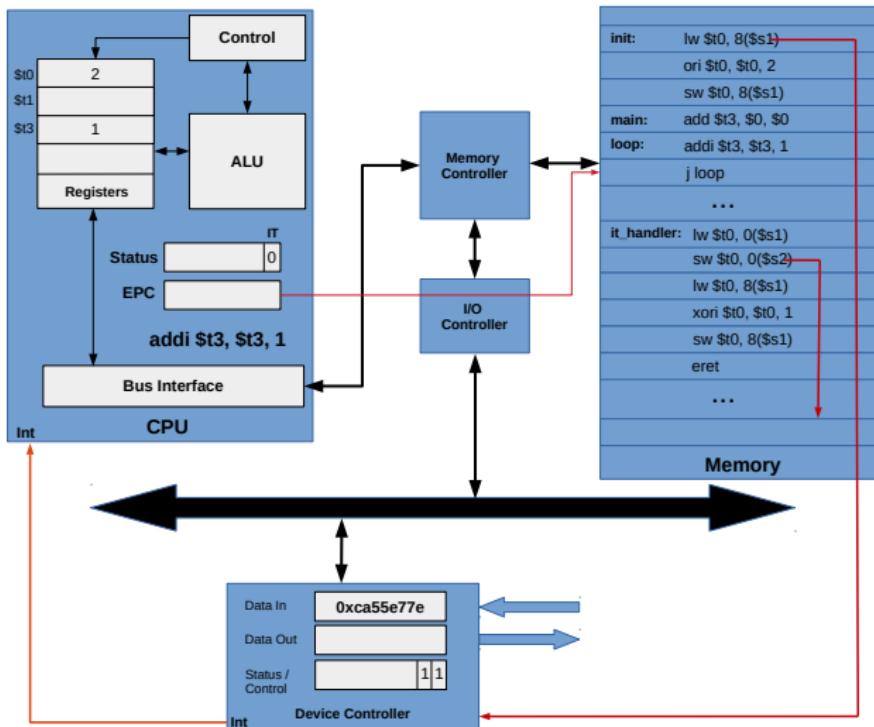
Getting Notified by Interrupts



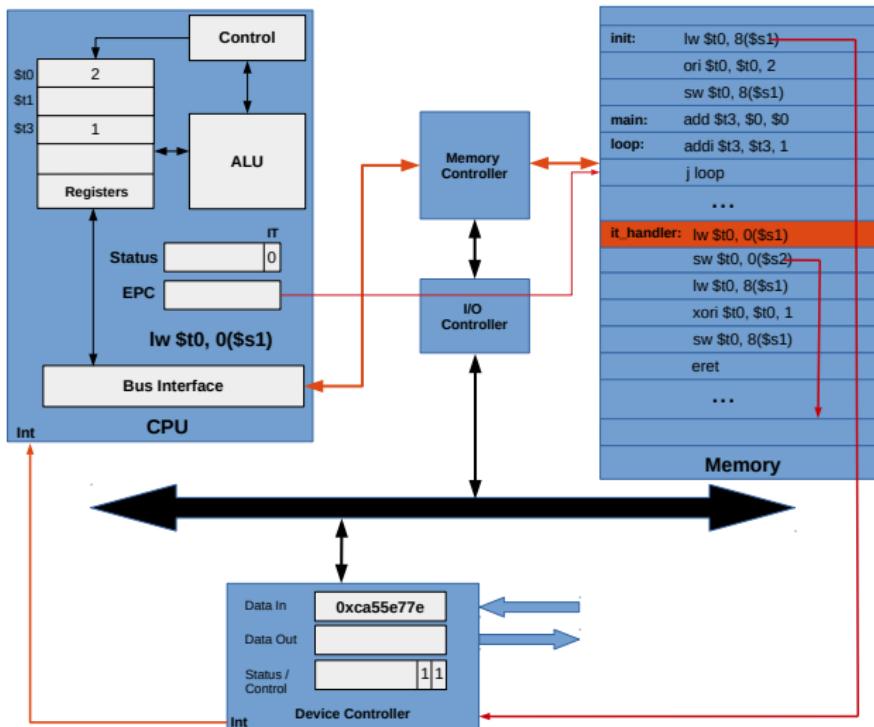
Getting Notified by Interrupts



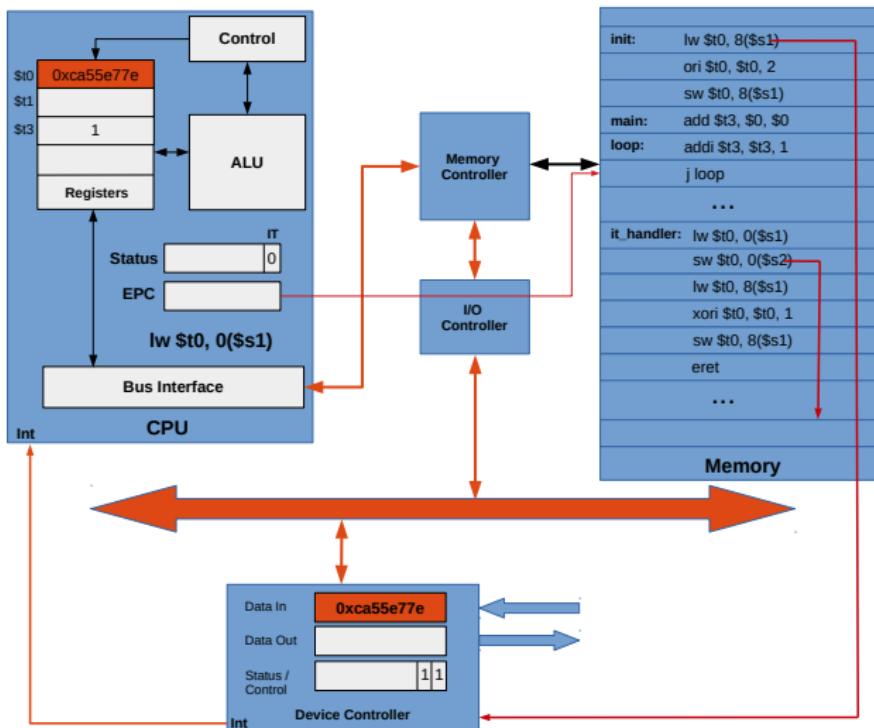
Getting Notified by Interrupts



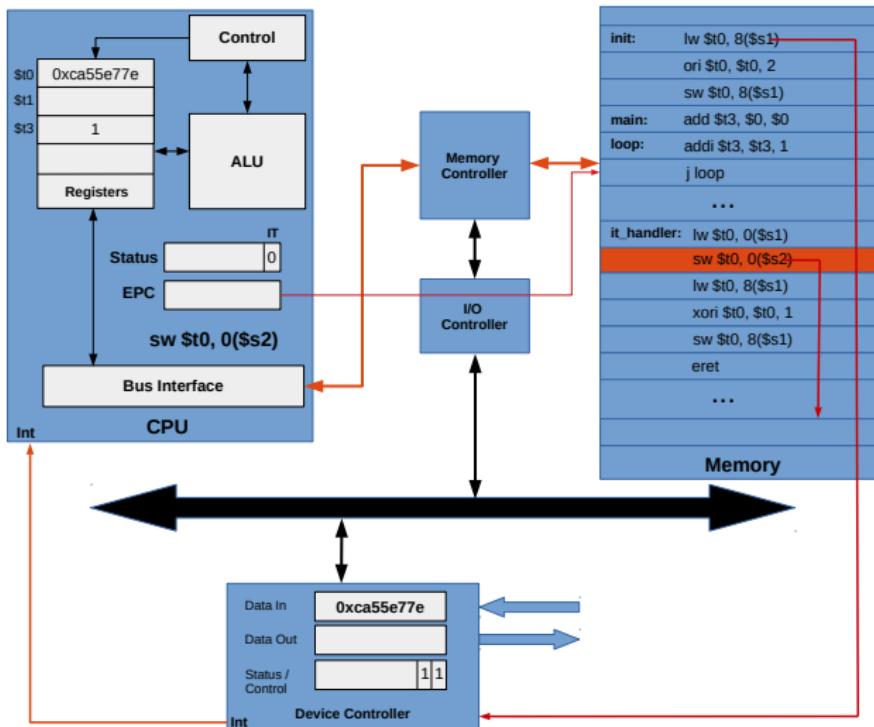
Getting Notified by Interrupts



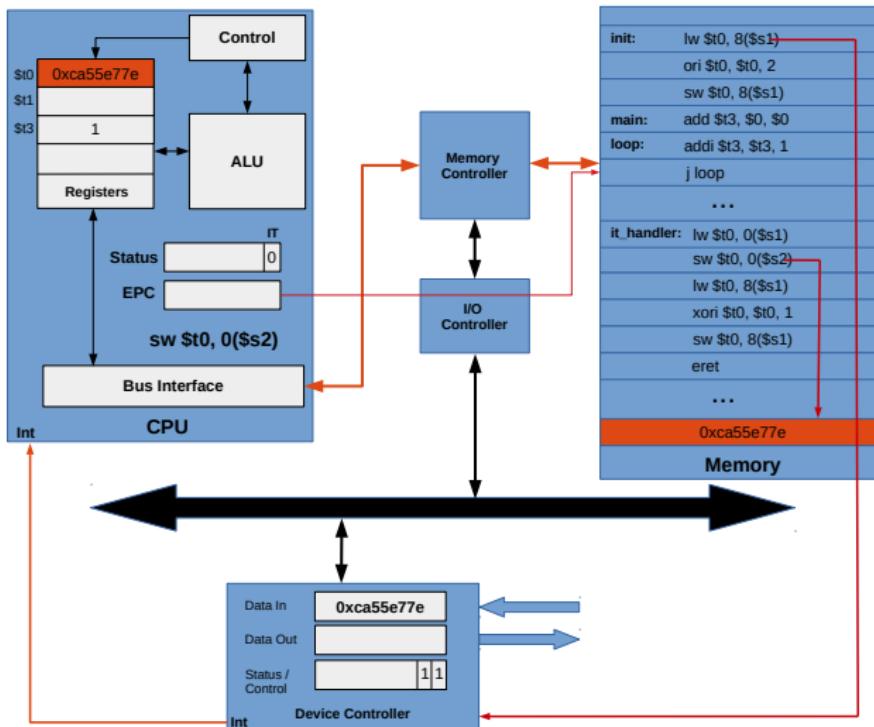
Getting Notified by Interrupts



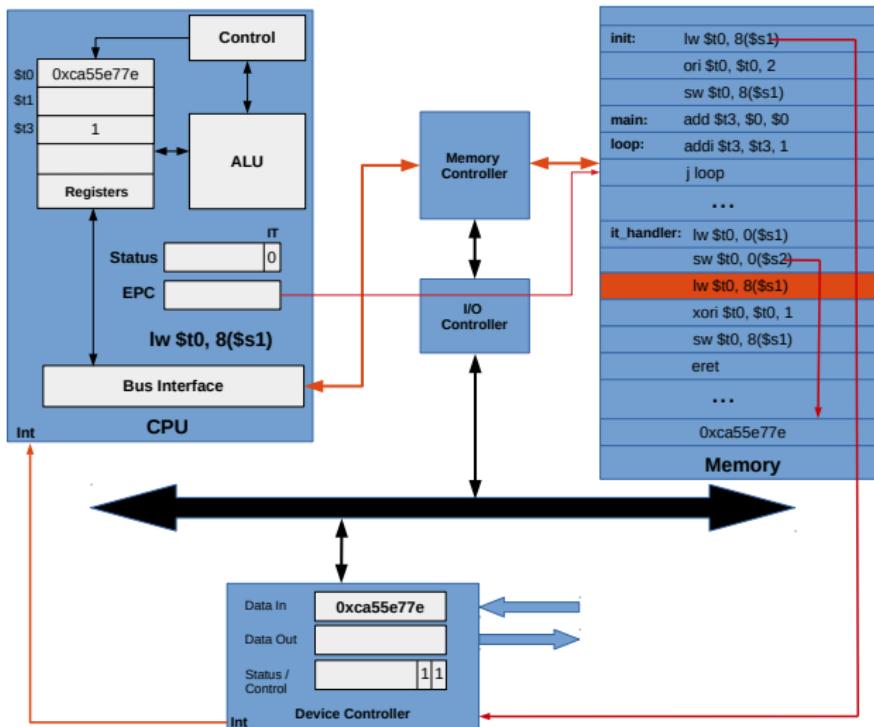
Getting Notified by Interrupts



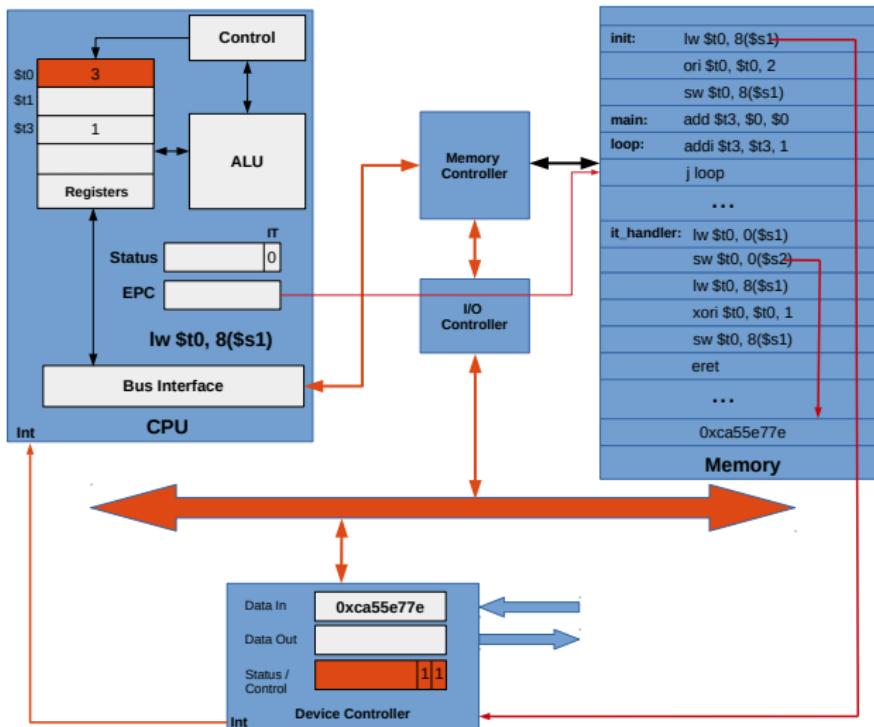
Getting Notified by Interrupts



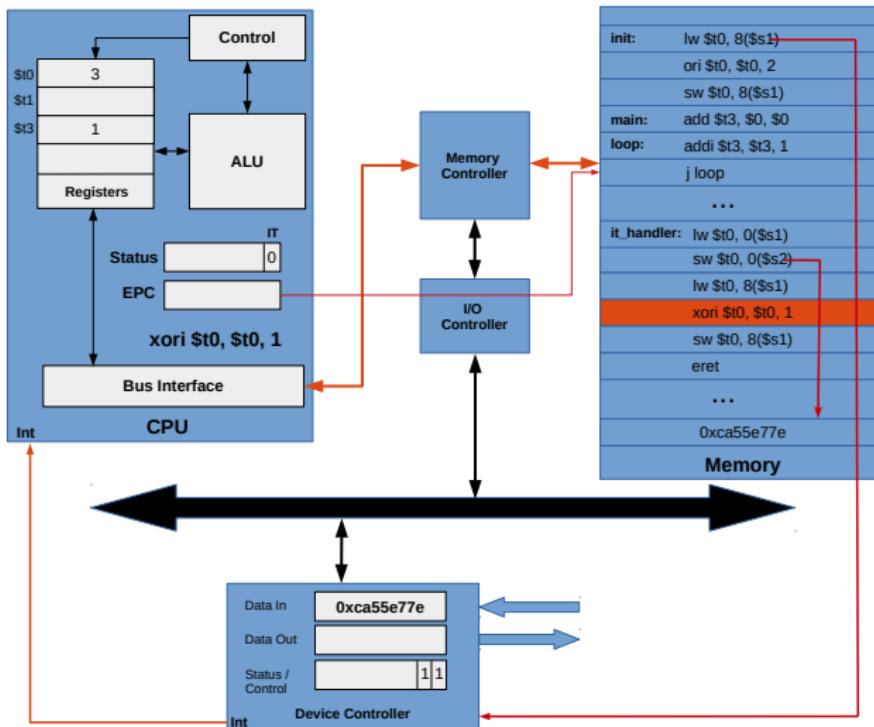
Getting Notified by Interrupts



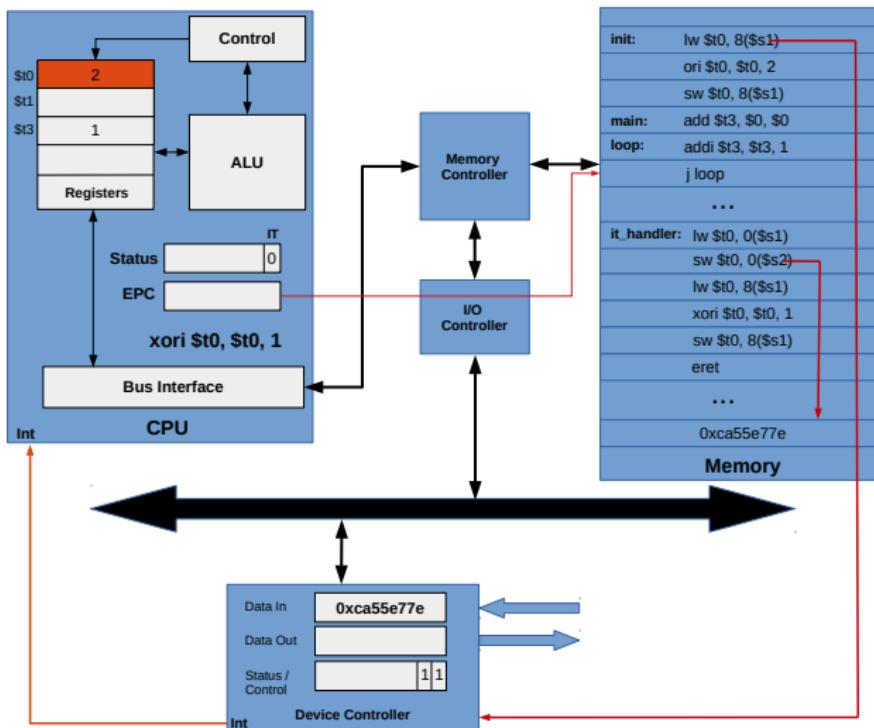
Getting Notified by Interrupts



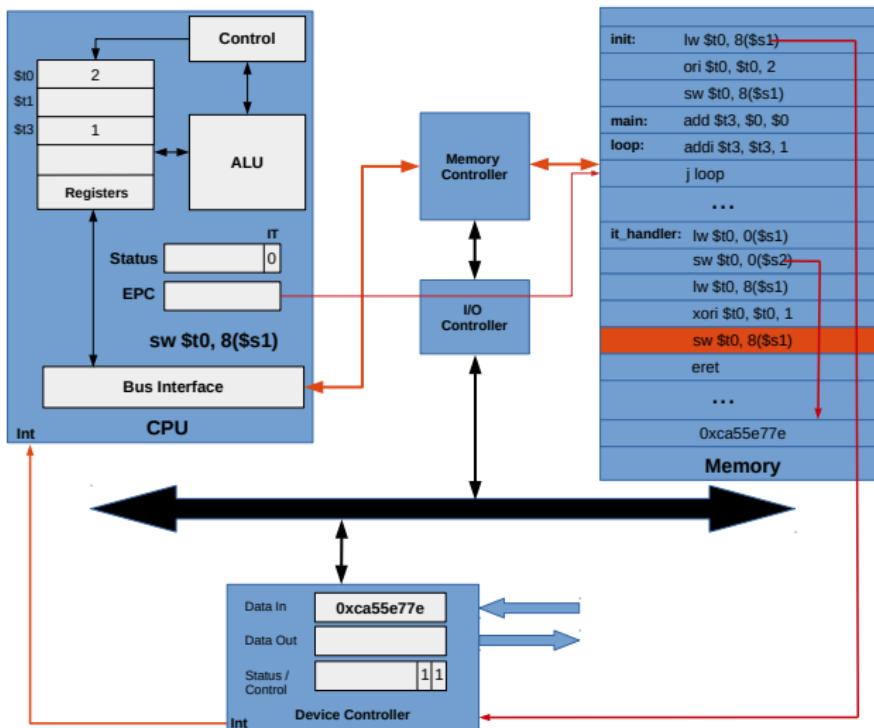
Getting Notified by Interrupts



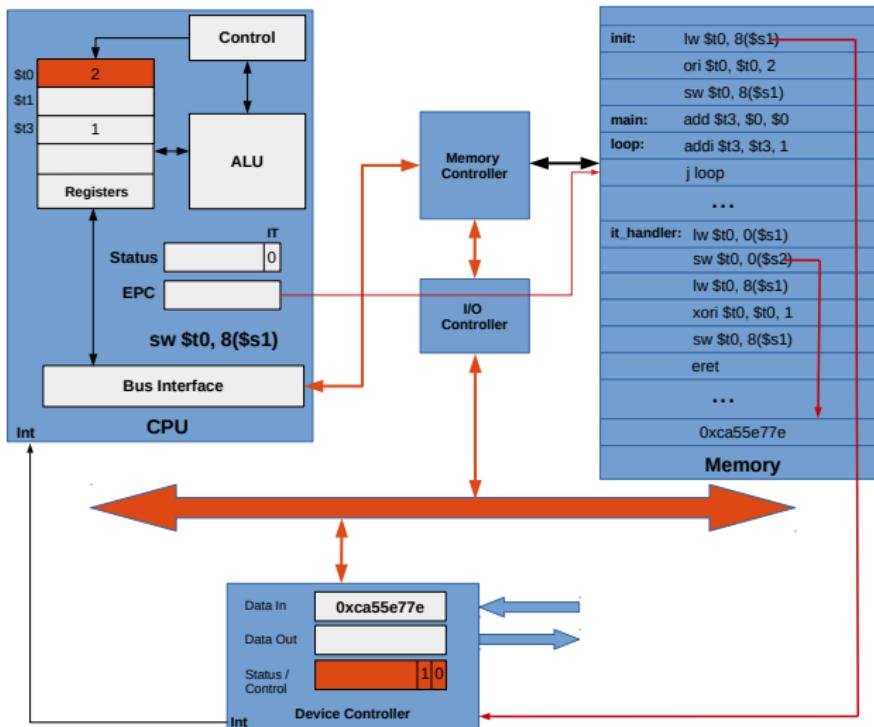
Getting Notified by Interrupts



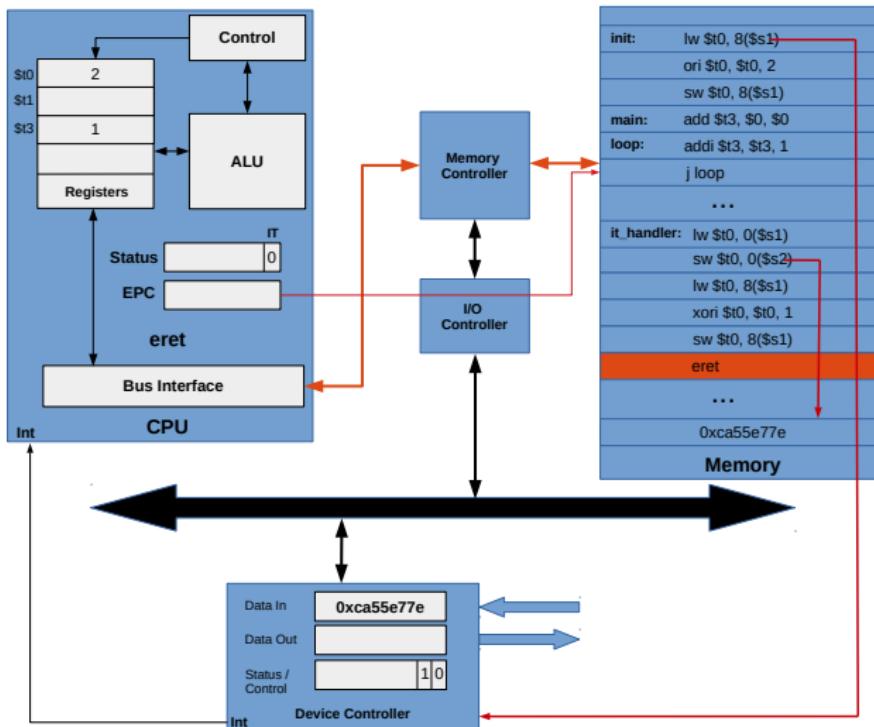
Getting Notified by Interrupts



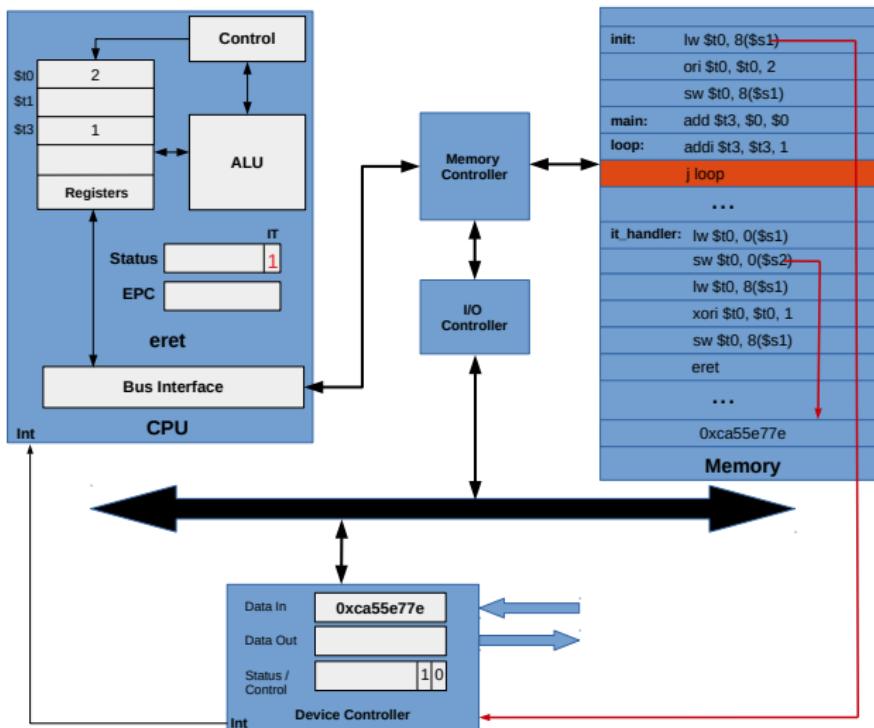
Getting Notified by Interrupts



Getting Notified by Interrupts



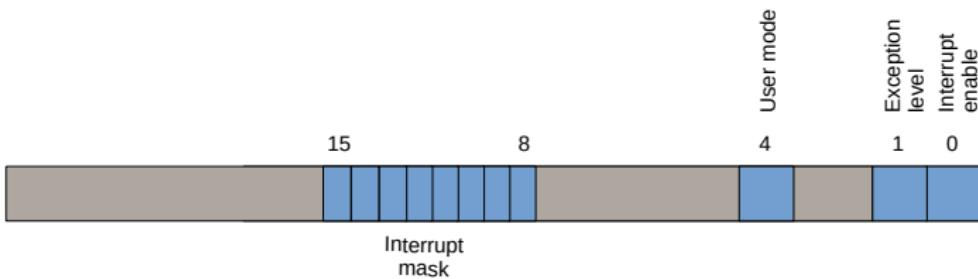
Getting Notified by Interrupts



MIPS Interrupts and Exceptions

- Integer arithmetic and logical operations are executed directly by the CPU
- Floating point operations are executed by Coprocessor 1
- Coprocessor 0 is used to manage exceptions and interrupts
 - Normal user level code doesn't have access to Coprocessor 0. This is restricted to privileged level code
 - But interrupt and exception aware code has to use it
 - Coprocessor 0 has several registers which controls exceptions and interrupts
 - The **Status** register (number 12) which manages the exceptions and interrupts behaviour and user or privileged level code
 - The **Cause** register (number 13) which gives information about the cause of the exception or interrupt
 - The **EPC** register (number 14) which saves the old value of the PC

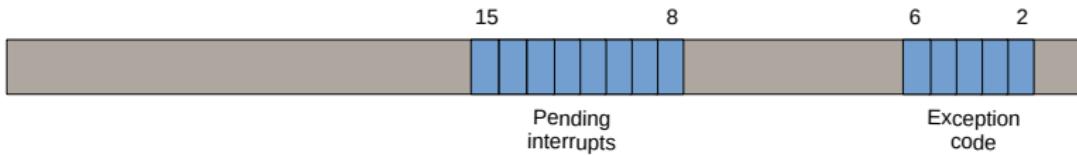
Status Register



Status Register

- The **interrupt mask** field contains a bit for each of the six hardware and two software interrupt levels
 - A mask bit that is 1 allows interrupts or exceptions at that level to interrupt the processor
 - A mask bit that is 0 disallows interrupts or exceptions at that level
 - When an interrupt or exception arrive, it sets its interrupt pending bit in the Cause register, even if the mask bit is disabled
- The **user mode** bit is 0 if the processor is running in kernel mode and 1 if it is running in user mode
- The **exception level** bit is normally 0, but is set to 1 after an exception or interrupt occur. When this bit is 1, interrupts or exceptions are disabled and the EPC is not updated if another interrupt or exception occur
- If the **interrupt enable** bit is 1, interrupts are allowed. If it is 0, they are disabled

Cause Register



Cause Register

- The **Pending interrupt** bit is set to 1 when an exception or interrupt is raised at that given hardware or software level
- The **Exception code** register describes the cause of an exception through the following codes:

Number	Name	Cause
0	Int	Interrupt (Hardware)
4	AdEL	Address error exception (load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error on instruction fetch
7	DBE	Bus error on data load or store
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
11	CpU	Coprocessor unimplemented
12	Ov	Arithmetic overflow exception
13	Tr	Trap
15	FPE	Floating point

Instructions Related to Interrupts and Exceptions

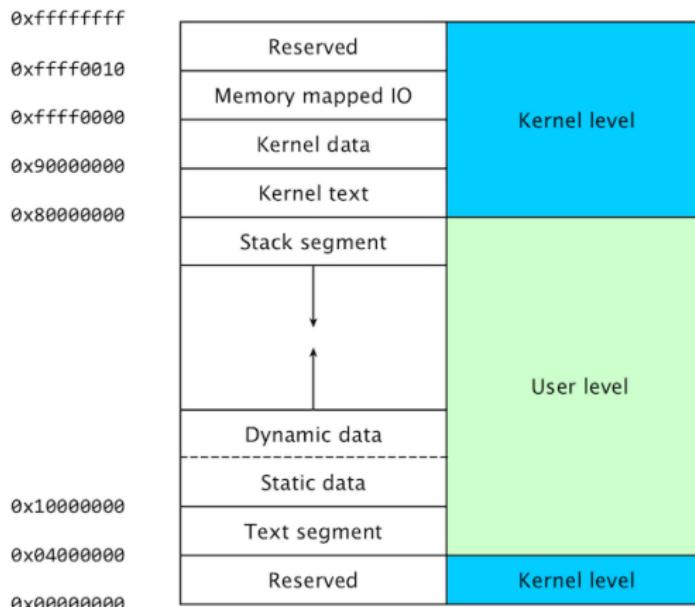
- **mtc0** reg, dst: Move the register reg to coprocessor 0 register dst. This instruction can only be used in privileged level mode
- **mfc0** dst, reg: Move the register reg of coprocessor 0 to register dst. This instruction can only be used in privileged level mode
- **syscall**: Launch the Sys exception. This instruction allows the user to ask service to the operating system
- **eret**: Set the EXL bit in coprocessor 0's Status register to 0 and return to the instruction pointed to by coprocessor 0's EPC register

Interrupt and Exception Sequence

- An interrupt is not ignored if the **IE** bit is set, the corresponding bit in the **Interrupt mask** is set and the **EXL** bit is unset
- An exception is not ignored if the corresponding bit in the **Interrupt mask** is set and the **EXL** bit is unset
- If an interrupt or exception has to be processed, the CPU does the following:
 - It sets up **EPC** to point to the restart location
 - The current instruction for an exception
 - The next instruction for an interrupt
 - It sets **EXL**, which forces the CPU into kernel (privileged) mode and disables interrupts and exceptions
 - **Cause** is set up so that software can see the reason for the interrupt or exception
 - The CPU then starts fetching instructions from the exception entry point at `0x80000180` and everything else is up to software

Memory Layout of a Program

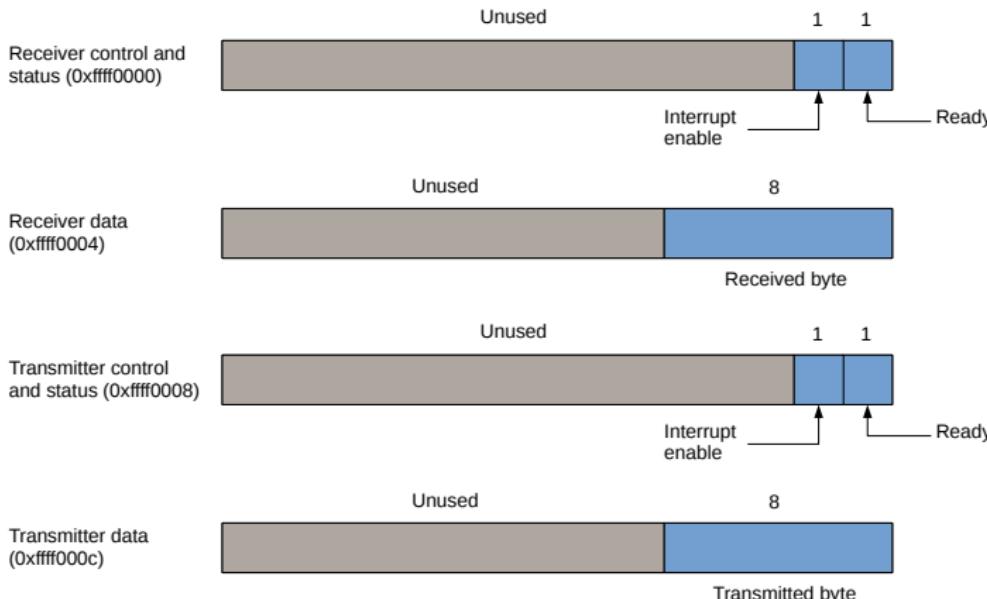
(From Uppsala university)



Memory Mapped Console

- We will describe one I/O device: A memory-mapped console on which a program can read and write characters
- The terminal device consists of two independent units: A receiver and a transmitter
 - The receiver reads characters typed on the keyboard
 - The transmitter display characters on the console
- The receiver and transmitter are independent. For example, the characters typed at the keyboard are not automatically echoed on the display
- A program controls the terminal with four memory-mapped device registers

Memory Mapped Registers



Receiver Control and Status

- Bit 0 is the ready bit
 - If it is set to 1, it means that a character has arrived from the keyboard but has not yet been read from the Receiver Data register
 - It changes from 0 to 1 when a character is typed at the keyboard, and it changes from 1 to 0 when the character is read from the Receiver Data register
- Bit 1 is the interrupt enable bit
 - If it is set to 1 by a program, the terminal requests an interrupt at hardware level 1 whenever a character is typed and the ready bit becomes 1.
 - Note that for the interrupt to affect the processor, interrupts must also be enabled in the Status register

Receiver Data

- The low-order 8 bits of this register contain the last character typed at the keyboard. All other bits contain 0s
- This register is read-only and changes only when a new character is typed at the keyboard
- Reading the Receiver Data register resets the ready bit in the Receiver Control register to 0
- The value in this register is undefined if the Receiver Control register is 0

Transmitter Control and Status

- Bit 0 is the ready bit
 - If this bit is 1, the transmitter is ready to accept a new character for output
 - If it is 0, the transmitter is still busy writing the previous character
- Bit 1 is the interrupt enable bit
 - If this bit is set to 1, then the terminal requests an interrupt at hardware level 0 whenever the transmitter is ready for a new character and the ready bit becomes 1

Transmitter Data

- When a value is written into this location, its low-order 8 bits (i.e., an ASCII character) are sent to the console
- When the Transmitter Data register is written, the ready bit in the Transmitter Control register is reset to 0. This bit stays 0 until enough time has elapsed to transmit the character to the terminal; then the ready bit becomes 1 again
- Reading the Receiver Data register resets the ready bit in the Receiver Control register to 0
- The Transmitter Data register should only be written when the ready bit of the Transmitter Control register is 1. If the transmitter is not ready, writes to the Transmitter Data register are ignored

Program Echoing Characters

The following program echoes characters from keyboard to screen.

```
1 #####  
2 # User text  
3 # MARS start to execute at label main in the user .text segment  
4 #####  
5 .globl main  
6 .text  
7 main:  
8     # Enable keyboard interrupts  
9     lw $s0, 0xffff0000  
10    ori $s1, $s0, 2  
11    sw $s1, 0xffff0000  
12 loop:  
13     # This infinite loop simulates the CPU doing something useful such as  
14     # executing another job while waiting for user keyboard input  
15     addi $s0, $s0, 1  
16     j loop
```

Program Echoing Characters

```
17 #####  
18 # Kernel data and code  
19 #####  
20 .kdata  
21 __save0: .word 0  
22 __save1: .word 0  
23 .ktext 0x80000180 # The handler address for MIPS32  
24 __handler:  
25     sw    $at, __save0  
26     mfc0 $k0, $13 # Get value in cause register  
27     andi $k1, $k0, 0x00007c # Mask all but the exception code to zero  
28     srl   $k1, $k1, 2 # Shift to get the exception code  
29 # Now $k0 = value of cause register  
30 #     $k1 = exception code  
31 # The exception code is zero for an interrupt  
32     beqz $k1, __interrupt  
33 __unhandled_exception:  
34     j    __resume_exception
```

Program Echoing Characters

```
35  __interrupt:
36      andi $k1, $k0, 0x00000100 # Reset interrupt pending bit
37      # Shift the interrupt pending bit as the least significant bit
38      srl $k1, $k1, 8
39      beq $k1, 1, __keyboard_interrupt # Branch on the interrupt pending
40      j __resume
41  __keyboard_interrupt:
42      # Get ASCII value of pressed key from the receiver data register
43      # Store content of the memory mapped receiver data register in $k1
44      lw $k1, 0xffff0004
45      sw $k0, __save1
46  __wait:
47      lw $k0, 0xffff0008 # Busy wait for console ready to accept data
48      andi $k0, 1
49      beqz $k0, __wait
50      sw $k1, 0xffff000c
51      lw $k0, __save1
52      j __resume
```

Program Echoing Characters

```
53  __resume_exception:
54      # When an exception or interrupt occurs, the value of the program
55      # counter ($pc) of the user level program is automatically stored
56      # in the exception program counter (EPC), the register $14 in
57      # Coprocessor 0
58      # Get value of EPC (Address of instruction causing the exception)
59      mfc0 $k0, $14
60      # Skip offending instruction by adding 4 to the value stored in EPC
61      # Otherwise the same instruction would be executed again causing the
62      # same exception again
63      addi $k0, $k0, 4
64      mtc0 $k0, $14 # Update EPC in coprocessor 0
65  __resume:
66      mtc0 $0, $13 # Clear cause register
67      lw $at, __save0
68      eret
```

Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

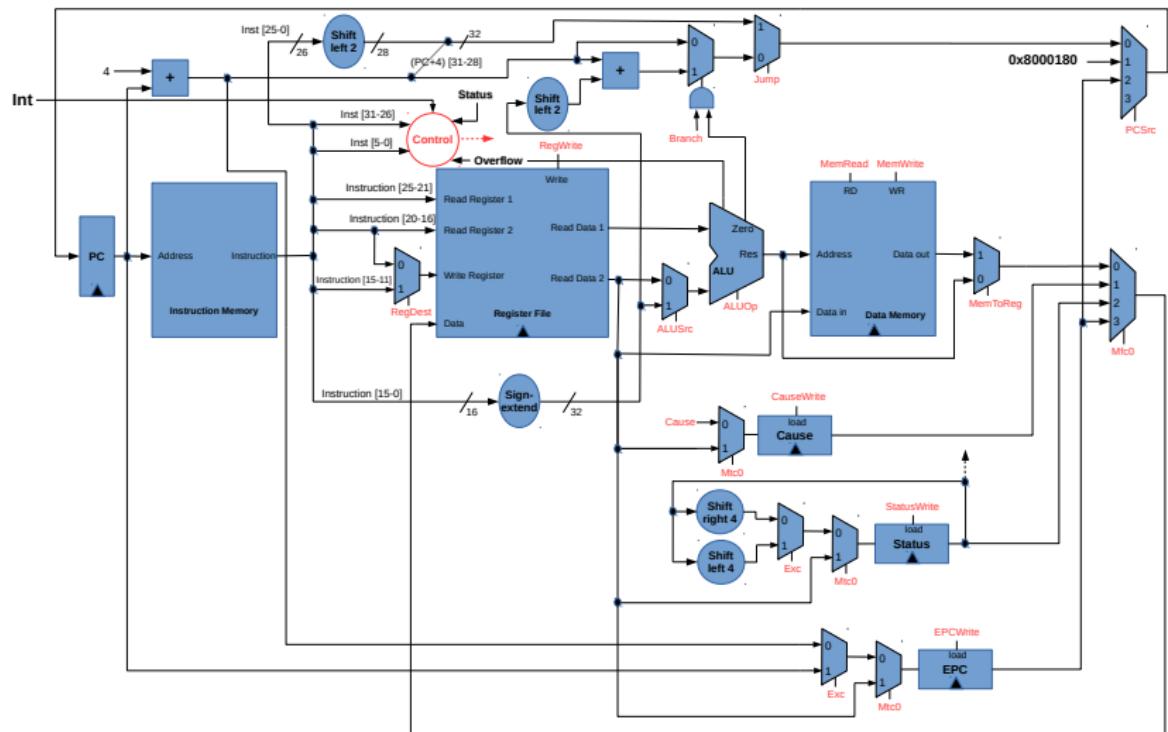
Interrupts and Exceptions
I/O Devices
Waiting for I/O Devices
Ints and Exceptions on MIPS

Single-Cycle with Ints and Exs
Ints and Exceptions on x86_64
Linux Signals

Simplified MIPS Interrupts and Exceptions

TO DO

Single-Cycle Datapath with Interrupts and Exceptions



Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Interrupts and Exceptions
I/O Devices
Waiting for I/O Devices
Ints and Exceptions on MIPS

Single-Cycle with Ints and Exs
Ints and Exceptions on x86_64
Linux Signals

x86_64 Interrupts and Exceptions

Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Interrupts and Exceptions
I/O Devices
Waiting for I/O Devices
Ints and Exceptions on MIPS

Single-Cycle with Ints and Exs
Ints and Exceptions on x86_64
Linux Signals

Linux Signals

Find the Bug

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4 #include <sys/time.h>
5
6 #define BUFFER_SIZE 1024
7 int buffer[BUFFER_SIZE];
8 volatile int head = 0;
9 volatile int size = 0;
10 volatile int counter = 0;
11
12 void producer(int signo)
13 {
14     if (size < BUFFER_SIZE)
15     {
16         buffer[(head + size) % BUFFER_SIZE] = counter++;
17         ++size;
18     }
19 }
```

Find the Bug

```
20 void consumer()
21 {
22     static int last_data = -1;
23     if (size == 0) return;
24     int data = buffer[head];
25     head = (head + 1) % BUFFER_SIZE;
26     --size;
27     if (last_data != data)
28     {
29         printf("%d\n", data);
30         last_data = data;
31     }
32 }
```

Find the Bug

```
33 | int check_consistency()
34 | {
35 |     for (int i = head; i < head + size - 1; ++i)
36 |     {
37 |         if (buffer[i % BUFFER_SIZE] + 1 !=
38 |             buffer[(i + 1) % BUFFER_SIZE])
39 |             return 0;
40 |     }
41 |     return 1;
42 | }
```

Find the Bug

```
43 int main() {
44     struct itimerval delay;
45     signal(SIGALRM, producer);
46     delay.it_value.tv_sec = 0;
47     delay.it_value.tv_usec = 10;
48     delay.it_interval.tv_sec = 0;
49     delay.it_interval.tv_usec = 10;
50     if (setitimer(ITIMER_REAL, &delay, NULL))
51     {
52         perror("setitimer");
53         return EXIT_FAILURE;
54     }
55     while (check_consistency())
56     {
57         consumer();
58     }
59     printf("Something has gone wrong\n");
60     return EXIT_SUCCESS;
61 }
```

Find the Bug

```
1 // gcc -O3 signal.c -o signal && ./signal
2
3 // ...
4 // 58051
5 // 58052
6 // 58053
7 // 58054
8 // 58055
9 // 58056
10 // 58057
11 // 58058
12 // 58059
13 // 58060
14 // 58061
15 // 58062
16 // 58063
17 // 58064
18 // 58065
19 // Something has gone wrong
```

Solution

```
1 void consumer()
2 {
3     static int last_data = -1;
4     if (size == 0) return;
5     int data = buffer[head];
6     mask_signal();
7     head = (head + 1) % BUFFER_SIZE;
8     --size;
9     unmask_signal();
10    if (last_data != data)
11    {
12        printf("%d\n", data);
13        last_data = data;
14    }
15 }
```

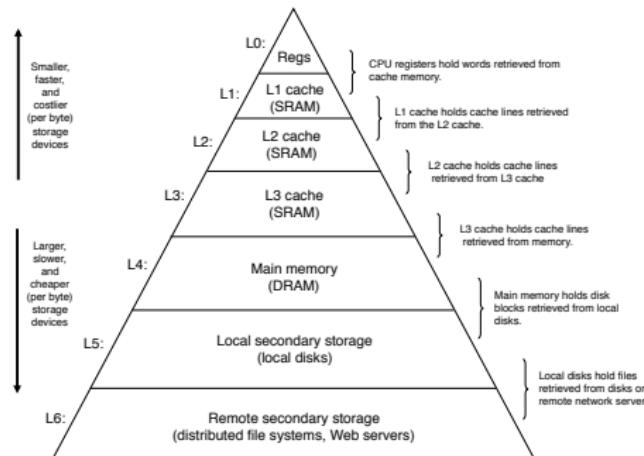
Solution

```
1 void mask_signal()
2 {
3     sigset_t set;
4     sigset_t old;
5     sigemptyset(&set);
6     sigaddset(&set, SIGALRM);
7     sigprocmask(SIG_SETMASK, &set, &old);
8 }
9
10 void unmask_signal()
11 {
12     sigset_t set;
13     sigemptyset(&set);
14     sigprocmask(SIG_SETMASK, &set, NULL);
15 }
```

Out of Order

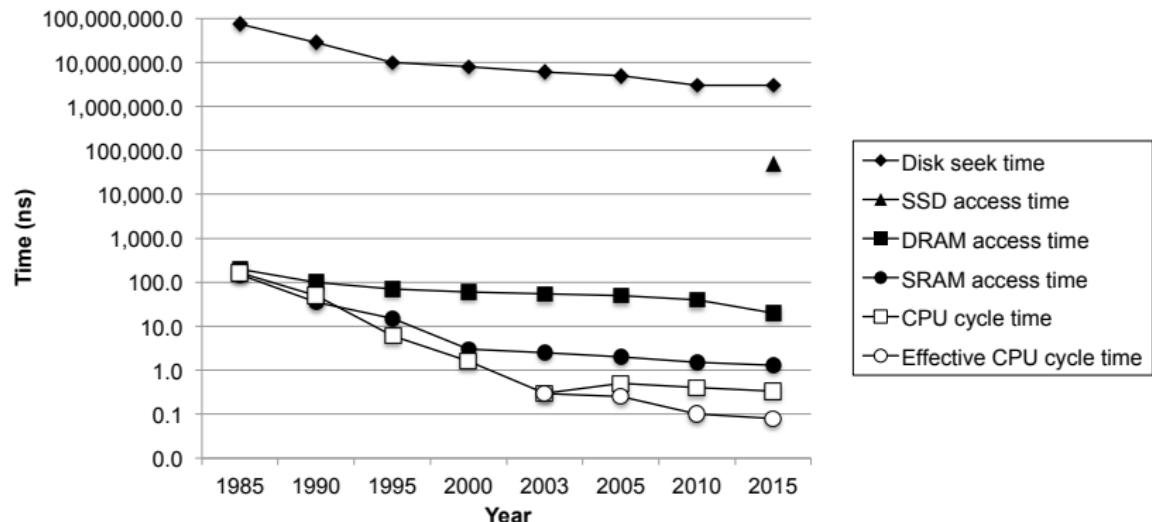
Memory Hierarchy

- Problem: Large memory access time is way bigger than the clock cycle time of the processor
- Problem: Fast memory technology is more expensive per bit than slower memory
- What we want: A big fast and cheap memory!
- Because of locality of memory references, we can build a memory hierarchy to obtain the illusion of a fast, big and cheap memory



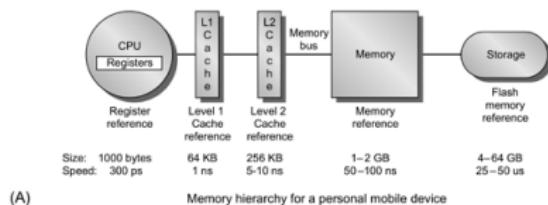
Memory Latency

(From *Computer Systems: A Programmer's Perspective*)



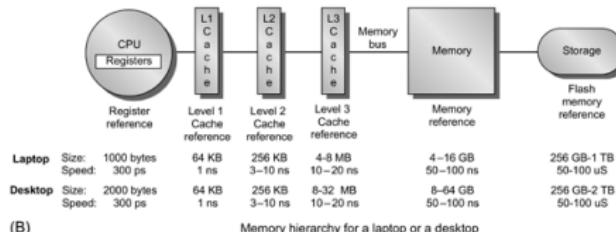
Hierarchy of Memories

(From *Computer Architecture: A Quantitative Approach*)



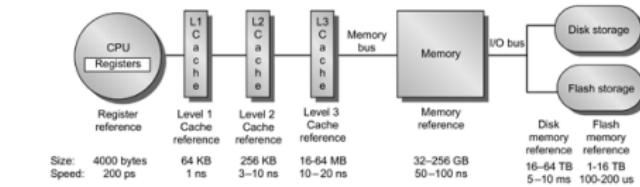
(A)

Memory hierarchy for a personal mobile device



(B)

Memory hierarchy for a laptop or a desktop

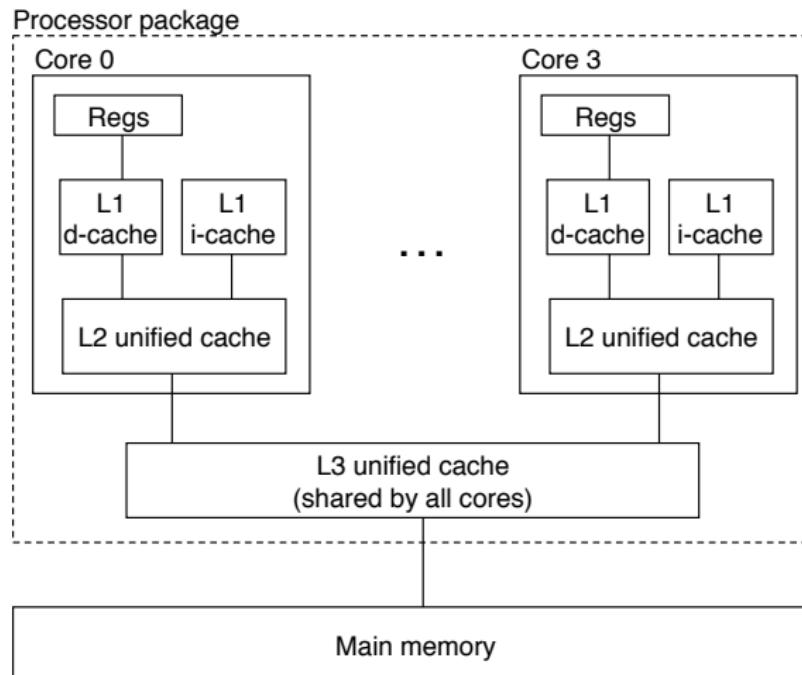


(C)

Memory hierarchy for server

Hierarchy of Memories

(From *Computer Systems: A Programmer's Perspective*)



Locality

- **Temporal Locality:** If a location is referenced, it is likely to be referenced again in the near future
- **Spatial Locality:** If a location is referenced, it is likely that locations near it will be referenced in the future

Code With Good Locality

```
1  const int N = 1200;
2
3  double A[N][N];
4  double B[N][N];
5  double C[N][N];
6
7  void mm_kij()
8  {
9      double r;
10     for (int k = 0; k < N; k++)
11     {
12         for (int i = 0; i < N; i++)
13         {
14             r = a[i][k];
15             for (int j = 0; j < N; j++)
16                 c[i][j] += r * b[k][j];
17         }
18     }
19 }
```

Code With Bad Locality

```
1 void mm_jki()
2 {
3     double r;
4     for (int j = 0; j < N; j++)
5     {
6         for (int k = 0; k < N; k++)
7         {
8             r = B[k][j];
9             for (int i = 0; i < N; i++)
10                C[i][j] += A[i][k] * r;
11         }
12     }
13 }
```

Cache Principle

- Cache memories are small and fast (*SRAM*-based) memories managed automatically in hardware
- Cache memories hold frequently accessed blocks of memory
- When a word is not found in the cache, a miss occurs:
 - Fetch word from lower level in hierarchy, requiring a higher latency reference and save it into the cache
 - Lower level may be another cache or the main memory
 - Also fetch the other words around this word

Caches Exploit Locality

- Caches exploit **temporal Locality** by remembering the contents of recently accessed locations
- Caches exploit **spatial Locality** by fetching blocks of data around recently accessed locations

Cache Big Picture

We will use the following program (translated in mips assembly on next slides) to illustrate the cache behaviour.

```
1 int m1[2][3];
2 int m2[3][2];
3 int res[2][2];
4
5 void matmult() {
6     for (int i = 0; i < 2; i++)
7     {
8         for (int j = 0; j < 2; j++)
9         {
10            int sum = 0;
11            for (int k = 0; k < 3; k++)
12                sum += m1[i][k] * m2[k][j];
13            res[i][j] = sum;
14        }
15    }
16 }
```

Cache Big Picture

```
1      .data
2
3 m1:    .word 1, 2, 3, 4, 5, 6      # 2 x 3
4 m2:    .word 7, 8, 9, 10, 11, 12 # 3 x 2
5 res:   .word 0, 0, 0, 0          # 2 x 2
6
7      .text
8
9 main:
10     addi $s0, $0, 2            # number of lines m1, number of columns of m2
11     addi $s1, $0, 3            # number of columns of m1, number of lines of m2
12
13     addi $t0, $0, 0          # i = 0
14 loop1:
15     beq $t0, $s0, end_loop1
16     addi $t1, $0, 0          # j = 0
17 loop2:
18     beq $t1, $s1, end_loop2
19     addi $t2, $0, 0          # k = 0
20     addi $t3, $0, 0          # sum = 0
21 loop3:
22     beq $t2, $s1, end_loop3
23
24     mul $t4, $t0, 3          # $t4 = i * 3
25     add $t4, $t4, $t2        # $t4 = i * 3 + k
26     sll $t4, $t4, 2          # $t4 = (i * 3 + k) * 4
27     lw $t4, m1($t4)         # $t4 = m1[i][k]
```

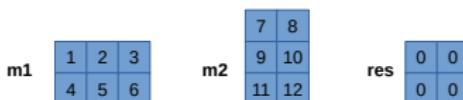
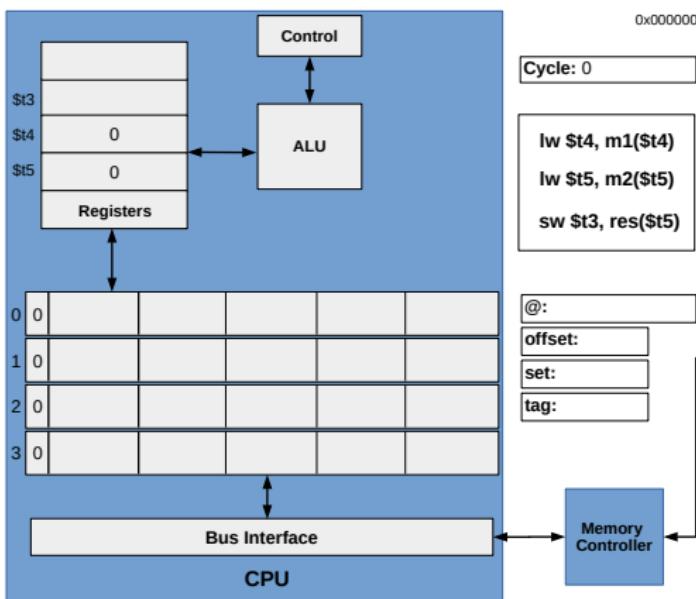
Cache Big Picture

```
28      sll  $t5, $t2, 1          # $t5 = k * 2
29      add  $t5, $t5, $t1        # $t5 = k * 2 + j
30      sll  $t5, $t5, 2          # $t5 = (k * 2 + j) * 4
31      lw   $t5, m2($t5)        # $t5 = m2[k][j]
32
33      mul  $t5, $t4, $t5        # $t5 = m1[i][k] * m2[k][j]
34      add  $t3, $t3, $t5        # sum += m1[i][k] * m2[k][j]
35
36      addi $t2, $t2, 1          # k++
37      j    loop3
38 end_loop3:
39      sll  $t5, $t0, 1          # $t5 = i * 2
40      add  $t5, $t5, $t1        # $t5 = i * 2 + j
41      sll  $t5, $t5, 2          # $t5 = (i * 2 + j) * 4
42      sw   $t3, res($t5)        # res[i][j] = sum
43
44      addi $t1, $t1, 1          # j++
45      j    loop2
46 end_loop2:
47      addi $t0, $t0, 1          # i++
48      j    loop1
49 end_loop1:
```

Cache Big Picture

