103011227_report

作者

周延儒

computer architecture

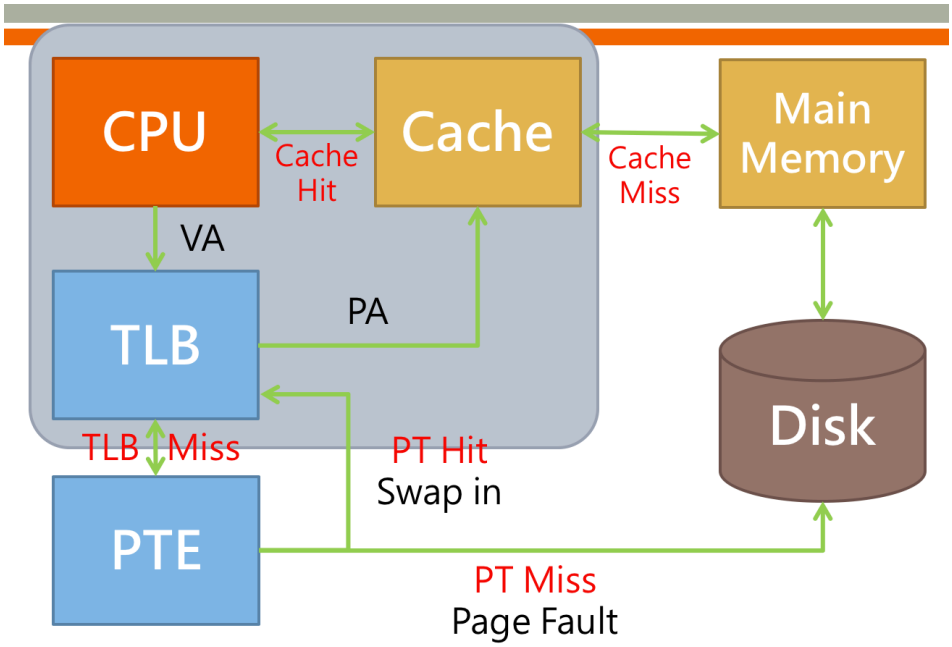
project 3

This is final project for the semester. In this project ,we are required to implement the memory hierarchical CPU base on the single cycle CPU we built in the first project . To achieve the memory hierarchy , we have to build the Translation lookaside Buffer(TLB), Virtual Page table (PTE),Cache, and memory .

Compared with the single cycle CPU ,where there is only disk for the data storage. The CPU in the project 3 can make the best use of the memory hierarchy for the memory , making the data transfer much faster .

So, let' s begin with the project. As the name suggests, the CMP is the abbreviation of the Cache Memory Pagetable. We introduce the concept of memory hierarchy .

The architecture would be as follows



Whenever we need to load and store the data, we have to get the Virtual address and thus we can know its Virtual page number (VPN) .

Next ,we know inquire whether in the the data is in the memory . If so , we can search data in the Cache . Then it is in the Cache , the cycle is complete. If it can not be found in the cache , which means it is not in the cache but in the memory . As as result , we need to search for the data in the memory , and update the cache after we find it. Back to the TLB, if we can find the

corresponding VPN in the TLB. Don’ t be upset, we can still query the Page table, which h can viewed as the larger and sophisticated TLB, although their behavior is quite different because the TLB is the real hardware , and on the contrary, the PTE is the data structure of the software.

If we can find the VPN in the PTE, which means the data is at least in the memory , we can do search the Cache. If it is in the cache, we have to update the TLB ,which is missed when we queried in the beginning. If we can not find it in the cache. That is , it is not in the cache but in the memory . Search for the data in the memory and update the TLB and Cache.

In the end, if the data is missed in both TLB and Page Table. Namely it is not in the memory . We have to swap, update Page Table, TLB an Cache

TLB	Page table	Cache	Action
Hit	Hit	Hit	
Hit	Hit	Miss	Search memory ➔ update cache
Miss	Hit	Hit	Update TLB
Miss	Hit	Miss	Update TLB & search memory ➔ update cache
Miss	Miss	Miss	Swap ➔ update PageTable ➔ update TLB ➔ update cache

Something we need to notice is that the Cache and memory is write-back

Moreover the TLB and memory is least recently used(LRU),which means we always replace the least used data.

