

Compilers for Embedded Systems

Summer Term 2022

Heiko Falk

Institute of Embedded Systems
Electrical Engineering, Computer Science and Mathematics
Hamburg University of Technology

Chapter 2

Compilers for Embedded Systems

—

Requirements & Dependencies

Outline

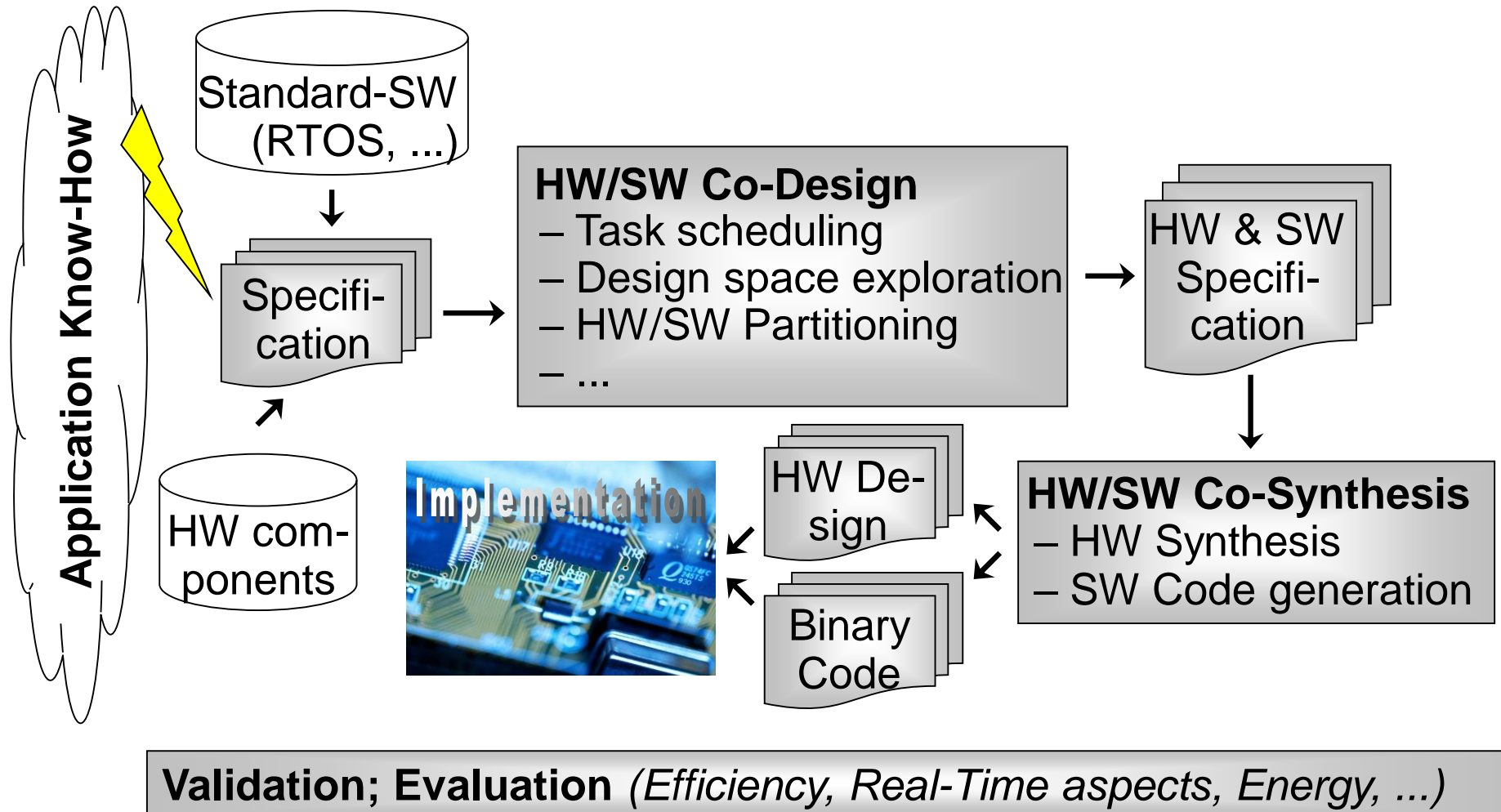
1. Introduction & Motivation
- 2. Compilers for Embedded Systems – Requirements & Dependencies**
3. Internal Structure of Compilers
4. Pre-Pass Optimizations
5. HIR Optimizations and Transformations
6. Code Generation
7. LIR Optimizations and Transformations
8. Register Allocation
9. WCET-Aware Compilation
10. Outlook

Chapter Contents

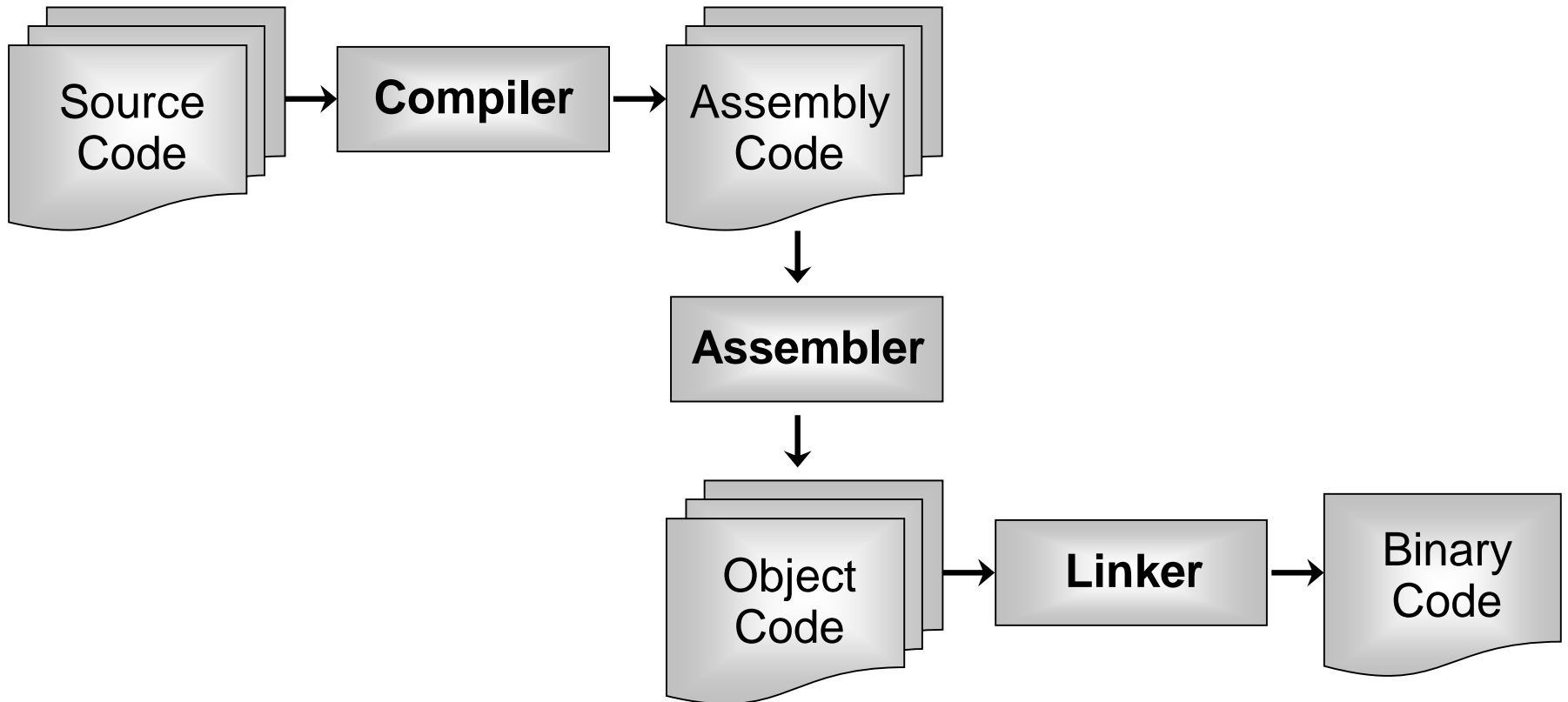
2. Compilers for Embedded Systems – Requirements & Dependencies

