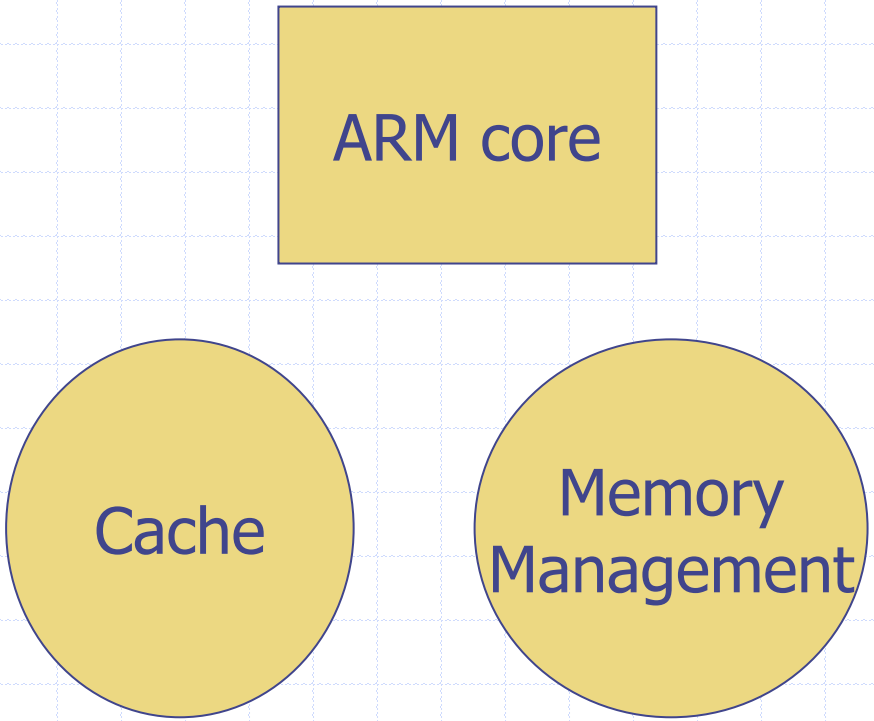
ARM Cache Memory

ARM Processor Organization



Cache Policy

- Write policy: WB or WT
- Cache line replacement policy
  - Round-robin/Pseudorandom
    - When victim counter reaches the max. vlaue, reset to the base value
  - LRU
- Allocation policy on a cache miss
  - Read-allocate
  - Read-write-allocate
- E.g., ARM920T: WB/WT, RR/PR, RA
  Xscale: WB/WT, RR, RA/RWA

RR vs. Pseudorandom Replacement

- RR
  - (+) Predictability improved
  - (-) large change in performance for a small change in memory access

EX: ARM 940T Cache

- 4KB I-Cache
- 4 Sets, each set with 1KB
- 16B line size
- 64-way set associative

```
void cache_RRtest(unsigned int times,unsigned int numset)
{
   clock_t count;
   printf("Round Robin test size = %d\r\n", numset);
   enableRoundRobin();
   cleanFlushCache();
   count = clock();
   readSet(times,numset);
   count = clock() - count;
   printf("Round Robin enabled = %.2f seconds\r\n",
                  (float)count/CLOCKS_PER_SEC);
   enableRandom();
   cleanFlushCache();
   count = clock();
   readSet(times, numset);
   count = clock() - count;
   printf("Random enabled = %.2f seconds\r\n\r\n",
                  (float)count/CLOCKS_PER_SEC);
}
```

```
int readSet( unsigned int times, unsigned int numset)
{
  int setcount, value;
  volatile int *newstart;
  volatile int *start = (int *)0x20000;
  __asm
  {
    timesloop:
      MOV     newstart, start
      MOV     setcount, numset
    setloop:
      LDR     value,[newstart,#0];
      ADD     newstart,newstart,#0x40;
      SUBS    setcount, setcount, #1;
      BNE     setloop;
      SUBS    times, times, #1;
      BNE     timesloop;
  }
  return value;
}
```

Cache_RRtest(0x10000, 64) vs.
Cache_RRtest(0x10000, 65)?

System Control Coprocessor 15

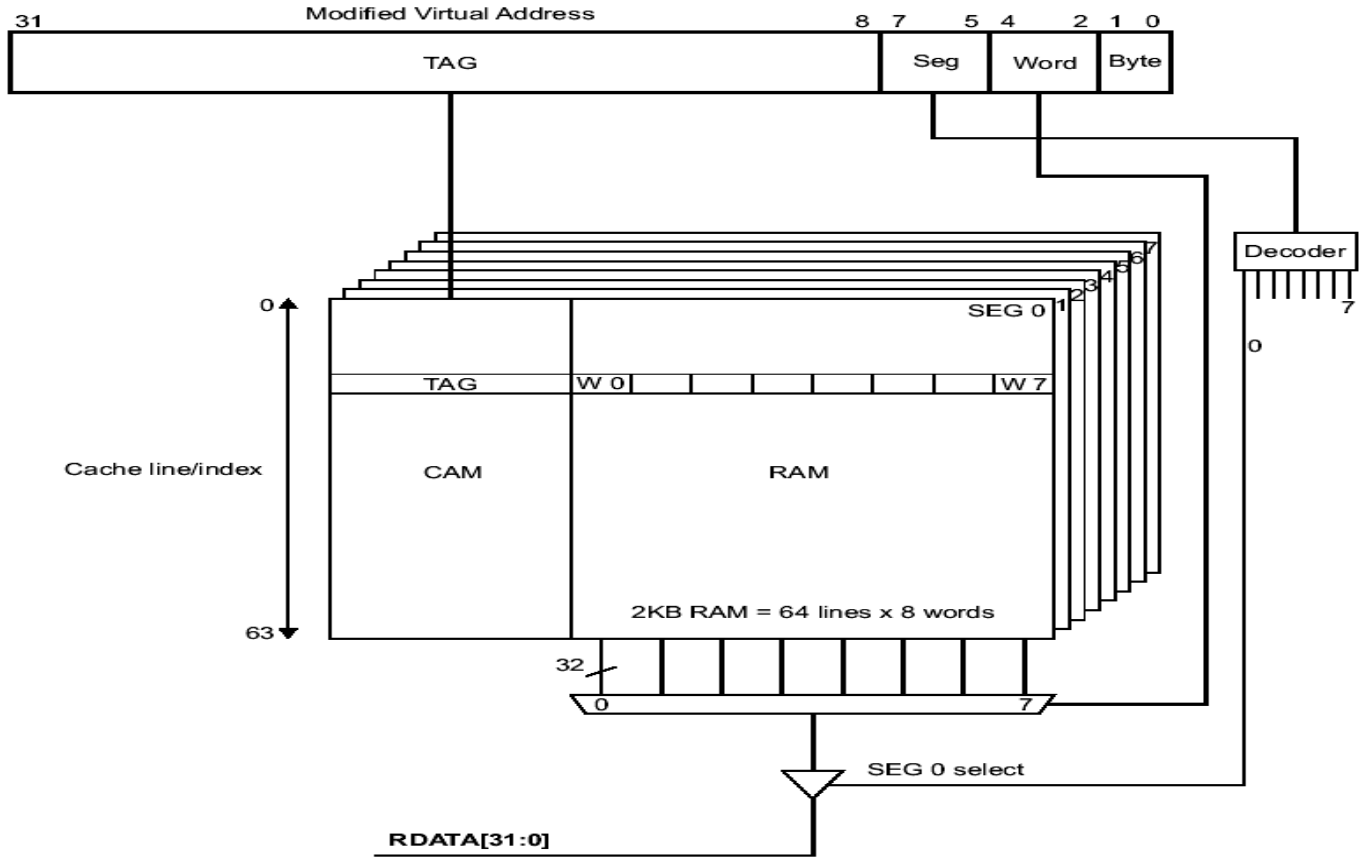
- Used to configure & control ARM cached cores
- Clean & Flush cache
  - Flush in ARM == Invalidate
    - Clear the valid bit in the cache line
  - Clean in ARM == Flush
    - Write dirty cache lines to memory
- Drain write buffer
- Cache lockdown

Cache Lockdown

- Improve the predictability but reduce the cache size
- Candidates for lockdown
  - Vector interrupt table
  - ISR
  - Performance critical code
  - Global variables (frequently used)

ARM 920T I-Cache Organization

- 16KB (512 lines * 32 bytes, arranged as a 64-way set associative cache)

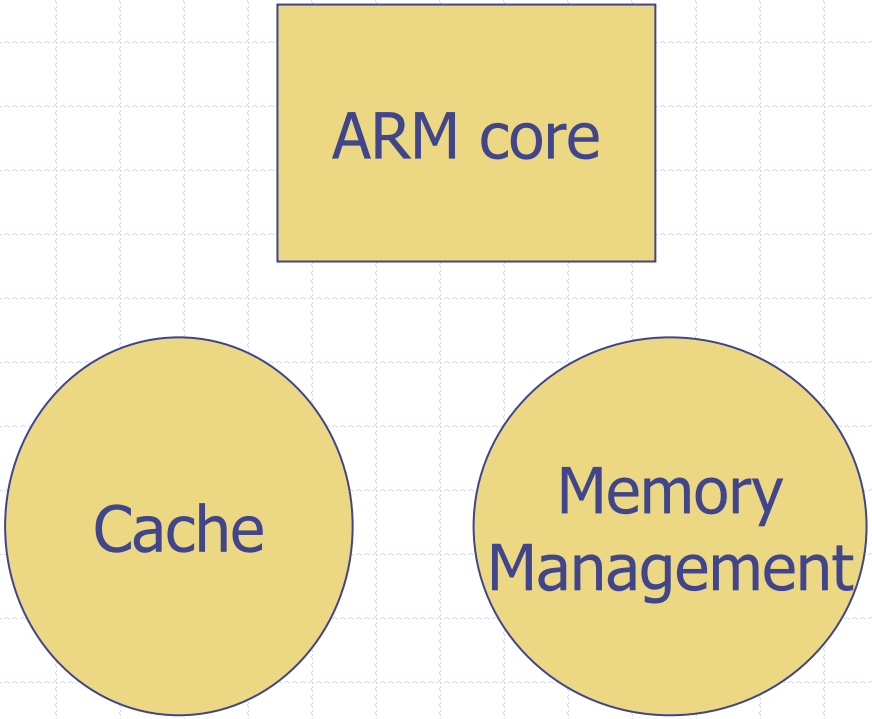


EX: I-Cache Locking

```
        ADRL    r0,start_address        ; address pointer
        ADRL    r1,end_address
        MOV     r2,#lockdown_base<<26   ; victim pointer
        MCR     p15,0,r2,c9,c0,1        ; write ICache victim and lockdown base

loop    MCR     p15,0,r0,c7,c13,1       ; Prefetch ICache line
        ADD     r0,r0,#32               ; increment address pointer to next ICache line

;; do we need to increment the victim pointer?
;; test for segment 0, and if so, increment the victim pointer
;; and write the ICache victim and lockdown base.
        AND     r3,r0,#0xE0             ; extract the segment bits from the address
        CMP     r3,#0x0                 ; test for segment 0
        ADDEQ   r2,r2,#0x1<<26          ; if segment 0, increment victim pointer
        MCREQ   p15,0,r2,c9,c0,1        ; and write ICache victim and lockdown base

;; have we linefilled enough code?
;; test for the address pointer being less than or equal to the
;; end_address and if so, loop and perform another linefill
        CMP     r0,r1                   ; test for less than or equal to end_address
        BLE     loop                    ; if not, loop

;; have we exited with r3 pointing to segment 0?
;; if so, the ICache victim and lockdown base has already been set to one
;; higher than the last entry written.
;; if not, increment the victim pointer and write the ICache victim and
;; lockdown base.
        CMP     r3,#0x0                 ; test for segments 1 to 7
        ADDNE   r2,r2,#0x1<<26          ; if address is segment 1 to 7,
        MCRNE   p15,0,r2,c9,c0,1        ; write ICache victim and lockdown base
```

Efficient Programming for ARM

ARM Processor Organization



Optimization Targets

- Execution Time
  - Cache
  - Instruction Selection
  - Instruction Scheduling
  - Register Allocation
- Code Size
- Power Consumption

Efficient C Programming

Efficient ASM Programming

Outline

- Part 1: Efficient C Programming
- Part 2: Memory Hierarchy Optimization
- Part 3: ASM Optimization

Part 1: Efficient C Programming

P1.1: Avoid `char` or `short` type

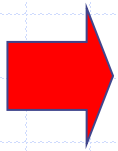
```
char i;
for (i=0; i<64; i++)
  sum += data[i]
```

```
AND r1, r1, #0xff necessary
```

- Same for function arguments or return values
  Avoid unnecessary casts

Type Example

```
int wordinc(int a)
{
    return a+1;
}

short shortinc(short a)
{
    return a+1;
}

char charinc(char a)
{
    return a+1;
}
```



```
wordinc
    add     a1, a1, #1
    mov     pc, lr

shortinc
    add     a1, a1, #1
    mov     a1, a1, lsl #16
    mov     a1, a1, asr #16
    mov     pc, lr

charinc
    add     a1, a1, #1
    and     a1, a1, #255
    mov     pc, lr
```

P1.2: Countdown to Zero

- No need to hold the termination value
- Comparison with 0 is free

```
for (i=0; i<64; i++)
     sum += *(data++);
```

```
LDR r3, [r2], #4; r3 = *(data++)
ADD r1, r1, #1 ;i++
CMP r1, #0x40
ADD r0, r3, r0 ;sum += r3
BCC forLoop
```

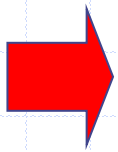
```
for (i=64; i!=0; i--)
     sum += *(data++);

    LDR r3, [r2], #4; r3 = *(data++)
    SUBS r1, r1, #1 ;i-- & set flags
    ADD r0, r3, r0 ;sum += r3
    BNE forLoop
```

P1.3: Loop Efficiency

- Use do-while instead of for if the loop iterates at least once
- Unroll important loops to reduce the loop overhead

Unrolling Example

```
int countbit1(unsigned int n)
{
   int bits=0;
   while(n!=0)
   {
      if (n&1) bits++;
      n>>=1;
   }
   return bits;
}
```



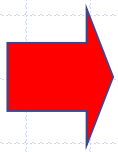
```
int countbit2(unsigned int n)
{
   int bits=0;
   while(n!=0)
   {
      if (n&1) bits++;
      if (n&2) bits++;
      if (n&4) bits++;
      if (n&8) bits++;
      n>>=4;
   }
   return bits;
}
```

P1.4: Register Allocation & Function Calls

- Limit # of local variables to 12
- Get familiarize with ATPCS (ARM-THUMB Procedure Call Standard)
  - First four registers used for arguments
    - Restrict to four arguments!

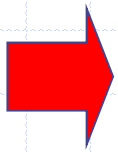
Number of Function Arguments

```
int f1(int a, int b, int c, int d)
{
    return a+b+c+d;
}

int g1()
{
    return f1(1,2,3,4);
}
```



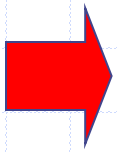
```
f1
   add      r0,r0,r1
   add      r0,r0,r2
   add      r0,r0,r3
   mov      pc,r14

g1
   mov      r3,#4
   mov      r2,#3
   mov      r1,#2
   mov      r0,#1
   b        f1
```

```
int f2(int a, int b, int c,
                int d, int e, int f)
{

    return a+b+c+d+e+f;

}
```



```
f2
    str     r14,[r13,#-4]!
    ldmib   r13,{r12,r14}
    add     r0,r0,r1
    add     r0,r0,r2
    add     r0,r0,r3
    add     r0,r0,r12
    add     r0,r0,r14
    ldr     pc,[r13],#4
```

```
int g2()
{

    return f2(1,2,3,4,5,6);

}
```



```
g2
    str     r14,[r13,#-4]!
    sub     r13,r13,#8
    mov     r3,#6
    mov     r2,#5
    stmia   r13,{r2,r3}
    mov     r3,#4
    mov     r2,#3
    mov     r1,#2
    mov     r0,#1
    bl      f2
    add     r13,r13,#8
    ldr     pc,[r13],#4
```

P1.5: Register Allocation

```
void f(int *a);
void g(int a);

int test1(int i)
{
   f(&i);
   i+=g(i);
   i+=g(i);
   return i;
}
```

Cannot allocate to Reg

```
void f(int *a);
void g(int a);

int test2(int i)
{
   int dummy=i;
   f(&dummy);
   i=dummy;
   i+=g(i);
   i+=g(i);
   return i;
}
```

Can allocate to Reg

```
int f(void);
int g(void);

int errs;

void test1(void)
{
   errs+=f();
   errs+=g();
}
```

```
int f(void);
int g(void);

int errs;

void test2(void)
{
   int localerrs=errs;
   localerrs+=f();
   localerrs+=g();
   errs=localerrs;
}
```

P1.6 Avoid Division

- No divide instruction in H/W

  Software implementations are EXPENSIVE!

Example: Circular Buffer

```
offset = (offset + increment) % size;
```

vs.

```
offset += increment;
if (offset >= size)  offset -= size;
```

Part 2: Memory Hierarchy Optimization

Cache-Aware Optimizations

Key: Cache-Aware Locality Optimizations

P2.1: Data Prefetching (ARM v5E)

- Prefetch load instruction (`pld`)
  - Advance load of a cache line (e.g., 32 bytes) to D-cache
  - Overlapped execution of `pld` with another instruction
    - Latency hiding
  - For a null pointer, no fault
    - Useful for recursive data structures

Data Prefetching Example 1

```
struct {
    long a;
    long b;
    long c;
    long d;
} data[100];
```

```
for (i=0; i<100; i++){
    PREFETCH(data[i+1]);
    data[i].a = data[i].b + data[i].c + data[i].d;
    ……
}
```

Data Prefetching Example 2

```
while (p) {
    do_something (p    data);
    p = p    next;
}
```



```
while (p) {
    PREFETCH (p    next);
    do_something (p    data);
    p = p    next;
}
```

```
preorder (treeNode *t) {
    if(t) {
        process(t    data);
        preorder(t    left);
        preorder(t    right);
    }
}
```



```
preorder (treeNode *t) {
    if(t) {
        PREFETCH (t    right);
        PREFETCH (t    left);
        process(t    data);
        preorder(t    left);
        preorder(t    right);
    }
}
```

P2.2: Merging Arrays

```
int a[SIZE];
int b[SIZE];
```



```
struct merge {
    int a;
    int b;
};
struct merge merged_array[SIZE];
```

P2.2: Merging Arrays

```
int a[SIZE];
int b[SIZE];
```

```
struct merge {
    int a;
    int b;
};
struct merge merged_array[SIZE];
```

```
#define ARRAY_SIZE (256 * 1024)
int A[ARRAY_SIZE];
int B[ARRAY_SIZE];
int C[ARRAY_SIZE];

for (i=0; i<ARRAY_SIZE; i++)
    A[i] = B[i] + C[i];
```

Direct-mapped 64KB D$
with 16-byte cache blocks

Using these data structures…

```
struct record {
   struct record *left;
   struct record *right;
   char key[KEYSIZE];
   /* ... many bytes, perhaps KB, of  record data ... */
} *rootptr = NULL;
```

```
 struct recData {
     /* ... many bytes, perhaps KB, of record data ... */
 };

 struct recIndex {
        struct recIndex *left;
        struct recIndex *right;
        char key[KEYSIZE];
        struct recData *datap;
 } *rootptr = NULL
```

Running this Program …

```
struct record *findrec(char *searchKey)
{
        for (struct record *rp = rootptr; rp != NULL; ) {
                int result = strncmp(rp->key, searchKey, KEYSIZE);
        if (result == 0) {
                return rp;
        } else if (result < 0) {
                rp = rp->left;
        } else { // result > 0
                rp = rp->right;
                }
        }

        /* if we fall out of loop, it's not in the tree */
        return (struct record *)NULL;
}
```

P2.3: Loop Interchange

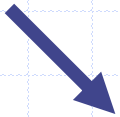
```
for (j = 0; j < 100; j = j+1)
        for (i = 0; i < 5000; i = i+1)
                x[i][j] = 2 * x[i][j];
```



```
for (i = 0; i < 5000; i = i+1)
        for (j = 0; j < 100; j = j+1)
                x[i][j] = 2 * x[i][j];
```

P2.4: Loop Fusion

```
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];
```



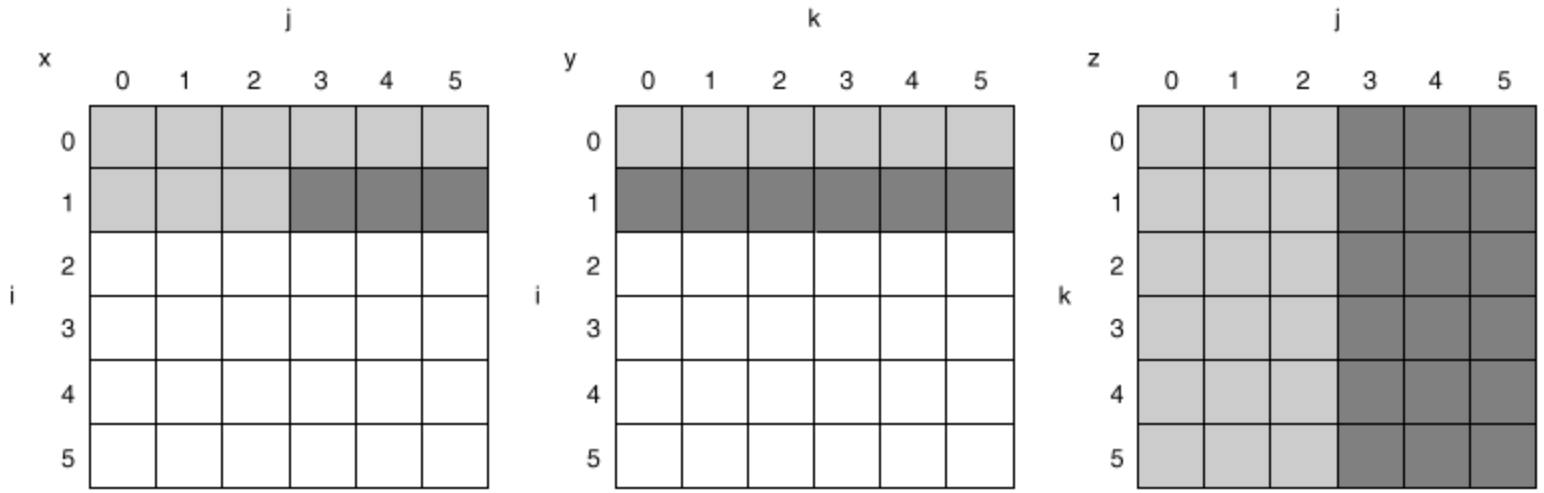
```
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
    {       a[i][j] = 1/b[i][j] * c[i][j];
            d[i][j] = a[i][j] + c[i][j];}
```