```
50      add  $t0, $0, $0
51      addi $t1, $t0, 16          # number of bytes in res
52 print_matrix:
53      lw   $a0, res($t0)
54      addi $v0, $0, 1
55      syscall                  # print_int(*(res + $t0))
56      addi $a0, $0, 10
57      addi $v0, $0, 11
58      syscall                  # print newline
59      addi $t0, $t0, 4
60      bne  $t0, $t1, print_matrix
61      addi $v0, $0, 10
62      syscall                  # exit
63
64 # spim -f matmult.s
65 # 58
66 # 64
67 # 139
68 # 154
```

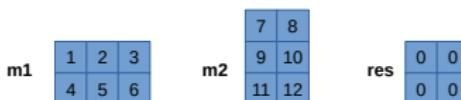
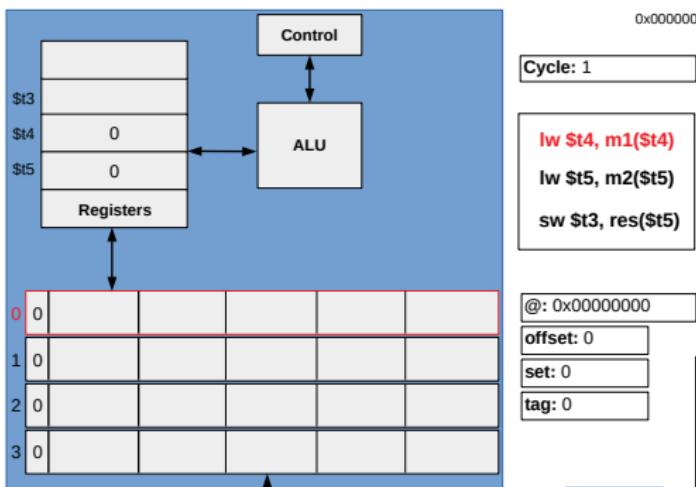
Cache Big Picture



0x00000000		
1	8	0
0	0	0
0	0	0
2	9	0
0	0	0
0	0	0
3	10	0
0	0	0
0	0	0
0	0	0
4	11	0
0	0	0
0	0	0
5	12	0
0	0	0
0	0	0
6	0	0
0	0	0
7	0	0
0	0	0
0	0	0
0	0	0

Memory

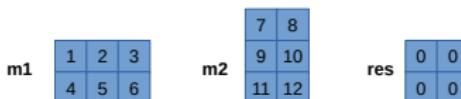
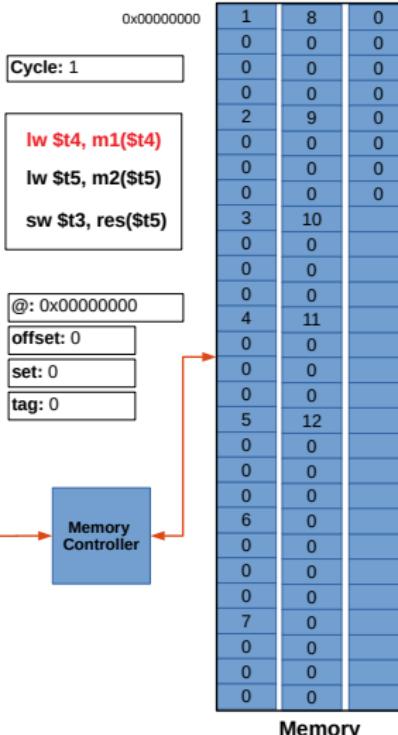
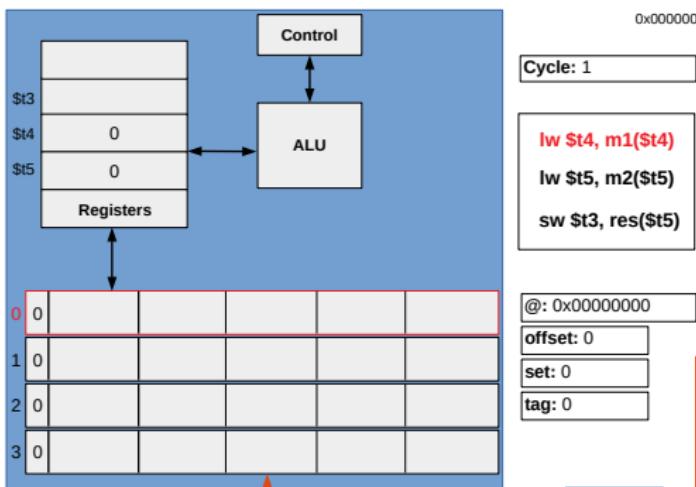
Cache Big Picture



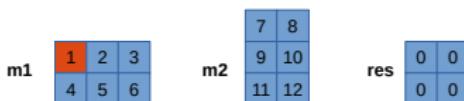
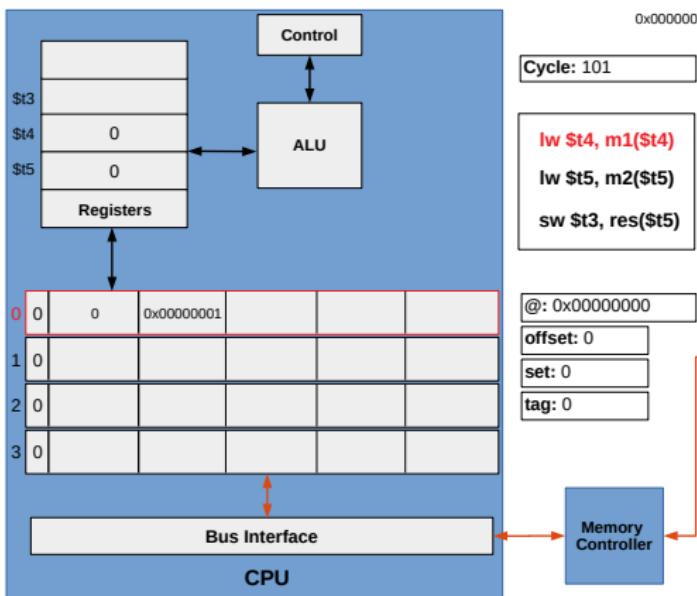
0x00000000	1	8	0
	0	0	0
	0	0	0
	0	0	0
	2	9	0
	0	0	0
	0	0	0
	0	0	0
	3	10	0
	0	0	0
	0	0	0
	0	0	0
	4	11	0
	0	0	0
	0	0	0
	5	12	0
	0	0	0
	0	0	0
	6	0	0
	0	0	0
	0	0	0
	7	0	0
	0	0	0
	0	0	0
	0	0	0

Memory

Cache Big Picture



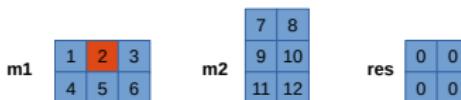
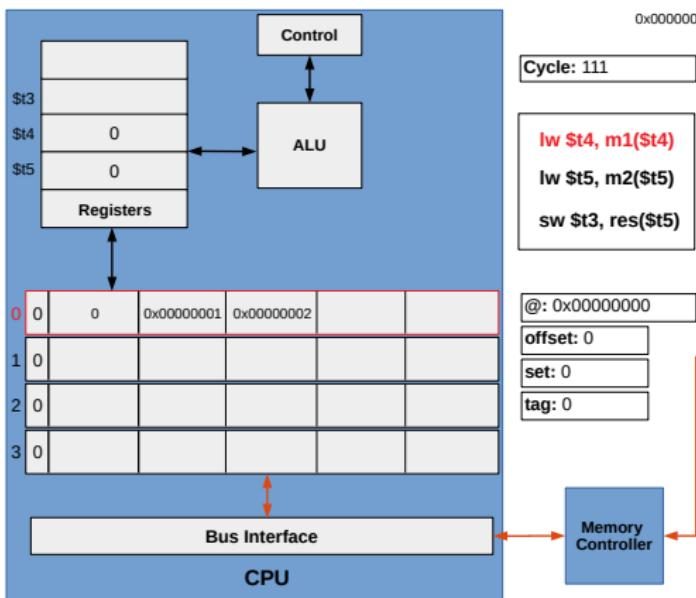
Cache Big Picture



0x00000000	1	8	0
	0	0	0
	0	0	0
	0	0	0
	2	9	0
	0	0	0
	0	0	0
	0	0	0
	3	10	0
	0	0	0
	0	0	0
	0	0	0
	4	11	0
	0	0	0
	0	0	0
	5	12	0
	0	0	0
	0	0	0
	6	0	0
	0	0	0
	0	0	0
	7	0	0
	0	0	0
	0	0	0
	0	0	0

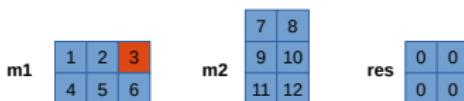
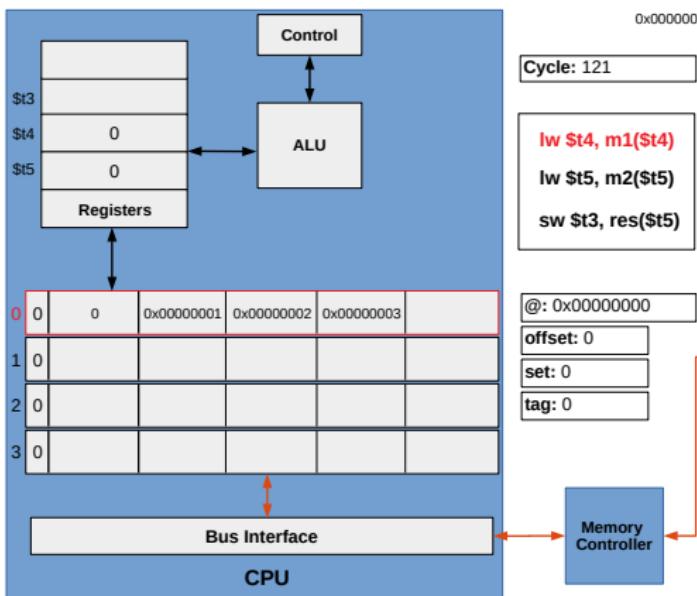
Memory

Cache Big Picture



1	8	0
0	0	0
0	0	0
0	0	0
2	9	0
0	0	0
0	0	0
0	0	0
3	10	
0	0	
0	0	
0	0	
4	11	
0	0	
0	0	
0	0	
5	12	
0	0	
0	0	
0	0	
6	0	
0	0	
0	0	
0	0	
7	0	
0	0	
0	0	
0	0	

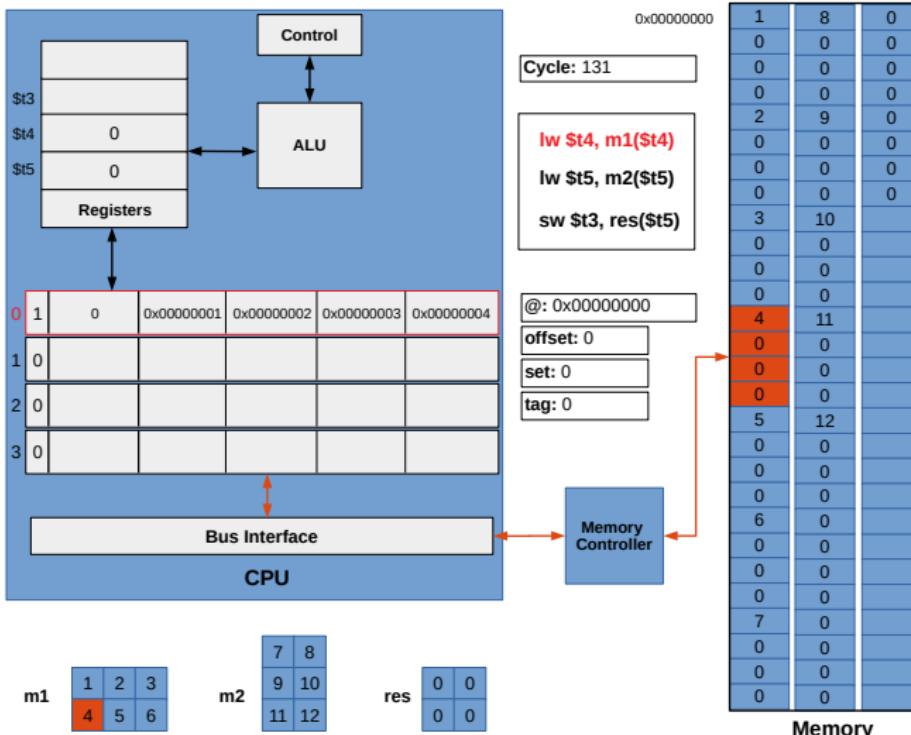
Cache Big Picture



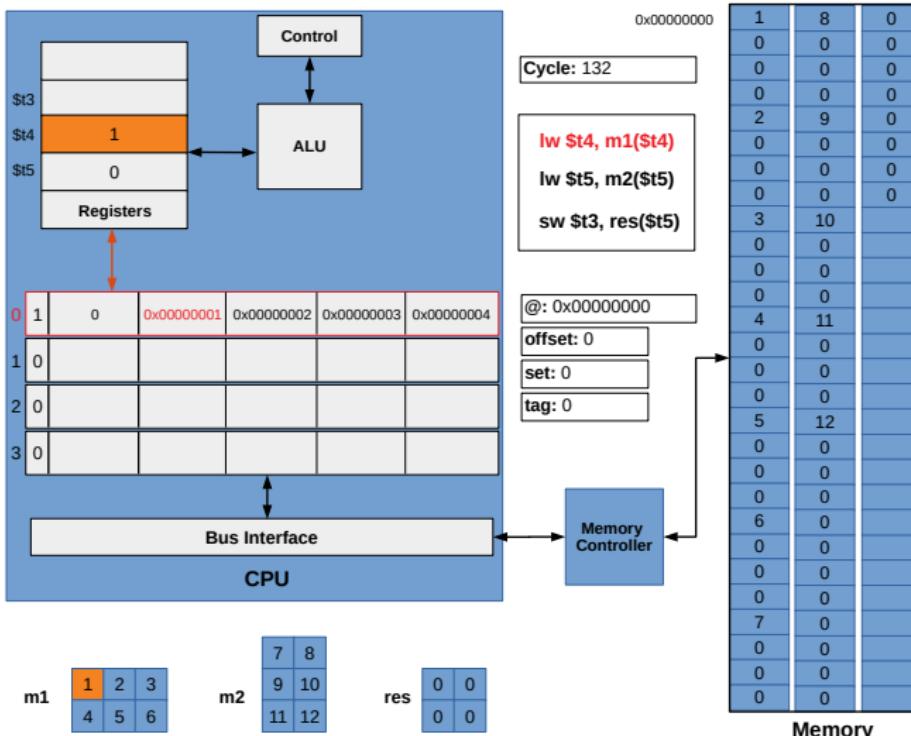
0x00000000	1	8	0
0	0	0	0
0	0	0	0
0	0	0	0
2	9	0	0
0	0	0	0
0	0	0	0
0	0	0	0
3	10	0	0
0	0	0	0
0	0	0	0
0	0	0	0
4	11	0	0
0	0	0	0
0	0	0	0
0	0	0	0
5	12	0	0
0	0	0	0
0	0	0	0
0	0	0	0
6	0	0	0
0	0	0	0
0	0	0	0
7	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Memory

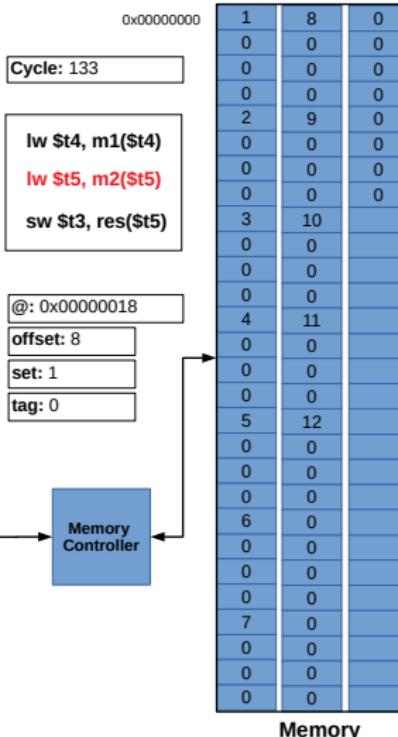
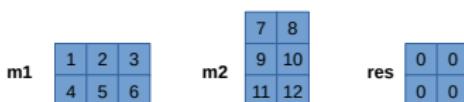
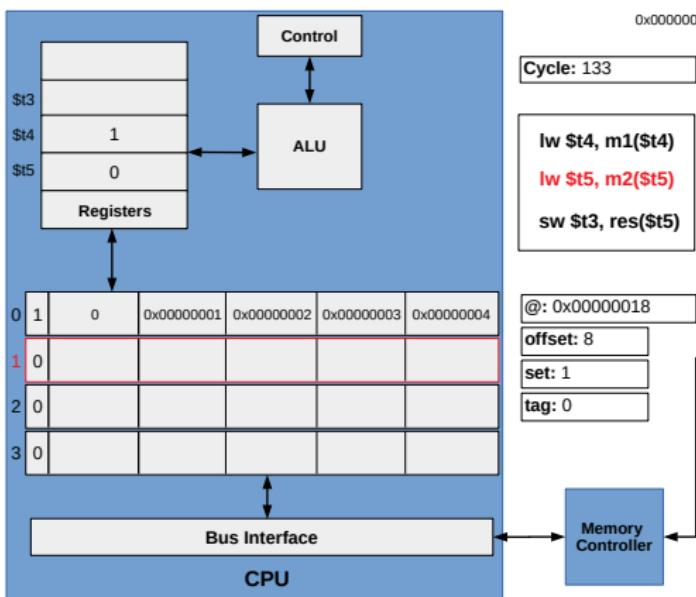
Cache Big Picture



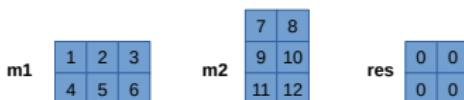
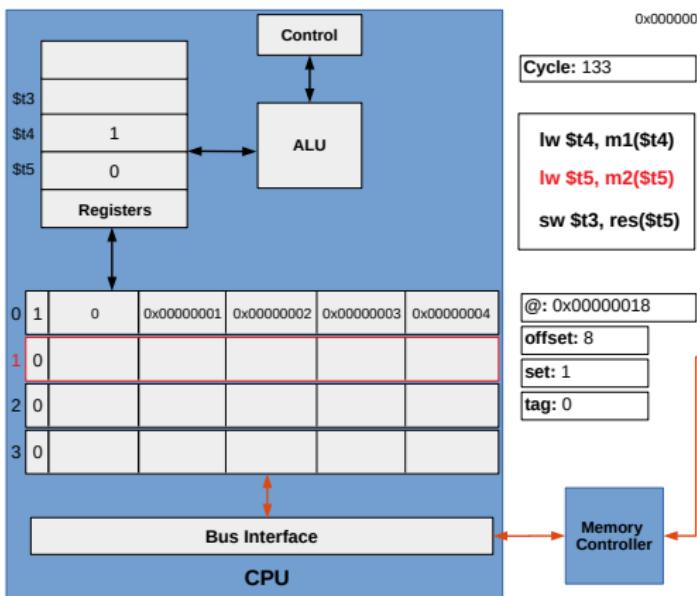
Cache Big Picture



Cache Big Picture



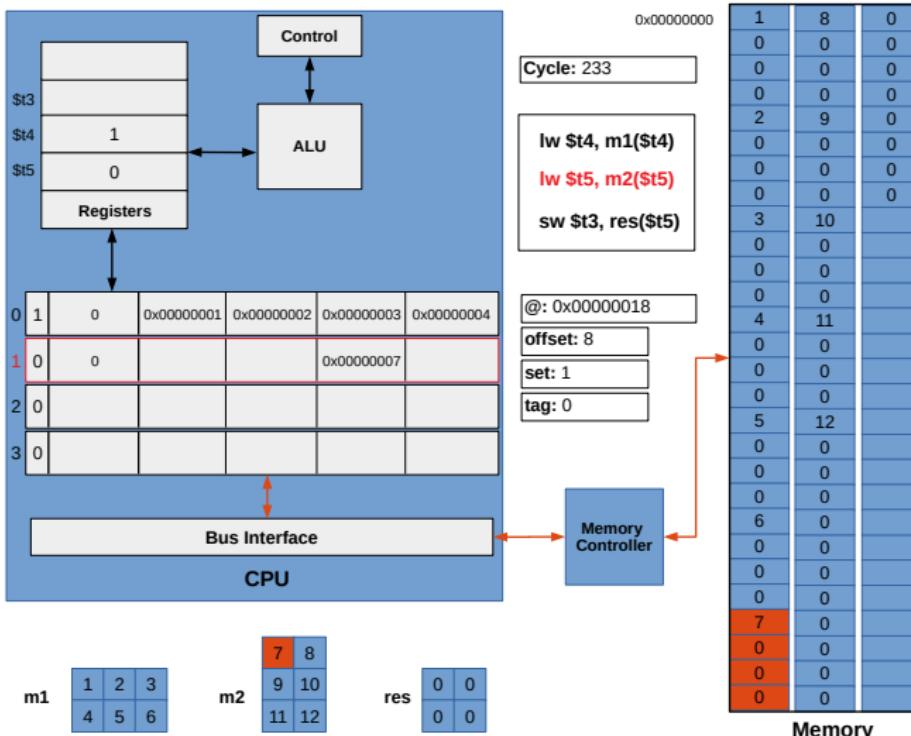
Cache Big Picture



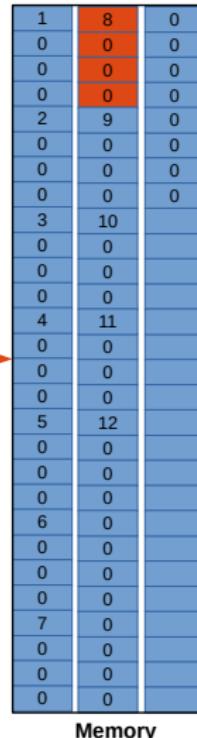
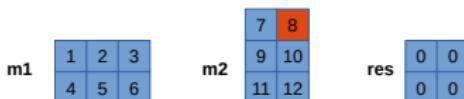
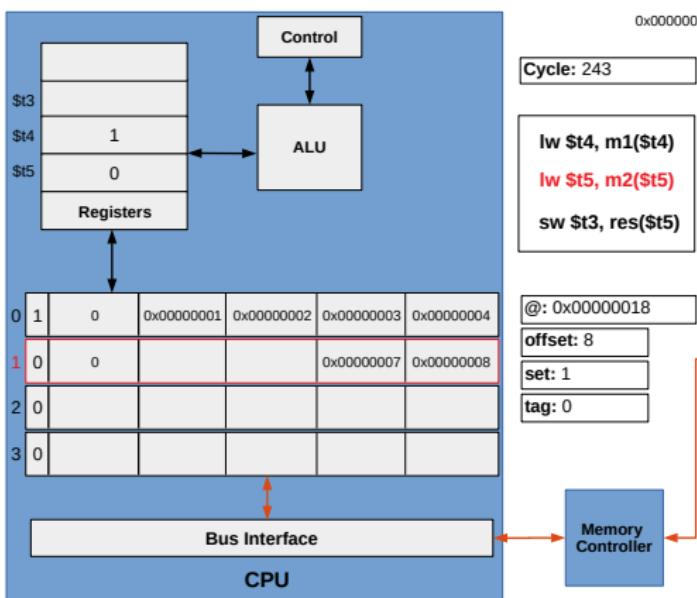
0x00000000	1	8	0
0	0	0	0
0	0	0	0
0	0	0	0
2	9	0	0
0	0	0	0
0	0	0	0
0	0	0	0
3	10	0	0
0	0	0	0
0	0	0	0
0	0	0	0
4	11	0	0
0	0	0	0
0	0	0	0
5	12	0	0
0	0	0	0
0	0	0	0
6	0	0	0
0	0	0	0
0	0	0	0
