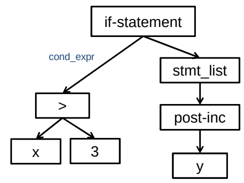
CS 1622: Code Generation & Register Allocation

Jonathan Misurda
jmisurda@cs.pitt.edu

Code Generation

Input: Intermediate representation
Output: Target code

Example:




```
        slti   $t1, 3, $s0
        beq    $t1, $zero, L1
        addi   $s1, $s1, 1
L1:
```

Why is Code Generation Hard?

If the goal is to simply generate target code, we have already done this when we generated IR:
- Walk the AST and emit target code.

However, if we want to generate ***good*** target code, there are many things to consider.

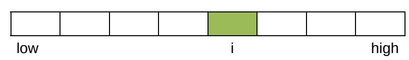
Ultimately, the back end of the compiler is the repository of machine-specific knowledge.

We need to be able to assess among the multiple possible ways to encode a calculation, which one is best.

Arrays

Consider converting the following to machine code and data:

```
int A[low … high];
A[i]++;
```



To deal with this array, we need to know the following things:
- width — width (size) of each element
- base — address of the first element
- low/high — lower/upper bound of subscript

Array Element Address

The address of element A[i] is then:

```
base + (i - low) * width =

i * width + (base-low*width) =
```

$$i * width + C_1$$

Where C_1 is a constant for this array.

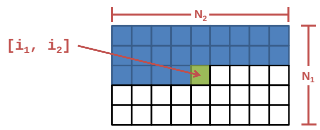
Multidimensional Arrays

How should we store the data for a 2-Dimensional array?

Memory is one dimensional and so we must linearize our multi-dimensional arrays.

Two choices for how to do this:
- Row Major Order
- Column Major Order

Row Major Order



Row Major Order — Store data elements row by row

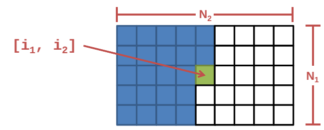
Blue elements are stored before `A[i1, i2]`

Address of Element `A[i1, i2]`:

$= base + ((i_1 - low_1) * N_2 + (i_2 - low_2)) * width$

$= (i_1 * N_2 + i_2) * width + C_{2R}$

Column Major Order



Column Major Order — Store data elements row by row

Blue elements are stored before `A[i1, i2]`

Address of Element `A[i1, i2]`:

$= base + ((i_2 - low_2) * N_1 + (i_1 - low_1)) * width$

$= (i_2 * N_1 + i_1) * width + C_{2C}$

Higher Dimensional Arrays

Row major: addressing a k-dimension array item (low_i = base = 0)

$$A_k = A_{k-1} * N_k + i_k * width$$

Column major: addressing a k-dimension array item (low_i = base = 0)

$$A_k = i_k * N_{k-1} * N_{k-2} * \dots * N_1 * width + A_{k-1}$$

C Arrays

C uses row major order:

```
int fun1(int p[][100])
{
    int a[100][100];
    …
    a[i1][i2] = p[i1][i2] + 1;
}
```

Why is p[][100] allowed?

- The information is enough to compute $p[i_1][i_2]$'s address
- $A_2 = (i_1 * N_2 + i_2) * width$ ...

Why is a[][100] is not allowed?

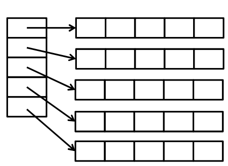
- Need to allocate space

Java Arrays

Java doesn't have 2+ dimensional arrays.

Arrays are arrays of arrays.

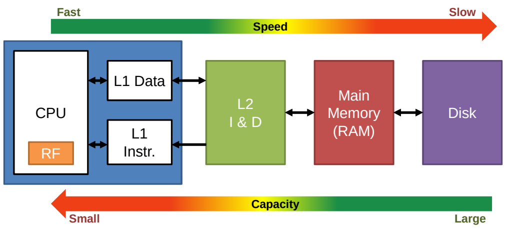
```
int [][] a = new int[5][5];
```



Why Does it Matter?

Caching

Memory Hierarchy



SRAM: 1-25 ns (2-100 cycles)
DRAM: 60-120 ns
Magnetic disk: 10-20 million ns

Locality

How do we know what to include in the levels that are faster but smaller?

Use principles of locality:

- **Temporal locality:** What you use now, you will likely use again soon.
- **Spatial locality:** When you access an address, you will likely access its neighbors soon.

Caches Exploit Locality

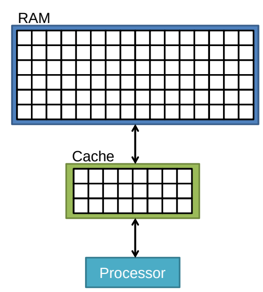
For **temporal locality,** keep more recently used items closer to the processor. Less recently used items can be kept farther away.

For **spatial locality**, get items nearby referenced item at the same time as the requested item. (That is, don't just bring what was requested but rather move larger blocks of contiguous memory.)

Cache Basics

Is address 16 in the cache?

If yes, we have a cache "hit".
If no, we have a cache "miss".

