ARQCP Course

Arquitetura de Computadores
Licenciatura em Engenharia Informática

2023/24

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Disclaimer

Material and Slides

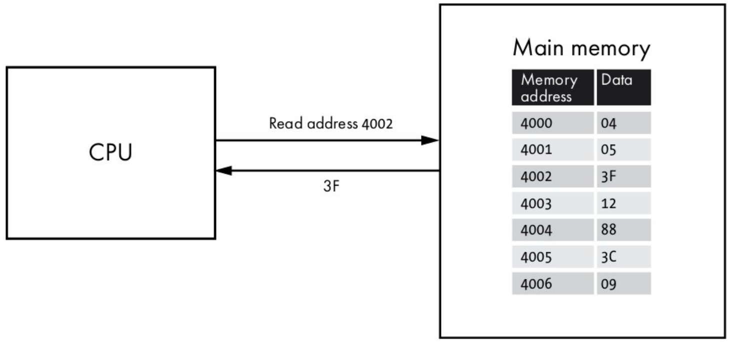
Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

Processor

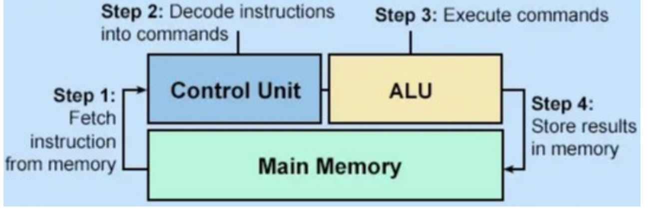
- The **central processing unit** (CPU), or simply processor, is the **engine that interprets (or executes) instructions stored in main memory**.



- At its core is a word-sized storage device (or register) called the **program counter (PC)**.
  - At any point in time, the **PC points at** (contains the address of) some **machine-language instruction in main memory**.

Processor (II)

- Instruction cycle
  - The **instruction cycle is the time required by the CPU to execute one single instruction**.
  - The **instruction cycle is the basic operation** of the CPU which consists on four steps:
    - **Fetch** the next instruction from memory
    - **Decode** the instruction just fetched
    - **Execute** this instruction as decoded
    - **Store** the result



  - At the end of each instruction cycle **CPU advances PC register**.

Instruction-Level Parallelism (IPL) (I)

- Optimizing requires the basic understanding of the **microarchitectures of processors**.
- At the code level, it appears **as if instructions are executed one at a time**, where each instruction involves fetching values from registers or memory, performing an operation, and storing results back to a register or memory location.
- In the actual processor, **a number of instructions are evaluated simultaneously**, a phenomenon referred to as **IPL**.
  - **Multiple instructions can be executed in parallel, while presenting an operational view of simple sequential instruction execution**.

IPL (II)

- **Superscalar** processors can perform multiple operations on every clock cycle, and **out-of-order**, meaning that the order in which instructions execute need not correspond to their ordering in the machine-level program.
- There is the need **to track the dependencies between instructions** to ensure that the out-of-order execution does not violate the program semantics.
- A dependency occurs **when an instruction reads or writes a register or memory location that is also accessed by another instruction**.

Coding

Main Objective

- "*The biggest speedup you'll ever get with a program will be when you first get it working*" – John K. Ousterhout
- The primary objective in writing a program **must be to make it work correctly under all possible conditions.**
  - A program that runs fast but gives incorrect results serves no useful purpose.
- Programmers **must write clear and concise code**, not only so that they can make sense of it, but also so that others can read and understand the code during code reviews and when modifications are required later.

Writing Code

- Writing an efficient program requires several types of activities.
  - Select **an appropriate set of algorithms and data structures**.
  - Write source code that **the compiler can effectively optimize** to turn into efficient executable code.
    - It is important **to understand the capabilities and limitations of optimizing compilers.**
    - Some features of C, such as the ability to perform **pointer arithmetic and casting, make it challenging** for a compiler to optimize.
    - Programmers **can often write their programs in ways that make it easier for compilers to generate efficient code.**
  - It **is to divide a task into portions that can be computed in parallel, on some combination of multiple cores and multiple processors**.
- In general, programmers must make **a trade-off between how easy a program is to implement and maintain, and how fast it runs**.