7	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Memory

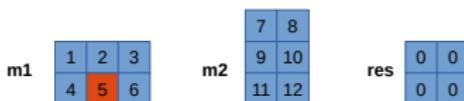
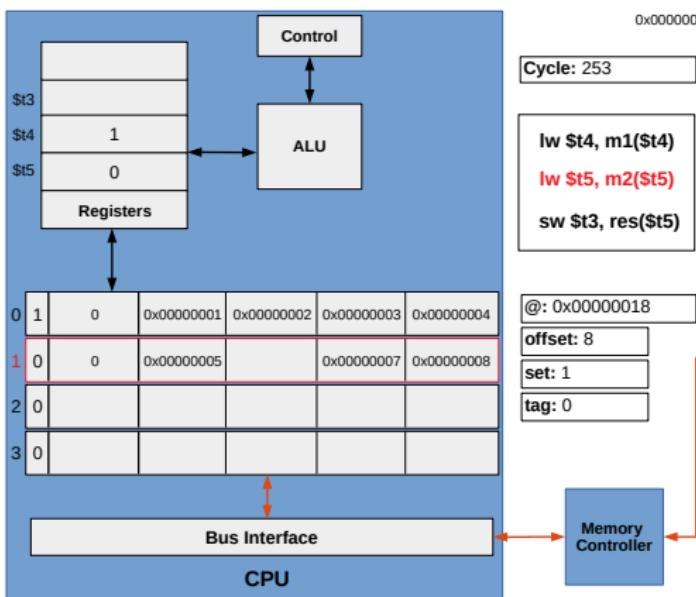
Cache Big Picture



Cache Big Picture



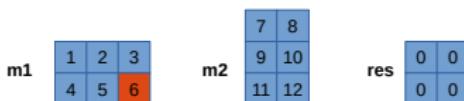
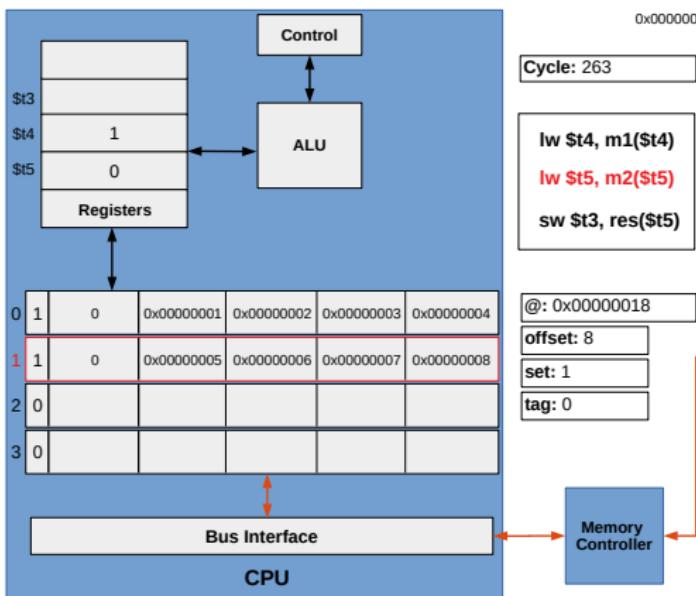
Cache Big Picture



0x00000000	1	8	0
0	0	0	0
0	0	0	0
0	0	0	0
2	9	0	0
0	0	0	0
0	0	0	0
0	0	0	0
3	10	0	0
0	0	0	0
0	0	0	0
0	0	0	0
4	11	0	0
0	0	0	0
0	0	0	0
5	12	0	0
0	0	0	0
0	0	0	0
6	0	0	0
0	0	0	0
0	0	0	0
7	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Memory

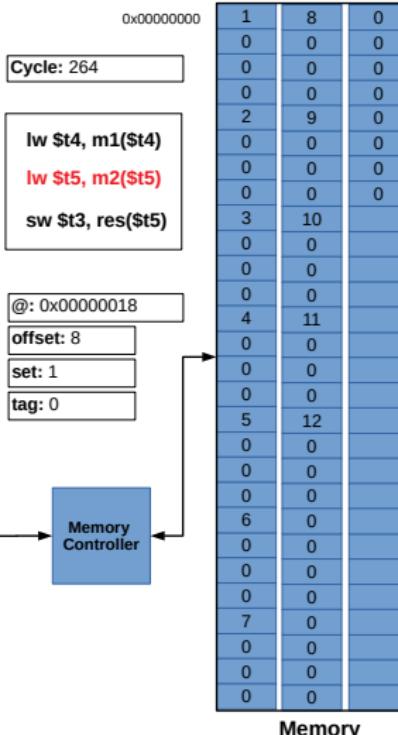
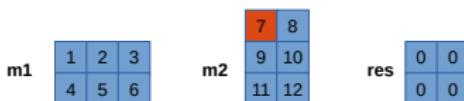
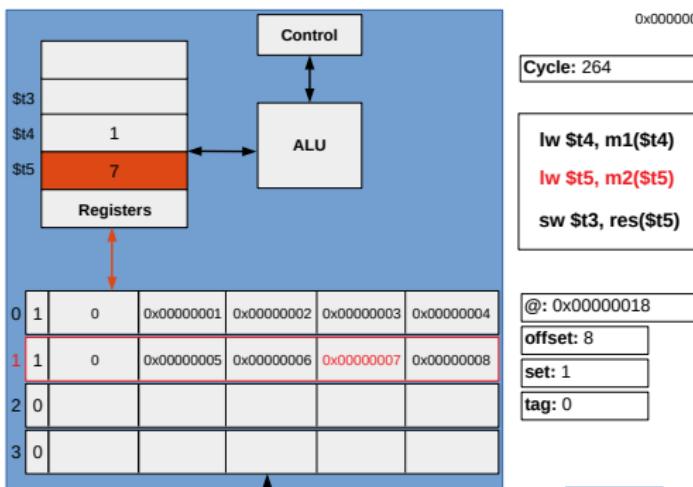
Cache Big Picture



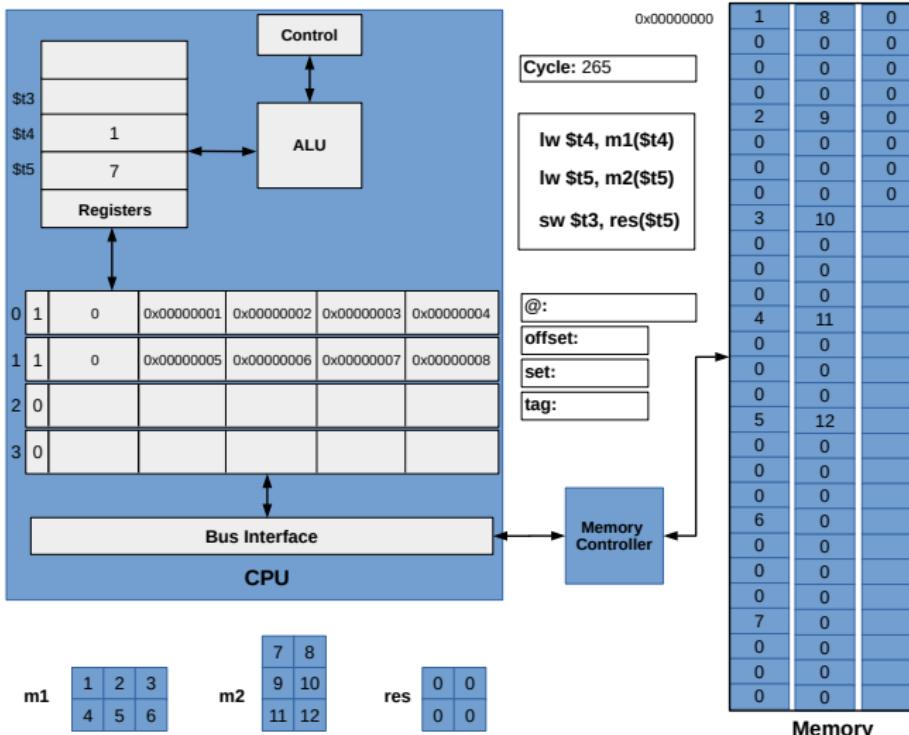
0x00000000	1	8	0
0	0	0	0
0	0	0	0
0	0	0	0
2	9	0	0
0	0	0	0
0	0	0	0
0	0	0	0
3	10	0	0
0	0	0	0
0	0	0	0
0	0	0	0
4	11	0	0
0	0	0	0
0	0	0	0
5	12	0	0
0	0	0	0
0	0	0	0
6	0	0	0
0	0	0	0
0	0	0	0
7	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Memory

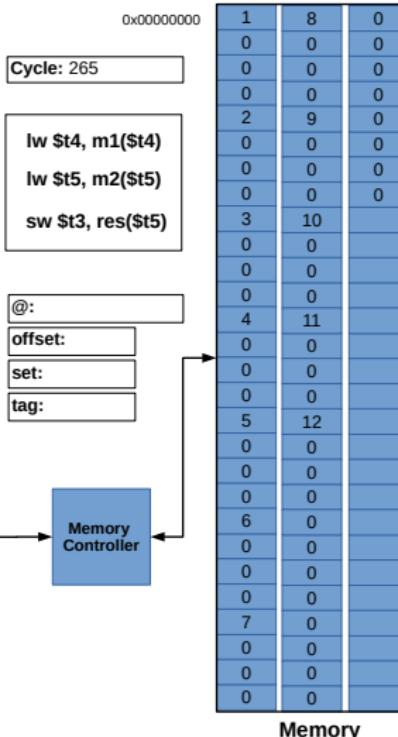
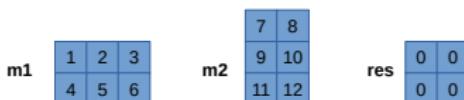
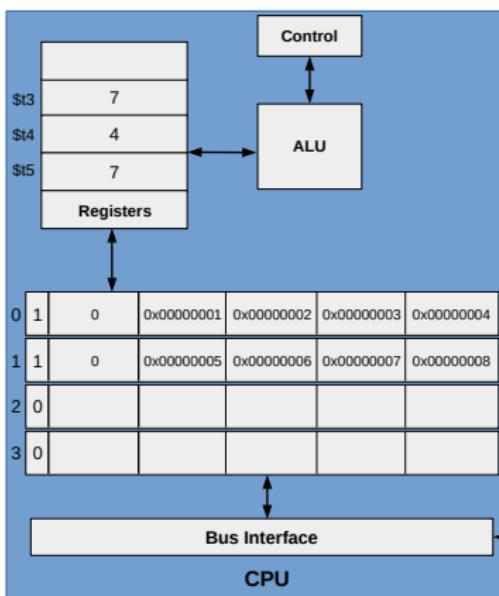
Cache Big Picture



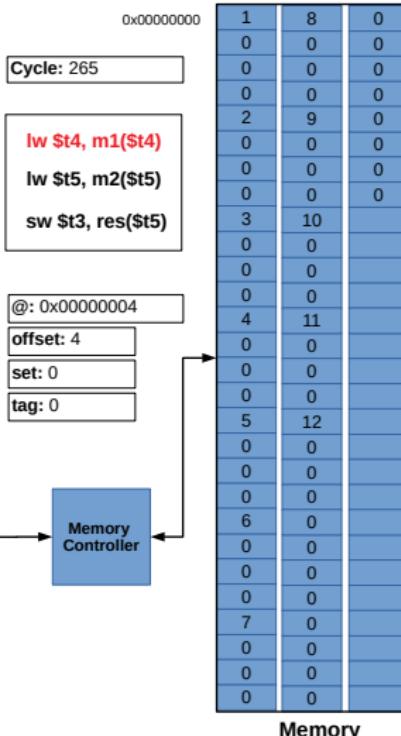
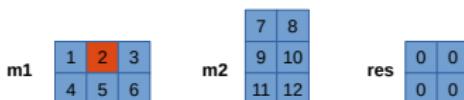
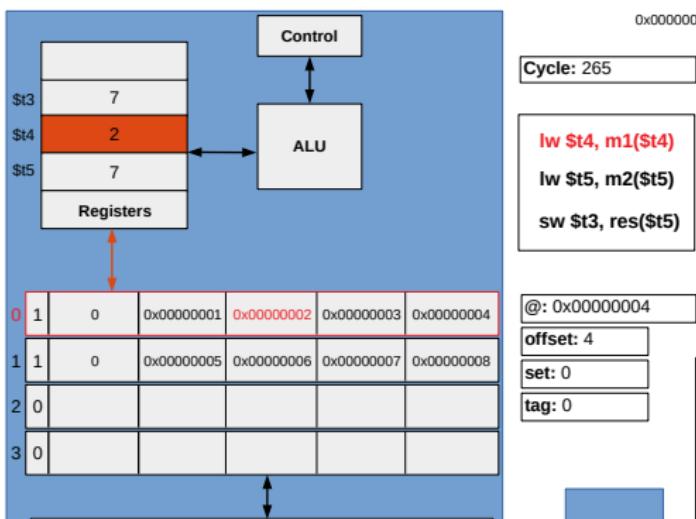
Cache Big Picture



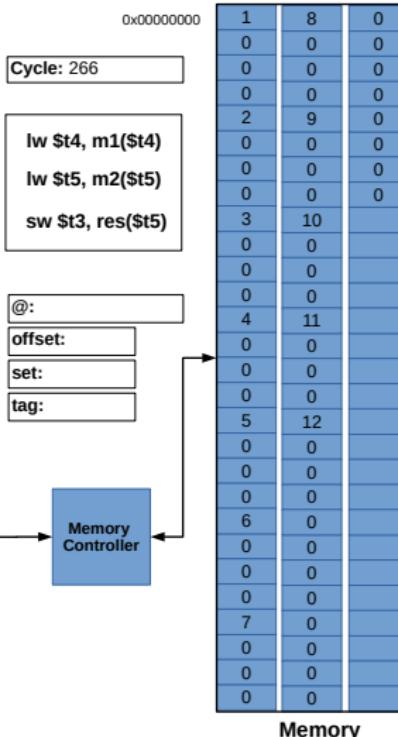
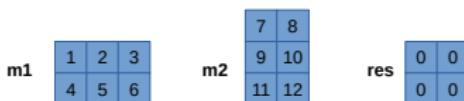
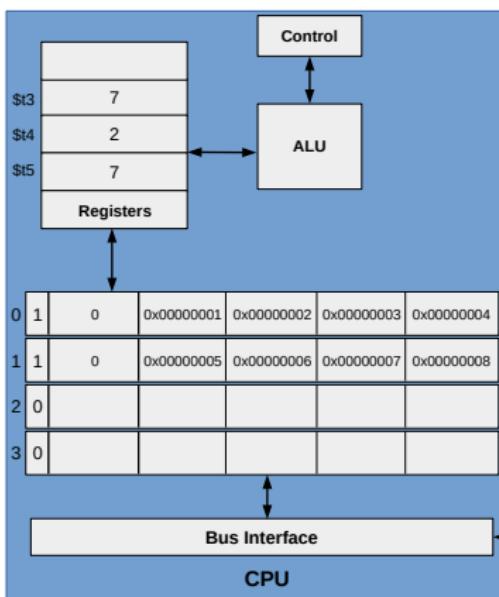
Cache Big Picture



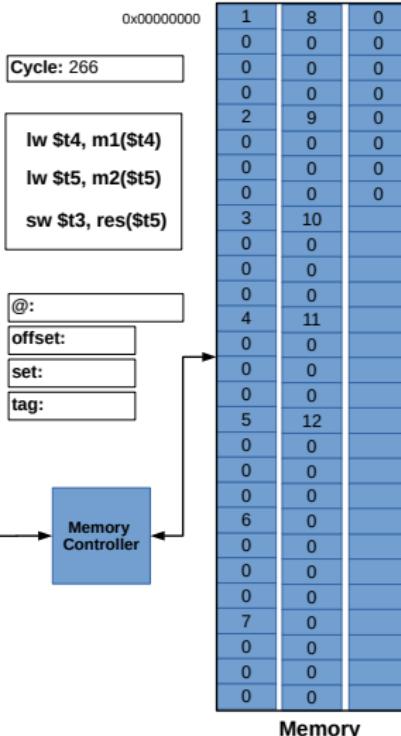
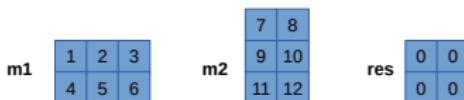
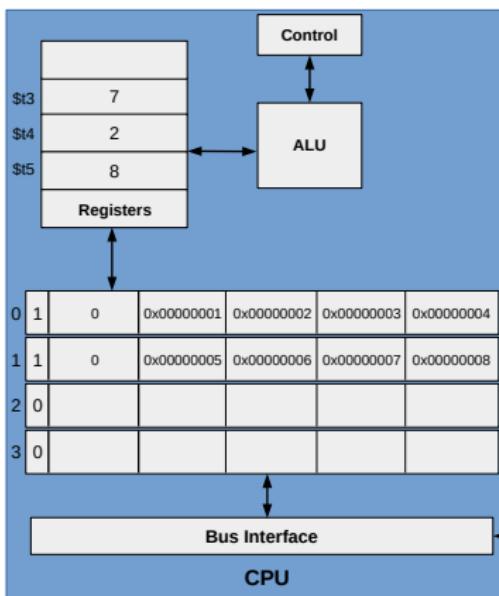
Cache Big Picture



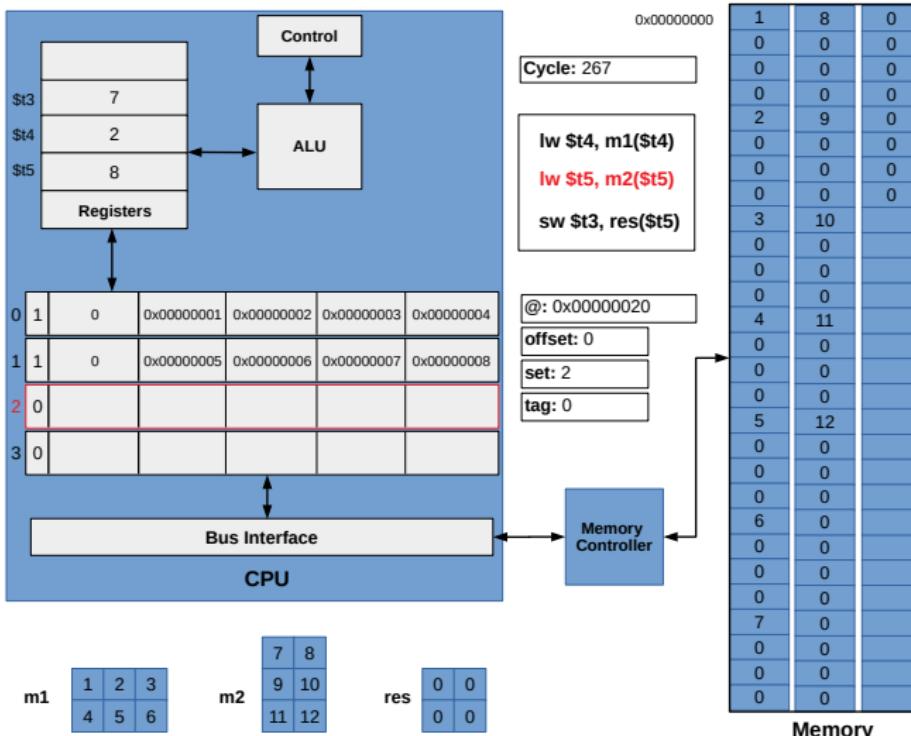
Cache Big Picture



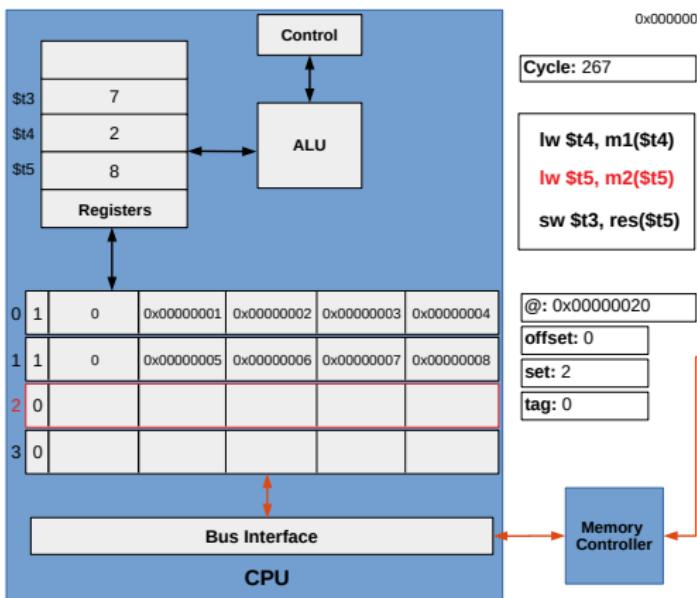
Cache Big Picture



Cache Big Picture



Cache Big Picture



$m1$

1	2	3
4	5	6

 $m2$

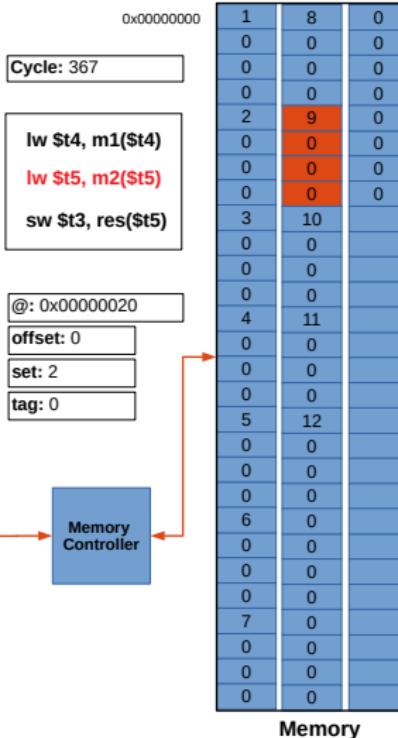
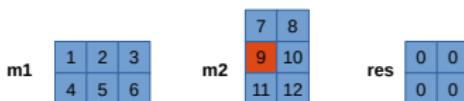
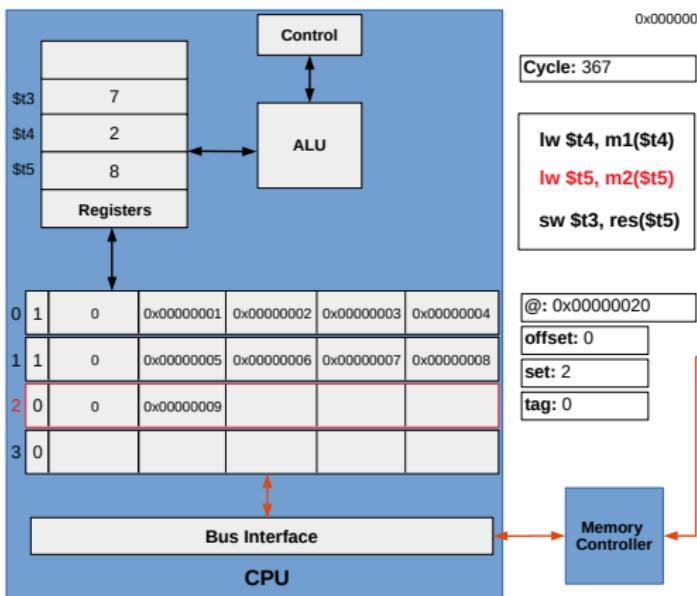
7	8
9	10
11	12

 res

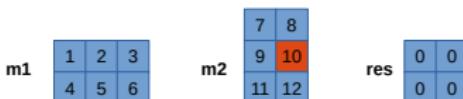
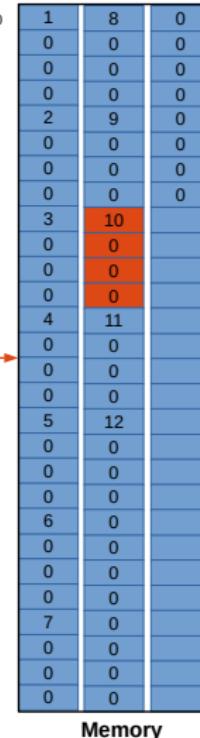
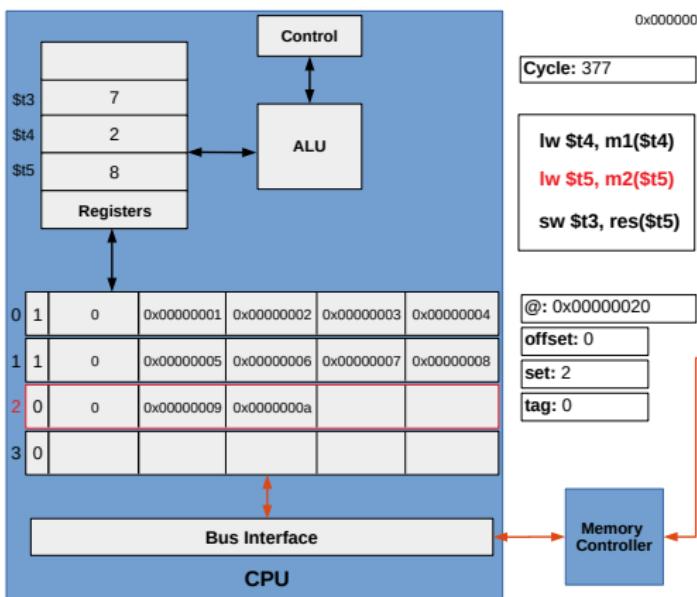
0	0
0	0

1	8	0
0	0	0
0	0	0
0	0	0
2	9	0
0	0	0
0	0	0
0	0	0
3	10	0
0	0	0
0	0	0
0	0	0
4	11	0
0	0	0
0	0	0
0	0	0
5	12	0
0	0	0
0	0	0
0	0	0
6	0	0
0	0	0
0	0	0
0	0	0
7	0	0
0	0	0
0	0	0
0	0	0

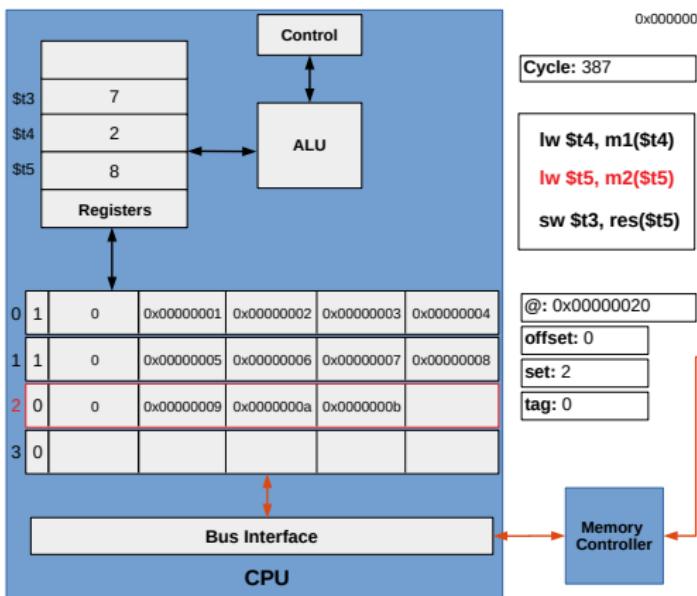
Cache Big Picture



Cache Big Picture



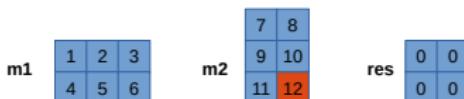
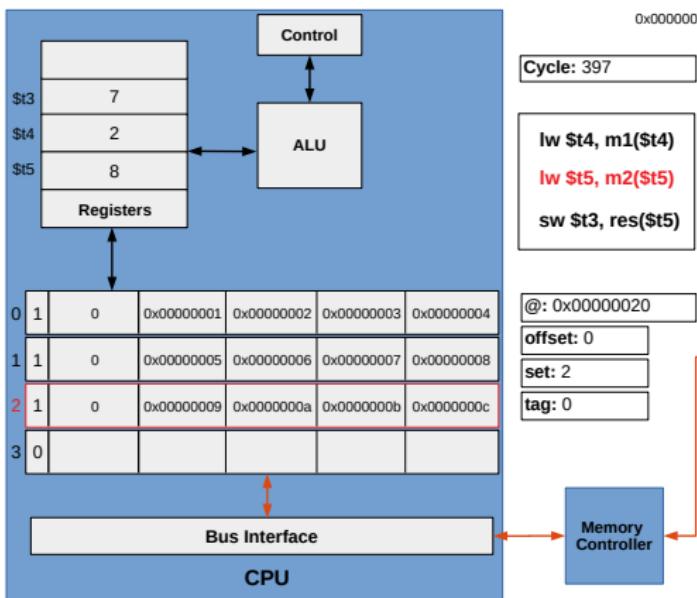
Cache Big Picture



1	8	0
0	0	0
0	0	0
0	0	0
2	9	0
0	0	0
0	0	0
0	0	0
3	10	0
0	0	0
0	0	0
0	0	0
4	11	0
0	0	0
0	0	0
0	0	0
5	12	0
0	0	0
0	0	0
0	0	0
6	0	0
0	0	0
0	0	0
7	0	0
0	0	0
0	0	0
0	0	0

Memory

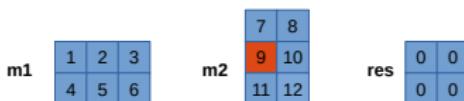
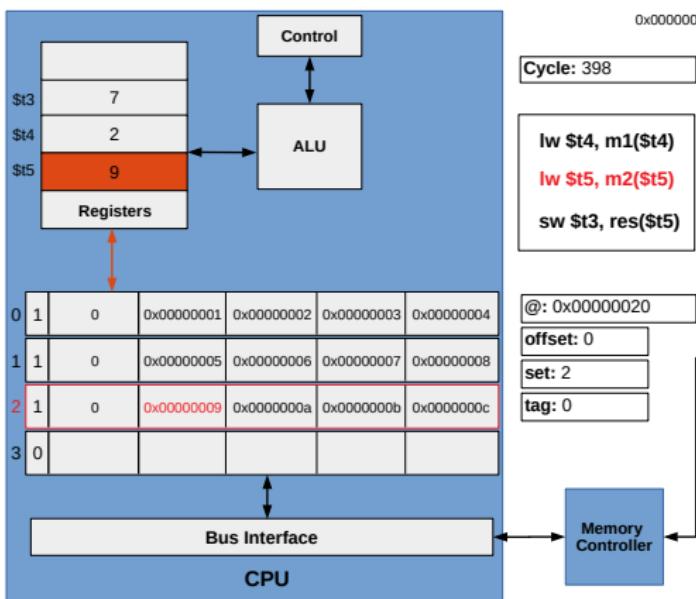
Cache Big Picture



0	1	8	0
0	0	0	0
0	0	0	0
2	9	0	0
0	0	0	0
0	0	0	0
3	10	0	0
0	0	0	0
0	0	0	0
4	11	0	0
0	0	0	0
0	0	0	0
5	12	0	0
0	0	0	0
0	0	0	0
6	0	0	0
0	0	0	0
0	0	0	0