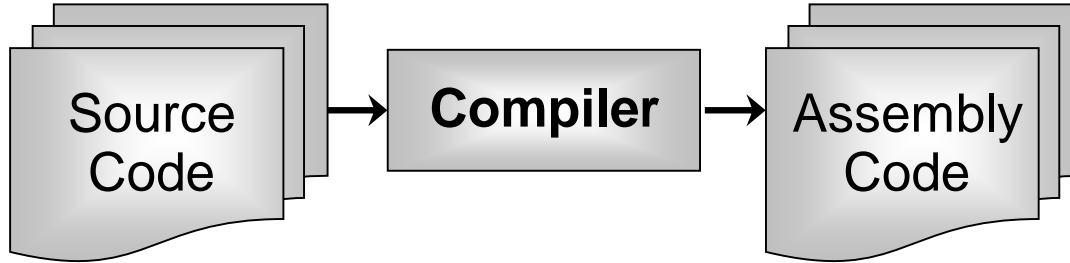
- Code Generation Tools
 - Compiler, Assembler, Linker
 - Source Code, Assembly Code, Object Code, Binary Code
- Source Languages for Compilers for Embedded Systems
 - C, C++, Java
- Embedded Processors
 - Digital Signal Processors
 - Multimedia Processors
 - *Very Long Instruction Word* Machines
 - Network Processors
- Requirements on Compilers for Embedded Systems
 - Code Quality vs. Compilation Times

Design Process of Embedded Systems



Code Generation Tools



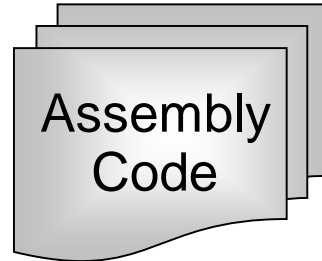


Source Code

- Programming language, legible and comprehensible for humans
- Standard constructs: Loops, procedures, variables, ...
- High abstraction level: Machine-independent algorithms

Assembly Code

- Symbolic machine code
- Limited legibility and comprehensibility for humans
- Machine language constructs: ALU operations, registers, ...
- Low abstraction level: Machine-specific representation



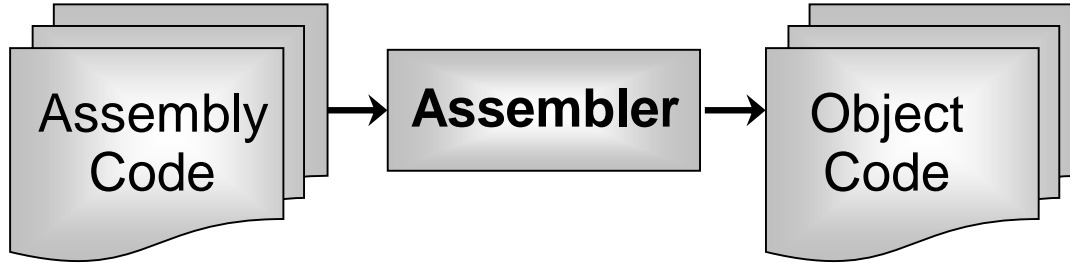
```
.align      1
.global    encode
.type      encode,@function
```

- Readable plain text
- No / few real addresses
- Instead: Symbolic addresses
e.g., `encode`, `h`, `tqmf`

```
encode:
```

```
mov      %d15, %d5
mov      %d12, %d4
movh.a  %a12, HI:h
lea      %a12, [%a12] LO:h
movh.a  %a13, HI:tqmf
lea      %a13, [%a13] LO:tqmf
ld.w    %d14, [%a13] 4
ld.w    %d10, [%a12] 4
mul      %d14, %d10
```

- # Load address of array `h` to A12*
- # Load address of array `tqmf` to A13*
- # Load `tqmf[1]` to D14*
- # Load `h[1]` to D10*
- # Multiply*

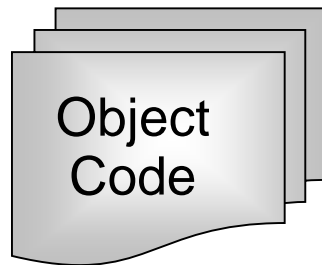


Object Code

- Binary representation of assembly code, no longer legible
- No plain text mnemonics, 0/1 bit sequences instead
- Whenever possible, symbolic addresses replaced by real ones

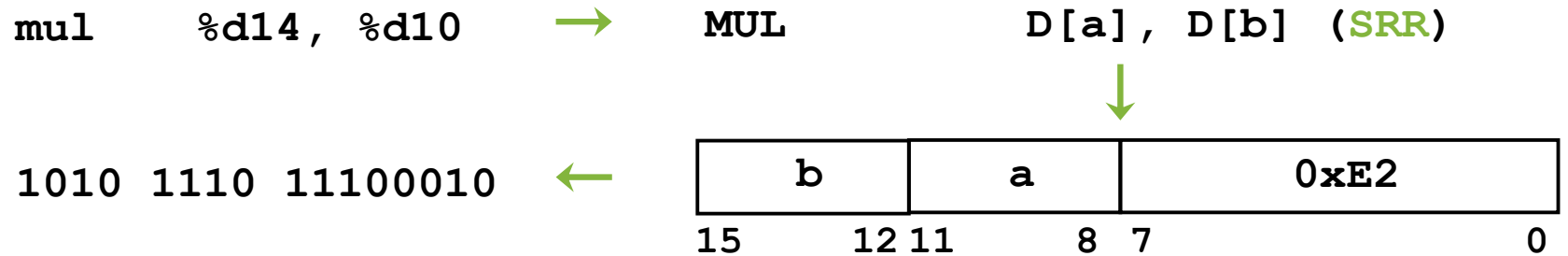
Assembler

- Line-by-line translation
Assembly instructions → machine instructions
- Within a single assembly file: Address Resolution