In this project , the interesting part is the Cache bits-pseudo LRU instead of the LRU.

In the bit_pseudo LRU, we add a additional bit ,called MRU(most rececntly used) to record the data of the cache block. In each time , we replace the least indexed invalid set if exists when the cache is missed. Otherwise,,

we replace the lowest indexed block of MRU=1 .If all block MRUs are 1,we just reset all MRUs to 0, except the one just replaced. The most weird part is that it take not effect if the cache is direct-map . Any replace policy won’ t work in this case.

The more detailed description in the appendix, we don't specified here.

Before we start with the design , we have the address our momery design .

First , we have two memory organization . One is for the VA(PC) ,and the other is for the data(load and store instruction).

Secondly , for the Cache, the replacement policy is Bit-Pseudo LRU

Thirdly , for TLB(Tranaslation lookaside Table), it is fully –associative and the size of it is 1/4 page table . As a result , the number of the TLB entries is 1/4*page table entries. It adopt LRU replacement policy

Next , for Page Table, although theoretically we are supposed to cover all the virtual space for the 32-bit architecture,for simplicity , the number of the page table entries is disk_size(1K bytes)/page_size.

Eventually ,for the memory ,it adopt write-back/allocate policy.

To enhance the Configurability, we can take argument from the command line to set up the memory parameters.

i. The instruction memory (I memory) size, in number of bytes

ii. The data memory (D memory) size, in number of bytes

iii. The page size of instruction memory (I memory), in number of bytes

iv. The page size of data memory (D memory), in number of bytes

v. The total size of instruction cache (I cache), in number of bytes
vi. The block size of I cache, in number of bytes
vii. The set associativity of I cache

viii. The total size of data cache (D cache), in number of bytes

ix. The block size of D cache, in number of bytes

x. The set associativity of D cache

After we comprehend the basic architecture of the memory hierarchy ,let's begin with the design !

First of all, the memory hierarchy works in each cycle ,where the CPU reads the PC (VA)value and get the I-instrcution, and data (load /store instruction).

The design flows are as follows:

disk size(1024byte) >= memory size >= page size >= block size >= 1word(4byte)

disk size(1024byte) >= cache size >= block size

```
struct _PAGE {
    int valid;
    int space[256];
    int LRU;
};
for each page in the memory
struct _MEMORY {
    int hits, misses;
    int size, pageSize;
    struct _PAGE page[256];
};
for the memory . Since the max size of the memory is 1024 which must be smaller
than that of disk , the memory can have at most 256 pages,and the page size is at
```

```
most 256 or the TLB would less than 1(#page_table_entries = disk_size / page_size.
#TLB_entries = 1/4*(#page_table_entries).)

struct _BLOCK {
    int valid;
    int tag, ind, off;
    int LRU, ppn;
    int MRU;
    int space;
};
struct _CACHE {
    int hits, misses;
    int totalSize, blockSize, set_associativity;
    struct _BLOCK block[256][256];
};
similarly.

struct _PPN {
    int valid;
    int ppn;
    int LRU;
    int vpn;
};
struct _TLB {
    int hits, misses;
    int entry;
    struct _PPN ppn[256];
};
struct _PTE {
    int hits, misses;
    int entry;
    struct _PPN ppn[256];
};
similarly.

struct _MEMORY IMEM, DMEM;
struct _CACHE ICACHE, DCACHE;
struct _TLB ITLB, DTLB;
struct _PTE IPTE, DPTE;

int LRU_Counter;

int IMSIZE;       // I memory size
int DMSIZE;       // D memory size
int IPGSIZE;      // Page size of I memory
int DPGSIZE;      // Page size of D memory
int ICTSIZE;      // Total size of I cache
int ICBSIZE;      // Block size of I cache
int ICSA;         // Set associativity of I cache
int DCTSIZE;      // Total size of D cache
int DCBSIZE;      // Block size of D cache
int DCSA;          // Set associativity of D cache
```