P2.5: Blocking (Tiling)

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1){
      r = 0;
      for (k = 0; k < N; k = k+1){
        r = r + y[i][k]*z[k][j];
          };
      x[i][j] = r;
  };
```

Blocking (Tiling)

Before blocking



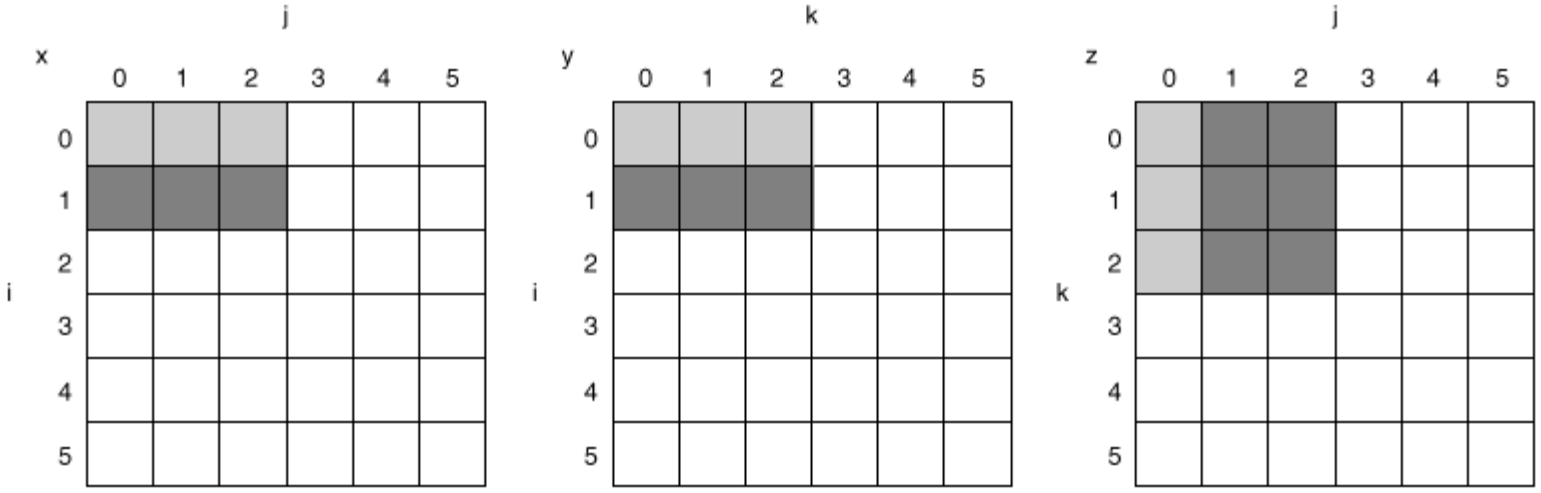
x[ ] y[ ] z[ ]

Blocking (Tiling)

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
    for (j = jj; j < min(jj+B-1,N); j = j+1) {
        r = 0;
        for (k = kk; k < min(kk+B-1,N); k = k+1) {
            r = r + y[i][k]*z[k][j];
        };
        x[i][j] = x[i][j] + r;
    };
```

Blocking (Tiling)

After blocking

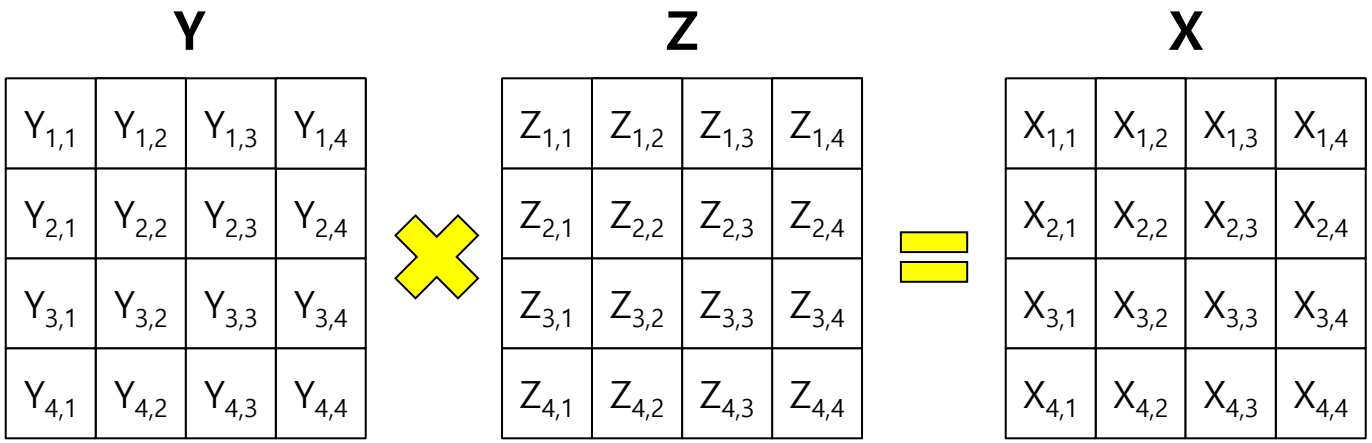


x[ ] y[ ] z[ ]

Matrix Multiplication: Example

Environments

- 4x4 matrix multiplication
- 16-entry fully-associative cache

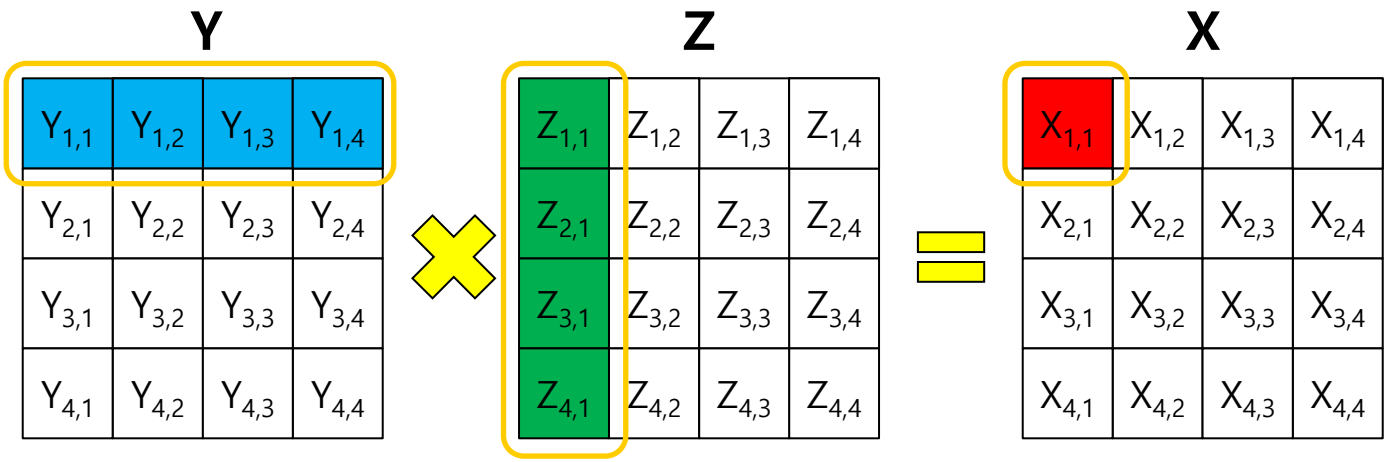


Cache (16 entries)

Matrix Multiplication: Example

Step 1

- $X_{1,1} = (Y_{1,1} \times Z_{1,1}) + (Y_{1,2} \times Z_{2,1}) + (Y_{1,3} \times Z_{3,1}) + (Y_{1,4} \times Z_{4,1})$

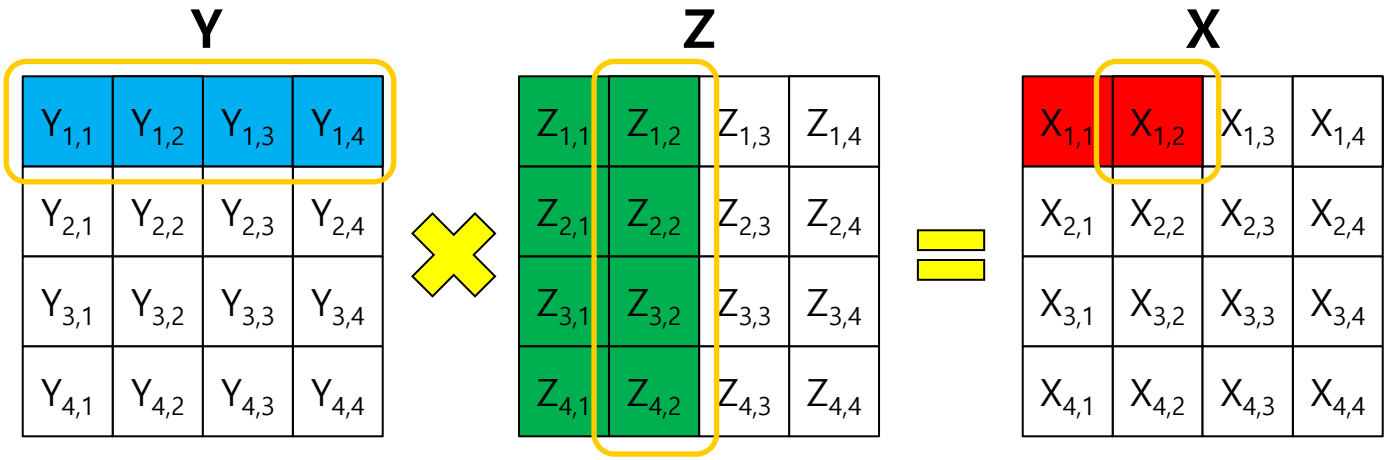


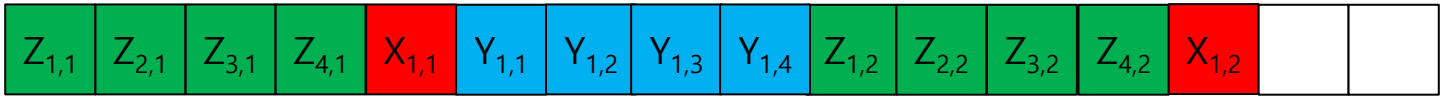
Cache (16 entries/LRU ordered)

Matrix Multiplication: Example

Step 2

- $X_{1,2} = (Y_{1,1} \times Z_{1,2}) + (Y_{1,2} \times Z_{2,2}) + (Y_{1,3} \times Z_{3,2}) + (Y_{1,4} \times Z_{4,2})$



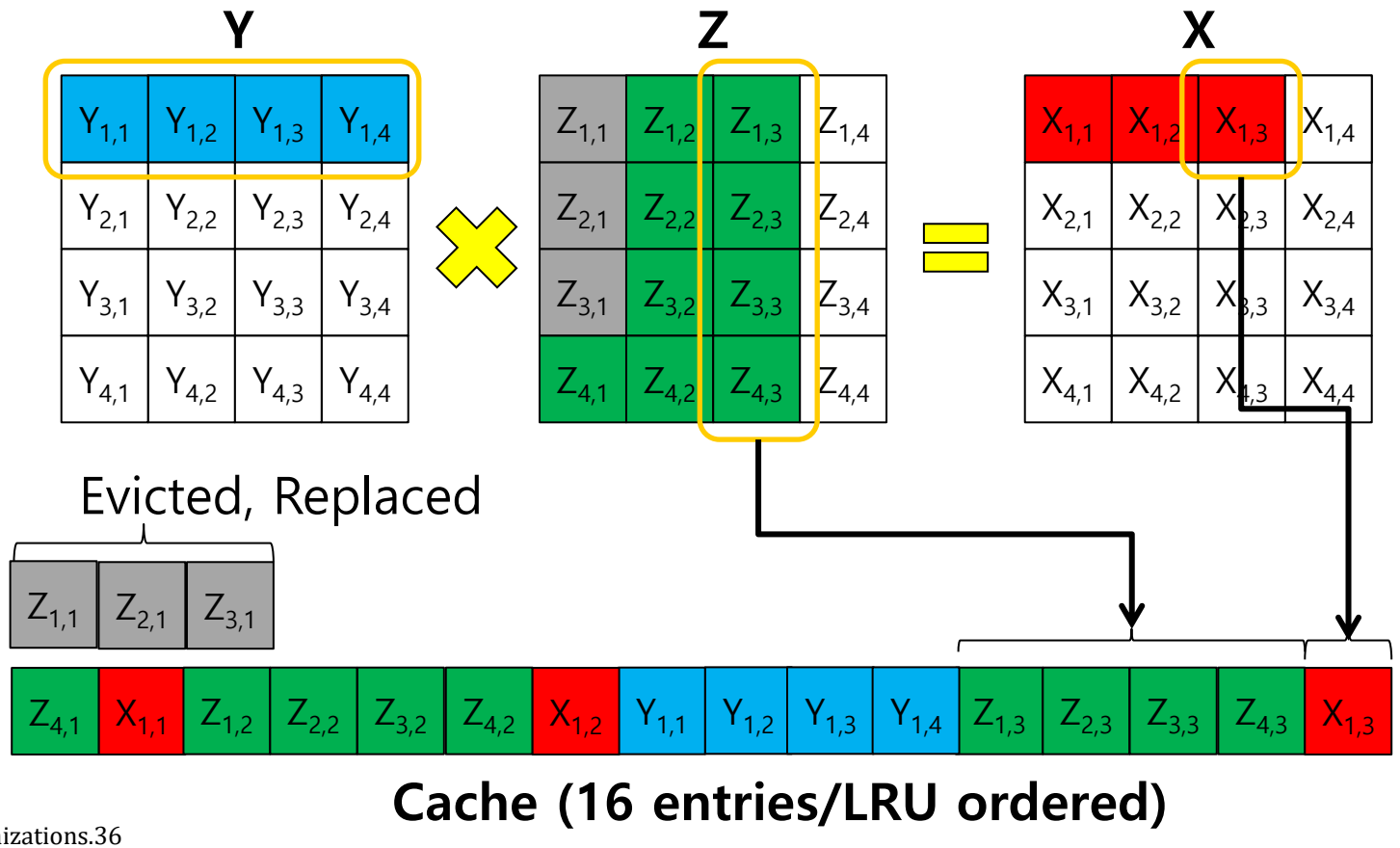


Cache (16 entries/LRU ordered)

Matrix Multiplication: Example

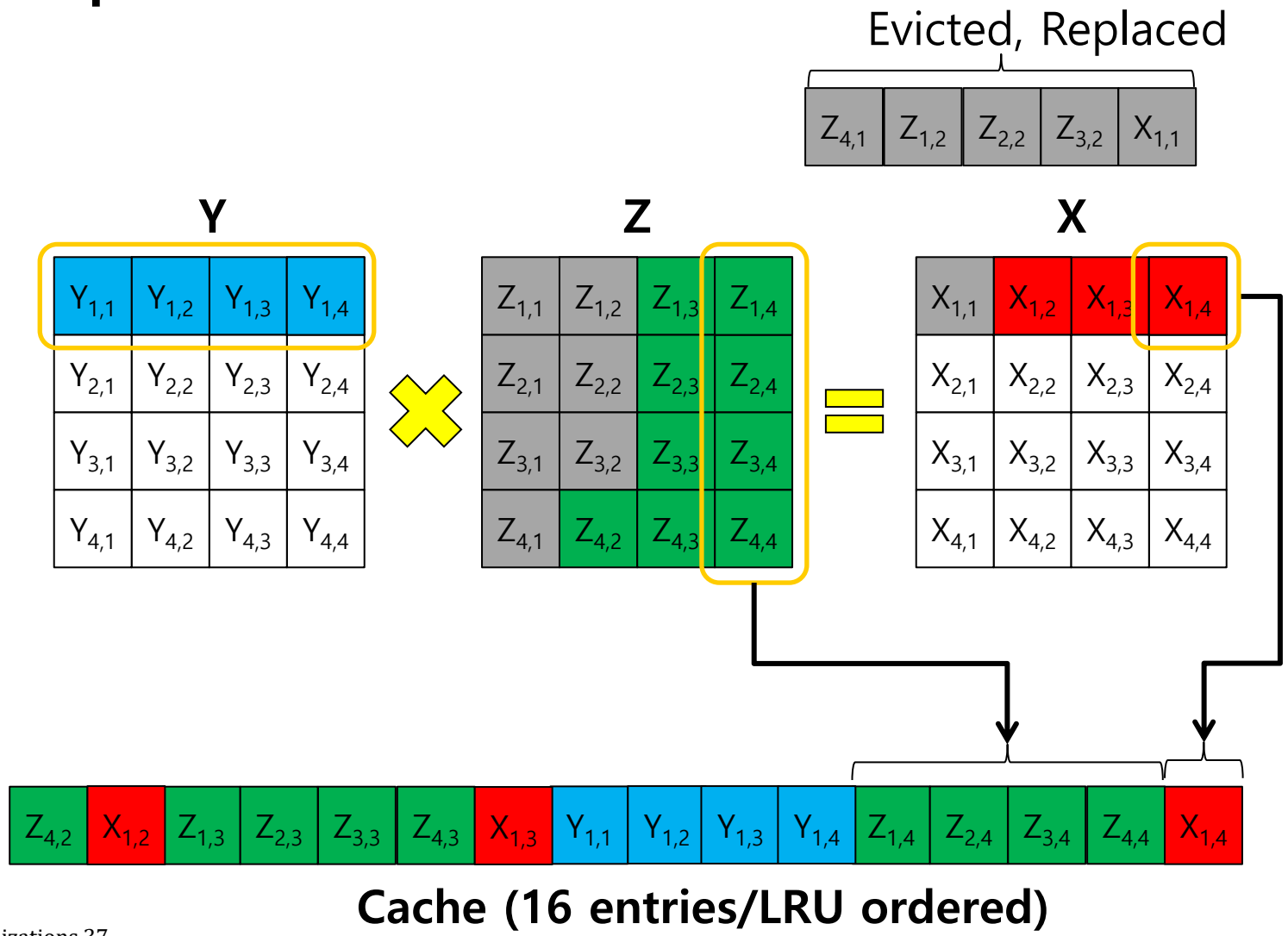
Step 3

- $X_{1,3} = (Y_{1,1} \times Z_{1,3}) + (Y_{1,2} \times Z_{2,3}) + (Y_{1,3} \times Z_{3,3}) + (Y_{1,4} \times Z_{4,3})$