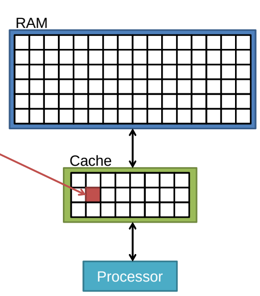


Cache Basics

Is address 16 in the cache?

If yes, we have a cache "hit".
If no, we have a cache "miss".

Address 16 is there, so we have a cache hit.



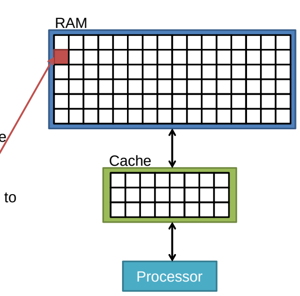
Cache Basics

Is address 16 in the cache?

If yes, we have a cache "hit".
If no, we have a cache "miss".

Address 16 isn't in the cache, so we must go to a farther away level of the memory hierarchy.

On a miss of our cache, we must go to main memory.



Cache Basics

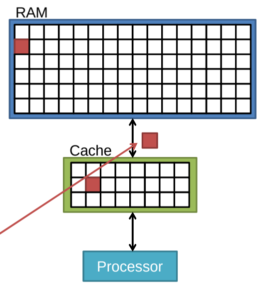
Is address 16 in the cache?

If yes, we have a cache "hit".
If no, we have a cache "miss".

Address 16 isn't in the cache, so we must go to a farther away level of the memory hierarchy.

On a miss of our cache, we must go to main memory.

Data can then be transferred between levels.



Cache Basics

Is address 16 in the cache?

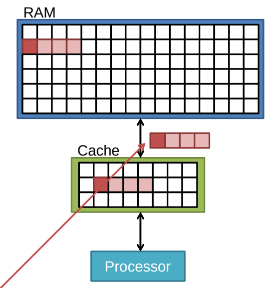
If yes, we have a cache "hit".
If no, we have a cache "miss".

Address 16 isn't in the cache, so we must go to a farther away level of the memory hierarchy.

On a miss of our cache, we must go to main memory.

Data can then be transferred between levels.

Data may be transferred together in some minimal unit, we'll call a block.



Row by Row vs. Col by Col

```
#define ROWS 20000
#define COLS 20000

int a[COLS][ROWS];

int main() {
    int i; int j;
    long long sum =0;

    for(i=0;i<COLS;i++)
        for(j=0; j<ROWS; j++)
            a[i][j]=rand()%10+1;

    for(i=0;i<COLS;i++)
        for(j=0; j<ROWS; j++)
            sum += a[i][j];
    return 0;
}
```

```
#define ROWS 20000
#define COLS 20000

int a[COLS][ROWS];

int main() {
    int i; int j;
    long long sum =0;

    for(i=0;i<ROWS;i++)
        for(j=0; j<COLS; j++)
            a[j][i]=rand()%10+1;

    for(i=0;i<ROWS;i++)
        for(j=0; j<COLS; j++)
            sum += a[j][i];
    return 0;
}
```

Results

```
gcc -m32 -o row roworder.c
time ./row

real    0m15.979s
user    0m14.651s
sys     0m1.326s
```

```
gcc -m32 -o col colorder.c
time ./col

real    0m38.640s
user    0m37.417s
sys     0m1.212s
```

$$\frac{37.417}{14.651} = 2.55$$

2.55x slower just by interchanging the loops!

Processing Boolean Expressions

Representation of True and False:

Like C:
- 0 – False
- Anything Else – True

Alternative:
- 0 – False
- -1 – True (-1 in Two's complement is the string of all 1s)

Short Circuiting

E = (a < b) or (c < d and e < f)

```
        if (a<b) goto TRUE_CODE
L1:     if (c<d) goto L2
        goto FALSE_CODE
L2:     if (e<f) goto TRUE_CODE
        goto FALSE_CODE
```

Processing Control Flow

Whenever we have forward control flow jumps (to locations we haven't translated yet) we are unable to generate the target labels for the code to jump to.

There are two options:
- Do it in a single pass and resolve unknown jumps using backpatching
- Generate the code in one pass and then the labels in a second pass

Backpatching

Create a worklist of "holes" to fill in as we gain the information necessary to do so.

100: if (a < b) goto ____	Process this branch and add (100) to our worklist
101: a := a + 1	
102: b: = b + a	
103: goto ___	

Backpatching

Create a worklist of "holes" to fill in as we gain the information necessary to do so.

100: if (a < b) goto ____	Process this branch and add (100) to our worklist
101: a := a + 1	
102: b: = b + a	
103: goto ___	Process this jump and add (103) to our worklist

Backpatching

Create a worklist of "holes" to fill in as we gain the information necessary to do so.

100: if (a < b) goto **104**	Process this branch and add (100) to our worklist
101: a := a + 1	
102: b: = b + a	
103: goto ___	Process this jump and add (103) to our worklist
104:	This is the first statement of the basic block (100) branches to. Go back and fill in the jump to 104.