Take D-memory design for example

```
    //get the VPN number in the disk
    // unsigned VPN = getVPN(VA, 1);
    unsigned VPN=getDVPN(VA);
    //get Pysical page number in the memory
    int PPN = getDTLBPPN(VPN);
    if (PPN != -1) {
        DTLB.hits++;
        //TLB hit and PTE hit
    } else {
        DTLB.misses++;

        PPN = getDPTEPPN(VPN);
        if (PPN != -1) {
            //TLB miss and PTE hit
            //Update TLB
            DPTE.hits++;
            updateDTLBPPN(VPN, PPN);
        } else {
            DPTE.misses++;
            //Swap  update PageTable  update TLB  update cache
            PPN = insertDMEM(VA);
            updateDPTEPPN(VPN, PPN);
            updateDTLBPPN(VPN, PPN);
        }
    }

    queryDCACHE(VA, PPN);
```

First of all, we need to know what the virtual page number(VPN) for the Virtual Addresss(VA) is .

The VA is viewed as [VPN][Page offset], where the VPN is what we desired , and the Page offset is related to the page size . Page size is 2^page offset.

VA

VPN | Page Offset

```
    int pageOffset=0;
    int DPageSize=0;
    DPageSize = DMEM.pageSize-1;
    while (DPageSize) {
        pageOffset++;
        DPageSize =(DPageSize>>1);
    }
```

```
return (VA >> pageOffset);
```

Then after we get VPN ,we get to know whether it is in the TLB

Since the The TLBs is fully-associative and its size is a quarter of the page table size, we can travers all the entries to look for the data .

```
  int PPN =0;
    int i;

    for ( i = 0; i < DTLB.entry; i++) {
        if (DTLB.ppn[i].valid == 1 && DTLB.ppn[i].vpn == VPN) {
            PPN = DTLB.ppn[i].ppn;
            DTLB.ppn[i].LRU = LRU_Counter++;
            return PPN;
        }
    }

    return -1;
```

We traverse the TLB entries. If the ppn(physical page number) is valid(1) and its VPN is exactly the same as the VPN we want , we hit the TLB ,and of course , we have to update the it LRU . Notice that I declare a global variable LRU_counter to record the LRU . For each hits ,it always plus one.

Otherwise ,we missed the TLB.

If we hit the TLB .it would return the corresponding PPN . On the contrary if we do miss it ,it would return -1 ,representing we miss it.

After traversing the TLB, we can know whether we find it or not.

If the PPN we find is non-negative, we hit the TLB and update DTLB.hits++;

If it is negative -1, we miss the TLB, and then we should try to find it in the Page Table ,`PPN = getDPTEPPN(VPN);`

```
int PPN = 0;
    int i;
    for ( i = 0; i < DPTE.entry; i++) {
        if (DPTE.ppn[i].valid == 1 && DPTE.ppn[i].vpn == VPN) {
            PPN = DPTE.ppn[i].ppn;
            DPTE.ppn[i].LRU = LRU_Counter++;
            return PPN;
        }
    }

    return -1;
```

The method is as same as that of TLB.

To to date, we can know whether the it is in the TLB, if it in the TLB updates the `DPTE.hits++;`and don't forget to TLB . `updateDTLBPPN(VPN, PPN);`

```
int ind = -1;
    int i;

    for ( i = 0; i < DTLB.entry; i++) {
        if (DTLB.ppn[i].valid == 0) {
            ind = i;
            break;
        }
    }
    if (ind != -1) {
        DTLB.ppn[ind].valid = 1;
        DTLB.ppn[ind].vpn = VPN;
        DTLB.ppn[ind].ppn = PPN;
    } else {
        int tmp = 0x7fffffff;
        for ( i = 0; i < DTLB.entry; i++) {
            if (DTLB.ppn[i].LRU < tmp) {
                tmp = DTLB.ppn[i].LRU;
                ind = i;
            }
        }
        DTLB.ppn[ind].valid = 1;
        DTLB.ppn[ind].vpn = VPN;
        DTLB.ppn[ind].ppn = PPN;

    }
```

To update the information in the TLB , we need to traverse the DTLB entries to see whether there exist empty . If there exist one place, the ind would the index ,and thus update it

```
DTLB.ppn[ind].valid = 1;
DTLB.ppn[ind].vpn = VPN;
DTLB.ppn[ind].ppn = PPN;
```

Or we adopt the the LRU replacement policy.