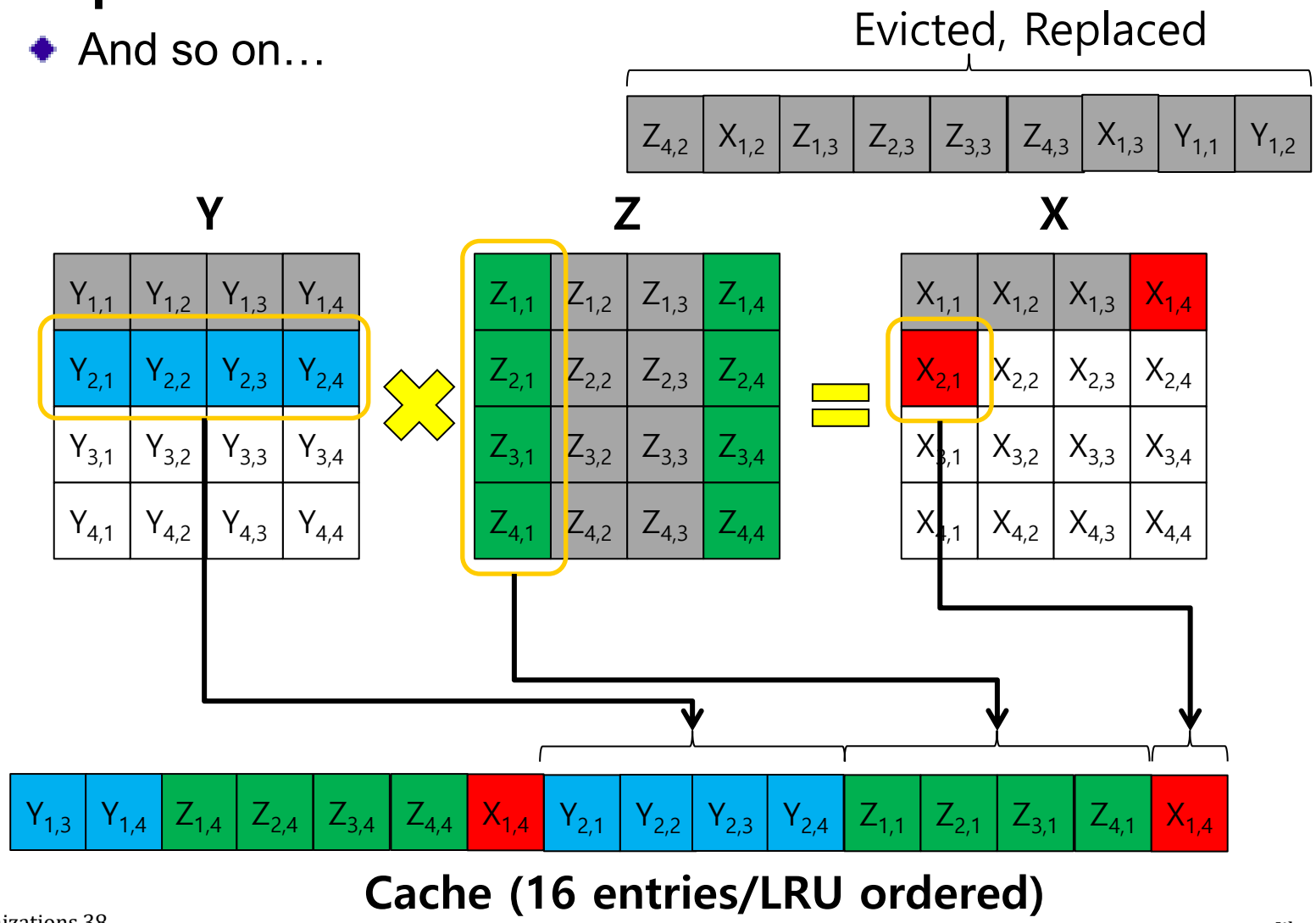
Matrix Multiplication: Example

- Step 4



Matrix Multiplication: Example

- **Step 5**
  - And so on…



Matrix Multiplication에서의 Cache Miss

문제점

X의 한 원소를 계산할 때마다 Z의 한 column을 I/O를 통해서 메모리로 가져오고 cache에 올려야 함

곧 다시 쓰이게 될 Z의 한 column을 cache에서 내려야 할 가능성이 큼

비효율적인 cache 운영으로 성능이 저하됨

해결 방안

Blocking

Blocking 을 통한 **Cache Miss** 감소

Blocking

각 matrix를 다수의 block으로 나누고, 각 block에 대해서 multiplication을 수행함

Matrix 전체가 아닌 일부이므로, 반복해서 matrix multiplication에 쓰일 원소를 ⇒ 모두 cache에 올려두는 것이 가능함

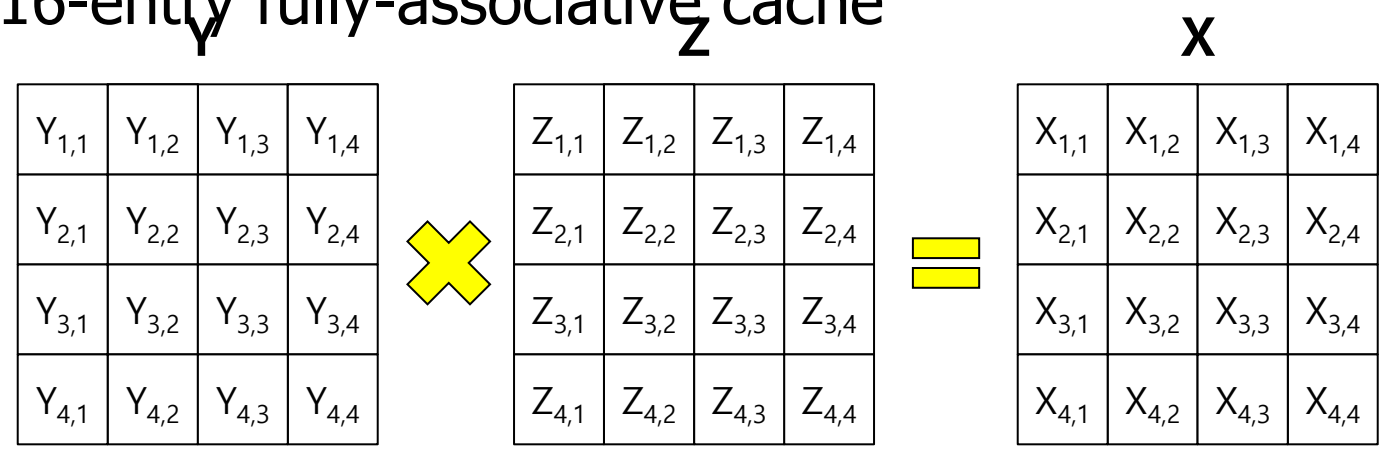
Cache miss가 감소하게 됨 ⇒ 성능 향상

Blocking 을 통한 Cache Miss 감소

Environments

4x4 matrix multiplication, **2x2 sub-block**

16-entry fully-associative cache

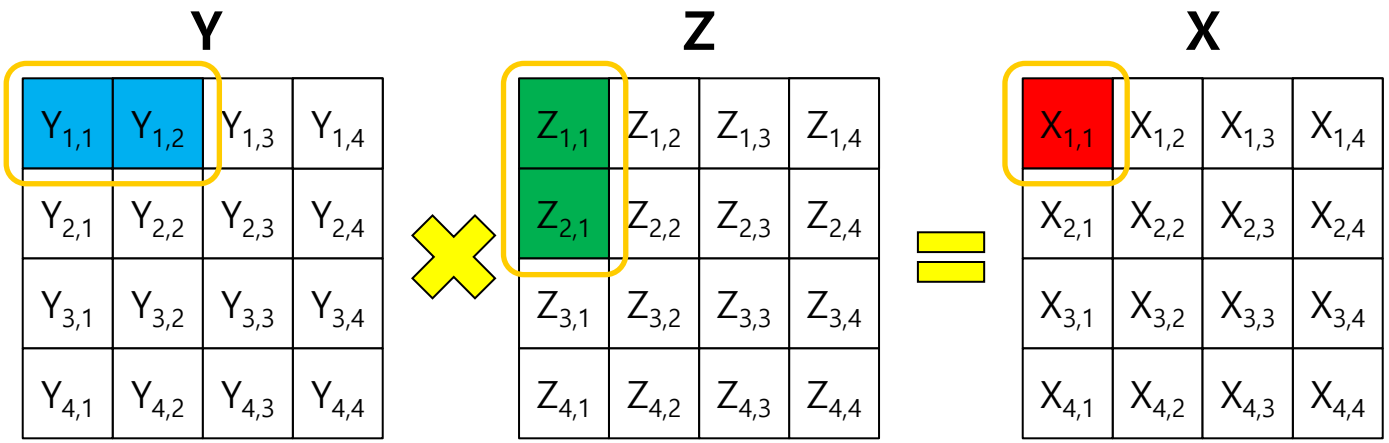


Cache (16 entries)

Blocking 을 통한 Cache Miss 감소

Step 1

- $X_{1,1} = (Y_{1,1} \times Z_{1,1}) + (Y_{1,2} \times Z_{2,1})$
  - Do the rest part later: $(Y_{1,3} \times Z_{3,1}) + (Y_{1,4} \times Z_{4,1})$

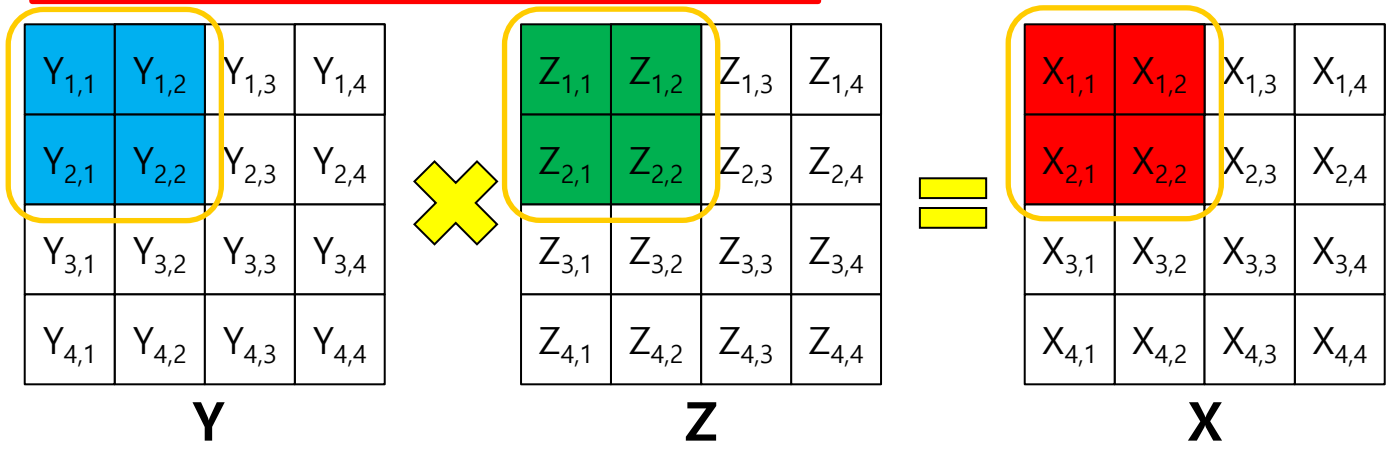


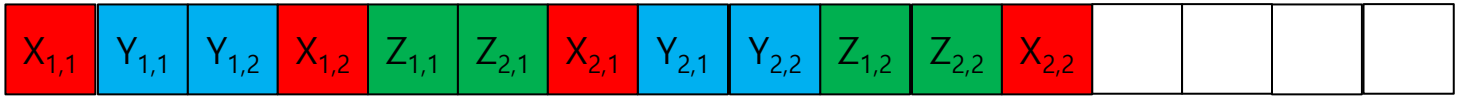
Cache (16 entries/LRU ordered)

Blocking 을 통한 Cache Miss 감소

Step 2 ~ 4

- $X_{1,2} = (Y_{1,1} \times Z_{1,2}) + (Y_{1,2} \times Z_{2,2}) + (Y_{1,3} \times Z_{3,2}) + (Y_{1,4} \times Z_{4,2})$
- $X_{2,1} = (Y_{2,1} \times Z_{1,1}) + (Y_{2,2} \times Z_{2,1}) + (Y_{2,3} \times Z_{3,1}) + (Y_{2,4} \times Z_{4,1})$
- $X_{2,2} = (Y_{2,1} \times Z_{1,2}) + (Y_{2,2} \times Z_{2,2}) + (Y_{2,3} \times Z_{3,2}) + (Y_{2,4} \times Z_{4,2})$



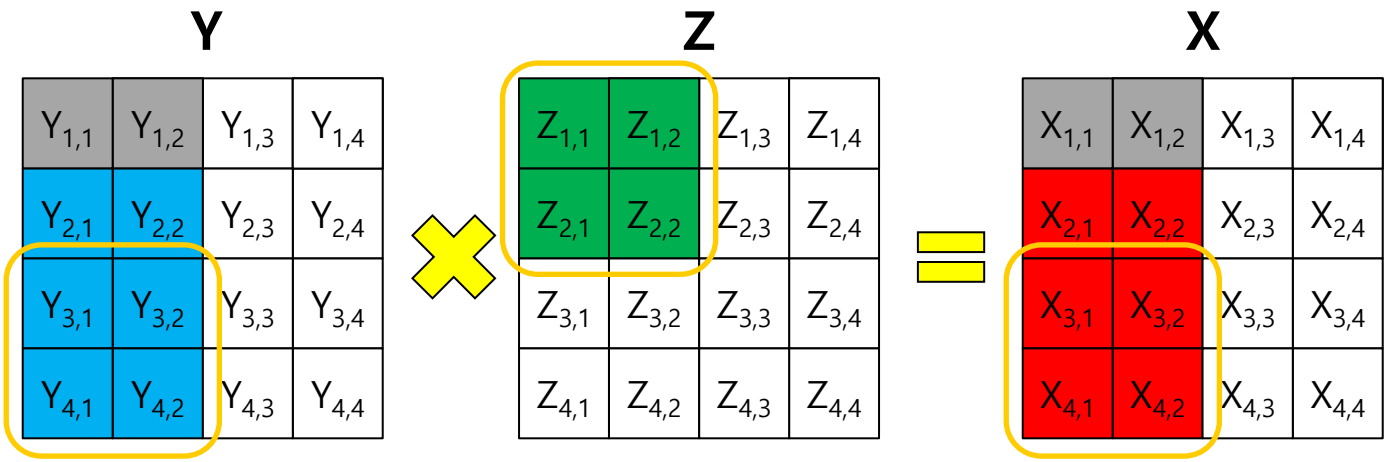


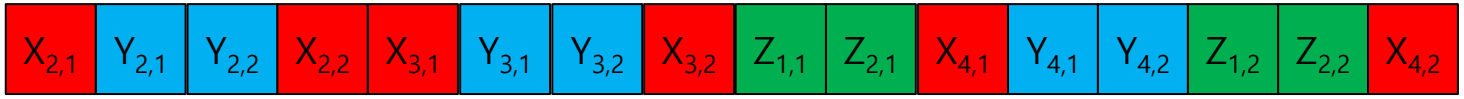
Cache (16 entries/LRU ordered)

Blocking 을 통한 Cache Miss 감소

Step 5 ~ 8

Do the former half of $X_{3,1}$, $X_{3,2}$, $X_{4,1}$, $X_{4,2}$



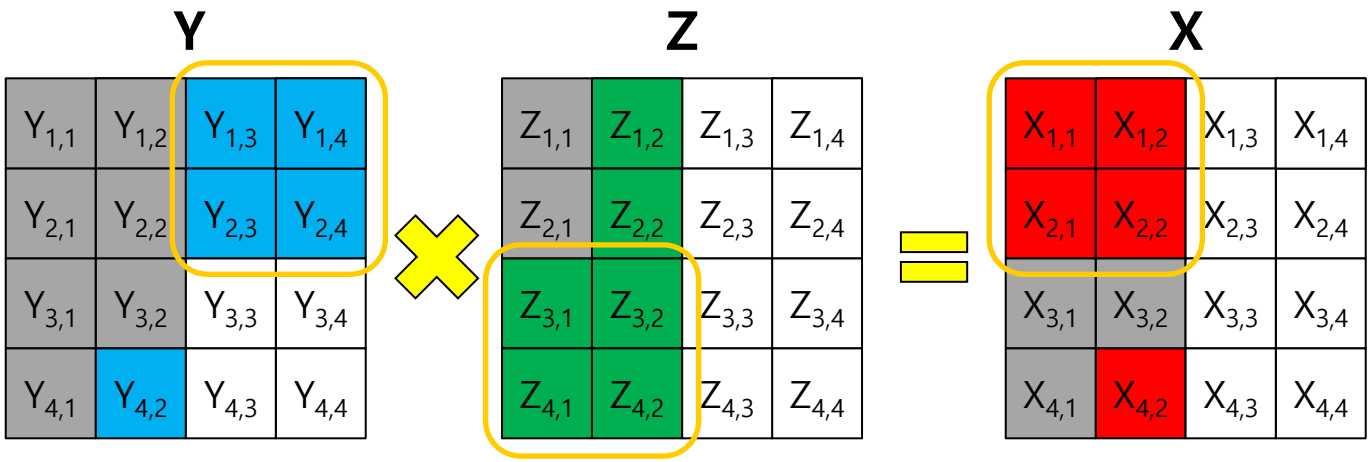


Cache (16 entries/LRU ordered)

Blocking 을 통한 Cache Miss 감소

Step 9 ~ 12

Complete $X_{1,1}$, $X_{1,2}$, $X_{2,1}$, $X_{2,2}$

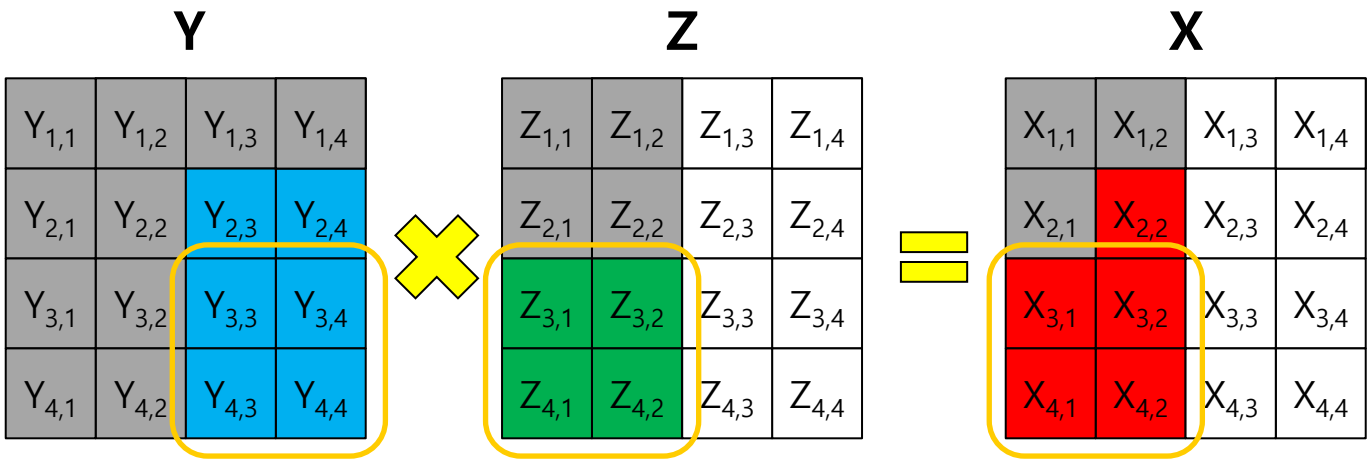


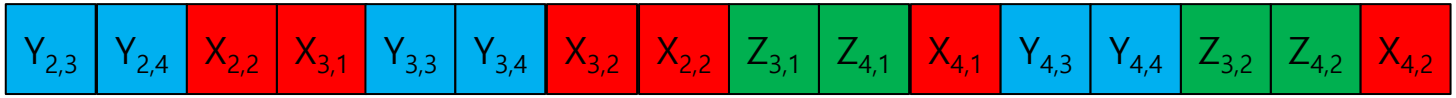
Cache (16 entries/LRU ordered)

Blocking 을 통한 **Cache Miss** 감소

Step 13 ~ 16

Complete $X_{3,1}$, $X_{3,2}$, $X_{4,1}$, $X_{4,2}$



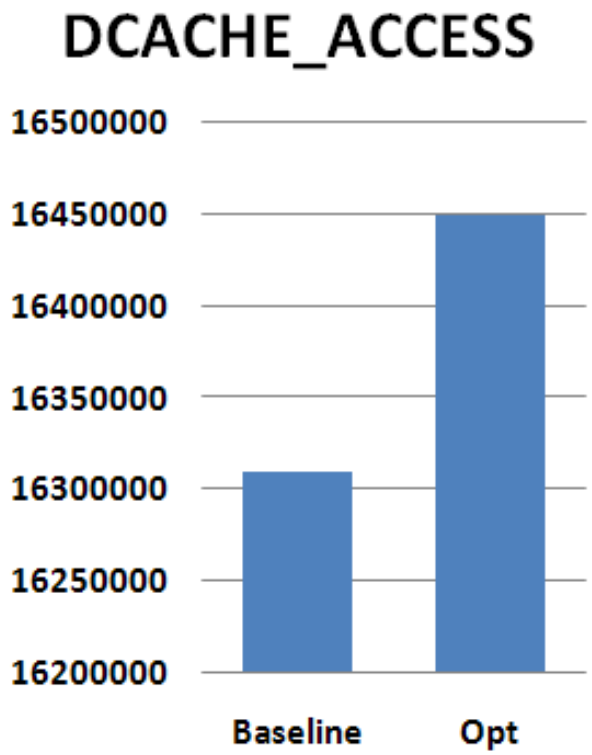


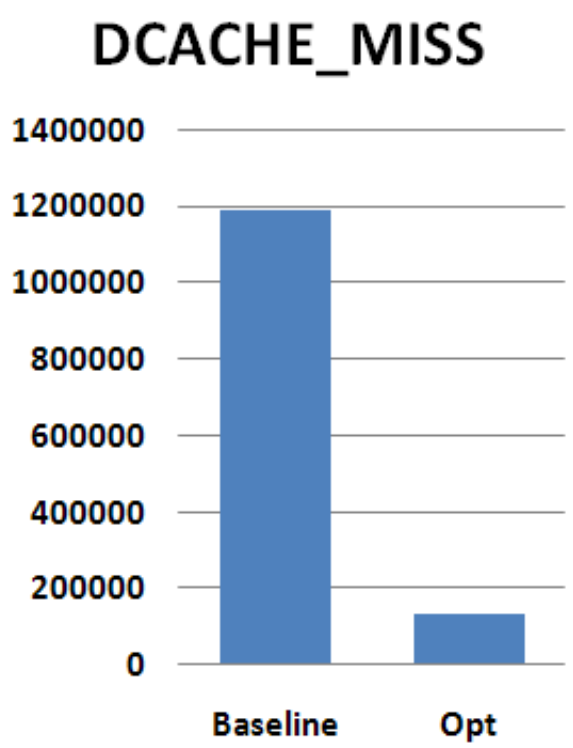
Cache (16 entries/LRU ordered)

