SENG440 Embedded Systems

– Lesson 3: Software Optimization Techniques I –

Mihai SIMA

msima@ece.uvic.ca

Academic Course

Copyright © 2019 Mihai SIMA

Disclaimer

The purpose of this course is to present general techniques and concepts for the analysis, design, and utilization of embedded systems. The requirements of any real embedded system can be intimately connected with the environment in which the embedded system is deployed. The presented design examples should not be used as the full design for any real embedded system.

Lesson 3: Software optimizations for embedded systems I

Lesson 3: Course Progress

- **Software optimization techniques**
  - What is wrong with plain software
  - Profile driven compilation
  - Efficient C programming
- **Standard peripherals for embedded systems**
  - **Timers, counters**, watchdog timers, real-time clocks
  - **Digital-to-Analog (D/A) and Analog-to-Digital (A/D) converters**
  - Pulse-Width Modulation (PWM) peripherals
  - Universal Asynchronous Receiver/Transmitters (UART)
- **Hardware – software – firmware.**
  - A taxonomy of processors

What's wrong with "plain" software?

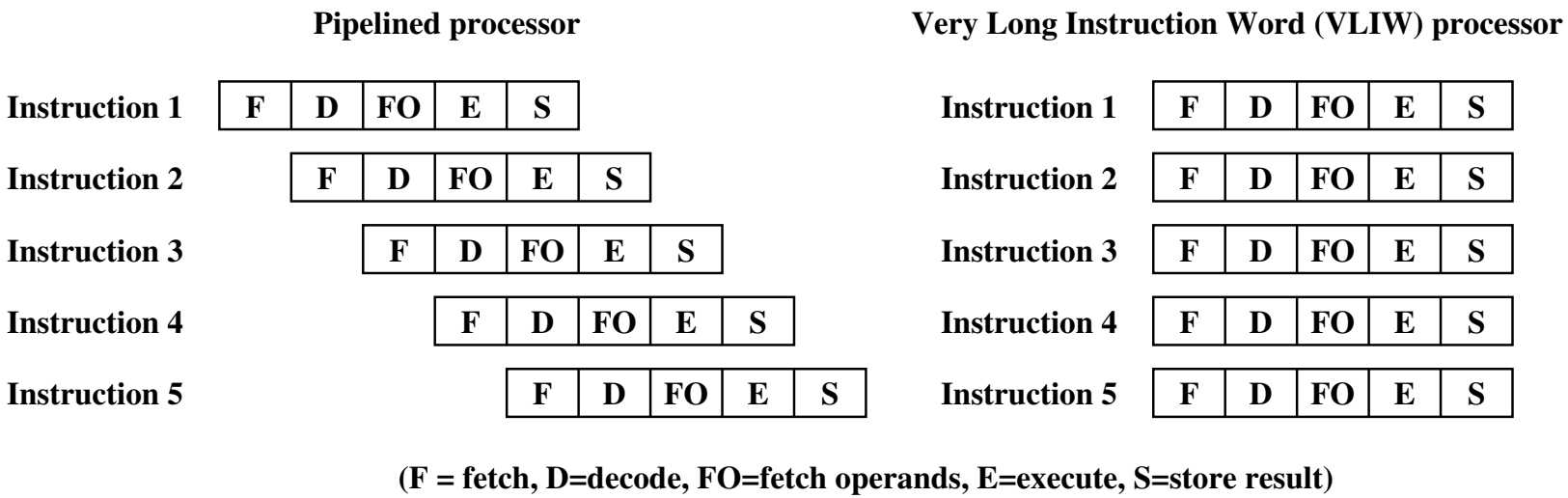
- Program and data memory space are often very limited in an embedded processor (64KB program, 256B data)
- As architectures diversify and become more complicated, the automatic optimization becomes more and more complicated
- For this reasons, the embedded programmer is typically exposed to the hardware itself and, therefore, has to optimize the code by hand
- There is still a long way till a compiler can generate an optimum code without human intervention
- Optimization requires strong knowledge of the processor, the algorithm, and the application
- Several optimization techniques are presented subsequently