We declare tmp to be the largest integer, and traverse all the entries to get least value .

Then we replace ppn[ind], which is the least recently used

```
DTLB.ppn[ind].valid = 1;
DTLB.ppn[ind].vpn = VPN;
DTLB.ppn[ind].ppn = PPN;
```

Next if we fail to find the VPN neither in the TLB nor Page Table. That means it it not in the memory .

What gives rise to the phenomenon may be that we don't load or store the data for a long period time depending size of the TLB and Page Table.

In general , the larger TLB and Page Table are, the larger Hit rate would be.

Also ,the higher space locality is , the larger hit rate would be,such as load or store data in a loop.

When the data is not in the memory. The situation is called page fault, where the accessed page is not present in main memory.

we are supposed to Swap ➔ update PageTable ➔ update TLB ➔ update cache.

```
PPN = insertDMEM(VA);
updateDPTEPPN(VPN, PPN);

updateDTLBPPN(VPN, PPN);
```

swap for the memory

```
int PPN = -1;
    int i;
    int Number_DMEM_Pages=DMEM.size/DMEM.pageSize;

    for ( i = 0; i < Number_DMEM_Pages; i++) {
        if (DMEM.page[i].valid == 0) {
            PPN = i;
            break;
        }
    }
    if (PPN != -1) {
        DMEM.page[PPN].valid = 1;
        for ( i = 0; i < DMEM.pageSize/4; i++) {
            DMEM.page[PPN].space[i] = VA+i*4;
        }
        DMEM.page[PPN].LRU = LRU_Counter++;
    } else {
        int LLRU = 0x7fffffff;
        for ( i = 0; i < Number_DMEM_Pages; i++) {
            if (DMEM.page[i].LRU < LLRU) {
                LLRU = DMEM.page[i].LRU;
                PPN = i;
            }
```

```
        }
        for ( i = 0; i < DTLB.entry; i++) {
            if (DTLB.ppn[i].ppn == PPN) {
                DTLB.ppn[i].valid = 0;
            }
        }
        for ( i = 0; i < DPTE.entry; i++) {
            if (DPTE.ppn[i].ppn == PPN) {
                DPTE.ppn[i].valid = 0;
            }
        }
        int set=DCACHE.totalSize/DCACHE.blockSize/DCACHE.set_associativity;
        //if set may be the block if it is directed mapped

        for ( i = 0; i < set; i++) {
            int j;
            for ( j = 0; j < DCACHE.set_associativity; j++) {
                if (DCACHE.block[i][j].ppn == PPN) {
                    DCACHE.block[i][j].valid = 0;
                    DCACHE.block[i][j].MRU=0;
                }
            }
        }
        DMEM.page[PPN].valid = 1;
        for ( i = 0; i < DMEM.pageSize/4; i++) {
            DMEM.page[PPN].space[i] = VA+i*4;
        }
        DMEM.page[PPN].LRU = LRU_Counter++;
    }
```

Still , we try to find a empty page in the memory . If we can find a empty page

,put the data in it

```
        DMEM.page[PPN].valid = 1;
        for ( i = 0; i < DMEM.pageSize/4; i++) {
            DMEM.page[PPN].space[i] = VA+i*4;
        }

        DMEM.page[PPN].LRU = LRU_Counter++;
```

If we can not find an empty page , replace the LRU set. Pick the least indexed set to be the victim in case of tie.

```
       int LLRU = 0x7fffffff;
        for ( i = 0; i < Number_DMEM_Pages; i++) {
            if (DMEM.page[i].LRU < LLRU) {
                LLRU = DMEM.page[i].LRU;
                PPN = i;
            }
```

```
    }
```