성능 평가 **(cont.)**

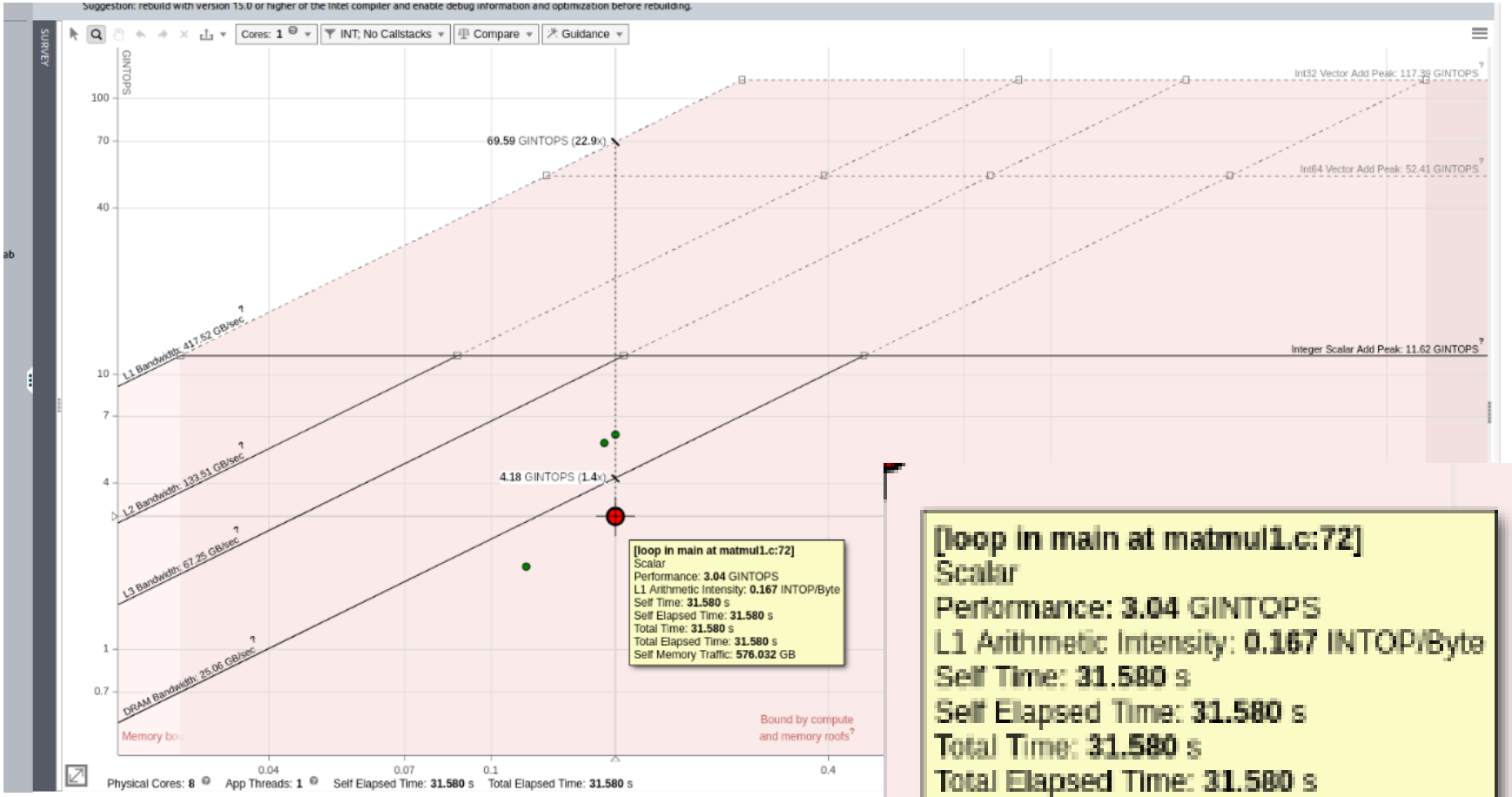
Data cache access는 거의 동일한 반면, data cache miss 수가 현격하게 감소함

Blocking 효과에 의한 data cache miss의 감소를 확인할 수 있음



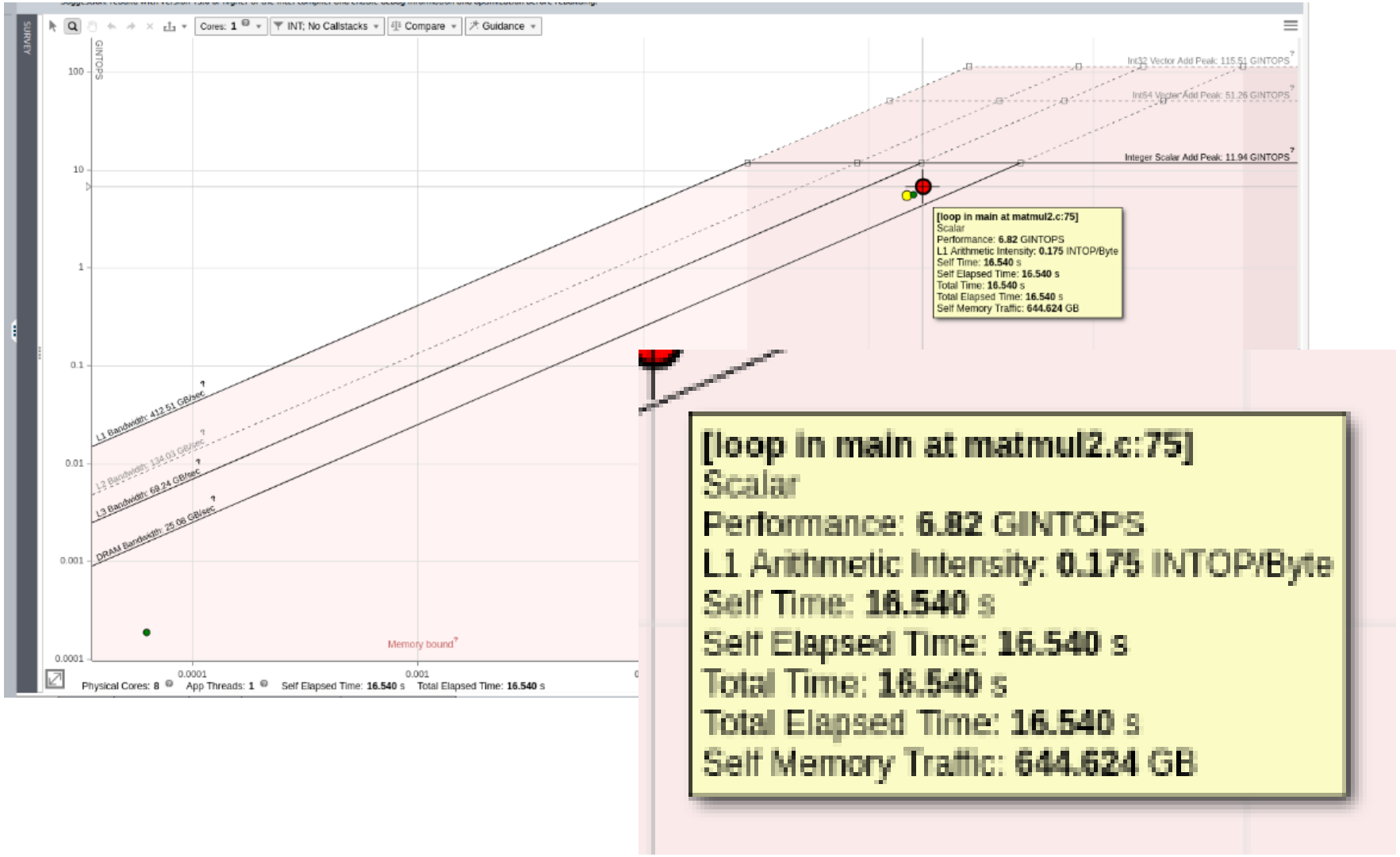


Roofline (Before Blocking)



[loop in main at matmul1.c:72]
Scalar
Performance: 3.04 GINTOPS
L1 Arithmetic Intensity: 0.167 INTOP/Byte
Self Time: 31.580 s
Self Elapsed Time: 31.580 s
Total Time: 31.580 s
Total Elapsed Time: 31.580 s
Self Memory Traffic: 576.032 GB

Roofline (After Blocking)



[loop in main at matmul2.c:75]
Scalar
Performance: 6.82 GINTOPS
L1 Arithmetic Intensity: 0.175 INTOP/Byte
Self Time: 16.540 s
Self Elapsed Time: 16.540 s
Total Time: 16.540 s
Total Elapsed Time: 16.540 s
Self Memory Traffic: 644.624 GB

Case Study: Huffman Decoding

Huffman Coding 소개

- 코드 압축의 종류
  - 무손실 압축
    - 데이터의 통계적 특징을 이용
    - 디지털 신호가 가지고 있는 중복성(redundancy)를 효율적으로 제거하여 손실 없이 데이터 양을 줄임
    - 대표적인 예: Huffman coding, ZIP파일
  - 손실 압축
    - 인간의 청각 또는 시각 특성을 이용
    - 불필요하거나 중요도가 낮은 정보를 삭제하여 데이터의 양을 줄이는 방법
    - 시청각 자료에 대해 높은 압축 효율
    - 대표적인 예: JPEG, MP3

Huffman Coding 소개 (cont.)

- Huffman Coding
  - 무손실 압축 방법
  - 통계적 특성을 이용한 부호화 방식
    - 매우 빈번하게 발생하는 문자는 적은 수의 bit를 할당
    - 드물게 발생하는 문자에 대해서는 많은 bit들을 할당
      - 가변 길이 부호화 방법
  - 10진수를 전송함에 있어 숫자 당 고정된 bit를 할당하는 경우
    - 원래 4개의 bit가 필요하지만 빈번하게 발생하는 숫자에 4개의 bit보다 짧은 bit를 할당하고 드물게 나오는 숫자에 더 많은 bit를 할당
    - 평균 codeword에 대한 bit 수는 4보다 짧아짐

Huffman Coding 소개 (cont.)

Huffman Coding 방법

압축하고자 하는 신호들의 발생 빈도를 검사

발생 횟수를 이용하여 각 신호에 최적화된 코드를 부여

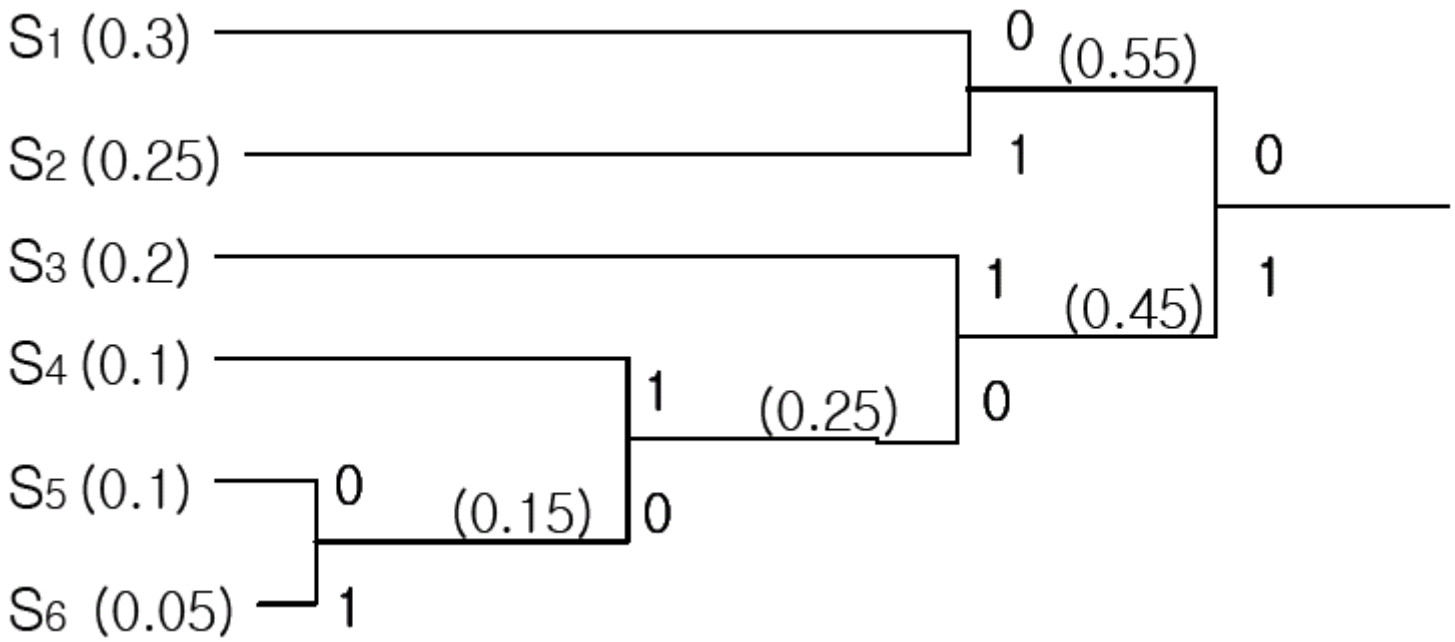
여섯 개의 문자 $S_1, S_2, S_3, S_4, S_5, S_6$ 에 대한 각각의 발생 빈도

심볼	S_1	S_2	S_3	S_4	S_5	S_6
확률	0.3	0.25	0.2	0.1	0.1	0.05

문자의 확률 순서를 정함

최소 확률 문자 두 개를 다음 단계에서 단일 기호로 결합

Huffman Coding 소개 (cont.)



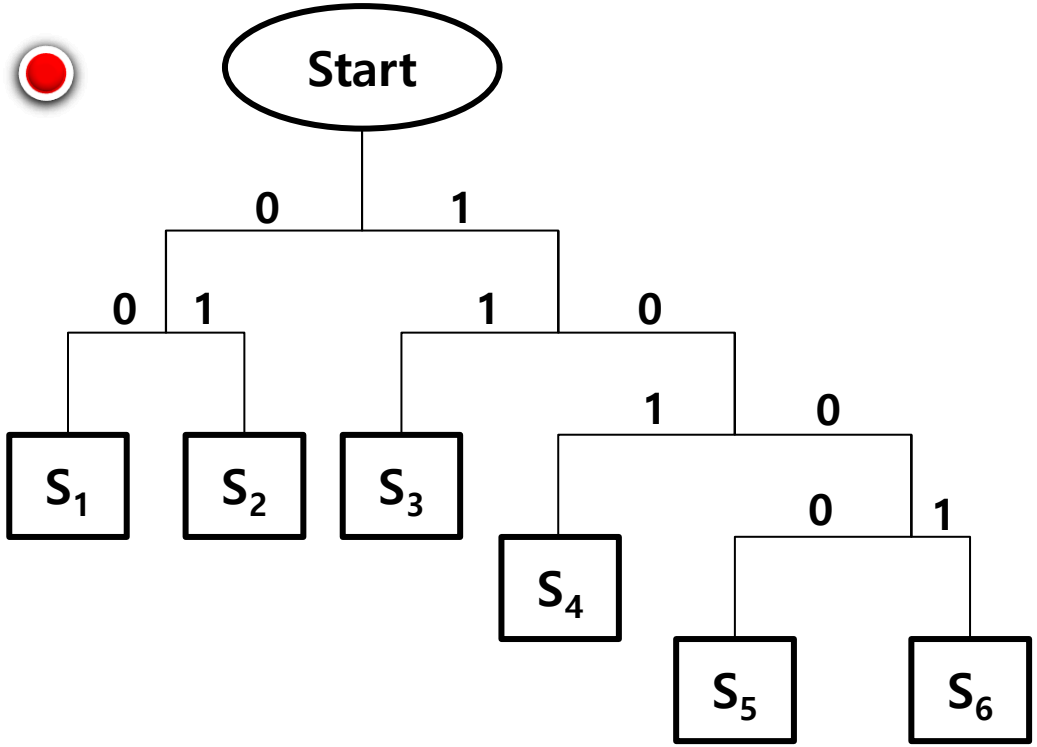
- 최종 Code

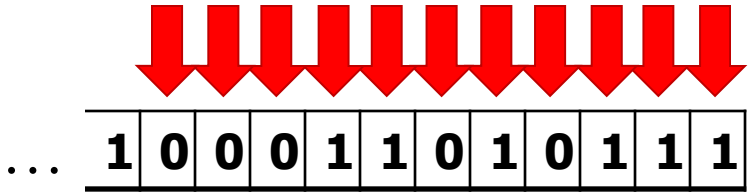
$$S_1 \quad 00, \quad S_2 \quad 01, \quad S_3 \quad 11,$$
$$S_4 \quad 101, \quad S_5 \quad 1000, \quad S_6 \quad 1001$$

Huffman Decoder

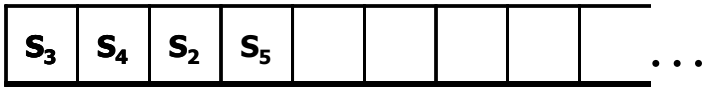
- **Simple 1-bit decoder**

S_1	00
S_2	01
S_3	11
S_4	101
S_5	1000
S_6	1001





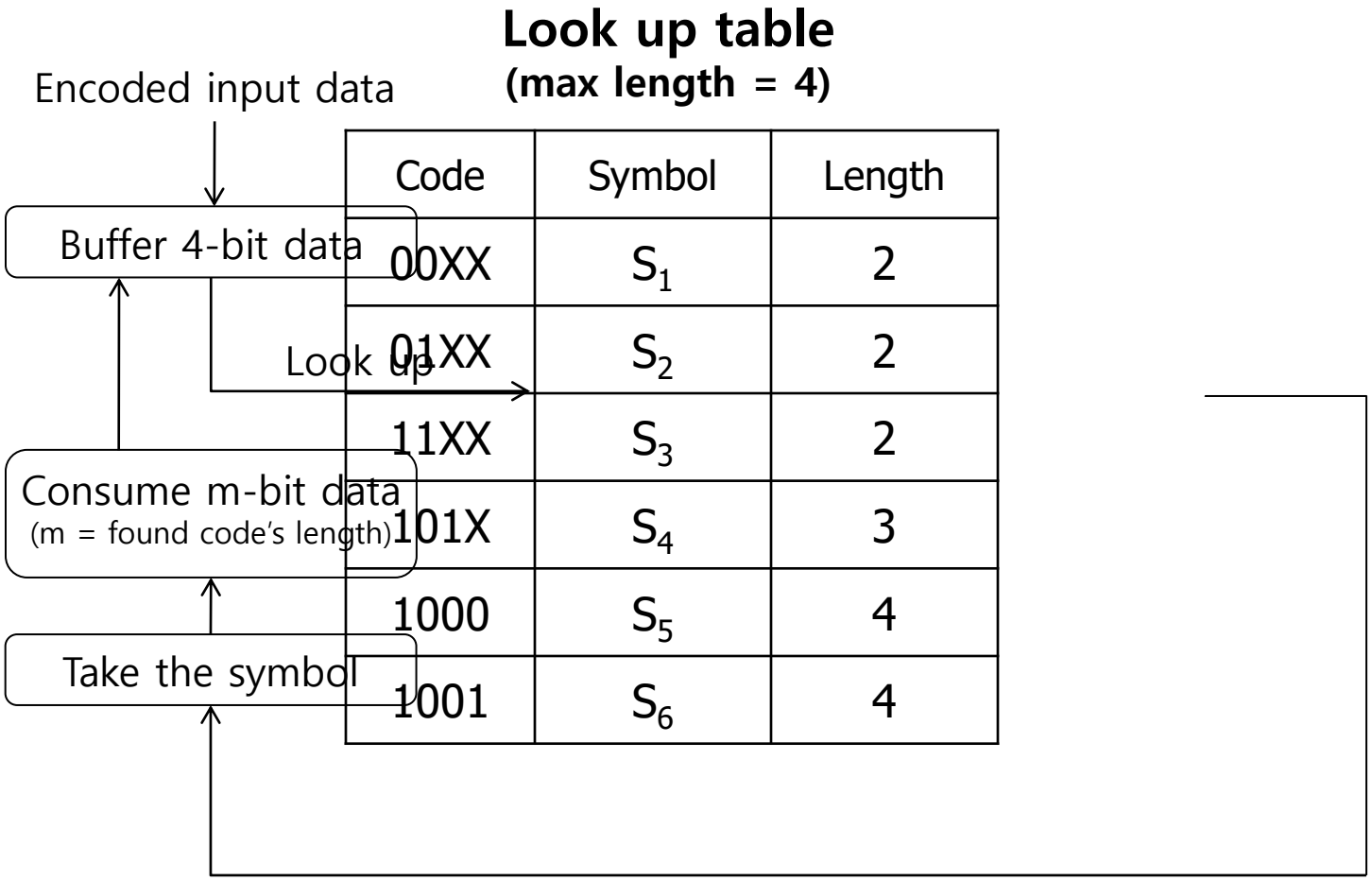
Encoded data



Decoded data

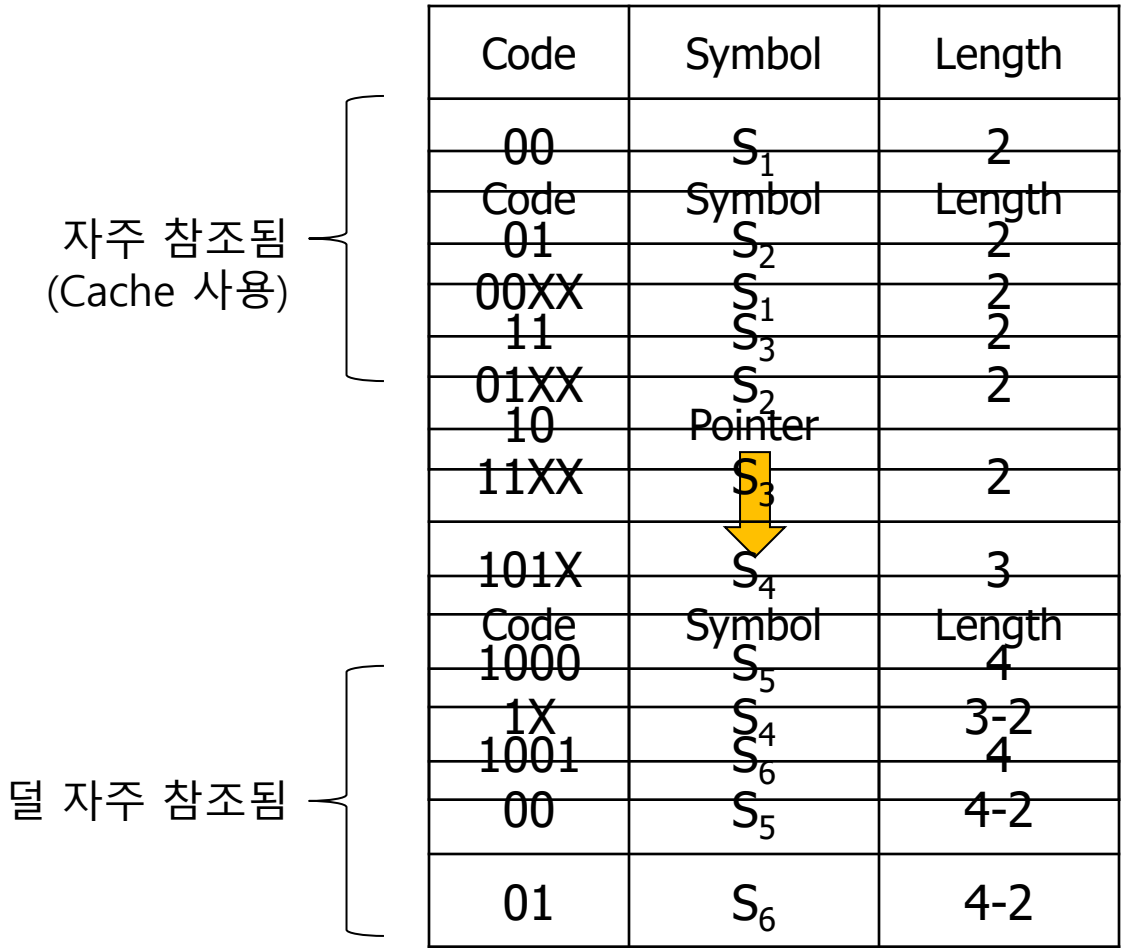
Lookup Table

Lookup table을 사용한 최적화



2-Level Lookup Table

- 다중 계층의 **lookup table**을 사용하여 사용 공간 감소



Part 3: Branch-Aware Optimization

P3.1: Branch Elimination

- In modern superscalar processors, the penalty of a branch misprediction is significant.
- Doing without branches can avoid miss penalty.
- Reducing the number of branches in a program can result in more accurate branch predictions.