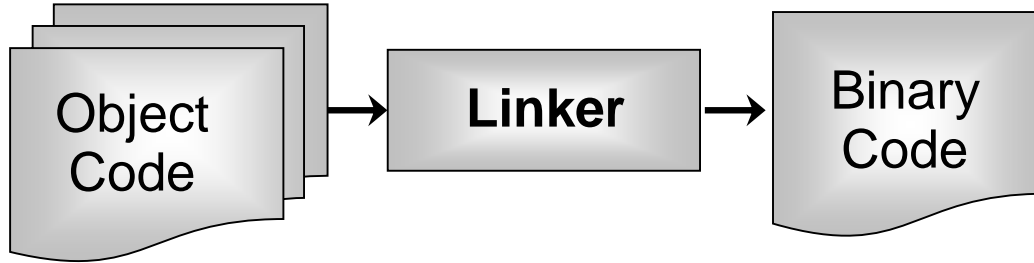
Translation

(Example: Infineon TriCore 1.3)



Address Resolution

- Symbolic address `h` declared in very same assembly file:
 - Symbol `h` is known to assembler
 - Replacement of `h` by relative address, relative within the object file
- `h` is unknown to assembler:
 - Defer address resolution at a later point in time

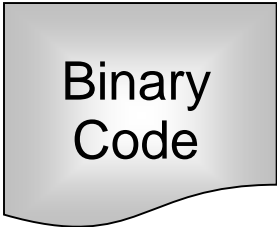


Binary Code

- Executable program representation
- All symbolic addresses replaced by real addresses
- Lowest possible abstraction level

Linker

- Union of many object files and libraries to a single, executable program
- Address resolution using libraries of object code
- Layout of code in memory

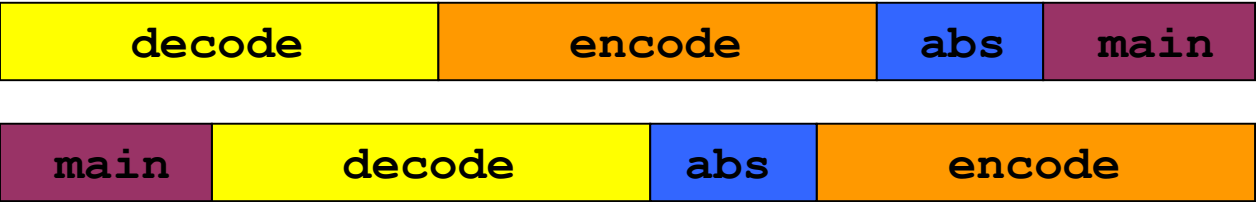


Example Address Resolution

- Object code contains jump to external function: `call abs`
- Search `abs` in all other object files and libraries
- Add code of `abs` to binary executable file

Example Memory Layout of Binary Code

- Binary code consists of functions `decode`, `encode`, `abs`, `main`



- Memory layout finally defines effective addresses

Chapter Contents

2. Compilers for Embedded Systems – Requirements & Dependencies

- Code Generation Tools
 - Compiler, Assembler, Linker
 - Source Code, Assembly Code, Object Code, Binary Code
- Source Languages for Compilers for Embedded Systems
 - C, C++, Java
- Embedded Processors
 - Digital Signal Processors
 - Multimedia Processors
 - *Very Long Instruction Word* Machines
 - Network Processors
- Requirements on Compilers for Embedded Systems
 - Code Quality vs. Compilation Times

Source Languages for Compilers for Embedded Systems

In the following

- Brief digest of most common programming languages
- No claim to be complete!

Imperative Programming Languages

- C

Object-Oriented Programming Languages

- C++
- Java

ANSI-C: Properties (1)

– Purely Imperative

- No object-orientation: No classes, no objects
- C-Program: Set of functions
- Function `main`: Standardized starting point
- Functions: Sequence of statements, execution in sequential order

```
int filtep( int r1t1, int a11, int r1t2, int a12 )
{
    long p1, p12;
    p1 = 2 * r1t1;
    p1 = (long) a11 * p1;
    p12 = 2 * r1t2;
    p1 += (long) a12 * p12;
    return( (int)(p1 >> 15) );
}
```

ANSI-C: Properties (2)

- **Standardized Programming Language**
 - ISO/IEC 9899:1999 (E)

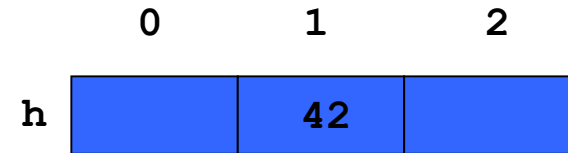
- **Standard Data Types**
 - `signed / unsigned char`
 - `signed / unsigned short`
 - `signed / unsigned int`
 - `signed / unsigned long`
 - `signed / unsigned long long`
 - `float, double, long double, _Bool`

ANSI-C: Properties (3)

– Composed Data Types

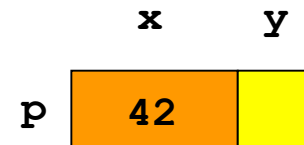
– Arrays

```
int h[ 3 ];
h[ 1 ] = 42;
```



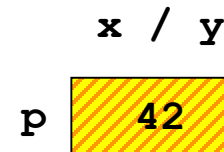
– Structures

```
struct point { int x; char y; } p;
p.x = 42;
```



– Variants

```
union point { int x; char y; } p;
p.y = 42;
```

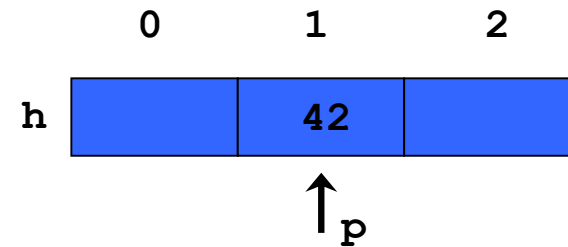


ANSI-C: Properties (4)

– Pointers and Memory Management

– Pointers

```
int h[ 3 ];  
int *p = &h[ 1 ];  
h[ 1 ] = 42;  
*p = 12;
```

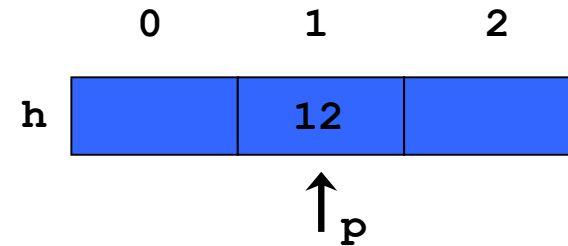


ANSI-C: Properties (5)

– Pointers and Memory Management

– Pointers

```
int h[ 3 ];
int *p = &h[ 1 ];
h[ 1 ] = 42;
*p = 12;
```



– Dynamic memory management

```
char *p = (char *) malloc( 100 ); /* Allocation of 100 Bytes */
p[ 1 ] = 42;
free( p ); /* Memory deallocation */
```

Dynamic memory management explicitly by programmer!