Efficient C programming

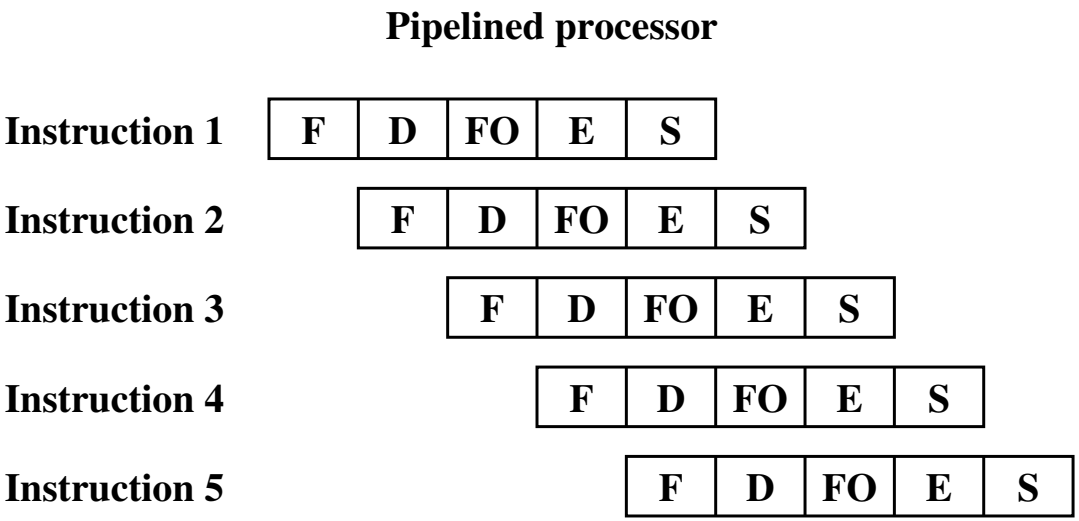
- **Goal**: write C code in a style that will compile efficiently on an *embedded system* architecture
- Once you have the feel for this translation process, you can distinguish:
  - Fast C code from slow C code
  - Small code size from large code size
- Most programming techniques concern parallel architectures
- To write efficient C code, the programmer must be aware of:
  - Where the C compiler has to be conservative
  - The limits of the processor architecture the C compiler is mapping to
  - The limits of a specific C compiler
- Be aware of portability issues!

What processor class is of interest?

- If our processor is really weak (cannot perform complex operations, it is non-pipelined, and can launch only one instruction per cycle) then we don't have much to do / optimize
- But if the processor is a parallel computing engine (see below), then we may try to exploit this parallelism by writing better (high-level) code



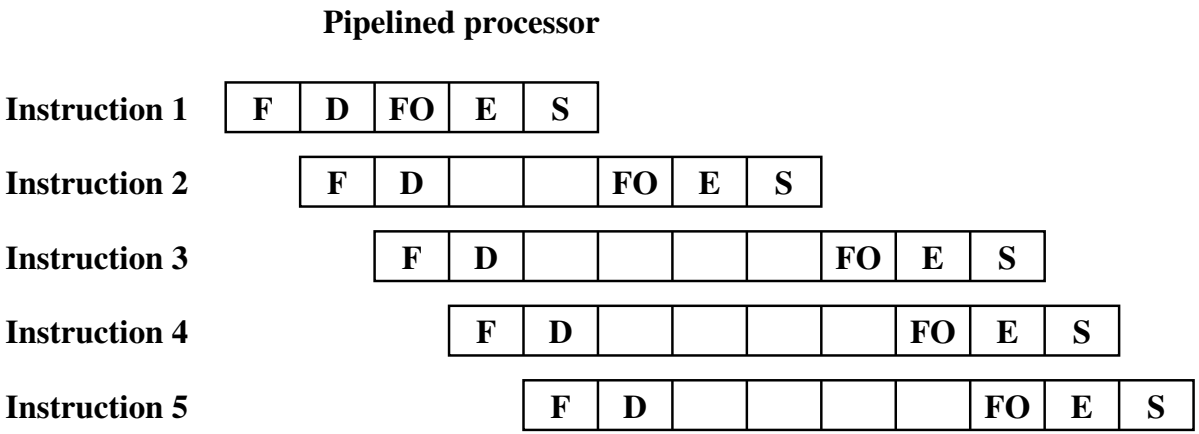
Pipelined Processors – (I)



(F = fetch, D=decode, FO=fetch operands, E=execute, S=store result)

- There are no **true dependencies** in the code
- The pipeline is filled in with operations (or sub-operations)
- The machine is running at full speed

Pipelined Processors – (II)