P3.1: Extra Work and Masking

Waring: a correctly predicted branch is free.

```
int foo(int x, int y)
{
      if (x > 0) y += x;
      return y;
}
```

```
int foo(int x, int y)
{
      unsigned signbit = (unsigned) x >> 31;
      return y + (x & (signbit -1));
}
```

P3.1: Conditional Instruction

Waring: a correctly predicted branch is free.

```
int foo(int x, int y)
{
        int tmp = x > 0 ? x : 0;
        return y + tmp;
}
```

`x > 0 ? x : 0;` is compiled using the conditional move.

```
mov    tmp, #0
cmp    x, #0
movgt  tmp, x
```

Conditional move works only when the condition is true.

P3.1: Reducing Executed Branches

- Arrange branches from the most influential branches to less influential branches.

```
if (cond1 && cond2 && cond3) { …}
```

Prob(cond1 == true) = 0.8
Prob(cond2 == true) = 0.6
Prob(cond3 == true) = 0.1

Rearrange to:

```
(cond3 && cond2 && cond1)
```

The less branches executed,
the better the accuracy of dynamic branch predictor

P3.1: Rearranging Branches

```
void foo(void)
{ if (very_likely_cond) return;
  if (somewhat_likely_cond) return;
  if (not_likely_cond) return;
     /* do something */
}

void bar(void)
{  if (not_likely_cond){
      if (somewhat_likely_cond) {
         if (very_likely_cond) { ..}
       }
    }
   /* do something */
}
```

P3.2: Branch Prediction-Aware Programming

- Dynamic branch predictors are very efficient at detecting patterns in the branch paths taken.
- Reorganize branches so that some predictable patterns exist.
  - Locality optimizations for branch behavior

Prog A

```
int data[arraySize];

for (unsigned c = 0; c < arraySize; ++c)
        data[c] = std::rand() % 256;

for (unsigned i = 0; i < 100000; ++i)
    {
        // Primary loop
        for (unsigned c = 0; c < arraySize; ++c)
        {
            if (data[c] >= 128)
                sum += data[c];
        }
    }
```

Prog B

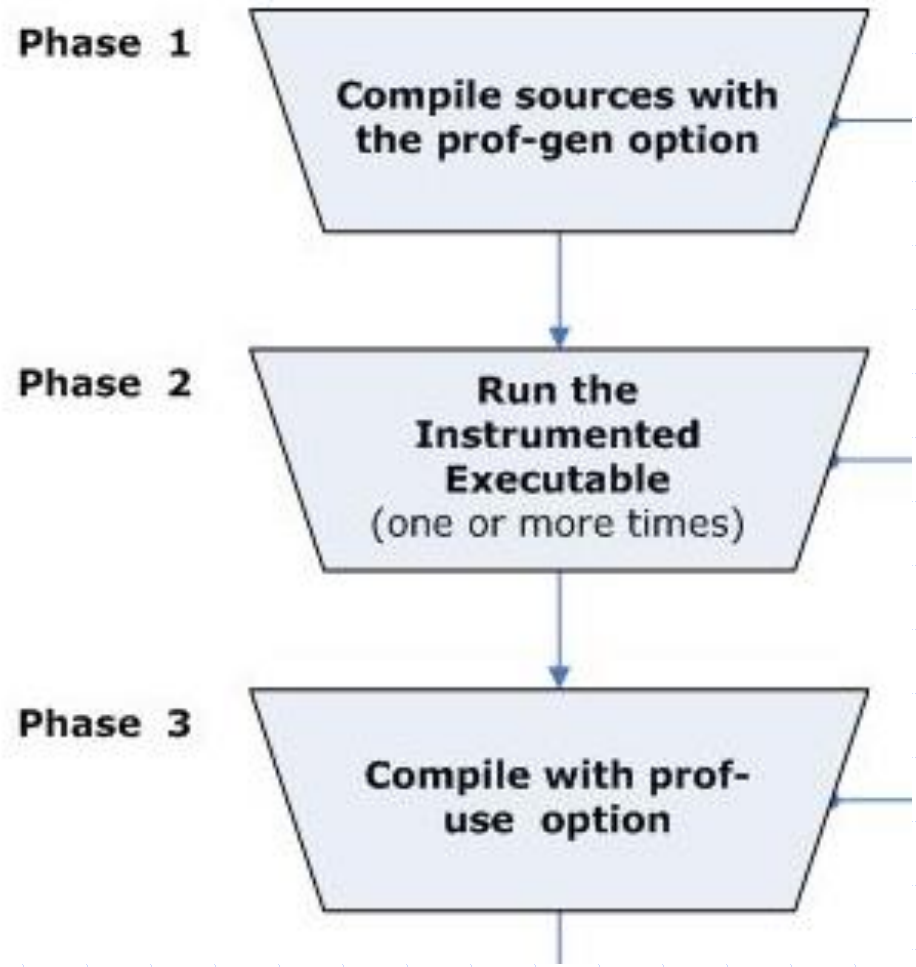
```
int data[arraySize];

for (unsigned c = 0; c < arraySize; ++c)
        data[c] = std::rand() % 256;
std::sort(data, data + arraySize);
for (unsigned i = 0; i < 100000; ++i)
    {
        // Primary loop
        for (unsigned c = 0; c < arraySize; ++c)
        {
            if (data[c] >= 128)
                sum += data[c];
        }
    }
```

Execution Time: Prog A vs. Prog B

	AMD	Intel
ExecTime(Prog B)	6.01(s)	4.43(s)
ExecTime(Prog A)	15.89(s)	14.16(s)

Impact of the branch prediction accuracy

P3.3 Profile Guided Optimization



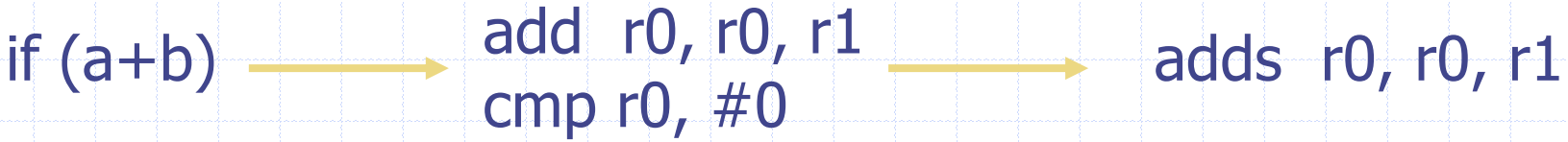
- PGO assumes an accurate representative workload.
- Architecture dependent optimizations.

Part 4: ASM Optimization

P4.1: Conditional Instruction

- ALU instruction + condition setting

  e.g. : adds add + cmp

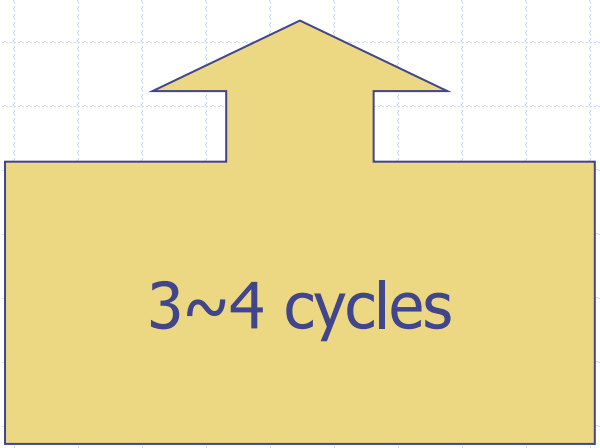


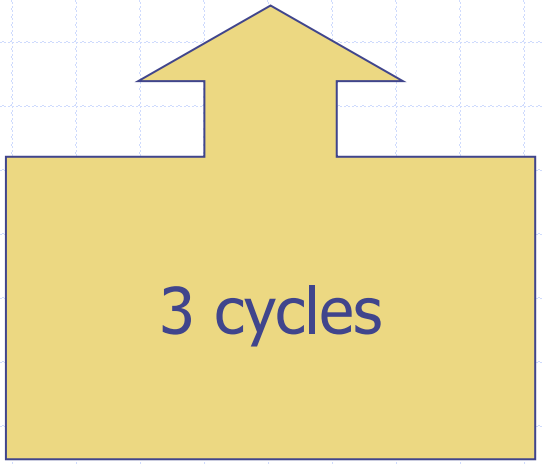
P4.1 Conditional Instruction

```
int foo(int a)
{
    if (a > 10)
        return 0;
    else
        return 1;
}
```

```
    cmp  r0, #10
    ble  L1
    mov  r0, #0
    b    L2
L1: mov r0, #1
L2:
```

```
cmp    r0, #10
movgt  r0, #0
movle  r0, #1
```





P4.1: Conditional Instruction

```
int foo(int a, int b)
{
    if (a != 0 && b != 0)
        return 0;
    else
        return 1;
}
```



```
cmp     r0, #0
cmpne   r1, #0
```

```
int foo(int a, int b)
{
    if (a != 0 || b != 0)
        return 0;
    else
        return 1;
}
```



```
cmp     r0, #0
cmpeq   r1, #0
```

P4.2: Integer Multiply/Divide

- mov r0, r0, LSL #n — $r0 \times 2^n$
- add r0, r0, r0, LSL #n — $r0 \times (2^n+1)$
- add r0, r0, r0, LSL #n — $r0 \times (2^n+1) \bullet 2^m$
  mov r0, r0, LSL #m
- mov r0, r0, LSR #n — $r0/2^n$ (unsigned)
- mov r1, r0, ASR #31 — $r0/2^n$ (signed)
  add r0, r0, r1, LSR #(32-n)
  mov r0, r0, ASR #n

P4.3: Addressing Mode

- str r1, [r0], #4
  mem [r0] = r1; r0 += 4
- str r1, [r0, #4] !
  r0 += 4; mem[r0] = r1
- str r1, [r0], #-4
  mem[r0] = r1; r0 -= 4
- str r1, [r0, #-4] !
  r0 -= 4; mem[r0] = r1

P4.4: Instruction Scheduling

```
add  r1, r2, r3
ldr  r0, [r5]
add  r6, r0, r1
sub  r8, r2, r3
mul  r9, r2, r3
```

```
ldr  r0, [r5]
add  r1, r2, r3
sub  r8, r2, r3
add  r6, r0, r1
mul  r9, r2, r3
```

ARM Thumb

The Thumb Instruction Set

16-bit Instructions (vs. 32-bit ARM instructions)

Used to improve the code density

About 30% reduction over ARM for the same code

Each Thumb instruction mapped to the equivalent ARM instruction:

ADD r0, #3 ADDS r0, r0, #3

Not conditionally executed except for 'B'

Separate instructions for the barrel shift operations

Code Size: ARM vs. Thumb

- C-code

```
if (x>=0)      return x;
else           return -x;
```

- ARM assembly version

4 b x 3 insts = 12 bytes

```
iabs    CMP r0,#0       ;Compare r0 to zero
        RSBLT r0,r0,#0  ;If r0<0 (less than=LT) then do r0= 0-r0
        MOV pc,lr       ;Move Link Register to PC (Return)
```

- Thumb assembly version

2 b x 4 insts = 8 bytes

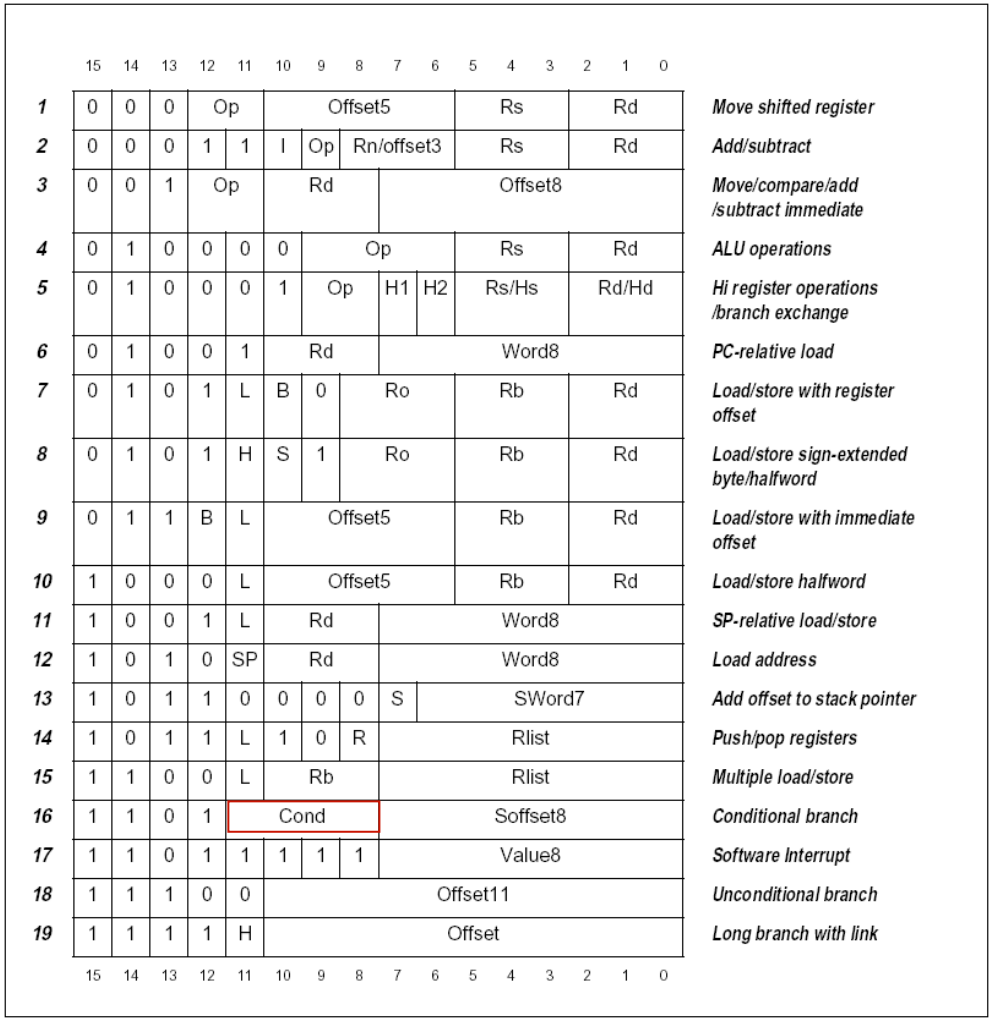
```
        CODE16          ;Directive specifying 16-bit
(Thumb)
                        ;instructions
iabs    CMP r0,#0       ;Compare r0 to zero
        BGE return      ;Jump to Return if greater or equal to zero
        NEG r0,r0       ;If not, negate r0
return  MOV pc,lr       ;Move Link register to PC (Return)
```

Thumb-ARM Differences

- Most Thumb instructions executed unconditionally
  - All the ARM instructions executed conditionally
- Many Thumb data processing instructions use a 2-address format (destination reg == one of source reg)
- Less regular instruction formats over ARM (for the code density)

Thumb instruction set

- 19 instruction formats



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	Op		Offset5					Rs			Rd			*Move shifted register*
2	0	0	0	1	1	I	Op	Rn/offset3			Rs			Rd			*Add/subtract*
3	0	0	1	Op		Rd		Offset8									*Move/compare/add /subtract immediate*
4	0	1	0	0	0	0	Op				Rs			Rd			*ALU operations*
5	0	1	0	0	0	1	Op		H1	H2	Rs/Hs			Rd/Hd			*Hi register operations /branch exchange*
6	0	1	0	0	1	Rd			Word8								*PC-relative load*
7	0	1	0	1	L	B	0	Ro			Rb			Rd			*Load/store with register offset*
8	0	1	0	1	H	S	1	Ro			Rb			Rd			*Load/store sign-extended byte/halfword*
9	0	1	1	B	L	Offset5					Rb			Rd			*Load/store with immediate offset*
10	1	0	0	0	L	Offset5					Rb			Rd			*Load/store halfword*
11	1	0	0	1	L	Rd			Word8								*SP-relative load/store*
12	1	0	1	0	SP	Rd			Word8								*Load address*
13	1	0	1	1	0	0	0	0	S	SWord7							*Add offset to stack pointer*
14	1	0	1	1	L	1	0	R	Rlist								*Push/pop registers*
15	1	1	0	0	L	Rb			Rlist								*Multiple load/store*
16	1	1	0	1	Cond				Soffset8								*Conditional branch*
17	1	1	0	1	1	1	1	1	Value8								*Software Interrupt*
18	1	1	1	0	0	Offset11											*Unconditional branch*
19	1	1	1	1	H	Offset											*Long branch with link*
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Thumb Register Usage

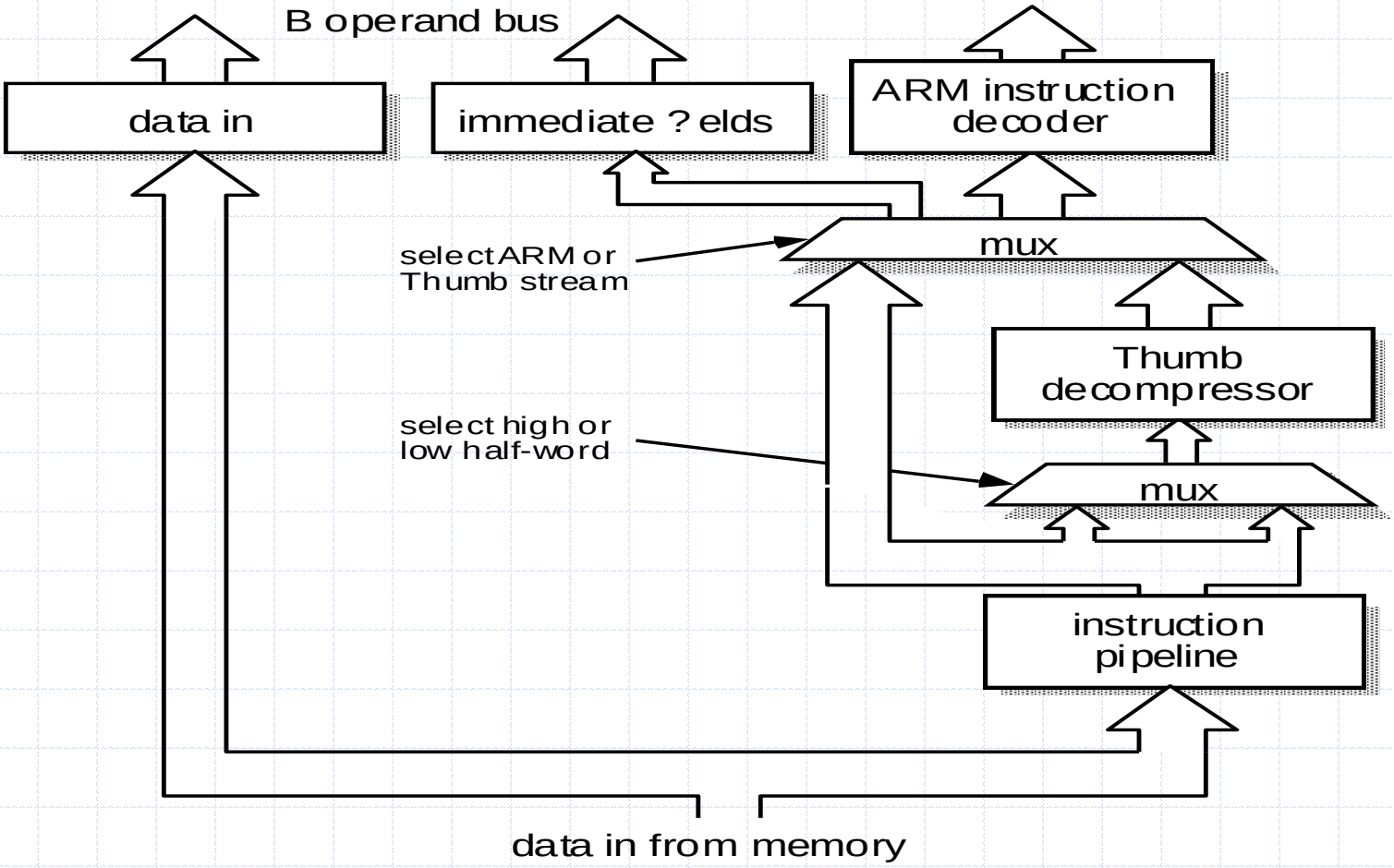
- r0 ~ r7: fully accessible
- r8 ~ r12: only accessible w/ `MOV, ADD, CMP`
- r13, r14, r15: limited accessibility
- cpsr/spsr: no direct access
  - Must switch to ARM state to access cpsr/spsr
  - No coprocessor instructions