Here we can find the “victim”

```
    for ( i = 0; i < DTLB.entry; i++) {
        if (DTLB.ppn[i].ppn == PPN) {
            DTLB.ppn[i].valid = 0;
        }
    }
    for ( i = 0; i < DPTE.entry; i++) {
        if (DPTE.ppn[i].ppn == PPN) {
            DPTE.ppn[i].valid = 0;
        }
    }
```

remember the reset the all data related to the victim

```
    int set=DCACHE.totalSize/DCACHE.blockSize/DCACHE.set_associativity;
    //if set may be the block if it is directed mapped

    for ( i = 0; i < set; i++) {
        int j;
        for ( j = 0; j < DCACHE.set_associativity; j++) {
            if (DCACHE.block[i][j].ppn == PPN) {
                DCACHE.block[i][j].valid = 0;
                DCACHE.block[i][j].MRU=0;
            }
        }
    }

    DMEM.page[PPN].valid = 1;
    for ( i = 0; i < DMEM.pageSize/4; i++) {
        DMEM.page[PPN].space[i] = VA+i*4;
    }

    DMEM.page[PPN].LRU = LRU_Counter++;
```

In the end, we have to query the cache. Here we take D-Cache.

```
queryDCACHE(VA, PPN);
```

Depeding on the cache set associativity , the behavior is quite different.

For a cache , if it's direct map, they have blocks.if it's it set associative , they have sets. Here we declare them as `block[256][256]` for simplicity .

In each of block ,they all have

```
int valid;
int tag, ind, off;-> tag index,offset;
```

```
    int LRU, ppn;
    int MRU;
    int space;
```

To query the cache , the first thing we need to do is get the value of PA(physical address),and then we can get the value of tag and ind(index).

```
unsigned PA = getDPA(VA, PPN);

:
    unsigned int PA = (unsigned int)PPN;
    unsigned PageOffset = VA % (1 << getDOffset());


    getDOffset():
      {
            int pageoffset=0;
            int tmp=DMEM.pageSize-1;
            while (tmp) {
            pageoffset++;
            tmp/=2;
                  }
            return pageoffset;
```

Here we can get the page offset

```
    }


    Here the pageoffset value is just VA % (1 << getDOffset())

    //physical address
    //PPN_PageOffset

    PA = ((unsigned)PPN << getDOffset())+PageOffset;

    return PA;
```

Once we have the PA,we can know the tag and the ind(index)

Somethin important is that the PA is composed of tag ,index and the o ffset.

---

tag | index |offset

The tag value is

```
unsigned getDTag(unsigned PA) {
    unsigned tmp = PA;
    tmp = tmp >> getDBlockOffset();
    tmp = tmp >> getDIndexOffset();
    return tmp;
}

where the getDIndexOffset() is
int getDIndexOffset() {
    int indexOffset = 0, tmp;
    tmp = DCACHE.totalSize/DCACHE.blockSize/DCACHE.set_associativity-1;

    notice that when the cache is direct map, DCACHE.set_associativity is 1.
    while (tmp) {
        indexOffset++;
        tmp/=2;

    }

    return indexOffset;
}
```

Thus , we just the Tag.

Moreover, the index would be

```
unsigned ind = getDInd(PA);


unsigned getDInd(unsigned PA) {
    unsigned tmp = PA;
    tmp = tmp >> getDBlockOffset();
    return (tmp % (1 << getDIndexOffset()));
}
```

We can get the value of the ind(index)

Now we have the tag and index, it is time that we queried the cache.

In the beginning , we still traverse the each set to find whether the information exists.

```
    for ( i = 0; i < DCACHE.set_associativity; i++) {
        if (DCACHE.block[ind][i].valid == 1 && DCACHE.block[ind][i].tag ==
tag) {
            hit = i;
            break;
        }

    }
```

In this part, we need to check each of block in the set ,to see whether it is valid and the tag is the tag we want.

Then if we can find the it , cache hits. All we need to do is update its MRU. Beware that we may to be reset the MRU

```
    if (hit != -1) {
        DCACHE.hits++;

        DCACHE.block[ind][hit].LRU = LRU_Counter++;
        DCACHE.block[ind][hit].MRU=1;

        if (DCACHE.set_associativity==1) {
            return;
```

Here if it is direcy map,without a doubt, any replacement policy never works. We just skip and return

```
        }


        int travese;
        int check=0;
        for (travese=0; travese<DCACHE.set_associativity; travese++) {
            if (DCACHE.block[ind][travese].MRU==1 ) {
                check++;
            }
        }

        if (check==DCACHE.set_associativity) {
            printf("reset\n");
            for (travese=0; travese<DCACHE.set_associativity; travese++) {

                DCACHE.block[ind][travese].MRU = 0;
            }
```

```
            DCACHE.block[ind][hit].MRU=1;
        }



    }
```

If we can not find any tag and valid matching the data. We miss the cache.