7	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Memory

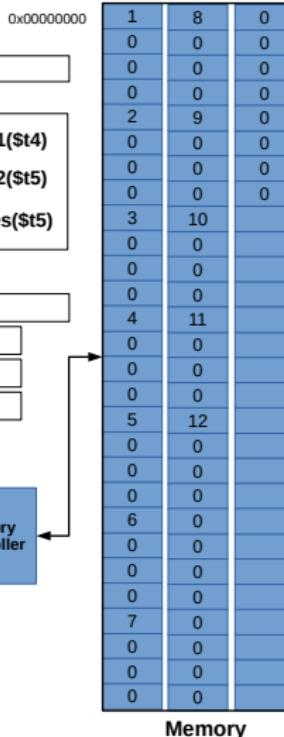
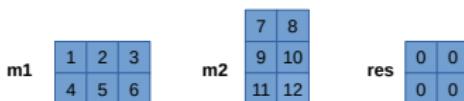
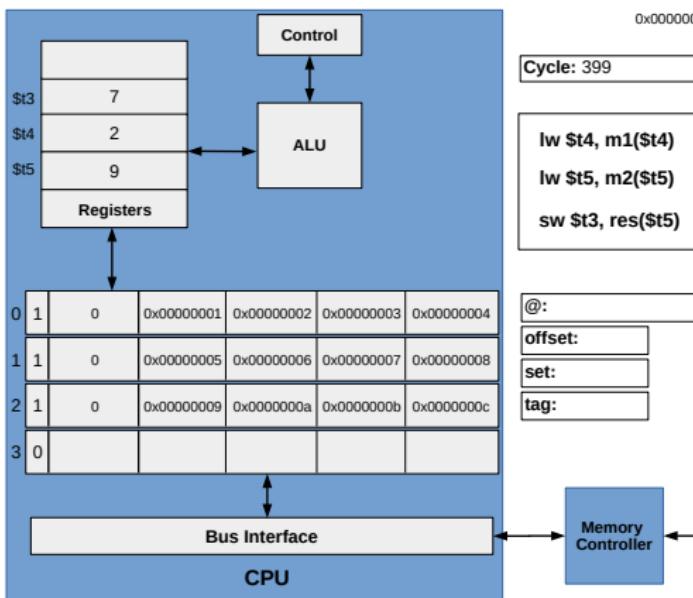
Cache Big Picture



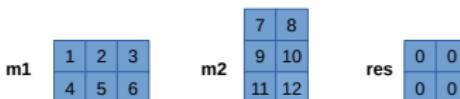
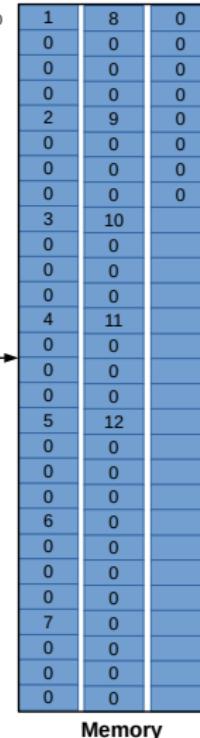
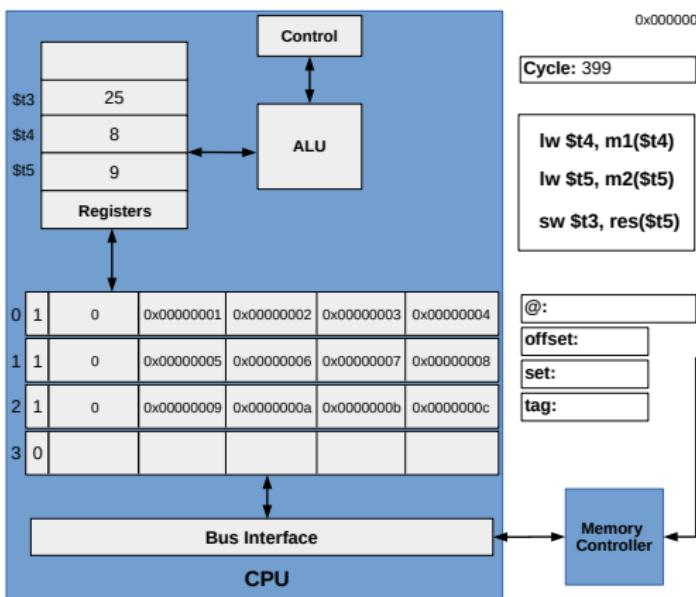
1	8	0
0	0	0
0	0	0
0	0	0
2	9	0
0	0	0
0	0	0
0	0	0
3	10	0
0	0	0
0	0	0
0	0	0
4	11	0
0	0	0
0	0	0
0	0	0
5	12	0
0	0	0
0	0	0
0	0	0
6	0	0
0	0	0
0	0	0
7	0	0
0	0	0
0	0	0
0	0	0

Memory:

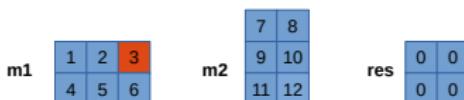
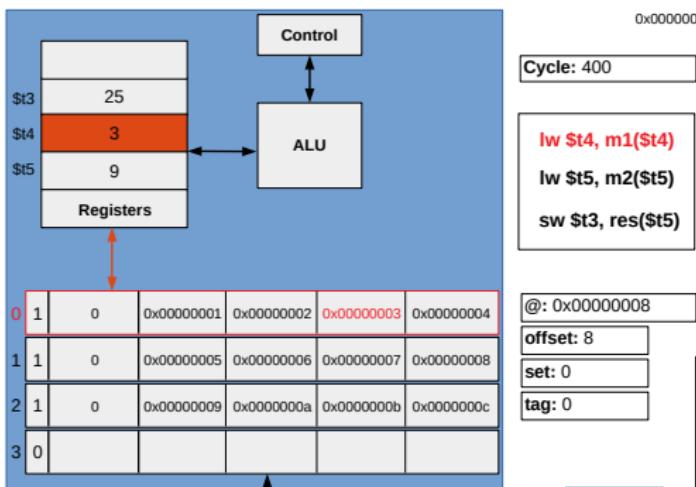
Cache Big Picture



Cache Big Picture



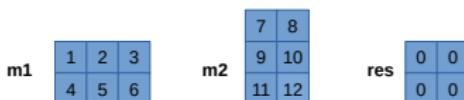
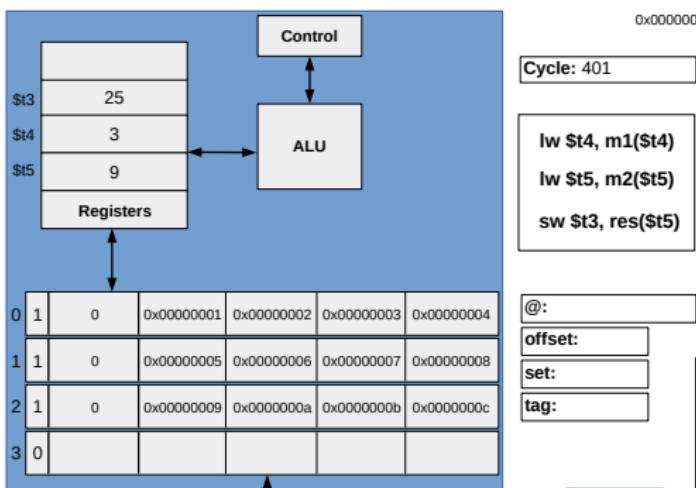
Cache Big Picture



1	8	0
0	0	0
0	0	0
0	0	0
2	9	0
0	0	0
0	0	0
0	0	0
3	10	0
0	0	0
0	0	0
0	0	0
4	11	0
0	0	0
0	0	0
0	0	0
5	12	0
0	0	0
0	0	0
0	0	0
6	0	0
0	0	0
0	0	0
7	0	0
0	0	0
0	0	0
0	0	0

Memory

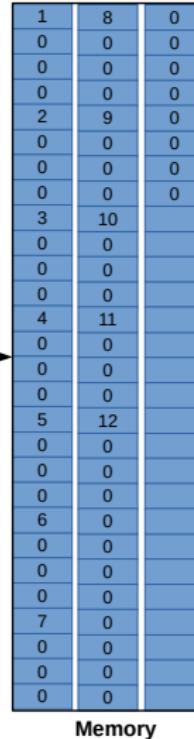
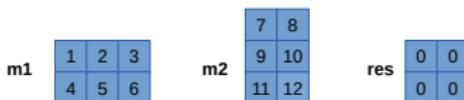
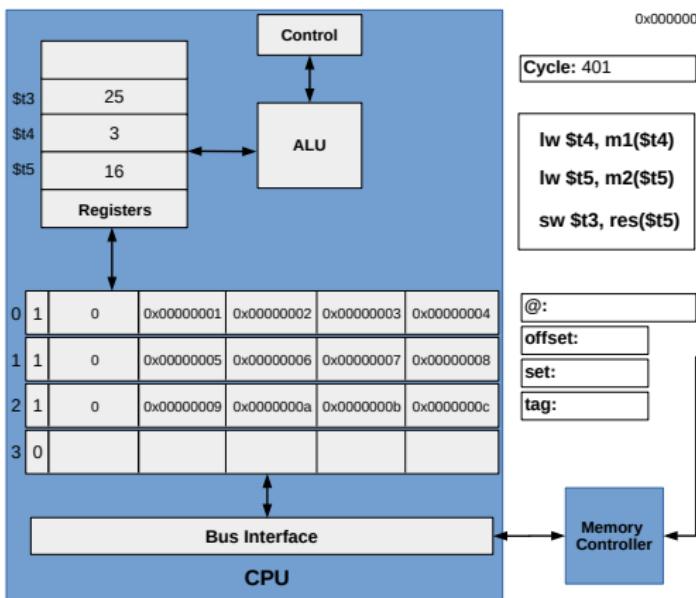
Cache Big Picture



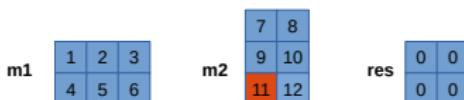
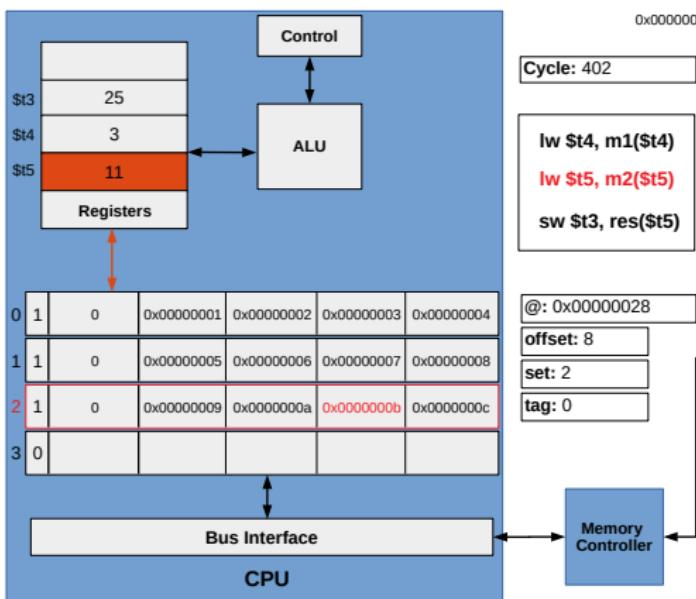
0x00000000	1	8	0
	0	0	0
	0	0	0
	0	0	0
	2	9	0
	0	0	0
	0	0	0
	0	0	0
	3	10	0
	0	0	0
	0	0	0
	0	0	0
	4	11	0
	0	0	0
	0	0	0
	5	12	0
	0	0	0
	0	0	0
	6	0	0
	0	0	0
	7	0	0
	0	0	0
	0	0	0
	0	0	0

Memory

Cache Big Picture



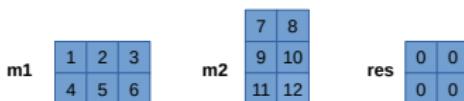
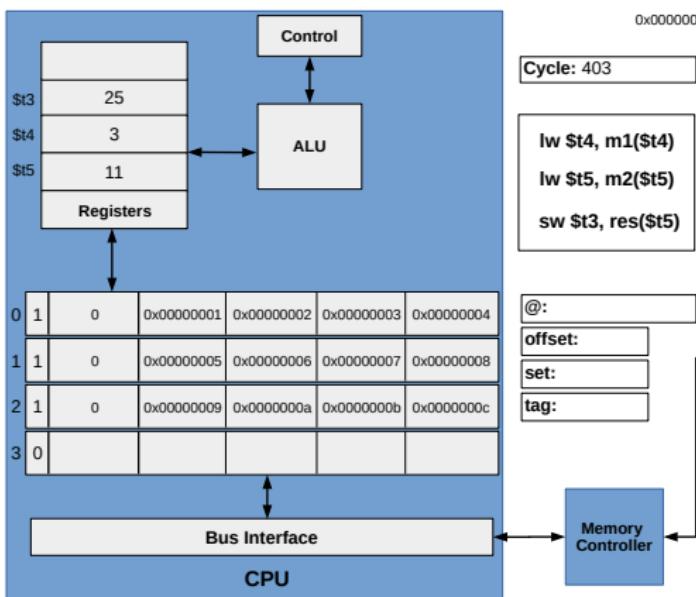
Cache Big Picture



0	1	8	0
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	0	0	0
13	0	0	0
14	0	0	0
15	0	0	0

Memory

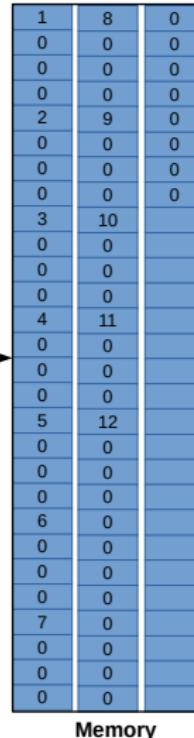
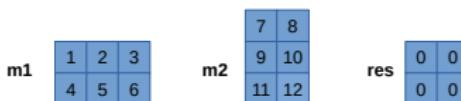
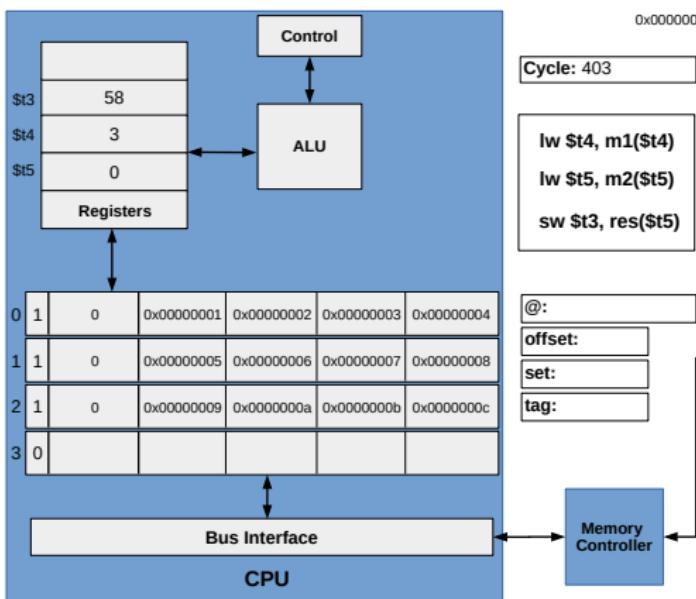
Cache Big Picture



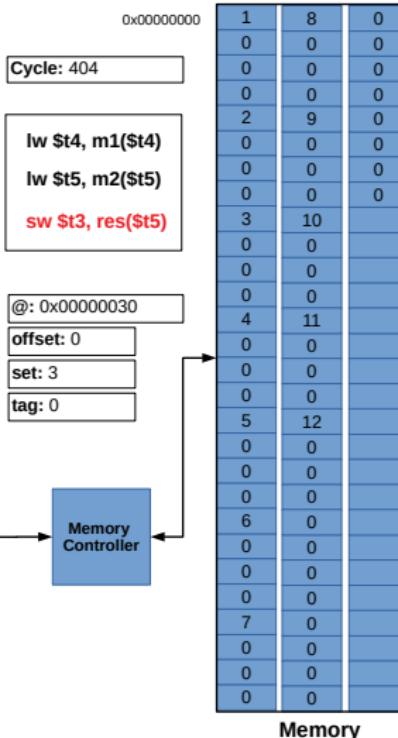
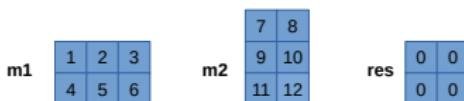
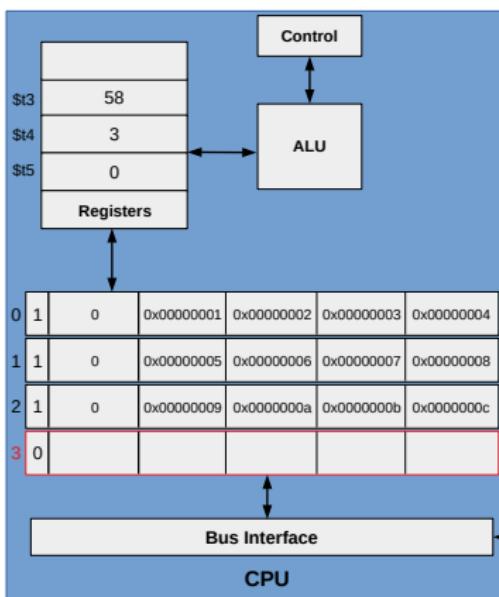
0x00000000	1	8	0
	0	0	0
	0	0	0
	0	0	0
	2	9	0
	0	0	0
	0	0	0
	0	0	0
	3	10	0
	0	0	0
	0	0	0
	0	0	0
	4	11	0
	0	0	0
	0	0	0
	5	12	0
	0	0	0
	0	0	0
	6	0	0
	0	0	0
	0	0	0
	7	0	0
	0	0	0
	0	0	0
	0	0	0

Memory

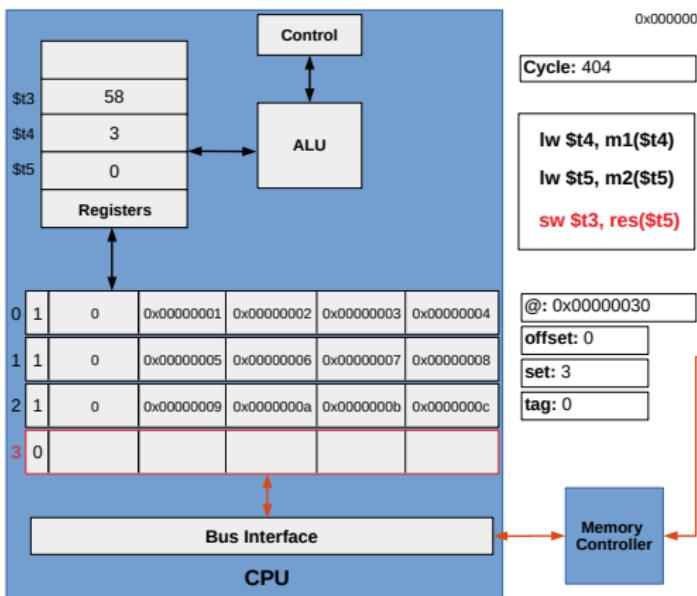
Cache Big Picture



Cache Big Picture



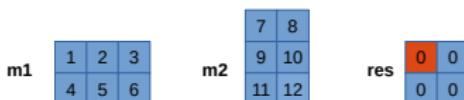
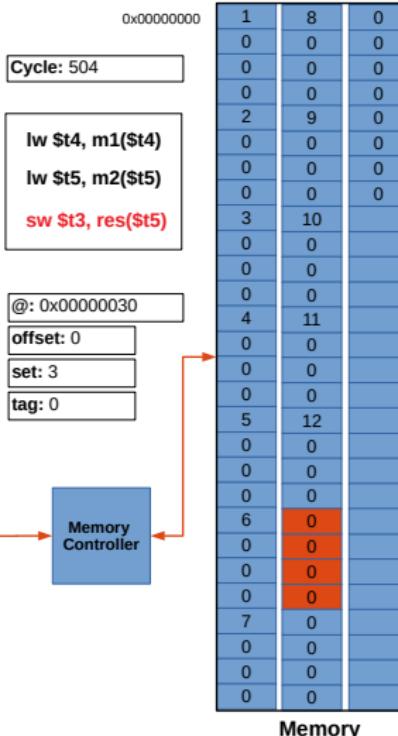
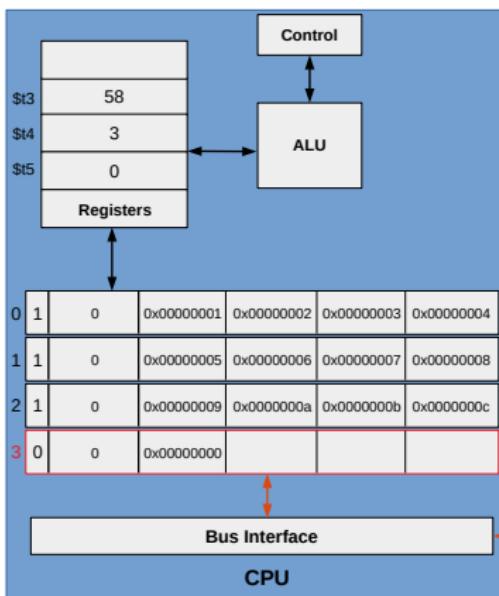
Cache Big Picture



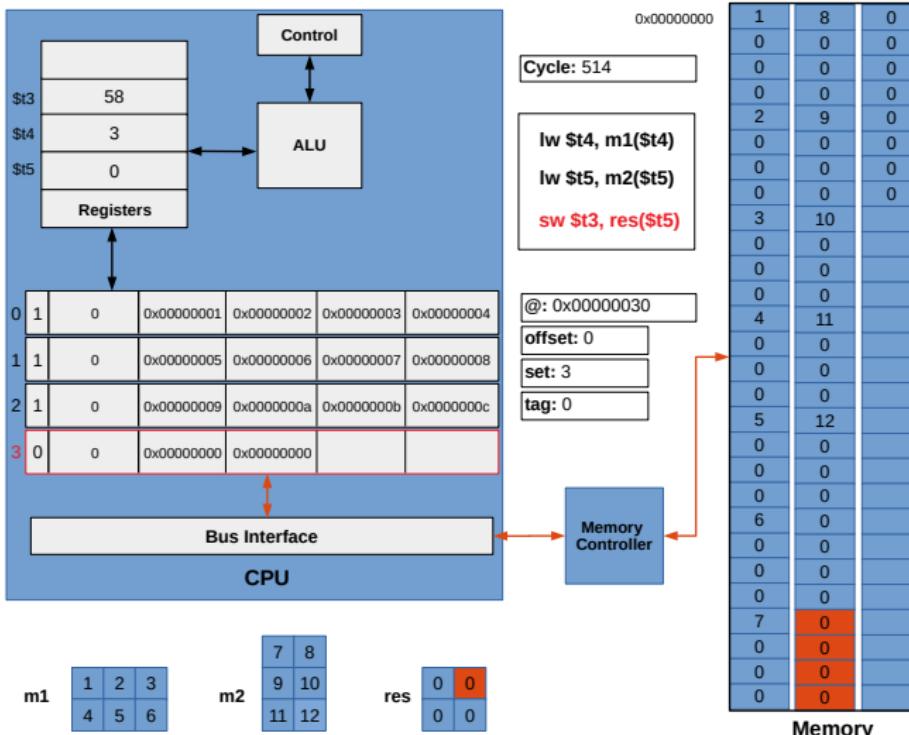
0x00000000	1	8	0
0	0	0	0
0	0	0	0
0	0	0	0
2	9	0	0
0	0	0	0
0	0	0	0
0	0	0	0
3	10	0	0
0	0	0	0
0	0	0	0
0	0	0	0
4	11	0	0
0	0	0	0
0	0	0	0
5	12	0	0
0	0	0	0
0	0	0	0
6	0	0	0
0	0	0	0
0	0	0	0
7	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Memory

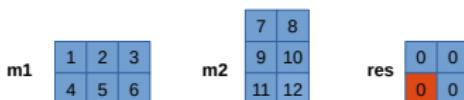
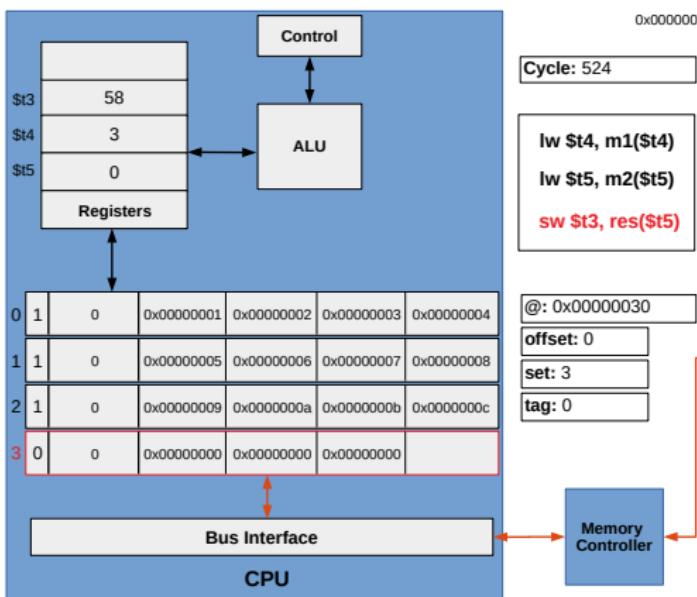
Cache Big Picture



Cache Big Picture



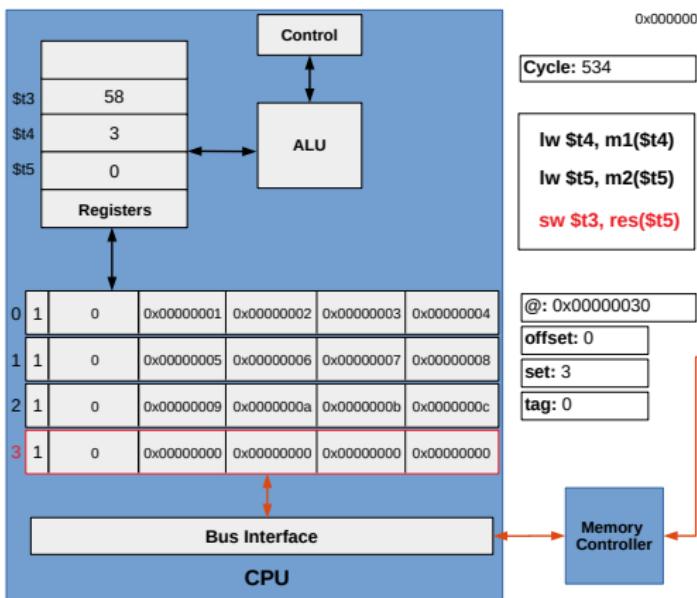
Cache Big Picture



0	1	8	0
0	0	0	0
0	0	0	0
0	0	0	0
2	9	0	0
0	0	0	0
0	0	0	0
0	0	0	0