ARM Thumb Instruction Opcodes

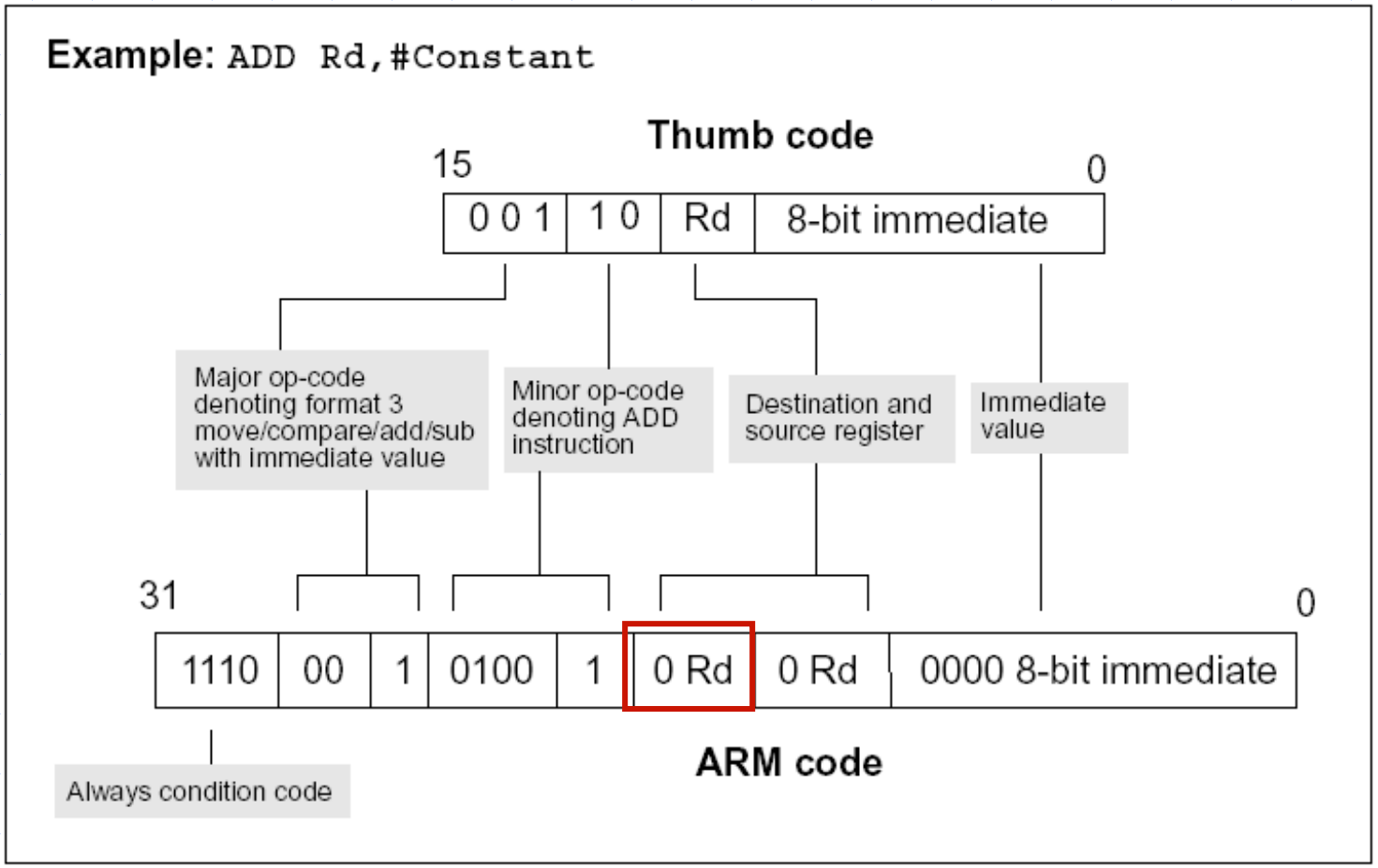
Mnemonic	Instruction	Lo register operand	Hi register operand	Condition codes set
NEG	Negate	✔		✔
ORR	OR	✔		✔
POP	Pop registers	✔		
PUSH	Push registers	✔		
ROR	Rotate Right	✔		✔
SBC	Subtract with Carry	✔		✔
STMIA	Store Multiple	✔		
STR	Store word	✔		
STRB	Store byte	✔		
STRH	Store halfword	✔		
SWI	Software Interrupt			
SUB	Subtract	✔		✔
TST	Test bits	✔		✔

Mnemonic	Instruction	Lo register operand	Hi register operand	Condition codes set
ADC	Add with Carry	✔		✔
ADD	Add	✔	✔	✔①
AND	AND	✔		✔
ASR	Arithmetic Shift Right	✔		✔
B	Unconditional branch	✔		
B*xx*	Conditional branch	✔		
BIC	Bit Clear	✔		✔
BL	Branch and Link			
BX	Branch and Exchange	✔	✔	
CMN	Compare Negative	✔		✔
CMP	Compare	✔	✔	✔
EOR	EOR	✔		✔
LDMIA	Load multiple	✔		
LDR	Load word	✔		
LDRB	Load byte	✔		
LDRH	Load halfword	✔		
LSL	Logical Shift Left	✔		✔
LDSB	Load sign-extended byte	✔		
LDSH	Load sign-extended halfword	✔		
LSR	Logical Shift Right	✔		✔
MOV	Move register	✔	✔	✔②
MUL	Multiply	✔		✔
MVN	Move Negative register	✔		✔

Thumb Instruction Decoder Organization



Thumb to ARM Instruction Mapping



Thumb: r0 – r7 only

ARM-THUMB Interworking

- To call a THUMB routine from an ARM routine, the core should switch to 'THUMB' mode:
- T flag in CPSR indicates the current mode.
- **`BX`** and **`BLX`** instructions are used to switch ARM/THUMB modes.

BX & BLX Instructions

- **`BX Rm`** ; branch exchange
  - **`pc`** = ***`Rm`*** & 0xfffffffe
  - T = ***`Rm`*****`[0]`**
- **`BLX Rm | label`** ; branch exchange w/link
  - **`lr`** = inst. addr after BLX + T
  - **`pc`** = ***`label, T = label[0]`***
  - **`pc`** = ***`Rm & 0xfffffffe, T = Rm[0]`***

BLX Example (ARM -> Thumb)

```
        CODE32
        LDR r0, =thumbCode + 1
        BLX r0

        CODE16
thumbCode
        ADD r1, #1
        BX    lr
```

BLX Example (Thumb ARM)

```
        CODE16
        LDR r0, =ARMCode
        BLX r0

        CODE32
ARMCode
        ADD r1, #1
        BX   lr              ; lr[0] was already set to 1.
```

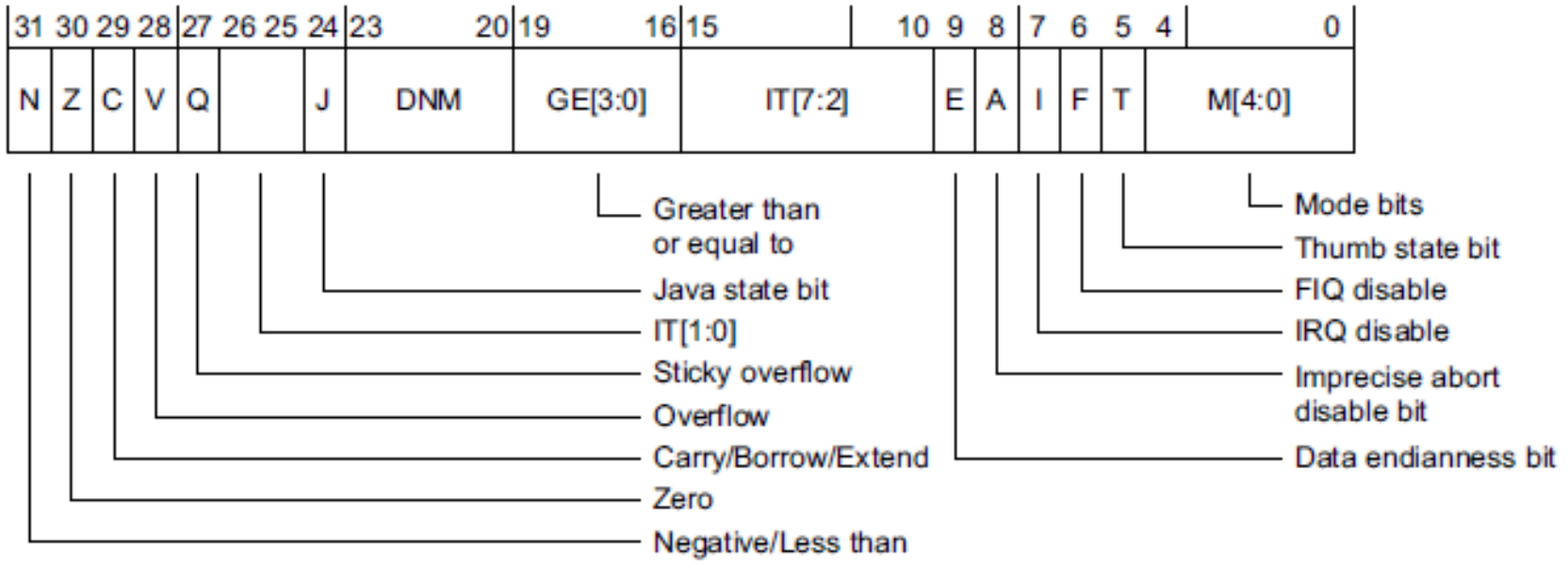
Thumb Advantages

- Space: About 70% of ARM code
- # of Instructions: About 140% of ARM code
- Exec. Time:

  With a 32-bit memory, ARM code is about 40% faster over Thumb code

  With a 16-bit memory, Thumb code is about 45% faster over ARM code
- Thumb code consumes about 30% less memory power.

ARM v7: CPSR

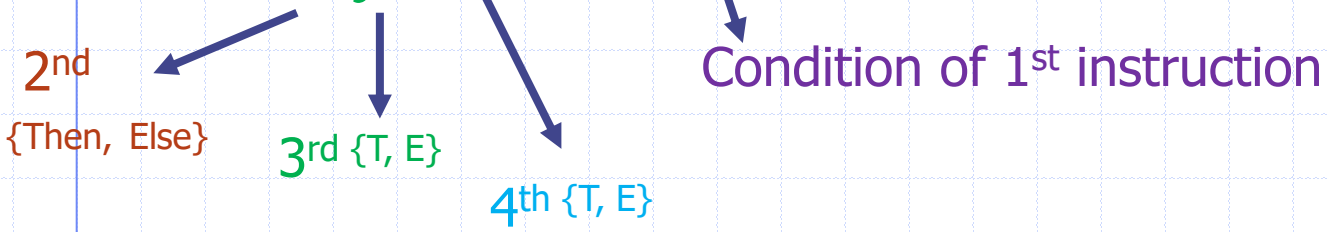


Execution state bits:

- If-Then state bits (IT)
- Java state bit (J)
- Thumb state bit (T)

IT (If-Then) Instruction

- Can make Thumb instructions to be conditionally executed.
- IT{x{y{z}}} {cond}




```
; flags set by a previous instruction
ITTE EQ
MOVEQ  r0, r1
BEQ  dloop
MOVNE r1, r0
```

ARM Instruction Set Overview - Condition Codes

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

IT[7:0] bits are split into:

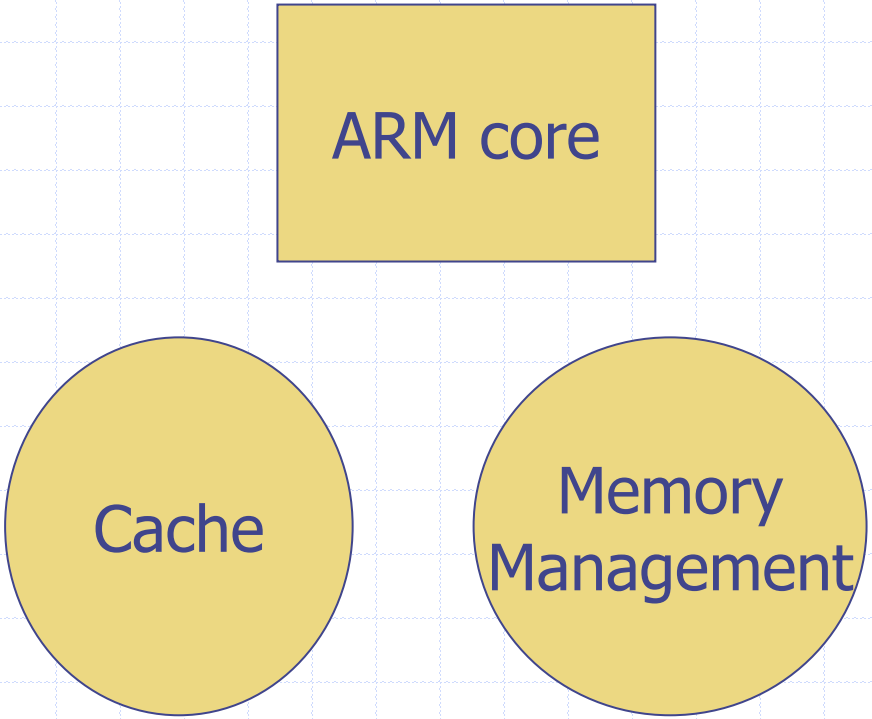
Base condition: 3 bits

Then/Else bits for up to 4 instructions: 4 bits

Stop bit of IT block: 1 bit

ARM Memory Management Units

ARM Processor Organization



MPU & MMU

- MPU == hardware protection over software-designated regions
- MMU == hardware protection + virtual memory support

- E.g., ARM920T vs. ARM940T

MMU Overview

- Support for private memory space for each task
  - Virtual memory system
  - Address relocation

Virtual to Physical Mapping

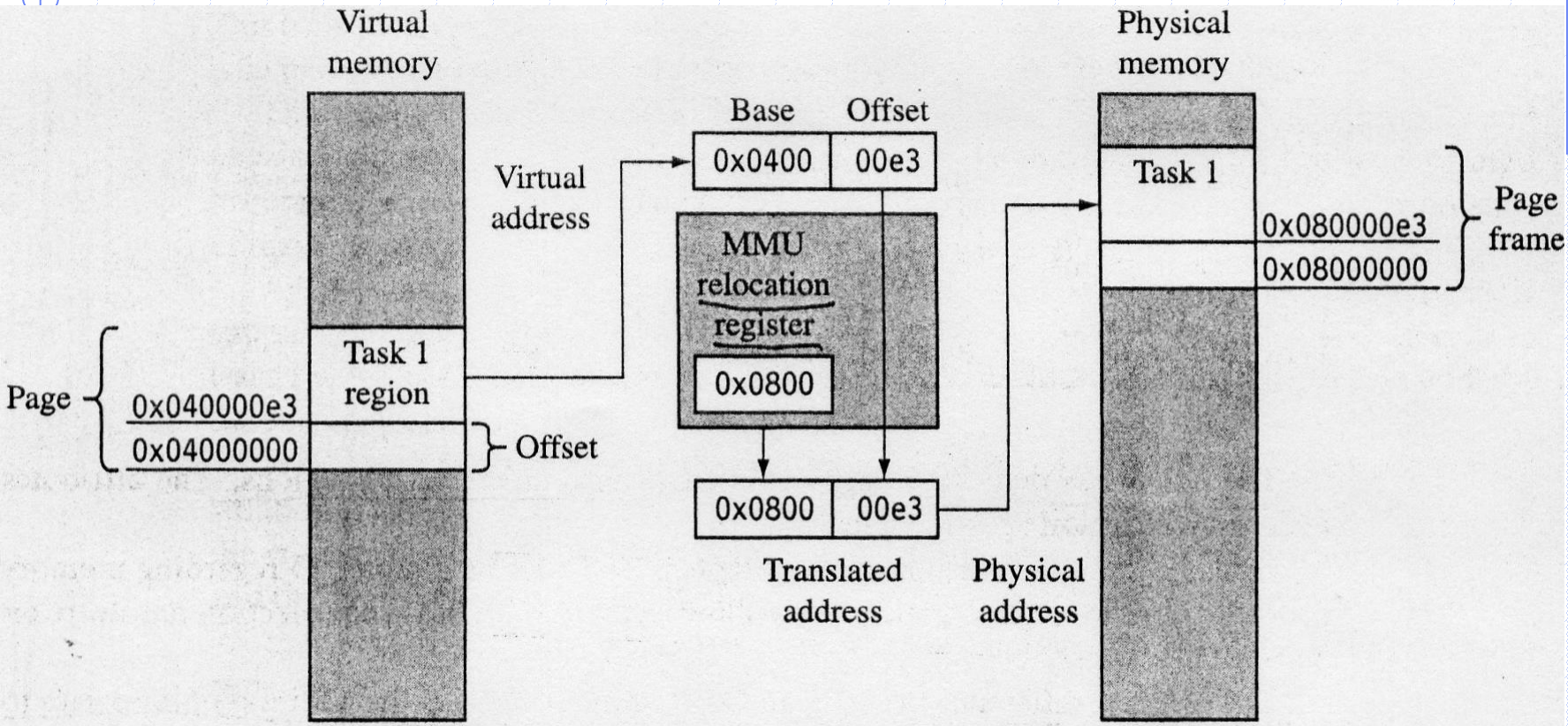


Figure 14.1 Mapping a task in virtual memory to physical memory using a relocation register.

TLB

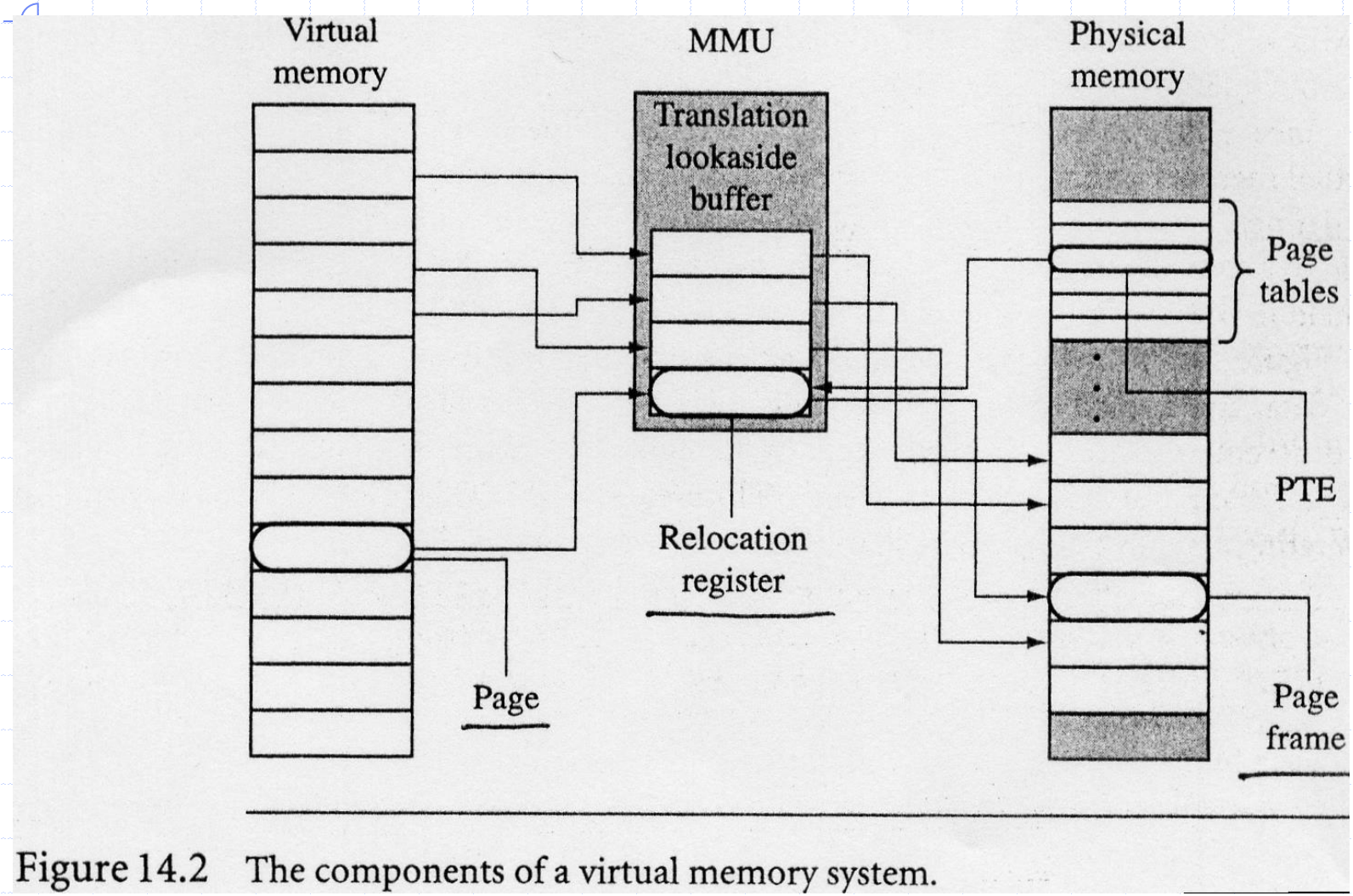


Figure 14.2 The components of a virtual memory system.