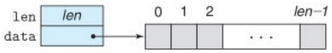
Optimizing Compilers

- Modern compilers employ sophisticated algorithms to determine what values are computed in a program and how they are used.
- They **can then exploit opportunities to simplify expressions, to use a single computation in several different places, and to reduce the number of times a given computation must be performed**.
- Most compilers provide optimizations options.
  - `gcc`
    - `-O1` : Turn on the most common optimizations
    - `-O2` : Provide maximum optimizations without increasing the executable size
    - `-O3` : Increase the speed of the resulting executable , and also increase its size
    - `-Os` : Select optimizations which reduce the size of an executable

Program Example

Initial Implementation

```c
#define IDENT 0
#define OP +
//#define IDENT 1
//#define OP *
typedef int data_t;
//typedef float data_t;
//typedef double data_t;
```




```c
typedef struct {
  long int len;
  data_t *data;
} vec_rec;

typedef vec_rec * vec_ptr;
```

```c
long int vec_length(vec_ptr v)
{
  return v->len;
}
```

```c
vec_ptr new_vec(long int len){
  /* Allocate header structure */
  vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
  if (!result)
    return NULL; /* Couldn't allocate storage */
  result->len = len;
  if (len > 0) { /* Allocate array */
    data_t *data = (data_t *)calloc(len,sizeof(data_t));
    if (!data) {
      free((void *) result);
      return NULL; /* Couldn't allocate storage */
    }
    result->data = data;
  }else
    result->data = NULL;
  return result;
}
```

```c
int get_vec_element(vec_ptr v, long int index, data_t *
      dest){
  if (index < 0 || index >= v->len)
    return 0;
  *dest = v->data[index];
  return 1;
}
```

Expressing Program Performance

- The **sequencing of activities by a processor is controlled by a clock providing a regular signal of some frequency**, usually expressed in gigahertz (GHz), billions of cycles per second.
  - For example, a **4 GHz** processor, it means that the processor clock runs at $4.0 * 10^{-9}$ cycles per second (4 cycles per nanosecond, which is 0.25 nanosecond per cycle or 250 picoseconds per cycle).
- From a programmer's perspective, it is more instructive to express measurements in **clock cycles** rather than nanoseconds or picoseconds.
  - Cycles Per Element (CPE) **it is used to express program performance** in a way that can guide us in improving the code.
    - CPE measurements help us understand the loop performance of an iterative program at a detailed level.

Implementation with maximum use of data abstraction

```
void combine1(vec_ptr v, data_t *dest)
{
  long int i;
  *dest = IDENT;
  for (i = 0; i < vec_length(v); i++) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
  }
}
```

		Integer		Floating point		
Function	Method	+	*	+	F *	D *
`combine1`	Abstract unoptimized	29.02	29.21	27.40	27.90	27.36
`combine1`	Abstract `-O1`	12.00	12.00	12.00	12.01	13.00

- CPE values: `Integer` addition (+) and multiplication (*), `Foating-point` addition (+), single-precision multiplication (labeled `F *`), and double-precision multiplication (labeled `D *`).

Eliminating Loop Inefficiencies (I)

- Observe that procedure `combine1` **calls function `vec_length` as the test condition of the for loop**.
- The **length of the vector does not change** as the loop proceeds.

```
void combine1(vec_ptr v, data_t *dest)
{
 long int i;
 *dest = IDENT;
 for (i = 0; i < vec_length(v); i++) {
   data_t val;
   get_vec_element(v, i, &val);
   *dest = *dest OP val;
 }
}
```

Eliminating Loop Inefficiencies (II)

- Loop-invariant code motion **is the process of moving loop-invariant code to a position outside the loop**, which may reduce the execution time of the loop by preventing some computations from being done twice for the same result.

```
void combine2(vec_ptr v, data_t *dest){
  long int i;
  long int length = vec_length(v);
  *dest = IDENT;
  for (i = 0; i < length; i++) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
  }
}
```

		Integer		Floating point		
Function	Method	+	*	+	F *	D *
combine1	Abstract -O1	12.00	12.00	12.00	12.01	13.00
combine2	Move vec_length	8.03	8.09	10.09	11.09	12.08

Reducing Procedure Calls

- **Invoking functions is costly**
  - Costs of call, return, arguments & result passing, stack frame maintenance
- Avoiding **function calls in the inner loop**.

```
data_t *get_vec_start(vec_ptr v){
 return v->data;
}
void combine3(vec_ptr v, data_t *dest){
 long int i;
 long int length = vec_length(v);
 data_t *data = get_vec_start(v);
 *dest = IDENT;
 for (i = 0; i < length; i++) {
   *dest = *dest OP data[i];
 }
}
```

		Integer		Floating point		
Function	Method	+	*	+	F *	D *
`combine2`	Move `vec_length`	8.03	8.09	10.09	11.09	12.08
`combine3`	Direct data access	6.01	8.01	10.01	11.01	12.02

Eliminating Unneeded Memory References

- Reading and writing of memory involves a set of operations
  - Using a temporary variable through **a register eliminates many read write memory operations**.

```
/* Accumulate result in local variable */
void combine4(vec_ptr v, data_t *dest)
{
 long int i;
 long int length = vec_length(v);
 data_t *data = get_vec_start(v);
 data_t acc = IDENT;
 for (i = 0; i < length; i++) {
  acc = acc OP data[i];
 }
 *dest = acc;
}
```