We have to replace the least indexed invalid set if exists; otherwise, replace the Bits-Pseudo LRU set.

```
if (hit != -1) {
            //find least invalid
            printf("the hit is %d\n",hit);



            DCACHE.block[ind][hit].valid = 1;
            DCACHE.block[ind][hit].tag = tag;
            DCACHE.block[ind][hit].LRU = LRU_Counter++;
            DCACHE.block[ind][hit].ppn = PPN;
            DCACHE.block[ind][hit].MRU=1;

            int travese;
            int check=0;
            for (travese=0; travese<DCACHE.set_associativity; travese++) {
                if (DCACHE.block[ind][travese].MRU==1 ) {
                    check++;
                }
            }

            if (check==DCACHE.set_associativity) {
                printf("reset\n");
                for (travese=0; travese<DCACHE.set_associativity; travese++) {

                    DCACHE.block[ind][travese].MRU = 0;
                }
                DCACHE.block[ind][hit].MRU=1;
            }



        } else {
            hit=-1;

            for ( i = 0; i < DCACHE.set_associativity; i++) {
                if (DCACHE.block[ind][i].MRU == 0) {
                    hit = i;
                    break;
                }
            }

            if (DCACHE.set_associativity==1) {
                //direct map
                DCACHE.block[ind][0].valid = 1;
                DCACHE.block[ind][0].tag = tag;
                DCACHE.block[ind][0].LRU = LRU_Counter++;
```

```
            DCACHE.block[ind][0].ppn = PPN;
            DCACHE.block[ind][0].MRU=1;
            return;

        }

        DCACHE.block[ind][hit].valid = 1;
        DCACHE.block[ind][hit].tag = tag;
        DCACHE.block[ind][hit].LRU = LRU_Counter++;
        DCACHE.block[ind][hit].ppn = PPN;
        DCACHE.block[ind][hit].MRU=1;


        int travese;
        int check=0;
        for (travese=0; travese<DCACHE.set_associativity; travese++) {
            if (DCACHE.block[ind][travese].MRU==1 ) {
                check++;
            }
        }

        if (check==DCACHE.set_associativity) {
            printf("reset D cache\n");
            printf("excpet %d\n",hit);

            for (travese=0; travese<DCACHE.set_associativity; travese++) {
                DCACHE.block[ind][travese].MRU = 0;

            }

            DCACHE.block[ind][hit].MRU=1;


        }



    }
```

Of course , we need to be ware of the set associativity ,especially the direct map

The the entire memory hierarchy design is done. The I-memory hierarchy is similar to that of load/store instruction (data). The VA is just the PC.

Testcase design:

I design the test case based on the previous one for the pipeline. The only difference is there is no memory misalignment .

```
 instruction(<#string a#>, <#int rs#>, <#int rt#>, <#int rd#>, <#int
shame#>, <#int function#>, <#int c_immediate#>)
    instruction("addi", 3, 4, 0, 0, 8, 10);
    instruction("addi", 3, 4, 0, 0, 8, 10);
    instruction("add", 3, 4, 5, 0, 32,0);
    instruction("add", 5, 8, 7, 0, 32, 0);
    instruction("BNQ", 3, 4, 0, 0, 0, -2);
```

```
    instruction("addi", 0, 8, 0, 0, 0, 0xfff0);
    instruction("bgtz", 8, 0, 0, 0, 0, 1);

    instruction("ori", 6, 7, 0, 0, 0, 0x4fab);
    instruction("nori", 2, 3, 0, 0, 0, 0xffab);
    instruction("and", 3, 10, 11, 0, 0x24, 0);
   instruction("sh", 0, 8, 0, 0, 0, 128);
    instruction("lw", 0, 15, 0, 0, 0, 128);
    instruction("sb", 0, 11, 0, 0, 0, 255);
    instruction("lw", 0,25, 0, 0, 0, 252);

    instruction("addi", 2, 14, 0, 0, 0, 0x0010);
    instruction("addi", 3, 17, 0, 0, 0, 0x0010);
    instruction("lw", 14, 17, 0, 0, 0, 0);
    instruction("addi", 11, 12, 0, 0, 0, 0x0050);
    instruction("addi", 11, 12, 0, 0, 0, 0x0050);
    instruction("addi", 11, 12, 0, 0, 0, 0x0050);
    instruction("addi", 12, 11, 0, 0, 0, 0x0abc);
    instruction("add", 12, 11, 25, 0, 32, 0);


    instruction("addi", 0, 18, 0, 0, 0, 0x0010);
    instruction("addi", 0, 19, 0, 0, 0, 0x0003);
    instruction("sw", 18, 19, 0, 0, 0, 0);
        instruction("lw", 18, 25, 0, 0, 0, 1);

    instruction("add", 0, 0, 26, 0, 32, 0);
    instruction("sw", 0, 26, 0, 0, 0, 0);
    instruction("lw", 0, 30, 0, 0, 0, 0);
    instruction("sub" ,7, 8, 16, 0, 0x22, 0);
    instruction("jal", 0, 0, 0, 0, 0, 48);
    instruction("jr", 31, 0, 0, 0, 0x08, 0);

    instruction("nop", 20, 0, 0, 0, 0, 0);
    instruction("ori", 0, 20, 0, 0, 0, 0xabcd);
    instruction("ori", 0, 21, 0, 0, 0, 0xabcd);
    instruction("beq", 20, 21, 0, 0, 0, 2);
    instruction("beq", 20, 21, 0, 0, 0, 1);

    instruction("addi", 0, 20, 0,0, 0, 0x2cde);
    instruction("sw", 0, 20, 0, 0, 0, 0);
    instruction("lb", 0, 10, 0, 0, 0, 0x0003);

    instruction("lw", 0, 27, 0, 0, 0, 0);
    instruction("sll", 0, 11, 6, 10, 0x00, 0);
    instruction("addi", 0, 28, 0, 0, 0, 0x00ff);
    instruction("bgtz", 28, 0, 0, 0, 0, 3);


    instruction("addi", 0, 23, 0, 0, 0, 0x9fab);
    instruction("addi", 22, 22, 0, 0, 0, 0x0001);
    instruction("bne", 23, 22, 0, 0, 0, -5);
    instruction("beq", 22, 23, 0, 0, 0, 3);
    instruction("nand", 15, 16, 17, 0, 0x28, 0);
    instruction("nor", 17, 3, 2, 0, 0x27, 0);
```

Besides in this testcase, I tried not to design a extreme case lest I should fail in my testcase. In that case, my testcase will viewed as invalid .

Last but not least, there is something I find in the project .

When the memory ,cache size increases , the hit rate is relatively increase.

That' s because the relative larger size memory element can store more without having discarding the least recently used data and replace it.

The program executes with more loop also cause higher hit rate.

This happens is due to the higher space locality . That is , whenever the program with higher locality will cause higher hit rate.

On the contrary , if a program with relative low locality will cause higher miss rate. So we may increase the hit rate by optimize the program to make it with high locality .

I' m of the opinion that the higher set associativity may increase the hit rate. It make the most of use of memory . As a result , it increase the hit rate. However, it also increase the hit time ,since it have to search each set to find the data we need. As the set associativity increase, it takes more time to search it . And also, it requires more supplicated and advanced hardware to implement it . So the choice among direct-mapped ,set-associative ,or fully associative mapping in any memory hierarchy will depend on the cost of a miss versus the cost of implementing associativity ,both in time and in extra hardare.

Reviews:

This is the final project for this coarse. It took me a huge amount time to finish it. As far as I am concerned, the most difficult part is comprehension of the memory hierarchy . After getting to know the architecture, the implementation is not so difficult ,compare with that of the pipeline project.

After these projects , it did make me understand more about the computer architecture although it exhaust me so much. Hope we learn more in the following days.