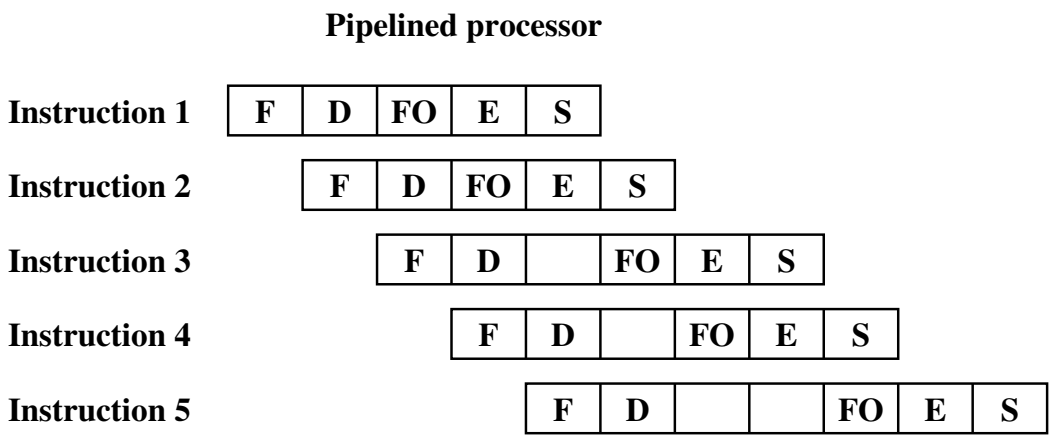


(F = fetch, D=decode, FO=fetch operands, E=execute, S=store result)

- The code is riddled with **true dependencies** (or **branches**)
- *Bubbles* are injected into the pipeline
- The machine is running at slower speed

Pipelined Processors – (III)

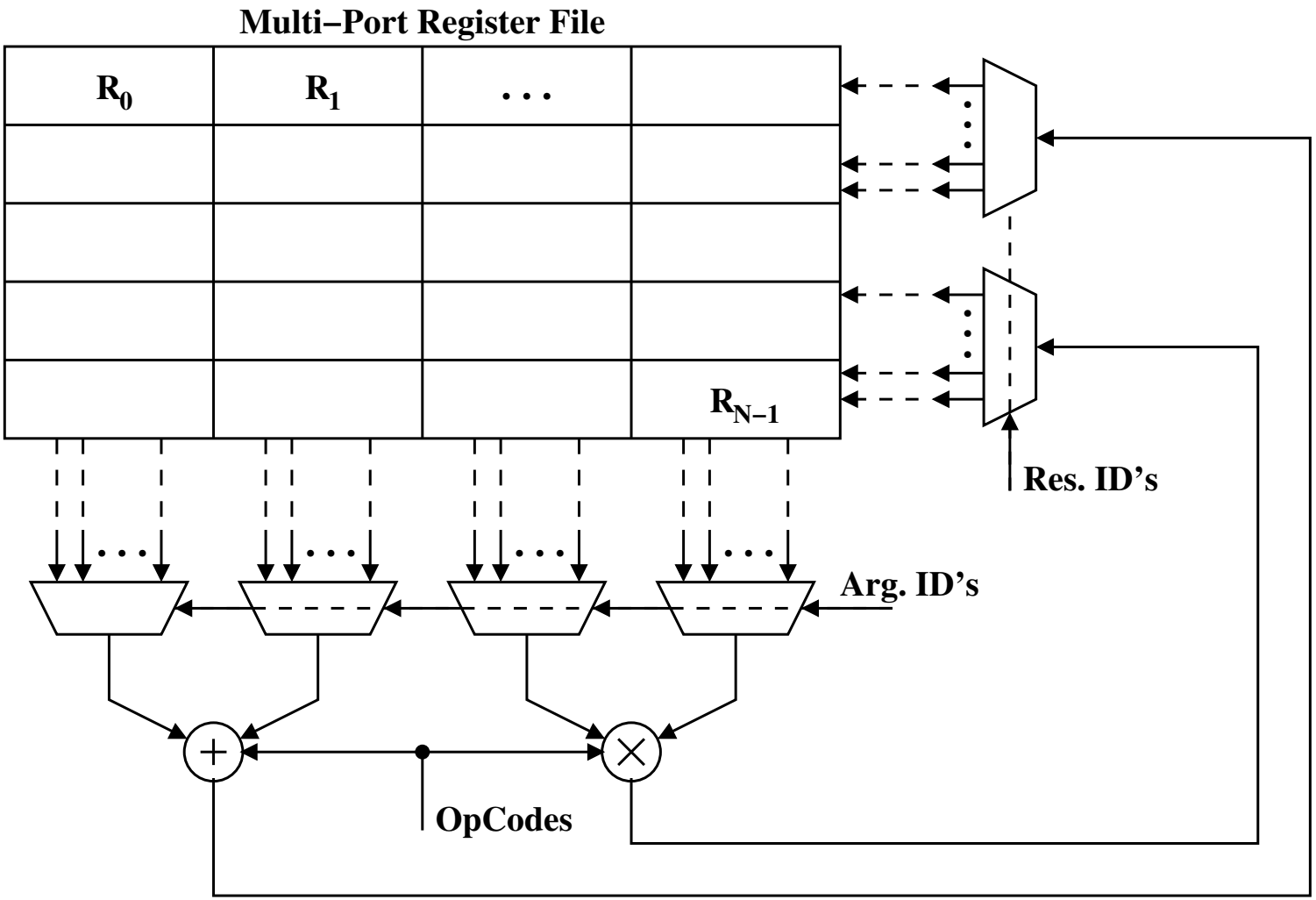
- Reordering the instructions increases speed
- Out-of-order issuing machines perform reordering at run time (Pentium is an out-of-order issuing machine)
- In-order issuing machines need compiler and/or programmer support (ARM is an in-order issuing machine)