		Integer		Floating point		
Function	Method	+	*	+	F *	D *
combine3	Direct data access	6.01	8.01	10.01	11.01	12.02
combine4	Accumulate in temporary	2.00	3.00	3.00	4.00	5.00

- Loop unrolling is a program transformation that **reduces the number of iterations for a loop by increasing the number of elements computed on each**.

```
void combine5(vec_ptr v, data_t *dest){
  long int i;
  long int length = vec_length(v);
  long int limit = length-1;
  data_t *data = get_vec_start(v);
  data_t acc = IDENT;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    acc = (acc OP data[i]) OP data[i+1];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    acc = acc OP data[i];
  }
  *dest = acc;
}
```

		Integer		Floating point		
Function	Method	+	*	+	F *	D *
combine4	No unrolling	2.00	3.00	3.00	4.00	5.00
combine5	Unroll by ×2	2.00	1.50	3.00	4.00	5.00
	Unroll by ×3	1.00	1.00	3.00	4.00	5.00

- **Multiple accumulators for a combining operation that is associative and commutative**, such as integer addition or multiplication

```
void combine6(vec_ptr v, data_t *dest){
  long int i;
  long int length = vec_length(v);
  long int limit = length-1;
  data_t *data = get_vec_start(v);
  data_t acc0 = IDENT;
  data_t acc1 = IDENT;
  for (i = 0; i < limit; i+=2) {
    acc0 = acc0 OP data[i];
    acc1 = acc1 OP data[i+1];
  }
  for (; i < length; i++) {
    acc0 = acc0 OP data[i];
  }
  *dest = acc0 OP acc1;
}
```

		Integer		Floating point		
Function	Method	+	*	+	F *	D *
`combine4`	Accumulate in temporary	2.00	3.00	3.00	4.00	5.00
`combine5`	Unroll by ×2	2.00	1.50	3.00	4.00	5.00
`combine6`	Unroll ×2, parallelism ×2	1.50	1.50	1.50	2.00	2.50

Optimization

Performance

- Up to programmer **to write the best overall algorithm**
  - **Big-O** savings are (often) more important than constant factors
    - In computer science, Big-O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows.
- Must **optimize at multiple levels**: algorithm, data representations, procedures, loops.
- Must understand the system to **optimize performance**.
  - How programs are **compiled and executed**.
  - How **to measure program performance and identify**
  - **bottlenecks**.
  - How to improve performance **without destroying code modularity and generality**.

Generally useful optimizations

- Optimizations that you (or the compiler) should do regardless of processor/compiler
- The first step in optimizing a program is to eliminate unnecessary work
  - This includes **eliminating unnecessary function calls**, conditional tests, and **memory references**.
  - These optimizations do not depend on any specific properties of the target machine
- Requires **a fair amount of trial-and-error experimentation**
  - **Seemingly small changes can cause major changes in performance**, while **some very promising techniques prove ineffective**.

Efficient memory use

- Keep the architecture's default **memory alignment**
- **Use registers as much as possible**.
  - If **local variables are few enough**, rather than in the stack, they can instead be stored in registers
  - **Eliminate unneeded memory references**
- Pass structures by reference, not by value
- Make best use of cache (**locality**)
  - Access memory in increasing addresses order

Constant Folding

- **Constant folding is computation of constants at compile time.**
- Consider this code:

```c
int i = 2 + 3;
```

- Could `i` be anything but 5 after this line? Nope! Add it at compile time!
- In the real world, this could be harder to find.
  - Consider: **macros**, **named constants**, etc

Code motion

- **Reduce frequency with which computation is performed**
  - If it will always **produce same result**
  - Common example **is moving code out of loop**
- Loop-invariant code motion
  - Code motion that specifically moves redundant computations outside the scope of a loop without affecting program's semantics
  - Often, very beneficial since most execution happens in loops

```
for(i=0; i<N; i++){
  x = y + z;
  a[i] = 10*i + 2*x;
}
```

```
x = y + z;
t = 2 * x;
for(i=0; i<N; i++)
   a[i] = 10*i + t;
```

```
for(i=0; i<N; i++){
  for(j=0; j<M; j++)
    a[i][j] = 2*x + 10*i + 2*j;
}
```

```
t1 = 2 * x;
for(i=0; i<N; i++){
  t2 = t1 + 10 * i;
  for(j=0; j<M; j++)
    a[i][j] = t2 + 2*j;
}
```