3	10	0	0
0	0	0	0
0	0	0	0
0	0	0	0
4	11	0	0
0	0	0	0
0	0	0	0
5	12	0	0
0	0	0	0
0	0	0	0
6	0	0	0
0	0	0	0
7	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Memory

Cache Big Picture

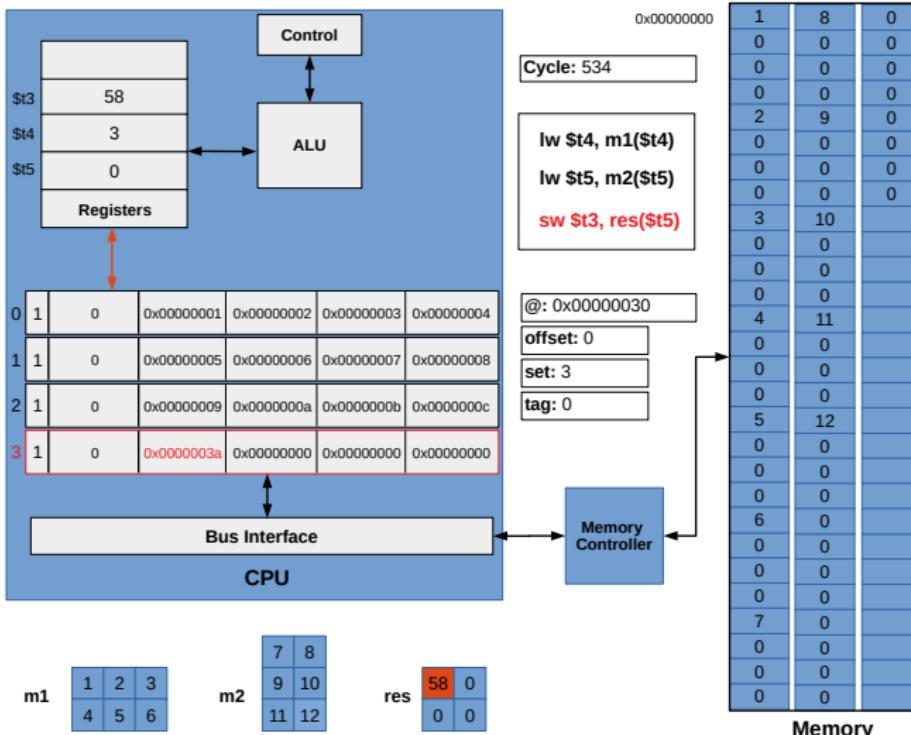


0x00000000	1	8	0
0	0	0	0
0	0	0	0
0	0	0	0
2	9	0	0
0	0	0	0
0	0	0	0
0	0	0	0
3	10		
0	0		
0	0		
0	0		
4	11		
0	0		
0	0		
0	0		
5	12		
0	0		
0	0		
0	0		
6	0		
0	0		
0	0		
7	0		
0	0		
0	0		
0	0		

Memory

m1	1	2	3	7	8	res	0	0
	4	5	6	9	10		0	0

Cache Big Picture



Cache Hit and Miss

- Cache organisation will have an impact on hit time and miss rate
- Simple designs will reduce hit time and more sophisticated ones will reduce miss rate but increase hit time
- Causes of misses
 - **Compulsory:** The very first access to a block cannot be in the cache, so the block must be brought into the cache
 - **Capacity:** If the cache cannot contain all the blocks needed during execution, some blocks will be discarded and later retrieved
 - **Conflict:** Due to the organisation of the cache, a block must be discarded to make room to another one, even if there is still room in the cache
 - **Coherence:** To keep different caches coherent in multicore environment, some blocks can be invalidated if another core write to the same block

Average Memory Access Time

- Average memory access time is
 - Hit time + Miss rate \times Miss penalty
- Example
 - Suppose cache hit time is 1 cycle and miss penalty is 100 cycles
 - If miss rate is 10% then average memory access time is

$$1 + 0.1 \times 100 = 11 \text{ cycles}$$

- If miss rate is 3% then average memory access time is

$$1 + 0.03 \times 100 = 4 \text{ cycles}$$

Exercice 10

We add a second level of cache with a hit time of 5 cycles and a miss rate of 1%. The first level cache has a miss rate of 5% and a hit time of 1 cycle. The memory access time is 100 cycles.

- What is the average memory access time?
- What is the speed-up compare to only having the first level of cache?

Solution

- Average memory access time with two caches

$$1 + 0.05 \times (5 + 0.01 \times 100) = 1.3 \text{ cycles}$$

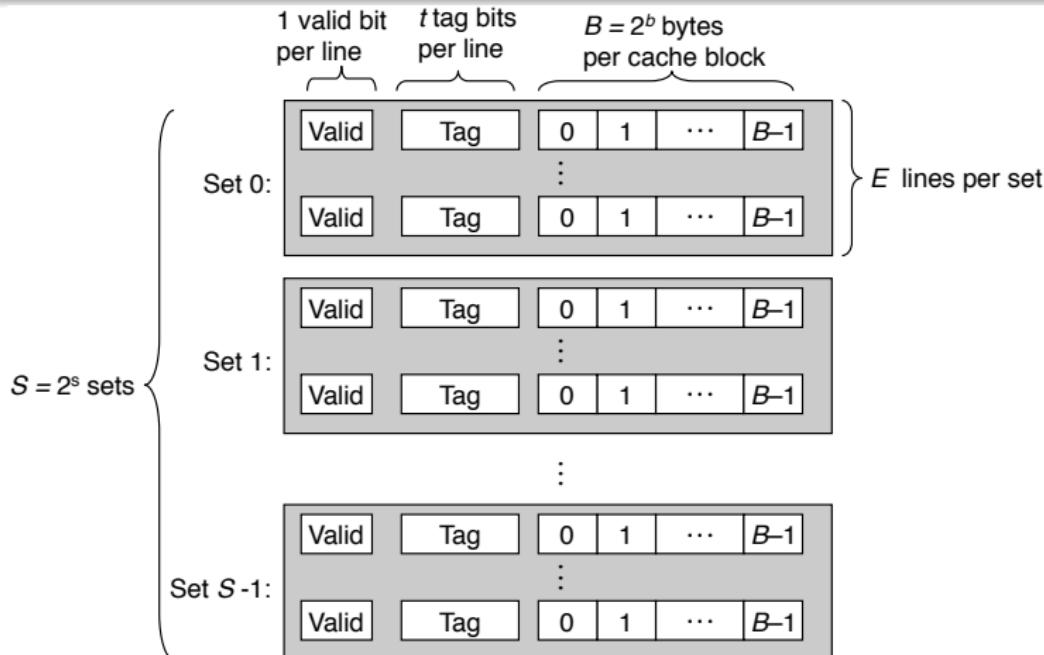
- Average memory access time with one cache

$$1 + 0.05 \times 100 = 6 \text{ cycles}$$

- Speed-up

$$\frac{6}{1.3} = 4.6$$

General Cache Organisation



Cache size: $C = B \times E \times S$ data bytes

Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Overview
Locality
Cache

Cache Organisation
Code Optimization for Cache

Eviction Policy

Introduction
Unsigned, Signed, Float
From C to Binary
MIPS
Single-Cycle Datapath
Multi-Cycle Datapath

Pipelining
Interrupts
Out of Order
Memory Hierarchy
Multicore

Overview
Locality
Cache

Cache Organisation
Code Optimization for Cache

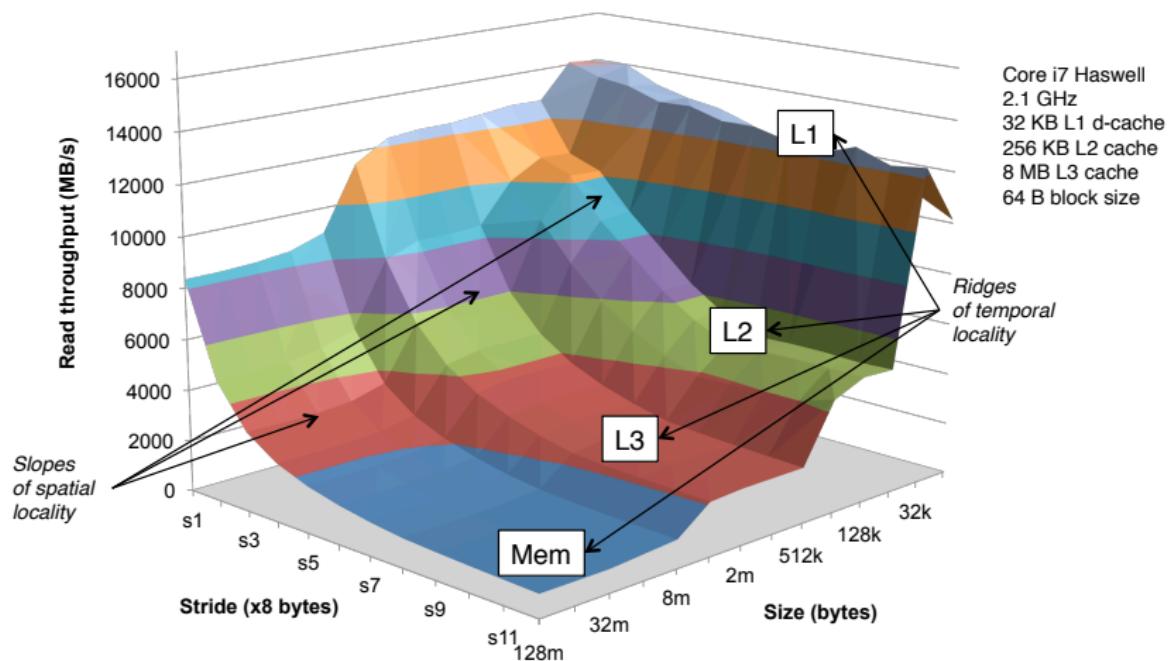
Writes

Memory Mountain