ANSI-C: Properties (6)

– Architecture-Dependence & Unspecified Behavior

– Bit-width of `int` \cong word-length of a CPU

`int` on 16-bit machine: [-32768, 32767]

`int` on 32-bit machine: [-2147483648, 2147483647]

– Behavior of `>>` operator (shift right)

logic shift – Most Significant Bit (MSB) filled with '0':

$$-8 \gg_1 1 = 1000 \gg_1 1 = 0100 = 4$$

arithmetic shift – MSB filled with old MSB:

$$-8 \gg_a 1 = 1000 \gg_a 1 = 1100 = -4$$


– Signedness of `char`:

`signed char` [-128, 127] vs. `unsigned char` [0, 255]



ANSI-C: Discussion

Pros

- Standardized, widespread programming language
- Many (freely available) existing code generation tools
- Large already existing source code base (open source & proprietary)
- Despite high abstraction level: Low-level programming still possible
- Machine-orientation
- Effort to construct a compiler still acceptable



Cons

- Machine-orientation, lacking portability of source code
- Programmer-responsible memory management error-prone
- No object-orientation



ANSI-C++: Properties

ANSI-C + Object-Orientation + ...

- Classes & Objects
- Elements, member functions/methods
- Constructors & destructors
- Inheritance
- Protection of class elements: **public**, **protected**, **private**
- Virtual methods & polymorphic classes
- Exception handling
- Generic programming: Templates
- Standard Template Library (STL)

ANSI-C++: Pros



- Higher-order programming language, fulfills desire for OO in Embedded Systems
- Existing ANSI-C source codes can often be adopted
- Wide dissemination
- Many (freely available) existing code generation tools
- Large already existing source code base (open source & proprietary)
- Despite high abstraction level: Low-level programming still possible
- Machine-orientation

ANSI-C++: Cons (1)




- Some C++ constructs lead to high overheads, too high for Embedded Systems

Example Exceptions:

```
try {
    object o;
    ...
    Code...;
}
catch( E ) {
    // Error handling
}
```

 *Exception E* is thrown here...

 ... and caught here



o must be destroyed!

- *Time between throwing and catching unclear due to destructors*
- *Increased memory consumption due to internal data structures*

ANSI-C++: Cons (2)



Example Purely Virtual Classes:

```

class A {
    virtual bar() = 0;
}

class B : public A {
    virtual bar();
}

class C : public A {
    virtual bar();
}

```

- *B and C both feature a very own implementation of `bar()`.*

```
A *foo; ...; foo->bar(); // B::bar()?? C::bar???
```

- *High run-times due to dynamic type checking and method lookup.*



Embedded C++

Subset of C++, Specifically Designed for Embedded Systems

- No purely virtual classes
- No exceptions
- No templates
- No namespaces

(“Features such as namespaces [...] are difficult to understand, increasing the chances of programmer errors.”)

- No multiple inheritance
- No STL data structures (Standard Template Library)

*[Embedded C++ Slashes Code Size And Boosts Execution.
www.ghs.com/wp/ec++article2.html]*



Java: Pros

Programming Language with consequent object-oriented design

- Modular structure, allows for excellent SW abstraction
- Good type mechanisms
- Good language constructs for modelling of behavior and control
- Mathematical model similar to C++, but said to be better
- Transparent memory protection, automatic Garbage Collection
- Code said to be better legible than C++
- No pointers
- Java Byte Code Interpreter: High portability of Java source code



Java: Cons

Enormous Resource Demands

- Disadvantages of Java's OO constructs similar to C++
- Byte code interpretation during run-time
- Just-In-Time translation often infeasible for Embedded Systems
- Real-time behavior of Garbage Collection?
- Today: Even lean Java (EmbeddedJava) unsuitable for systems that have to be fast and are resource-constrained

Excerpt from Sun's (ancient) license agreement for Java:

"Software is not designed or licensed for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility."

[Formerly at java.sun.com/products/plugin/1.2/license.txt]

Chapter Contents

2. Compilers for Embedded Systems – Requirements & Dependencies

- Code Generation Tools
 - Compiler, Assembler, Linker
 - Source Code, Assembly Code, Object Code, Binary Code
- Source Languages for Compilers for Embedded Systems
 - C, C++, Java
- Embedded Processors
 - Digital Signal Processors
 - Multimedia Processors
 - *Very Long Instruction Word* Machines
 - Network Processors
- Requirements on Compilers for Embedded Systems
 - Code Quality vs. Compilation Times

Digital Signal Processors

Properties

- Optimized for digital signal processing (e.g., filters, Fourier transformation, ...)
- Heterogeneous register files, grouped for special purposes
- Limited parallelism during instruction execution
- Dedicated address generation units / addressing modes
- Multiply-Accumulate instruction ($a = a + b * c$)
- Zero-Overhead Loops
- Saturating arithmetic
- Efficiency and real-time capabilities extremely important

DSPs: Heterogeneous Register Files (1)

Example Infineon TriCore 1.3:

- Separate address & data registers

<i>Address Registers</i>	<i>Data Registers</i>
A15	D15
A14	D14
A13	D13
A12	D12
A11	D11
A10	D10
A9	D9
A8	D8
A7	D7
A6	D6
A5	D5
A4	D4
A3	D3
A2	D2
A1	D1
A0	D0

DSPs: Limited Parallelism

Example Infineon TriCore 1.3:

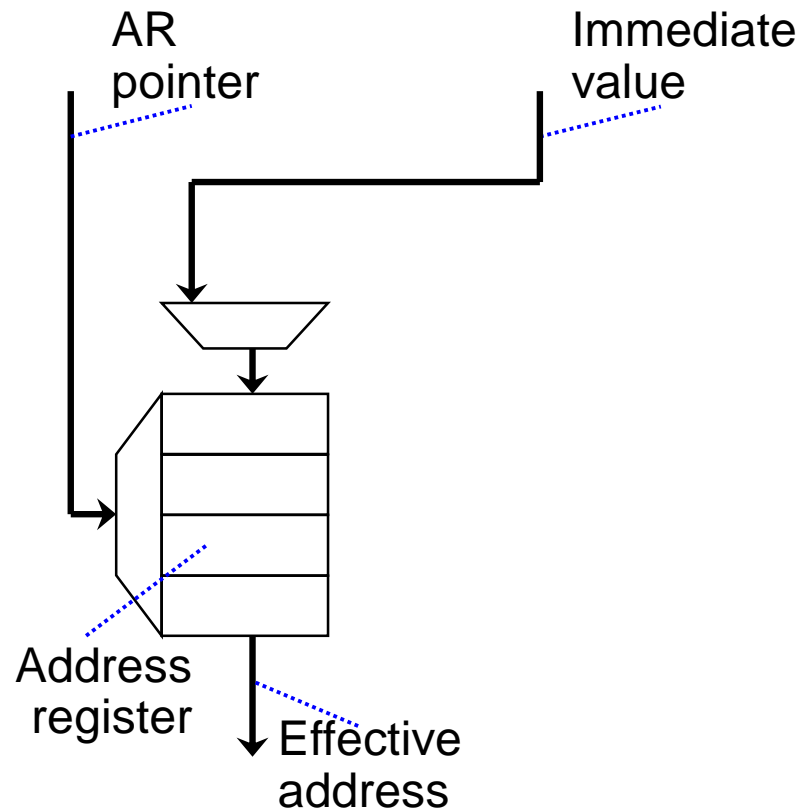
- Integer Pipeline: Arithmetical/logical instructions, conditional branches
- Load/Store Pipeline: Memory accesses, address computations, unconditional branches, function calls
- Loop Pipeline: Loop instructions

- Limited Parallelism:
 - Pipelines work independently / parallel in the ideal case
 - If not the ideal case:
Stall in L/S Pipeline → Stall in I-Pipeline and vice versa