Reduction in strength

- **Replace costly operations with simpler ones that still have an equivalent effect**
  - **Shift and add instead of multiply or divide** (utility is machine dependent – **depends on cost of multiply or divide instruction**)

```
x = w % 8;
y = x * 2;
z = y * 33;

for(i=0; i<MAX; i++){
  h = 14 * i;
  …
}
```

→

```
x = w & 7;
y = x << 1;
z = (y << 5) + y;

for(i=h=0; i<MAX; i++){
  h += 14;
  …
}
```

- **Array indexing** in C is basically **multiply and add**.
  - The multiply part can be subjected to strength reduction.

```
for(i=0; i<N; i++)
  for(j=0; j<M; j++)
    a[i][j] = b[j];
```

→

```
int mi = 0;
for(i=0; i<N; i++){
  for(j=0; j<M; j++)
    a[mi + j] = b[j];
  mi += M;
}
```

Share common sub-expressions

- **Reuse portions of expressions.**

```
sum1 = a + b + c;
sum2 = a + b + d;
sum3 = a + b + e;
```

```
tmp = a + b;
sum1 = tmp + c;
sum2 = tmp + d;
sum3 = tmp + e;
```

```
/* Sum neighbors of i,j */
up    = val[i-1][j];
down  = val[i+1][j];
left  = val[i][j-1];
right = val[i][+1];
sum   = up + down + left + right;
```

```
inj   = i*n + j;
up    = val[inj - n];
down  = val[inj + n];
left  = val[inj - 1];
right = val[inj + 1];
sum   = up + down + left + right;
```

Removing unnecessary function calls

- **Function calls incur overhead.**

```c
void lower (char *s){
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```c
void lower (char *s){
  int i;
  int len = strlen(s);
  for (i=0; i<len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

Compilers

Optimizing Compilers

- Provide efficient **mapping of program to machine**:
  - Register allocation
  - Code selection and ordering
  - Eliminating minor inefficiencies
- **Don't (usually) improve the computational complexity of algorithms**
  - The amount of time, storage, or other resources needed to execute them
- They **cannot improve algorithms**.
- They can also be fooled/tricked by certain constructions.
- Have difficulty overcoming **optimization blockers**
  - Potential memory aliasing
  - Potential procedure side-effects

Limitations of Optimizing Compilers

- Compilers operate under a fundamental constraint:
  - They must not cause **any change in program behavior under any possible condition**.
- Behavior that **may be obvious to the programmer can be obfuscated by languages and coding styles**
  - e.g., data ranges may be more limited than the variable type suggests.
- Most analysis **is performed only within functions**;
  - whole-program analysis is too expensive in most cases.
- Most analysis **is based only on static information**.
- **When in doubt, the compiler must be conservative**

Capabilities and Limitations of Optimizing Compilers

- Modern compilers employ sophisticated algorithms to determine **what values are computed in a program and how they are used**.
- They can then exploit **opportunities to simplify expressions**, to use a single computation in several different places, and to reduce the number of times a given computation must be performed.
- Compilers must be careful **to apply only safe optimizations to a program**
- The programmer **must make more of an effort to write programs in a way that the compiler can then transform into efficient machine-level code**.

Function calls

- Why are function calls problematic?
  - They might **have side effects** (alter global state, do I/O).
  - They might **not be deterministic**.
  - They might **modify pointers**.
- The compiler **must not change semantics**.
- Optimizations around **function calls are therefore weakened**.
- Why couldn't the compiler move `strlen` out of loop?

```c
void lower1(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

  - It cannot assume that `strlen` does not alter `s`.
  - It cannot assume that `strlen(s)` is always the same.
  - It treats `strlen()` as a **black box**.

Memory Aliasing (I)

```
void func1(int *xp, int *yp){
  *xp += *yp;
  *xp += *yp;
}
```

```
void func2(int *xp, int *yp){
  *xp += 2 * *yp;
}
```

- At first glance, **both functions seem to have identical behavior**
  - Function `func2` is more efficient
    - It requires only three memory references (read `*xp`, read `*yp`, write `*xp`)
    - `func1` requires six memory references (two reads of `*xp`, two reads of `*yp`, and two writes of `*xp`)

Memory Aliasing (II)

- Can the compiler **safely generate more efficient code based on the computations performed by `func2`**?
  - Consider the case in which `xp` and `yp` are equal (points to the same variable)
    - `func1` : `xp` will be increased by a factor of 4

```c
void func1(int *xp, int *yp){
  *xp += *yp; /* Double value at xp */
  *xp += *yp; /* Double value at xp */
}
```

    - `func2` : `xp` will be increased by a factor of 3

```c
void func2(int *xp, int *yp){
  *xp += 2 * *yp; /* Triple the value at xp */
}
```

  - The case where two pointers may designate the same memory location is known as **memory aliasing**.
  - **The compiler must assume that different pointers may be aliased**.