```
1 // From Computer Systems: A Programmer's Perspective
2 long data[MAXELEMS]; // The global array we'll be traversing
3 /* test - Iterate over first "elems" elements of array "data" with
4  *      stride of "stride", using 4x4 loop unrolling. */
5 int test(int elems, int stride) {
6     long i, sx2 = stride*2, sx3 = stride*3, sx4 = stride*4;
7     long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
8     long length = elems;
9     long limit = length - sx4;
10    for (i = 0; i < limit; i += sx4) { // Combine 4 elements at a time
11        acc0 = acc0 + data[i];
12        acc1 = acc1 + data[i+stride];
13        acc2 = acc2 + data[i+sx2];
14        acc3 = acc3 + data[i+sx3];
15    }
16    for (; i < length; i += stride) { // Finish any remaining elements
17        acc0 = acc0 + data[i];
18    }
19    return ((acc0 + acc1) + (acc2 + acc3));
20 }
```

Memory Mountain

(From *Computer Systems: A Programmer's Perspective*)



Code Optimization for Cache

- Make the common case fast, where most of the time is spent in your program
- Write code that exhibit good local and spatial locality
- Repeated references to variables are good (**temporal locality**)
 - Re-order data accesses to improve temporal locality
- Stride-1 reference patterns are good (**spatial locality**)
 - Group data accesses together to improve spatial locality

Loop Interchange