OpCode	C
ADD	c = a + b ;
SHIFT	j = k >> 2 ;
MUL	d = c * f ;
DIV	l = m / n ;
SUB	g = d – i ;

(F = fetch, D=decode, FO=fetch operands, E=execute, S=store result)

'Horizontal' (VLIW) Processor – (I)



'Horizontal' (VLIW) Processor – (II)

- The assembly-level programmer is responsible to specify two (or more, depending on the number of the issue slots) operations per instruction
- The VLIW toolchain includes a *Scheduler*, which determines the true dependencies in the code, and groups multiple operations per instruction
- True dependencies generate a situation similar to pipelined processors
- If the number of true dependencies is large, then many of these operations are NOPs (No OPeration)
- Scheduling branch operations is critical for parallel processors
- Operation reordering aims to reduce the number of NOPs

'Horizontal' (VLIW) Processor – (III)

- The assembly-level programmer is responsible to specify two (or more, depending on the number of the issue slots) operations per instruction
- The VLIW toolchain includes a *Scheduler*, which determines the true dependencies in the code, and groups multiple operations per instruction
- True dependencies generate a situation similar to pipelined processors
- If the number of true dependencies is large, then many of these operations are NOPs (No OPeration)
- Scheduling branch operations is critical for parallel processors
- Operation reordering aims to reduce the number of NOPs

- **What should the high-level programmer do?**

Profile-driven compilation

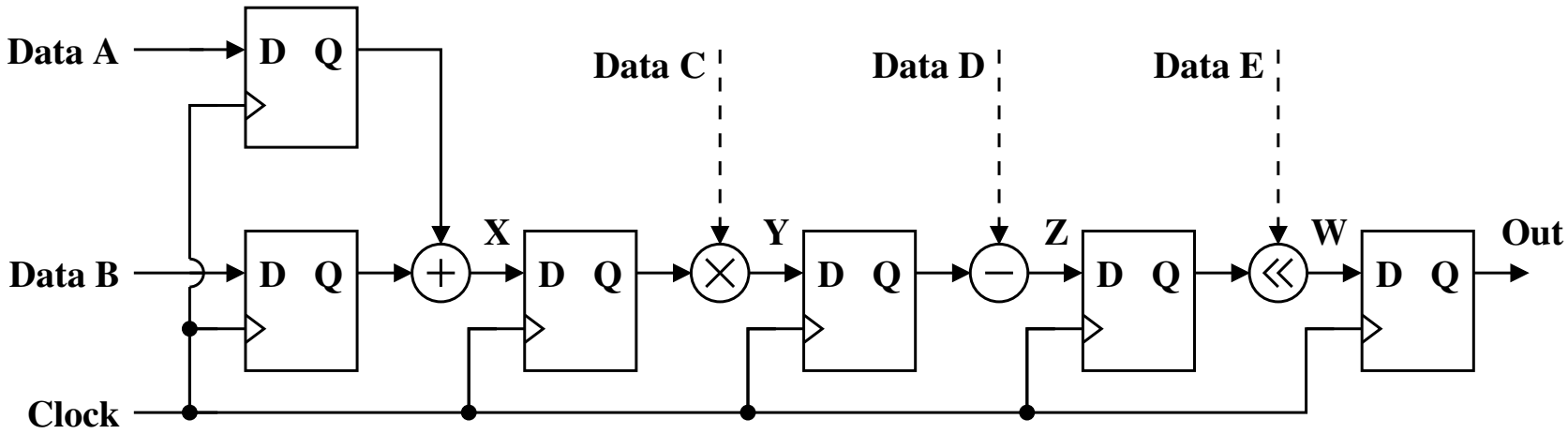
- **Compile–profile–recompile** cycle of performance tuning
- Why do we need to do so?
  - Loop unrolling and grafting use execution frequencies and branch probabilities (JUMP is an expensive operation in terms of its latency, so predicting its outcome helps in improving ILP)
  - Function inlining uses execution frequencies at the call site
  - Predicate execution uses profile information (for if conversion)
  - The optimization can address the code size or speed
- Basic idea:
  1. Obtain profile information about the program
  2. Identify the most frequently executed parts of the program
  3. Use optimization technics like loop unrolling, grafting, and function inlining