DSPs: Address Generation Units (AGUs)

General Structure of Address Generation Units:

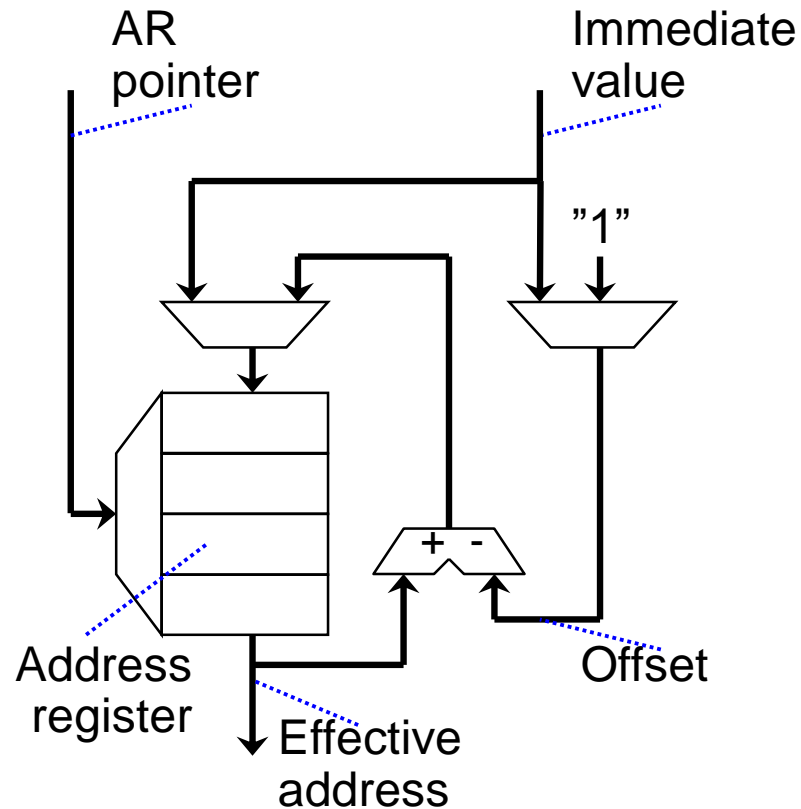
- Address registers (AR) contain *effective addresses* for memory accesses
- Instruction word encodes which AR to use (AR pointer)
- ARs can be loaded with constants explicitly encoded in the machine instruction (immediate operands)



DSPs: Address Generation Units (AGUs)

General Structure of Address Generation Units:

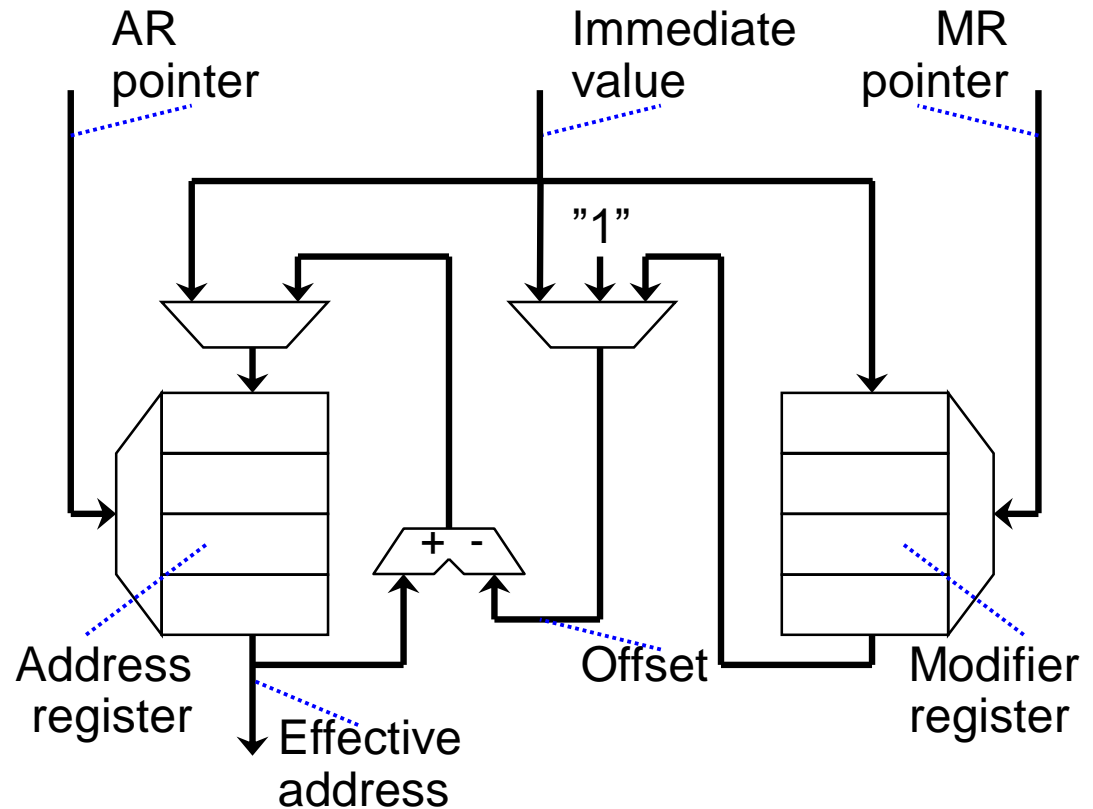
- ARs can be in-/decreased via a simple ALU
- Increment / decrement by offset given as immediate operand
- Increment / decrement by constant "1" as offset



DSPs: Address Generation Units (AGUs)

General Structure of Address Generation Units:

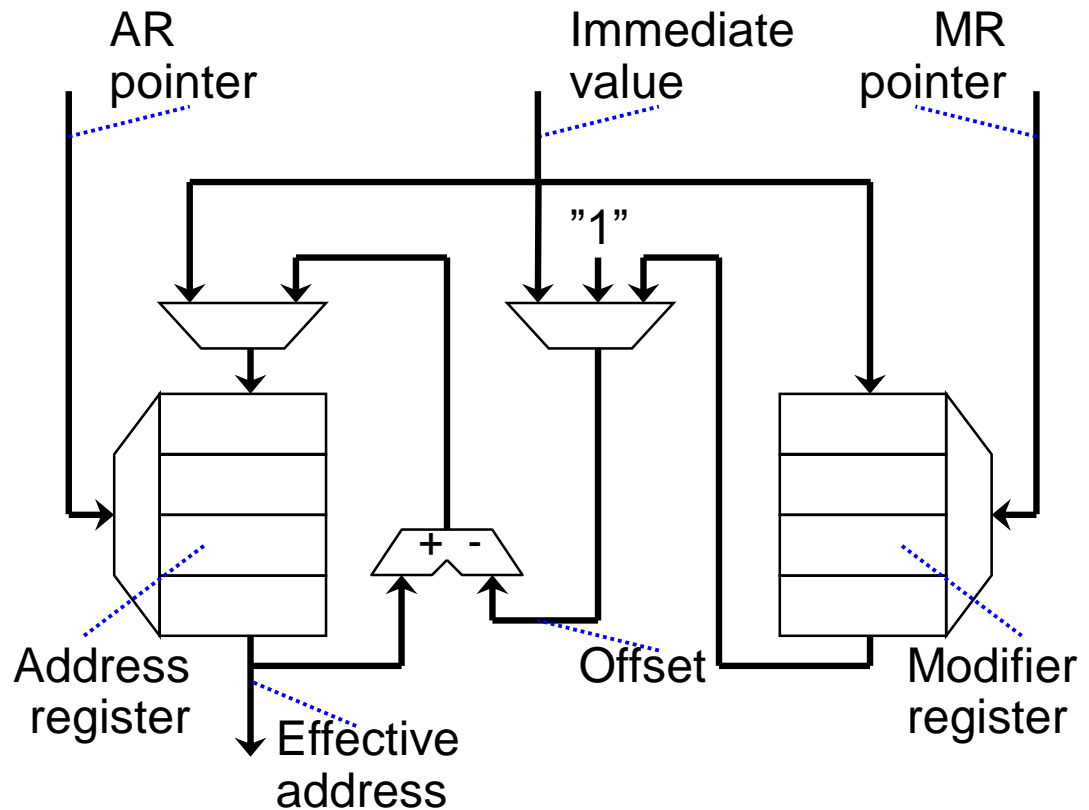
- Increment / decrement by contents of Modifier Register (MR)
- Instruction word encodes which MR to use (MR pointer)
- MRs can be loaded explicitly with immediate values