```
1 long long test(int nb)
2 {
3     long long sum = 0;
4     for (int i = 0; i < nb; ++i)
5         for (int k = 0; k < SIZE; ++k)
6             for (int j = 0; j < SIZE; ++j)
7             {
8                 sum += data[j][k];
9             }
10    return sum;
11 }
```

Loop Interchange

```
1 long long test(int nb)
2 {
3     long long sum = 0;
4     for (int i = 0; i < nb; ++i)
5         for (int j = 0; j < SIZE; ++j)
6             for (int k = 0; k < SIZE; ++k)
7             {
8                 sum += data[j][k];
9             }
10    return sum;
11 }
```

Loop Fusion

```
1 #define N 8000
2 #define M 6000
3 long A[N][M], B[N][M], C[N][M], D[N][M];
4
5 void no_fusion(int nb)
6 {
7     for (int k = 0; k < nb; k++)
8     {
9         for (int i = 0; i < N; i++)
10            for (int j = 0; j < M; j++)
11            {
12                A[i][j] = B[i][j] * C[i][j];
13            }
14         for (int i = 0; i < N; i++)
15            for (int j = 0; j < M; j++)
16            {
17                D[i][j] = A[i][j] * C[i][j];
18            }
19     }
20 }
```

Loop Fusion

```
1 #define N 8000
2 #define M 6000
3 long A[N][M], B[N][M], C[N][M], D[N][M];
4
5 void fusion(int nb)
6 {
7     for (int k = 0; k < nb; k++)
8     {
9         for (int i = 0; i < N; i++)
10            for (int j = 0; j < M; j++)
11            {
12                A[i][j] = B[i][j] * C[i][j];
13                D[i][j] = A[i][j] * C[i][j];
14            }
15    }
16 }
```

Blocking

```
1 #define N 2000
2 int A[N][N], B[N][N], C[N][N];
3 void matmult()
4 {
5     for (int i = 0; i < N; ++i)
6     {
7         for (int j = 0; j < N; ++j)
8         {
9             int sum = 0;
10            for (int k = 0; k < N; ++k)
11            {
12                sum += A[i][k] * B[k][j];
13            }
14            C[i][j] = sum;
15        }
16    }
17 }
18 // gcc -O3 blocking.c -o blocking
19 // ./blocking
20 // 16.705886 seconds
```

Blocking

```
1 #define BLOCK 32
2 #define MIN(X,Y) ((X) < (Y) ? (X) : (Y))
3 void matmult_blocking()
4 {
5     for (int jj = 0; jj < N; jj += BLOCK)
6         for (int kk = 0; kk < N; kk += BLOCK)
7             for (int i = 0; i < N; ++i)
8                 for (int j = jj; j < MIN(jj + BLOCK, N); ++j)
9                 {
10                     int sum = 0;
11                     for (int k = kk; k < MIN(kk + BLOCK, N); ++k)
12                     {
13                         sum += A[i][k] * B[k][j];
14                     }
15                     C[i][j] += sum;
16                 }
17 }
18 // gcc -O3 blocking.c -o blocking
19 // ./blocking
20 // 3.761188 seconds
```

Prefetching

```
1 #include <time.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define SIZE 10000000000
6 // from Zhirkov
7 int binary_search(int *array, size_t number_of_elements, int key) {
8     size_t low = 0, high = number_of_elements-1, mid;
9     while (low <= high) {
10         mid = low + (high - low) / 2;
11         __builtin_prefetch(&array[(mid + 1 + high) / 2], 0, 1);
12         __builtin_prefetch(&array[(low + mid - 1) / 2], 0, 1);
13         if (array[mid] < key)      low = mid + 1;
14         else if (array[mid] == key) return mid;
15         else if (array[mid] > key) high = mid-1;
16     }
17     return -1;
18 }
```

Prefetching

```
1 sub    $0x1,%rsi
2 xor    %ecx,%ecx
3 jmp    4004e9 <binary_search(int*, unsigned long, int)+0x19>
4 nopl   0x0(%rax,%rax,1)
5 lea    0x1(%rax),%rcx
6 cmp    %rsi,%rcx
7 ja    40052a <binary_search(int*, unsigned long, int)+0x5a>
8 mov    %rsi,%rax
9 sub    %rcx,%rax
10 shr   %rax
11 add   %rcx,%rax
12 lea    0x1(%rsi,%rax,1),%r8
13 shr   %r8
14 prefetcht2 (%rdi,%r8,4)
15 lea    -0x1(%rcx,%rax,1),%r8
16 shr   %r8
17 prefetcht2 (%rdi,%r8,4)
18 mov    (%rdi,%rax,4),%r8d
19 cmp    %edx,%r8d
20 jl    4004e0 <binary_search(int*, unsigned long, int)+0x10>
21 je    40052f <binary_search(int*, unsigned long, int)+0x5f>
22 sub   $0x1,%rax
23 cmp    %edx,%r8d
24 cmovg %rax,%rsi
25 cmp    %rsi,%rcx
26 jbe   4004e9 <binary_search(int*, unsigned long, int)+0x19>
27 mov    $0xffffffff,%eax
28 retq
```

Exercice 11

In this exercise we will compute the miss rate of different code for matrix multiplication. We assume that

- Block size is 32 bytes.
- The elements in the matrix are double, so a block can hold 4 elements
- Matrix dimension N is very large
- Cache can hold only one row
- No conflict misses

We have to compute the miss rate for each code on the next three slides, considering only the inner most loop.

Matrix Multiplication 1

```
1 void mm_ijk()
2 {
3     double sum;
4     for (int i = 0; i < N; i++)
5     {
6         for (int j = 0; j < N; j++)
7         {
8             sum = 0.0;
9             for (int k = 0; k < N; k++)
10                 sum += a[i][k] * b[k][j];
11             c[i][j] = sum;
12         }
13     }
14 }
```

Matrix Multiplication 2

```
1 void mm_kij()
2 {
3     double r;
4     for (int k = 0; k < N; k++)
5     {
6         for (int i = 0; i < N; i++)
7         {
8             r = a[i][k];
9             for (int j = 0; j < N; j++)
10                c[i][j] += r * b[k][j];
11         }
12     }
13 }
```

Matrix Multiplication 3

```
1 void mm_jki()
2 {
3     double r;
4     for (int j = 0; j < N; j++)
5     {
6         for (int k = 0; k < N; k++)
7         {
8             r = b[k][j];
9             for (int i = 0; i < N; i++)
10                c[i][j] += a[i][k] * r;
11         }
12     }
13 }
```

Multicore