Profile-driven compilation flow-chart

- **gcc** is used as an ilustration platform Read the manual!
- The basic steps are mostly the same for other platforms
- The compile-profile-recompile sequence:
  1. Generate the executable file with profiling turned on:
     $ gcc -pg example.c
  2. Run the executable to produce the profile information (**gmon.out** file):
     $ ./a.out
  3. Read the profile information:
     $ gprof
  4. Analyse the profile information and update the source file **example.c**
  5. Recompile the source file with profiling turned off:
     $ gcc example.c

Hardware pipeline – (I)

- Many computing kernels (especially the data-dominant ones as well as the loops) can be implemented in hardware as a pipeline:



- Vertical machine: only one pipeline stage is executed at any given time
- **Question**: how to implement such a pipeline in software on a horizontal/pipelined machine?

Hardware pipeline – (II)

- A pipeline is essentially a (long) string of true dependencies
- The straightforward mapping of a hardware pipeline (or of an algorithm comprising a string of true dependencies) into software does **not** benefit from parallel computing cabilities of a processor
- **The emulation of a hardware pipeline in software is difficult**
- Instruction Reordering is needed: this is called **Software Pipelining**
- Instruction reordering can be done by either an optimizing compiler or the programmer

Software pipelining

```
vecdiv( float a[], int size) {
  int i; float a_temp;

  for( i=0; i < size; i++) {
    /* loop kernel */
    a_temp = a[i];
    a[i] = a_temp / 3.14;
  } /* JMP */
}
```

- Optimizations difficult across jumps
- Sequential code: LOAD then DIVIDE

Software pipelining

```
vecdiv( float a[], int size) {
  int i; float a_temp;

  for( i=0; i < size; i++) {
    /* loop kernel */
    a_temp = a[i];
    a[i] = a_temp / 3.14;
  } /* JMP */
}
```

- Optimizations difficult across jumps
- Sequential code: LOAD then DIVIDE
- Eliminate true dependency and expose parallelism to the compiler

```
vecdiv( float a[], int size) {
  int i; float a_temp;

  /* loop prologue */
  a_temp = a[0];
  for( i=0; i < size-1; i++) {
    /* loop kernel */
    a[i] = a_temp / 3.14;
    a_temp = a[i+1];
  } /* JMP */
  /* loop epilogue */
  a[size-1] = a_temp / 3.14;
}
```

With the exception of prologue (firing-up code) and epilogue (flushing-out code) the code is parallel: DIVIDE and LOAD

What is software pipelining

- Operations from different iterations are executed in parallel
- Each loop iteration uses intermediate results generated by the previous iteration and performs operations whose intermediate results will be used in the next iteration
- It exploits the Instruction-Level Parallelism present across loop iterations
- The deeper the hardware pipeline and longer the operations latency, the more likely it is that software pipelining will be necessary
- Main advantage: it increases performance by making use of instruction slots that would otherwise be wasted waiting for results
- Main disadvantages:
  - It makes programs more complicated, harder to read and maintain
  - Increased pressure on Register File

Software pipelining: pressure on Register File

```
int a[64];
void  vecdiv_1( int a[]) {
  register int i;
  int a_temp;
  for( i=0; i<64; i++) {
    a_temp = a[i];
    a[i] = a_temp + 7;
  }
}

void vecdiv_2( int a[]) {
  register int i;
  for( i=0; i<64; i++) {
    a[i] = a[i] + 7;
  }
}
```

```
void vecdiv_3( int a[]) {
  register int i;
  int a_temp;

  a_temp = a[0];          // prologue
  for( i=0; i<63; i++) {
    a[i] = a_temp + 7;    // software
    a_temp = a[i+1];      // pipelining
  }
  a[63] = a_temp + 7;     // epilogue
}
```

Notive that the variable `a_temp` is not defined in `vecdiv_2`

arm-linux-gcc -O3 -S source.c

Software pipelining: pressure on Register File

```
vecdiv_1:
        mov     r2, #0
.L2:
        ldr     r3, [r0, r2]
        add     r3, r3, #7
        str     r3, [r0, r2]
        add     r2, r2, #4
        cmp     r2, #256
        bne     .L2
        bx      lr
```

```
vecdiv_2:
        mov     r2, #0
.L7:
        ldr     r3, [r0, r2]
        add     r3, r3, #7
        str     r3, [r0, r2]
        add     r2, r2, #4
        cmp     r2, #256
        bne     .L7
        bx      lr
```