DSPs: Address Generation Units (AGUs)

General Structure of Address Generation Units:

- Load AR: $AR = \langle const \rangle$
- Load MR: $MR = \langle const \rangle$
- Modify AR: $AR \pm \langle const \rangle$
- *Auto-Increment*: $AR \pm "1"$
- *Auto-Modify*: $AR \pm MR$
- “Auto”-operations: Parallel to data path, no extra run-time, highly efficient!
- *All others*: Need extra instruction for data path, less efficient.



DSPs: Conventional Loop Code

C-Code of a Loop:

```
int i = 10;
do {
    ...
    i--;
} while ( i );
```

Conventional Assembly Code: (TriCore 1.3)

```
    mov %d8, 10;
.L0:
    ...
    add %d8, -1;
    jnz %d8, .L0;
```

Properties

- Decrement & conditional branch: Both in Integer Pipeline
 - ☞ no parallel execution of these instructions
- 2 clock cycles * 10 iterations = 20 cycles (min.) loop overhead
- *In presence of delay slots for branches even worse!*

DSPs: Optimized Loop Code

C-Code of a Loop:

```
int i = 10;
do {
    ...
    i--;
} while ( i );
```

Zero-Overhead Loops: (TriCore 1.3)

```
    mov %a12, 10;
.L0:
    ...
    loop %a12, .L0;
```

Properties

- Decrement & conditional branch: Parallel in Loop Pipeline
- `loop` instruction: Consumes run-time only in 1st and last iteration
 - ☞ only 2 clock cycles loop overhead

Problem of Standard Wrap-Around Computer Arithmetic (1)

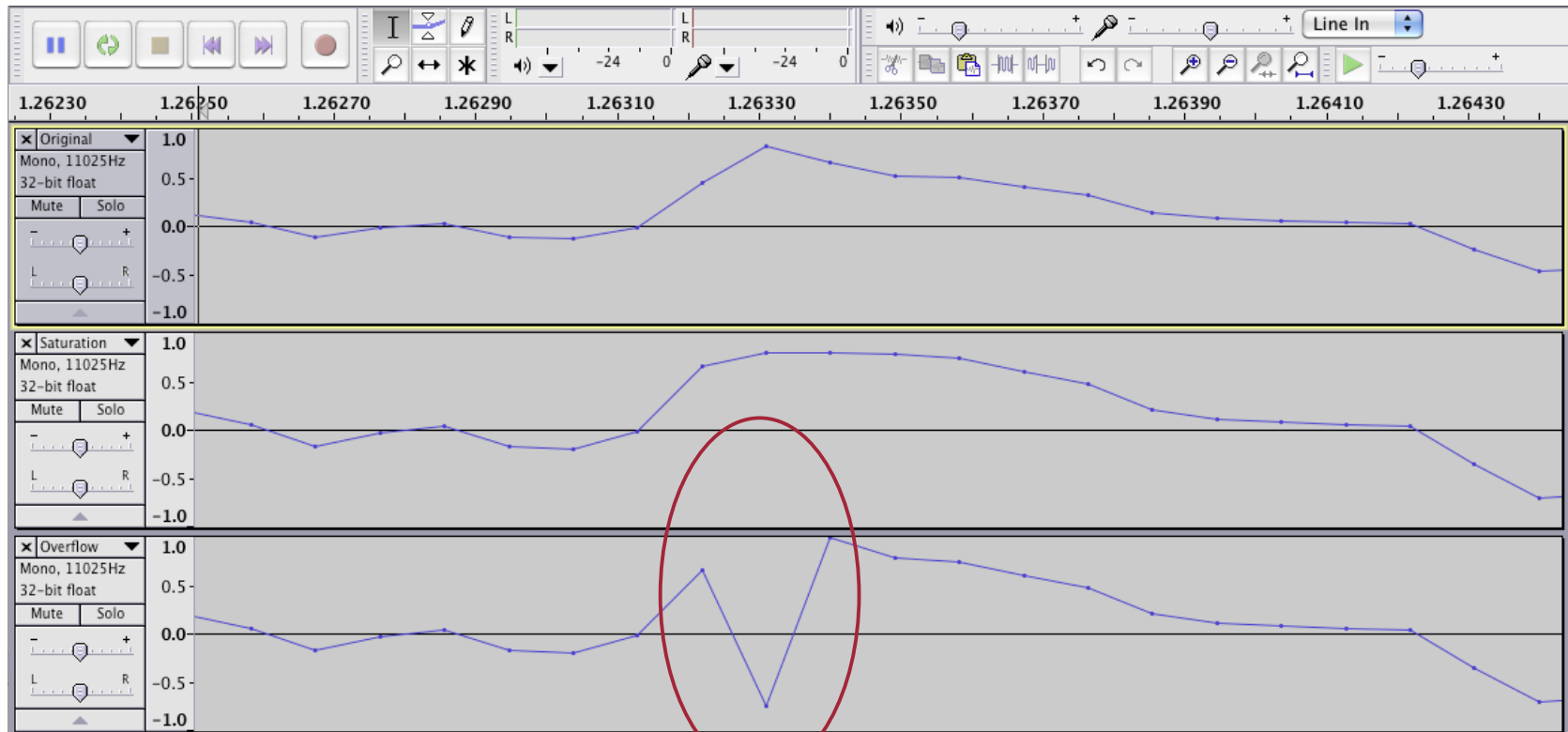
Standard Arithmetic leads to Wrap-Around in case of Over-/Underflows

- Problem: Computed results with Wrap-Around are...
 - ... not only **incorrect**
 - ... but also **implausible** / not even close to the correct solution
- The resulting error is **maximal** (most-significant bit position 2^n gets lost), not minimal! Example:

$$\begin{aligned}
 (4 \text{ bits, 2's compl.): } & |(7 +_{wrap} 1) -_{exact} (7 +_{exact} 1)| = \\
 & |(0111_{(2)} +_{wrap} 0001_{(2)}) -_{exact} 8| = \\
 & |1000_{(2)} -_{exact} 8| = \\
 & |-8 -_{exact} 8| = 16
 \end{aligned}$$