Regions

- Sequential set of page table entries (S/W)
  - c.f., H/W components in MPU
- Most page tables represent 1 MB areas of virtual memory

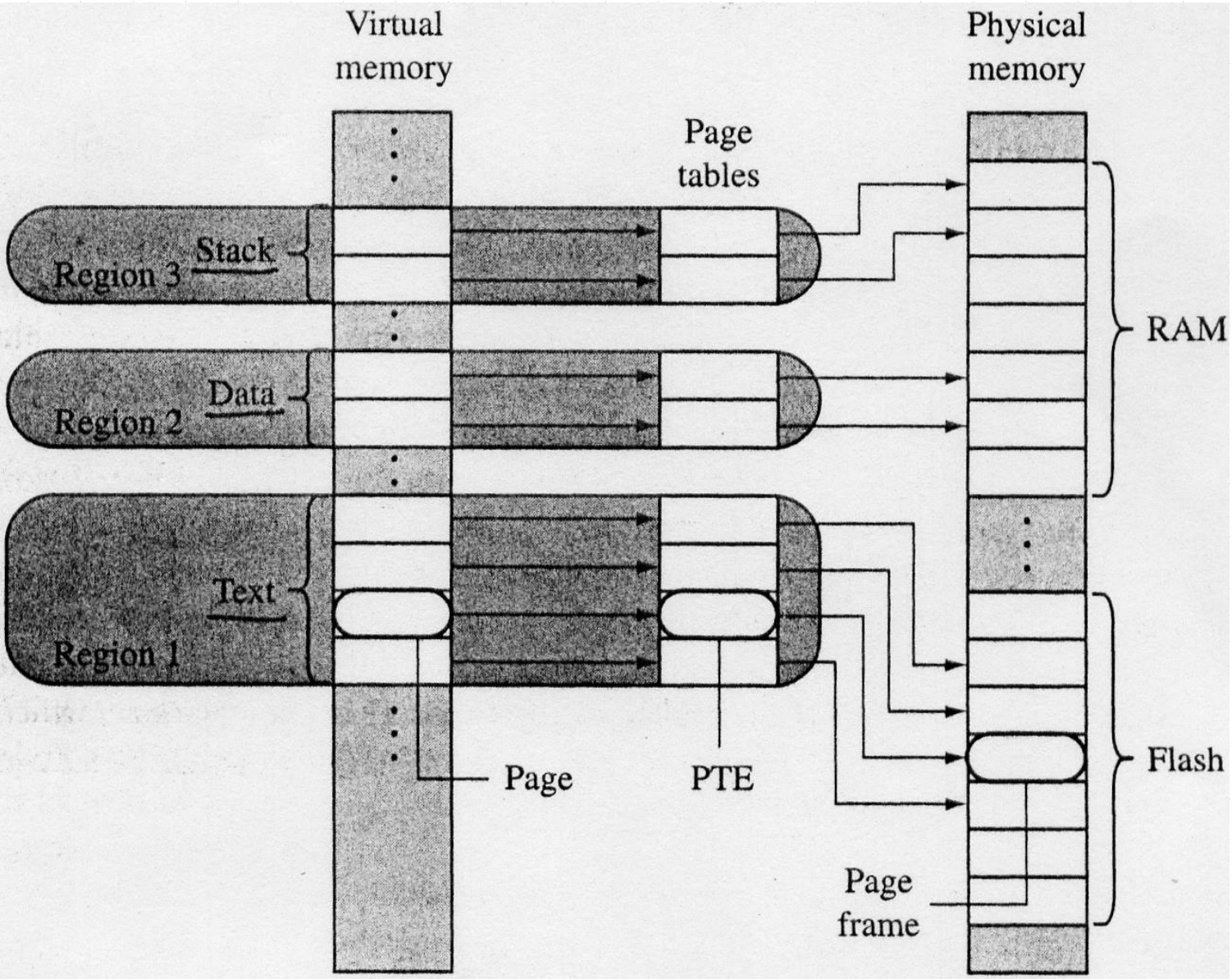


Figure 14.3 An example mapping pages to page frames in an ARM with an MMU.

Multitasking under MMU

- For context switch:
  1. Save the active task context & put it in a dormant state
  2. Flush caches & TLB
  3. Configure MMU to use new page tables
  4. Restore the context of new task
  5. Resume the execution
- Why flush caches?

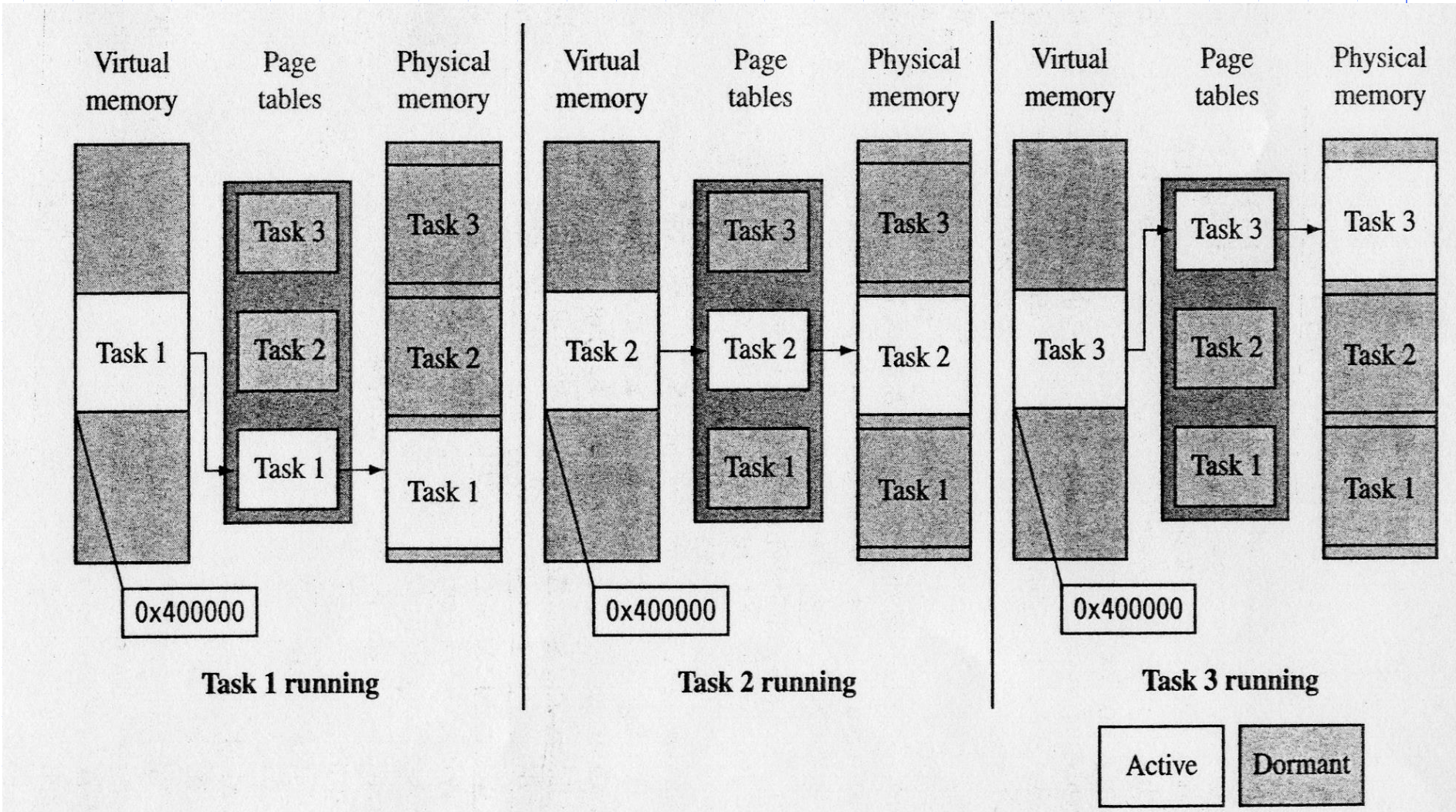


Figure 14.4 Virtual memory from a user task context.

Three tasks with the same virtual addresses

General Memory Organization

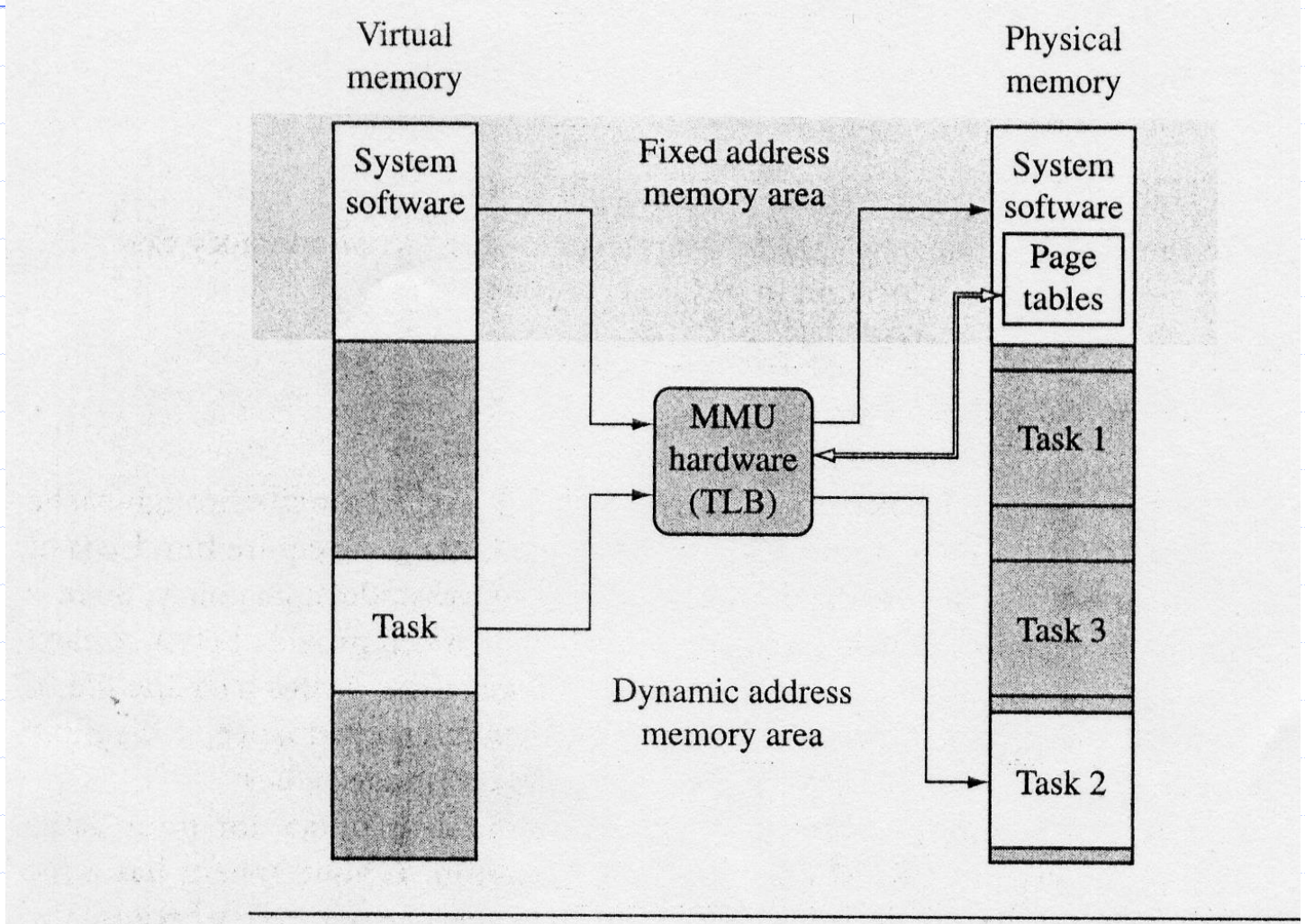


Figure 14.5 . A general view of memory organization in a system using an MMU.

MMU Functions

- Virtual to physical translation
- Memory access permission
- Cache & WB configuration for each page

- Generate abort exceptions for
  - Translation, permission, domain faults

ARM MMU Overview

- Two levels of page tables:
  - L1 page table
  - L2 page table
- L1 page table divides 4 GB address space into 1 MB sections:
  - 4096 entries

L1 Page Tables

- L1 master page table (base addr: CP15:c2)
  - Types 1 & 2: Base address of L2 page tables
    - Fine L2 page table
    - Coarse L2 page table
  - Type 3: Page table entry for 1 MB section
  - Type 4: Fault entry (will generate an abort exception)

L1 PTEs

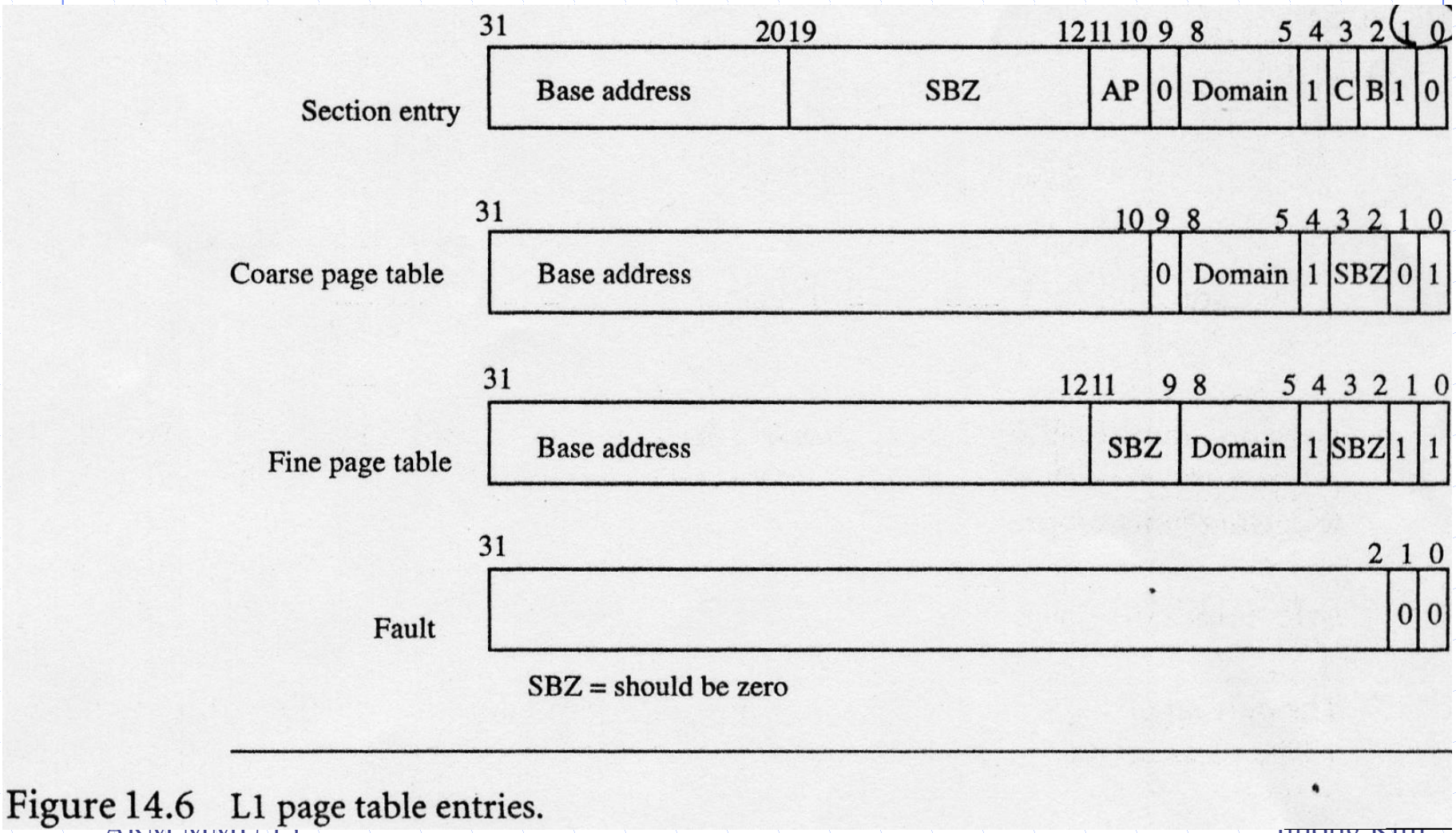


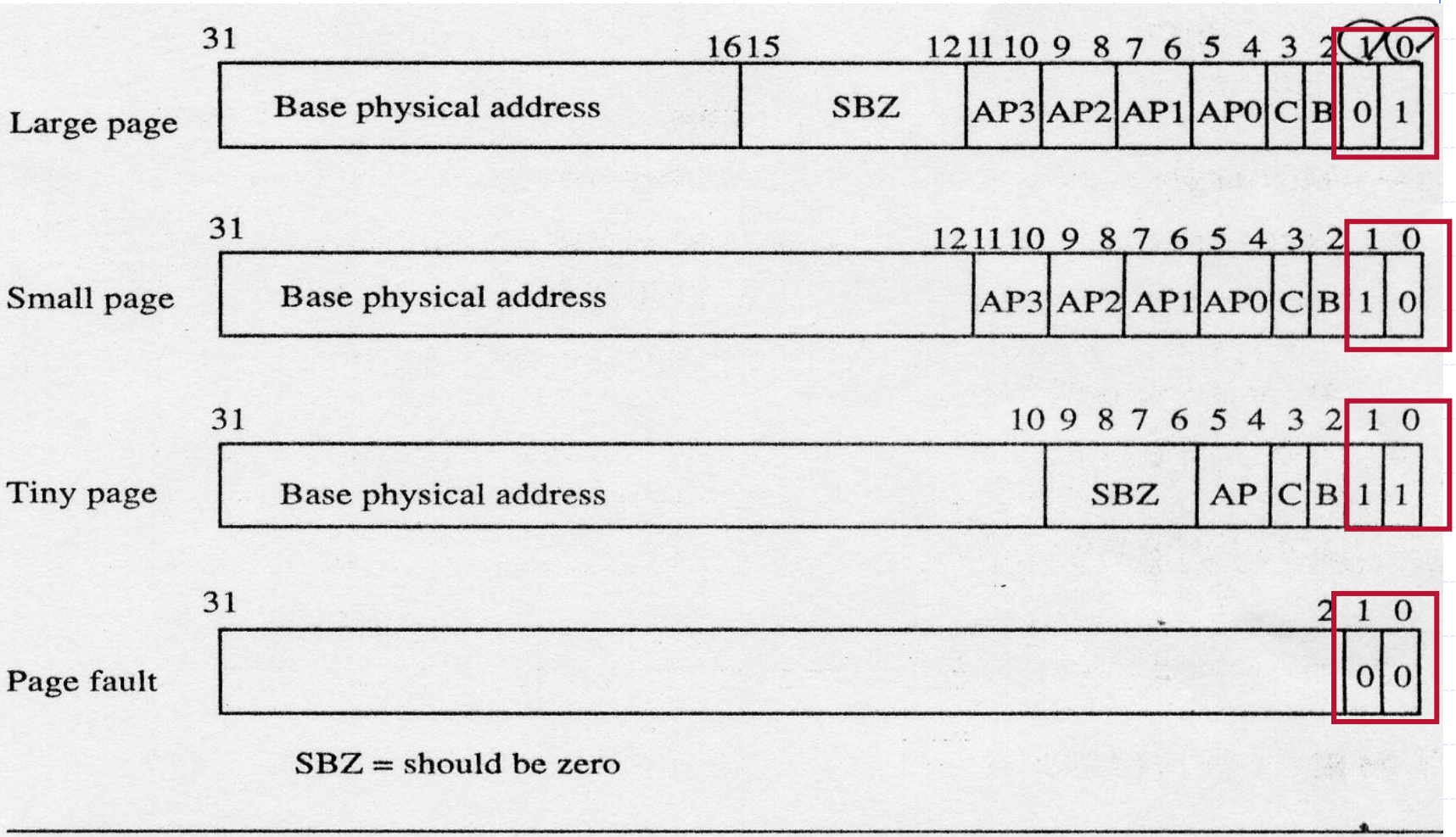
Figure 14.6 L1 page table entries.

L2 Page Tables

- Large page (64 KB)
- Small page (4 KB)
- Tiny page (1KB)
- Fault page entry

L2 PTEs

APs for four subpages



14.8 L2 page table entries.

Single-Step Page Table Walk

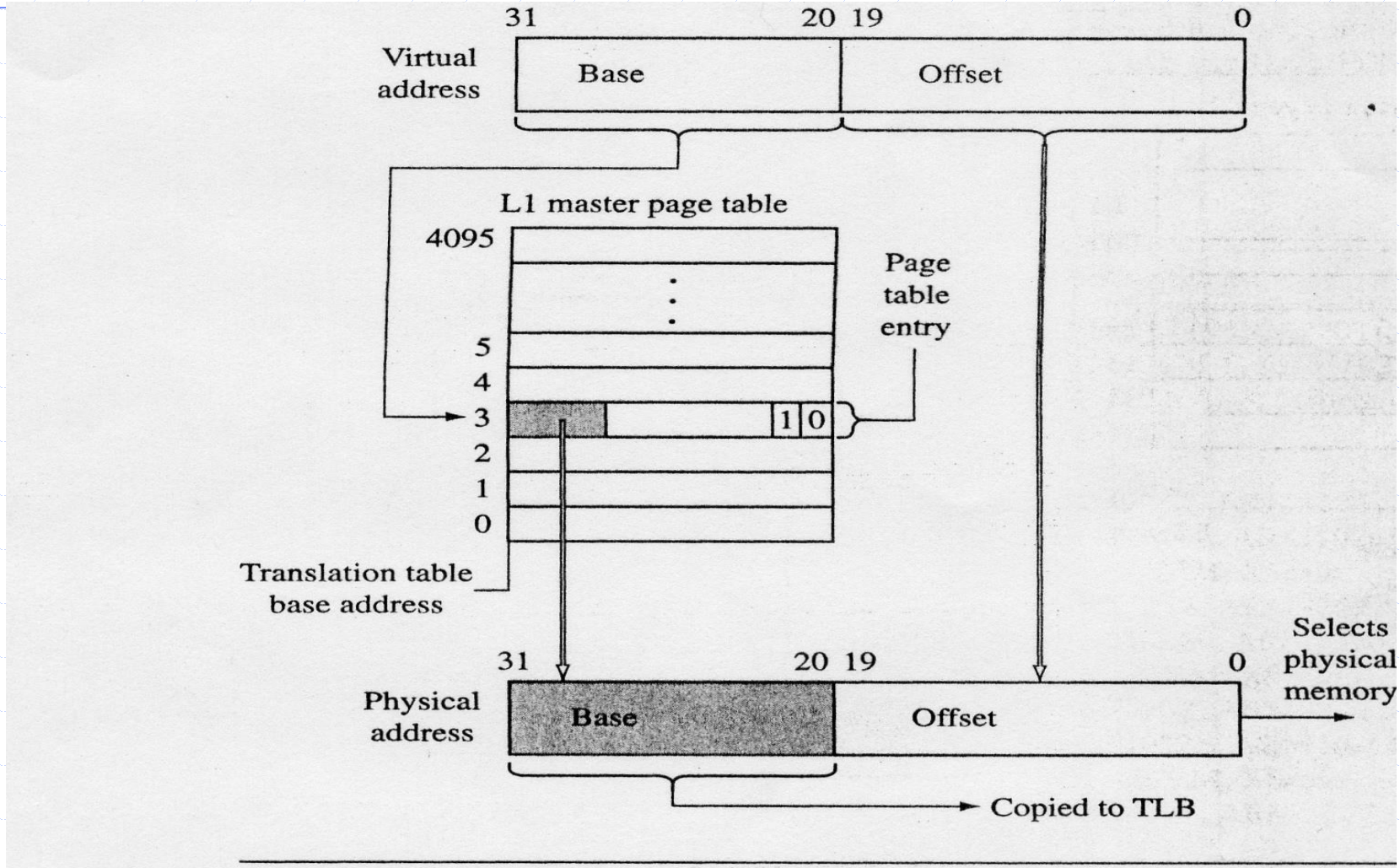


Figure 14.9 L1 Page table virtual-to-physical memory translation using 1 MB sections.