The routines `vecdiv_1` and `vecdiv_2` compile into identical assembly

Software pipelining: pressure on Register File

```
vecdiv_3:
        mov     r2, r0
        ldr     r0, [r0, #0]
        add     r1, r2, #252
.L11:
        add     r3, r0, #7
        str     r3, [r2, #0]
        ldr     r0, [r2, #4]!
        cmp     r1, r2
        bne     .L11
        add     r3, r0, #7
        str     r3, [r1, #0]
        bx      lr
```

The `vecdiv_3` routine uses four registers (r0, r1, r2, and r3)

Each of the `vecdiv_1` and `vecdiv_2` routines use three registers (r0, r2, r3)

Since a software pipelined routine requires more registers to store temporary variables, it puts a pressure on the register file

Loop unrolling

```
vecdiv( float a[], int size) {
  int i; float a_temp;

  for( i=0; i < size; i++) {
    /* loop kernel */
    a_temp = a[i];
    a[i] = a_temp / 3.14;
  } /* JMP */
}
```

```
vecdiv( float a[], int size) {
  int i; float a1_temp, a2_temp;

  for( i=0; i < size; i+=2) {
    /* loop kernel */
    a1_temp = a[i];
    a2_temp = a[i+1];
    a[i] = a1_temp / 3.14;
    a[i+1] = a2_temp / 3.14;
  } /* JMP */
}
```

- Somehow similar to software pipelining
- Repetition of loop-body instructions within a single loop iteration
- Drawback: it increases register and memory usage

Software pipelining versus loop unrolling

- Software pipelining can be thought of infinite loop unrolling
- Software pipelining generates lower code size than loop unrolling
- Loop unrolling code size increases linearly with the degree of loop unrolling, which may in turn increase the instruction cache misses
- Real-time response cannot be controlled in software pipelining
- Real-time response decreases linearly with the degree of loop unrolling
- In loop unrolling the programmer should trade-off the amount of parallelism obtained by unrolling the loop against code size and real-time response
- A software pipelined loop is significantly more difficult to generate than an unrolled loop

Interruptible and non-interruptible loops

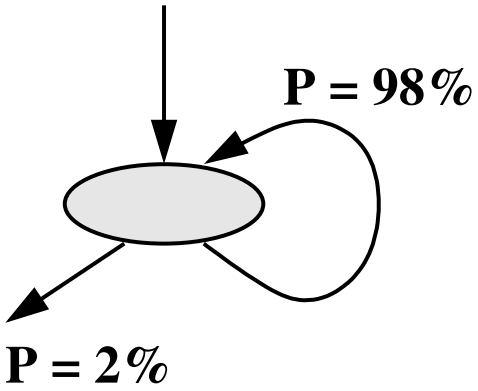
- Problem definition
  - Two types of JUMP operations: **interruptible** and **non-interruptible**
  - An event (e.g., a pending interrupt) is handled only when an interruptible jump is encountered
  - If an interruptible jump occurs too seldom, the response is not real-time
  - Since interrupting a software pipeline generates overhead (prologue and epilogue), interruptible jumps cannot be used in software-pipelined loops
- Trade-off is needed
- It is the programmer responsibility to force a jump to be compiled into an interruptible or non-interruptible jump.
- A way to do that is by using pragmas (read the C manual of the processor)
- A common pragma is **ATOMIC**: the next function cannot be interrupted

Interruptible loop code – an example

```
#pragma ATOMIC
vecdiv( float a[], int size) {
  int i; float a_temp;

  /* loop prologue */
  a_temp = a[0];

  for( i=0; i < size-1; i++) {
    /* loop kernel */
    a[i] = a_temp / 3.14;
    a_temp = a[i+1];
  } /* to be translated into non-interruptible JUMP */

  /* loop epilogue */
  a[size-1] = a_temp / 3.14;
} /* to be translated into interruptible JUMP */
```

Grafting

- Code replication technique that eliminates branches
- Conditional branches inject *bubbles* into the processor's pipeline
- Very similar to loop unrolling
- Useful when a branch is known as likely to be taken

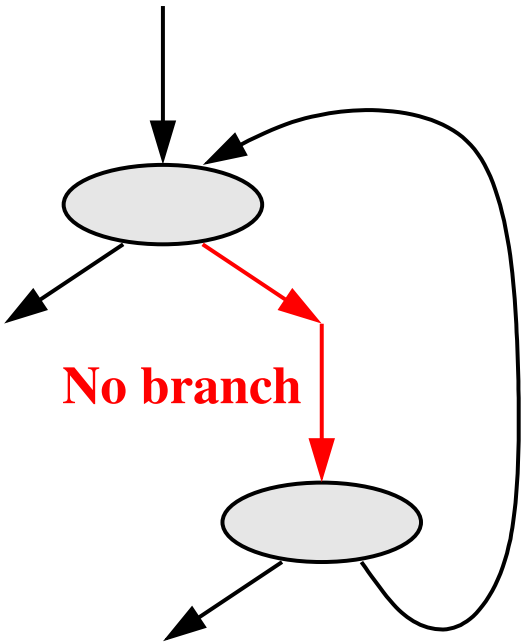


Original routine

```
for( i=0; i<50; i++) {

  temp = a[i];
  temp = temp + (1 << 3);
  temp = temp >> 2;
  a[i] = temp;

} /* Conditional JUMP back to for */
```

Grafting – replicate the code to eliminate branches

- Very similar to loop unrolling, but performed for a different reason



Grafted routine

```
for( i=0; i<25; i+=2) {

  temp = a[i];
  temp = temp + (1 << 3);
  temp = temp >> 2;
  a[i] = temp;
  temp = a[i+1];
  temp = temp + (1 << 3);
  temp = temp >> 2;
  a[i+1] = temp;

} /* Conditional JUMP back to for */
```

Operator strength reduction

- Replacement of a costly operator by a less expensive one
- Two procedures to normalize a vector
- Left: individual elements are divided by the sum of values
- Right: the division is replaced by a multiplication by the reciprocal
- Typical latency figures: ADD = 1 cycle, MULT = 3 cycles, DIV = 10 cycles
- Evaluate the reduction in total execution time for a 100-element vector.

```
norm( float *a, int size) {
  int i;
  float sum = 0.0;
  for( i=0; i < size; i++)
    sum += a[i];
  for( i=0; i < size; i++)
    a[i] /= sum;
}
```

```
norm( float *a, int size) {
  int i;
  float sum = 0.0, invsum;
  for( i=0; i < size; i++)
    sum += a[i];
  invsum = 1.0 / sum;
  for( i=0; i < size; i++)
    a[i] *= invsum;
}
```

Move externals and reference parameters to locals

- Copying an external variable to a local variable can substantially improve the performance in time-critical parts of the code.
- Two procedures to calculate the sum of the array elements.
- The sum is to be contained in the external variable **result**.
- The local variable **local_result** is used instead of the external variable **result**. This way, a register is accessed instead of a memory location.
- It is not possible to allocate external variables to registers and require memory references.
- Evaluate the reduction in the total execution time.

Move externals and reference parameters to locals (cont'd)

```
int result;

int sum( int nargs, int *p) {
  int i;

/*initialize the global result*/
  result = 0;

/*add each vector element*/
  for( i=0; i < nargs; i++)
    result += p[i];

/*return the result*/
  return result;
}
```

```
int result;

int sum( int nargs, int *p) {
  int i;
  register int local_result;

/*initialize the local result*/
  local_result = 0;

/*add each vector element*/
  for( i=0; i < nargs; i++)
    local_result += p[i];

/*return the result*/
  return( result = local_result);
}
```

Remove function calls

- Calculate the square of the distance from the origin for a set of points
- Left: grafting does not cross function call boundaries. It's useless
- Automated function inlining by using compilation arguments (**-O3** for **gcc**)
- Specific function inlining can be achieved by using pragmas
- Other solution: hand inlining by defining macros (right program)

```
float hypot( float x, float y) {
  return x*x + y*y;
}
main() {
  float x[100], y[100], rad[100];
  int i;
  for( i=0; i < 100; i++)
    rad[i] = hypot( x[i], y[i]);
}
```

```
#define hypot(x,y) (x)*(x)+(y)*(y)

main() {
  float x[100], y[100], rad[100];
  int i;
  for( i=0; i < 100; i++)
    rad[i] = hypot( x[i], y[i]);
}
```

Writting time- or size-critical code in assembly

- We get optimized code at the expense of high development effort
- For calling assembly routines from high-level code, the programmer has to be aware of a number of conventions
- **Argument passing**: in standard C, all arguments are passed by value
- **Register usage convention** – an example:
  - R2 stores the return pointer (return address)
  - R3 stores the stack pointer
  - R4 stores the return value
  - R5...R10 store the arguments
  - R11...R32 comprise the register pool

Writting time- or size-critical code in assembly (cont'd)

- **Callee versus caller saved registers**
  - Compiled C functions need a register working set
  - Any register might already be allocated for use by the calling function
  - Convention between callers and callees must exist to ensure that none of the caller's values are lost during the call
  - Callee saving = the content of each register used in a function is saved and restored by the function itself
  - Caller saving = the content of each register used in a function is saved and restored by the caller itself
- The compiler databook of the particular processor you are using should describe the above conventions