Two-Step Page Table Walk

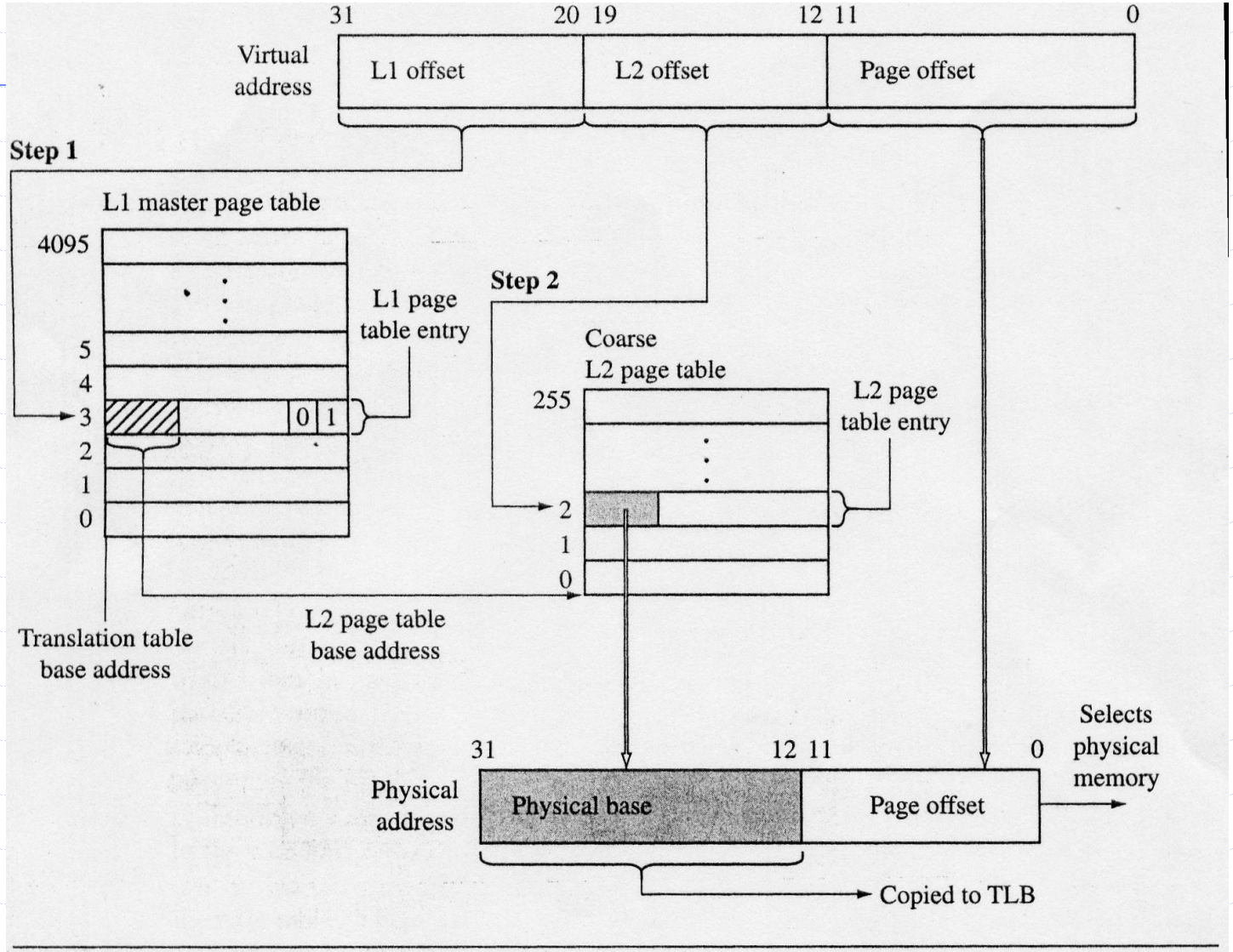


Figure 14.10 Two-level virtual-to-physical address translation using coarse page tables and 4 KB pages

Domains

- 16 different domains
  - Assigned to a section by setting the domain field in L1 PTE

Value	Status	Description
00	No access	Any access will generate a domain fault
01	Client	Page and section permission bits are checked
10	Reserved	Do not use
11	Manager	Page and section permission bits are not checked

Fast Context Switch Extension

- Avoid cache & TLB flushes during context switch
- Virtual Address (VA)
- Modified Virtual Address (MVA)

- MVA = VA + (0x2000000 * process ID)

CP15 register 13

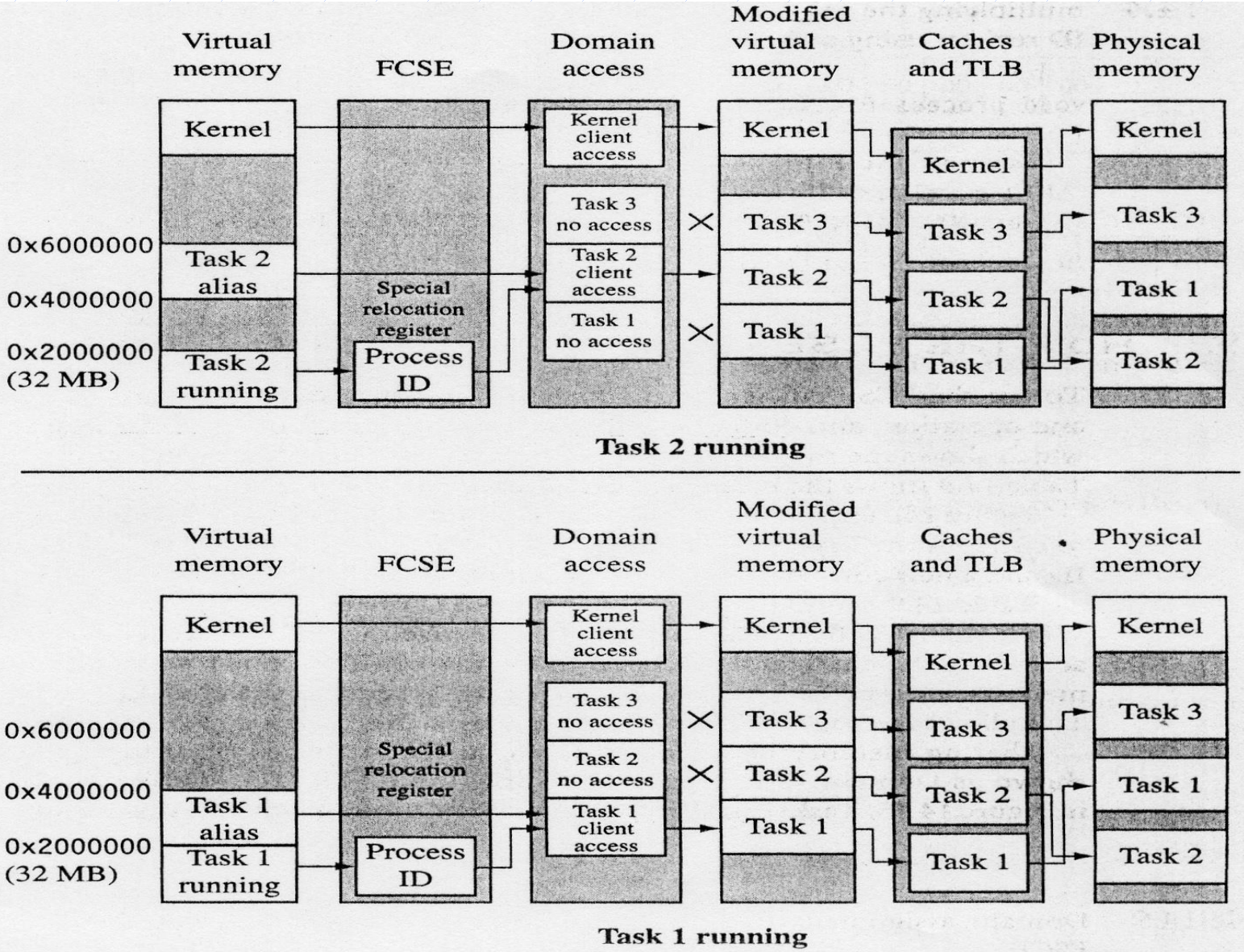
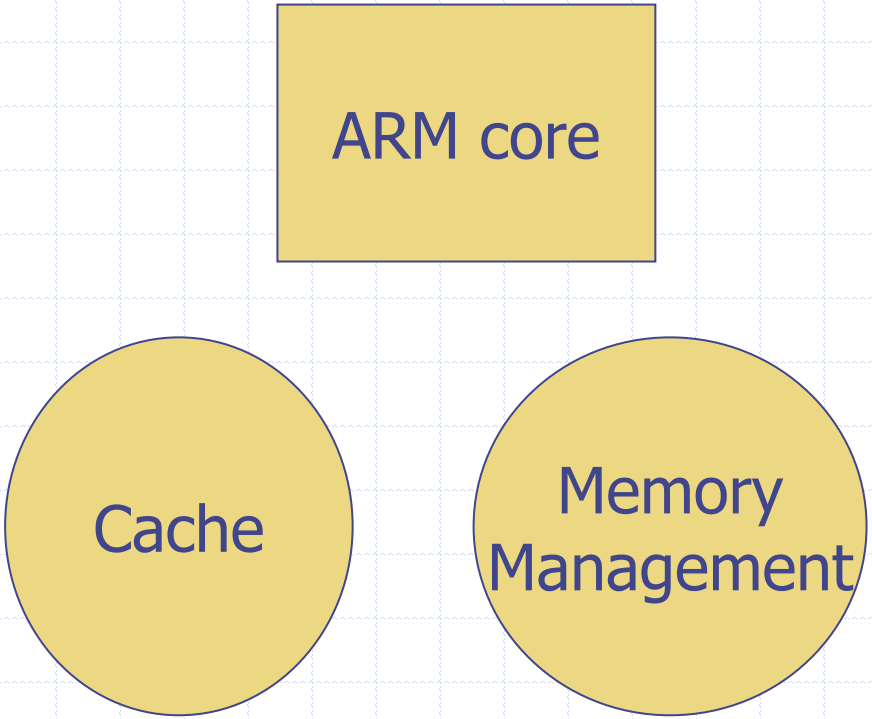


Figure 14.14 Fast Context Switch Extension example showing task 1 before a context switch and task 2 running after a context switch in a three-task multitasking environment.

ARM Memory Protection Units

ARM Processor Organization



MPU & MMU

- MPU == hardware protection over software-designated regions
- MMU == hardware protection + virtual memory support

- E.g., ARM920T vs. ARM940T
  - MMU
  - MPU

Why Protection Necessary?

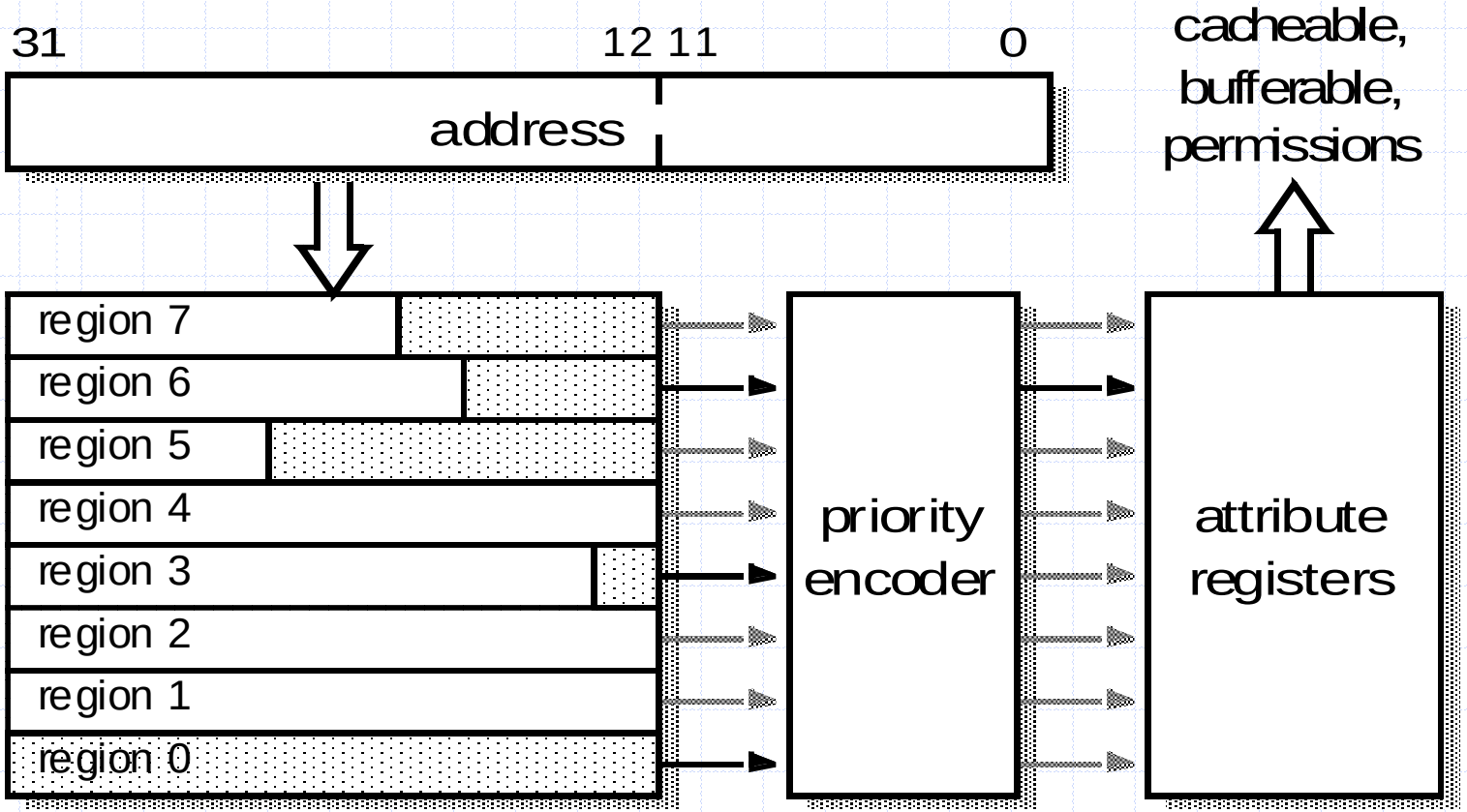
Protected Regions

- A basic management unit of system protection
- 8 regions: region 0 – region 7
- Attributes:
  - Starting address
  - Length (4KB ~ 4GB, power of two size only)
  - Access rights, cache/WB policies, cache write policy
    - Read-write, read-only, no access
    - Based on the current processor mode, [privileged] [user]
    - Memory access violations → abort exceptions

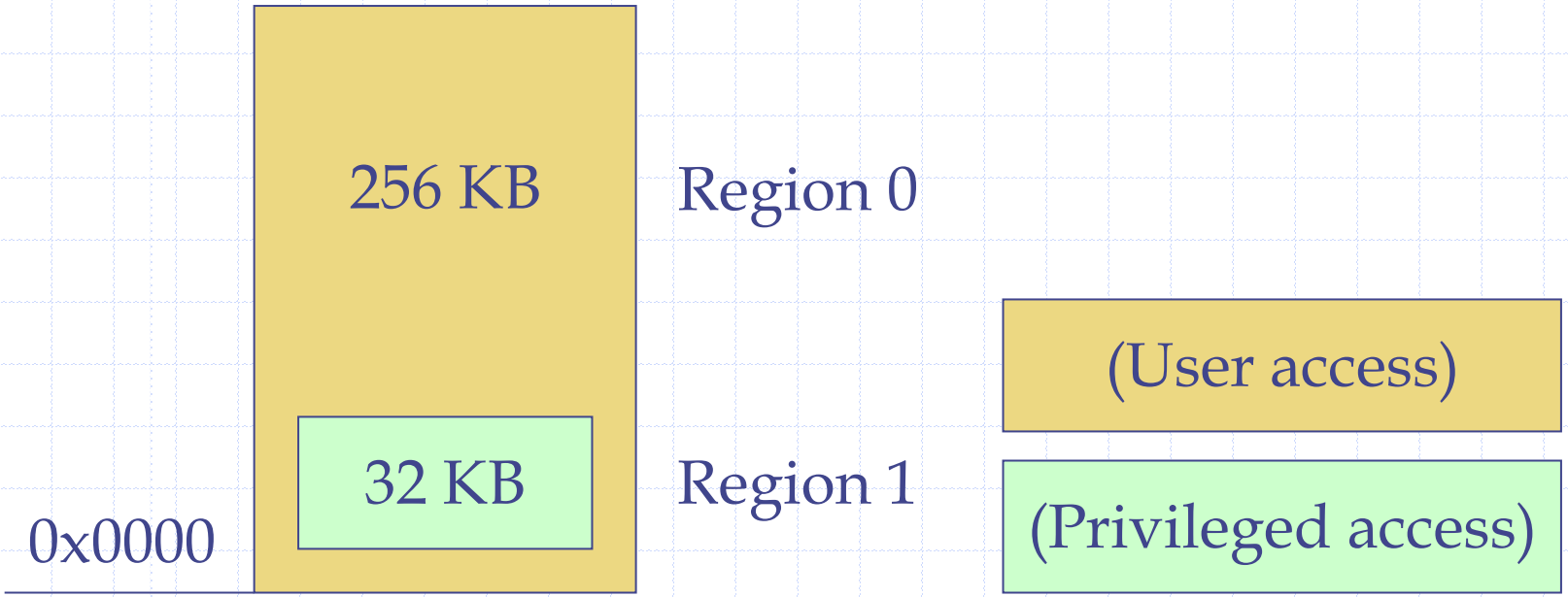
Region Rules

- Can overlap other regions
- Each region has its own priority
  - The higher the region number, the higher its priority
  - For overlapped areas, the attributes of the highest priority are applied
- Starting address must be aligned to its size
- Accessing an area outside of a defined region results in an abort.

MPU Organization



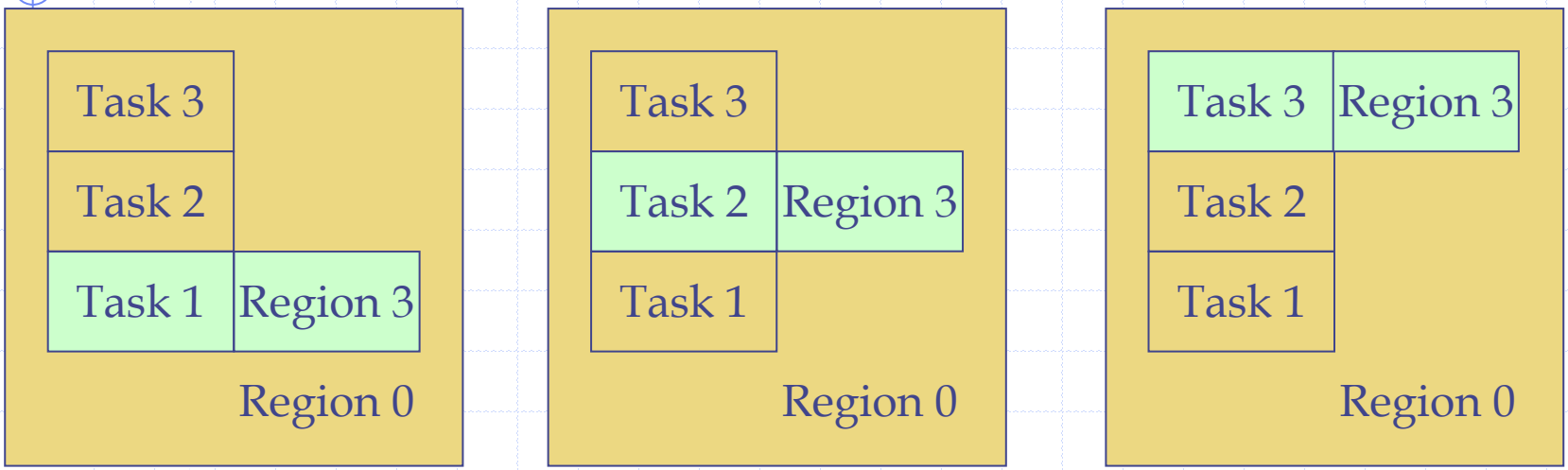
Ex: Overlapping Regions



Background Regions

- A low-priority region used to assign the common attributes to a large memory area.
- Other higher regions can overrule when necessary.

Ex: Background Regions



Task 1 running

Task 2 running

Task 3 running

Region 0: Background Region
Region 3: Active Task

(User access)

(Privileged access)

MPU Initialization Steps

1. Define regions. (CP15:c6)
2. Set access permission for each region. (CP15:c5)
3. Set the cache/WB attributes for each region. (CP15:c2 (cache) c3 (WB))
4. Enable caches/MPU. (CP15:c1)

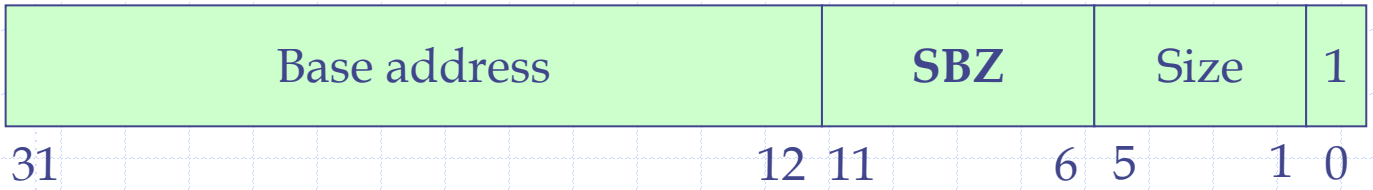
- Standard AP

Supervisor	User
No Access	No Access
R/W	No Access
R/W	Read Only
R/W	R/W

- Inst $: Not Cached/Cached
- Data $: Not Cached & Not Buffered,
  Not Cached & Buffered
  Cached (WT)
  Cached (WB)

Ex: Region size and location

- One secondary register for each region:
  - CP15:c6:c0:0 for region 0



Size of region: $2^{(Size+1)}$

SBZ == Should Be Zero