Specific Boolean expressions in critical parts of the code

- Used in conjunction with guarding operations (predicate execution)
- Guarding uses a flag or the least-significant (guard) bit of a register to determine whether a condition is true/false: **IF Rguard ADD Rs1, Rs2, Rt**
- Predicate execution avoids the injection of bubbles into the pipeline
- Basic principle:
  - In C, true and false are represented by non-zero and zero, respectively
  - Conditional jump is expensive (comparison + jump itself)
  - If the value of the expression is known to depend only on the least-significant bit, then the comparison is not necessary
  - The compiler may recognize specific constructions such as:

    (E & 1) ! (E & 1)
  - Better code can be generated if these constructions are used instead of:

    (E != 1) (E == 1)
  - The following constructions may also be recognized:

    (E & 2^n) ! (E & 2^n)

Dirty float

- Compiler optimizations on floating-point expressions are illegal Why?
- Commutative and associative properties that hold for integer operations such addition and multiplication do NOT hold for floating-point operations
- Example: let us assume a a computer system where the number representation is normalized 4-digit mantissa and 2-digit exponent
  - Consider the following calculations, assuming rounding is used in the 4-digit arithmetic registers,
    1. $p+q+r+s=20$
    2. $p+q+s+r=26$
    3. $q+r+p+s=30$

    where $p=75640$, $q=4$, $r=6$, $s=-75620$
  - The answer in the next slide
- The programmer can give the compiler more freedom in expression optimizations by using compilation flags or pragmas; however, this might cause incorrect results

Dirty float (cont'd)

- The floating point representations are: $p = 0.7564 \times 10^5$, $q = 0.4000 \times 10^1$, $r = 0.6000 \times 10^1$, $s = -0.7562 \times 10^5$
- $p + q + r + s$

$p+q$	0.7564	$\times$	10^5	0.7564	$\times$	10^5	
	0.4000	$\times$	10^1	0.0000	$\times$	10^5	(matching exponent)
				0.7564	$\times$	10^5	
$p+q+r$	0.7564	$\times$	10^5	0.7564	$\times$	10^5	
	0.6000	$\times$	10^1	0.0000	$\times$	10^5	(matching exponent)
				0.7564	$\times$	10^5	
$p+q+r+s$	0.7564	$\times$	10^5	0.7564	$\times$	10^5	
	- 0.7562	$\times$	10^5	- 0.7562	$\times$	10^5	
				0.0002	$\times$	10^5	= 20

Dirty float (cont'd)

- The floating point representations of the numbers are:
  $p = 0.7564 \times 10^5$, $q = 0.4000 \times 10^1$, $r = 0.6000 \times 10^1$,
  $s = -0.7562 \times 10^5$
- $p + q + s + r$

$p+q$	0.7564	$\times$	10^5	0.7564	$\times$	10^5	
	0.4000	$\times$	10^1	0.0000	$\times$	10^5	(matching exponent)
				0.7564	$\times$	10^5	
$p+q+s$	0.7564	$\times$	10^5	0.7564	$\times$	10^5	
	- 0.7562	$\times$	10^5	- 0.7562	$\times$	10^5	
				0.0002	$\times$	10^5	$= 0.2000 \times 10^2$
$p+q+s+r$	0.2000	$\times$	10^2	0.2000	$\times$	10^2	
	0.6000	$\times$	10^1	0.0600	$\times$	10^2	(matching exponent)
				0.2600	$\times$	10^2	$= 26$

Dirty float (cont'd)

- The floating point representations of the numbers are:
  $p = 0.7564 \times 10^5$, $q = 0.4000 \times 10^1$, $r = 0.6000 \times 10^1$,
  $s = -0.7562 \times 10^5$
- $q + r + p + s$

$q+r$	0.4000	$\times$	10^1	0.4000	$\times$	10^1	
	0.6000	$\times$	10^1	0.6000	$\times$	10^1	
				1.0000	$\times$	10^1	$= 0.1000 \times 10^2$
$q+r+p$	0.1000	$\times$	10^2	0.0001	$\times$	10^5	(matching exponent)
	0.7564	$\times$	10^5	0.7564	$\times$	10^5	
				0.7565	$\times$	10^5	
$q+r+p+s$	0.7565	$\times$	10^5	0.7565	$\times$	10^5	
	- 0.7562	$\times$	10^5	- 0.7562	$\times$	10^5	
				0.0003	$\times$	10^5	$= 30$

What we learned

- Embedded programmer is typically exposed to the hardware itself
- Handcrafted code is common in embedded systems
- A number of software optimization technics for embedded systems
- What a C handbook should contain

Questions, feedback

Notes I