Problem of Standard Wrap-Around Computer Arithmetic (2)

- Large deviation between computed result (with overflow) and desired result, highly dramatic during signal processing (e.g., amplifying of an audio signal / luminance adjustment of an image pixel)



Lesser Error with Saturating Arithmetic

Saturating arithmetic for additions or multiplications yields the **maximally/minimally representable value in case of over-/underflows.**

Examples

- Absolute value representation (4 bits, unsigned):

$$8 +_{sat} 8 = 1000_{(2)} +_{sat} 1000_{(2)} = 7 +_{sat} 11 \rightarrow 15 \neq 18$$

$$10000_{(2)} \rightarrow 1111_{(2)} = 15 \neq 16$$

- 2's complement representation (4 bits, signed):

$$7 +_{sat} 1 = 0111_{(2)} +_{sat} 0001_{(2)} = -5 -_{sat} 7 \rightarrow -8 \neq -12$$

$$1000_{(2)} \rightarrow 0111_{(2)} = 7 \neq 8$$

In particular, saturating arithmetic never produces sign inversions!

Another Example

– a		0111
b	+	1001
<hr/>		
Standard Wrap-Around arithmetic		(1)0000
Saturating arithmetic		1111
<hr/>		
(a+b)/2:	correct	1000
	Wrap-Around arithmetic	0000
	Saturating arithmetic using >>	0111

„almost correct“

- Suitable for DSP / multimedia applications
 - Interrupts generated by overflows
 - ☞ Real-time requirements violated...?
 - Exact values of less importance anyway
 - Wrap-Around arithmetic produces inferior results

Saturating Arithmetic: Discussion

Pros

- More plausible results in case of over-/underflows

Cons

- More time-consuming to compute
- Associativity and other laws violated

DSPs usually allow to choose between saturating and standard arithmetic (they feature correspondent variants of machine instructions)

“Saturation” in IEEE 754 floating point standard:

- Over-/underflows produce \pm infinity as result
- Further IEEE 754 operations do not change this result!

DSPs: Real-Time Capabilities

The Timing Behavior of a Processor should be predictable!

Properties causing trouble:

- Accesses to shared hardware resources
 - Caches with replacement strategies with problematic timing behavior
 - Unified caches for both code and data (conflicts between data and instructions)
 - Pipelines with stall cycles (“bubbles”)
 - Multi-cores with unpredictable communication/bus delays
- Branch prediction, speculative instruction execution
- Interrupts that can happen at any time
- Memory refreshes at any time
- Machine instructions with data-dependent execution latencies

 **Avoid as many of these problematic properties as possible**

Multimedia Processors

Properties

- Optimized for, e.g., image and audio processing
- Well-known commercial products:
Intel MMX, SSE or SSE2; AMD 3DNow!; Sun VIS;
PowerPC AltiVec; HP MAX
- Motivation: Multimedia software often does not use the full word-length of a processor (i.e., `int`), but only parts of it (e.g., `short` or `char`).
- SIMD principle: *Single Instruction, Multiple Data*
- Parallel handling of several “little” pieces of data by 1 machine instruction

SISD vs. SIMD Execution

Exercise: Perform two Additions of 2 short Variables each

- SISD principle (Single Instruction, Single Data):

Load first 2 summands in registers,

`int` addition,

Load second 2 summands in registers,

`int` addition

☞ Costs: 2 full additions

- SIMD principle (Single Instruction, Multiple Data):

Load first 2 summands in upper halves of registers,

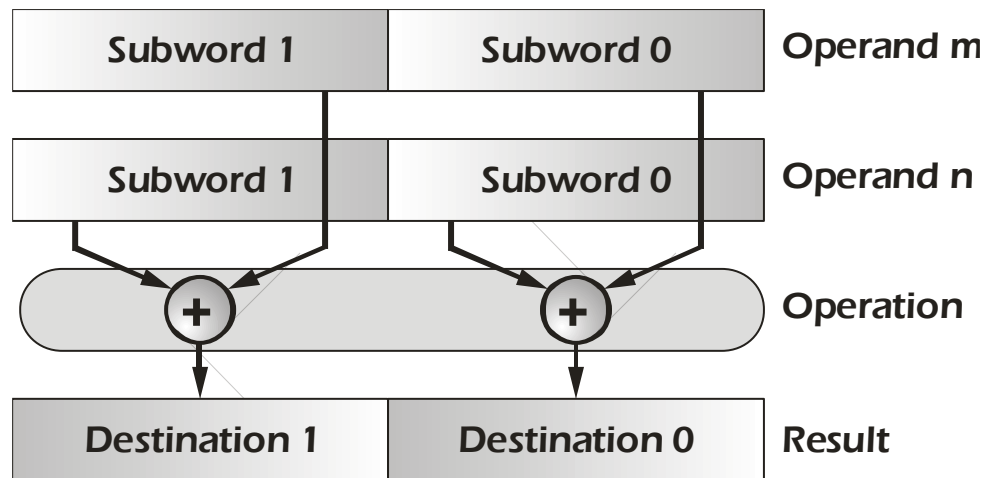
Load second 2 summands in lower halves of registers,

SIMD addition

☞ Costs: 1 addition

Illustration SIMD Addition

SIMD Half-Word Addition:



- SIMD instructions also common for quarter-words:
 - ☞ 4 parallel `char` additions for a 32-bit processor

Very Long Instruction Word (VLIW)

Motivation

Performance boost by exploiting instruction-level parallelism

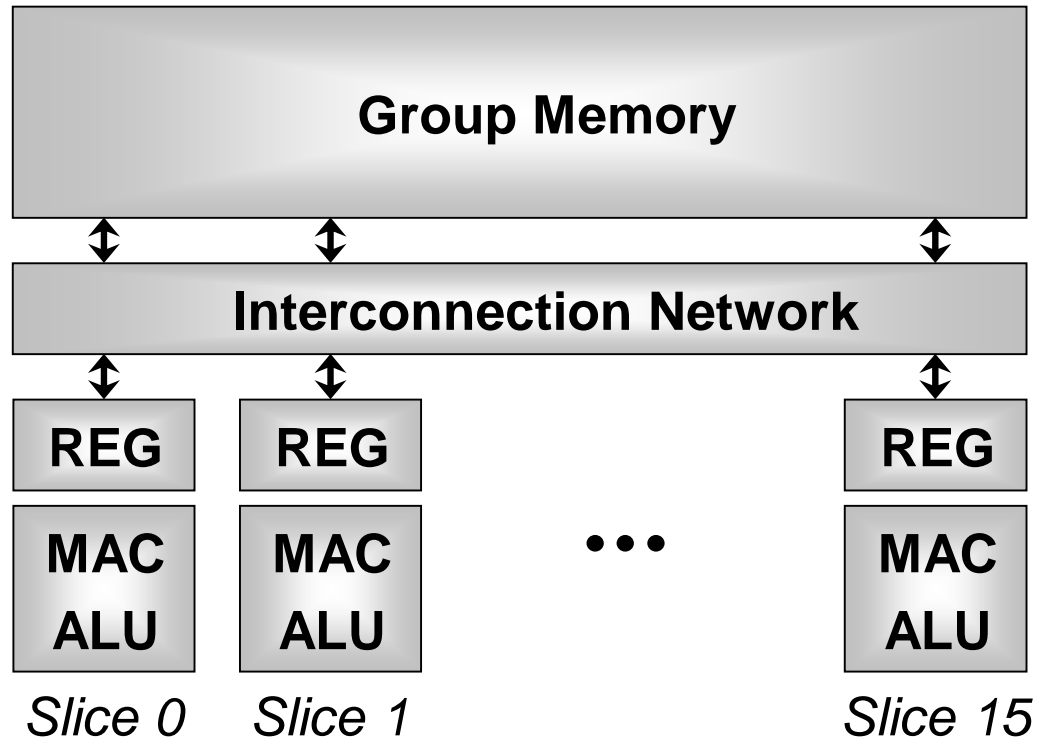
Conventional Processors:

- 1 integer ALU
- 1 multiplier
- 1 (heterogeneous) register file

VLIW Processors:

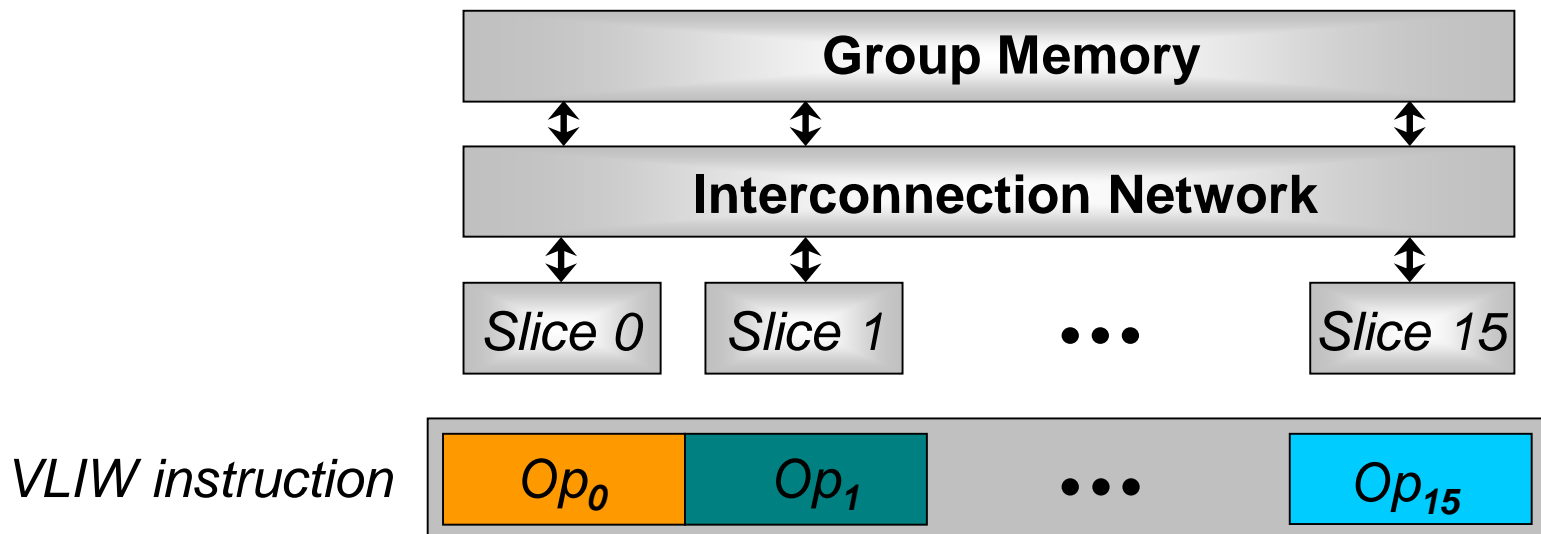
- n integer ALUs
- n multipliers
- n (heterogeneous) register files
- Interconnection network

Example: M3 VLIW Processor



VLIW Instruction Word

- 1 instruction word contains 1 VLIW machine instruction
- 1 VLIW instruction contains n VLIW operations
- Each operation controls exactly one Functional Unit (FU)
- Fixed matching of operations in the instruction with FUs:
Operation 0 \leftrightarrow FU 0, Operation 1 \leftrightarrow FU 1, ...



Network Protocols

Communication between distant Processors

- Communication media error-prone
- Payload is divided into packets
- Packets are augmented with additional information (Header)

Example IPv4 Header:

0	7	15	23	26	31
Version	Length	Service Code	Total Packet Length		
Identification			Flags	Fragment Offset	
Time To Live		Protocol	Header Checksum		
Source IP Address					
Destination IP Address					

Bit Packets (1)

Bit Packets in Protocol Headers

- Headers are divided in regions with different meaning
- These bit regions are not aligned with processor word-lengths
- Bit packet:
 - Sequence of consecutive bits
 - of arbitrary length
 - starting at arbitrary bit positions
 - and eventually crossing word boundaries

☞ Efficient manipulation of data at bit-level necessary!

Bit Packets (2)

Network Processing Units (NPUs)

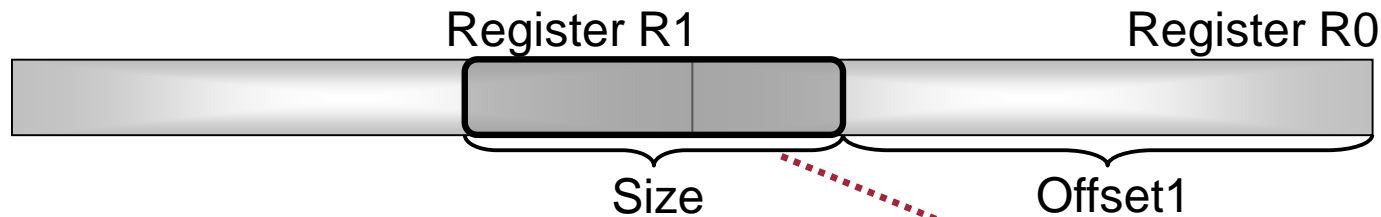
- Software for network protocol processing:
High amount of code dealing with processing of bit packets
- Typical C-Code (GSM kernel, TU Berlin):

```
xmc[0] = (*c >> 4) & 0x7;  
xmc[1] = (*c >> 1) & 0x7;  
xmc[2] = (*c++ & 0x1) << 2;  
xmc[2] |= (*c >> 6) & 0x3;  
xmc[3] = (*c >> 3) & 0x7;
```

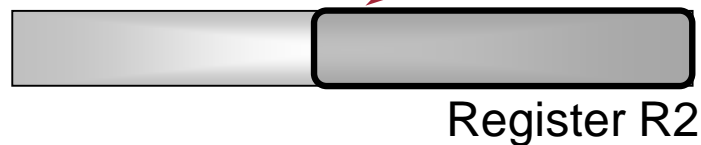
- Instruction set of NPUs:
Special-purpose machine instructions to extract, insert & manipulate bit packets

Operations on Bit Packets

Extraction of Bit Packets

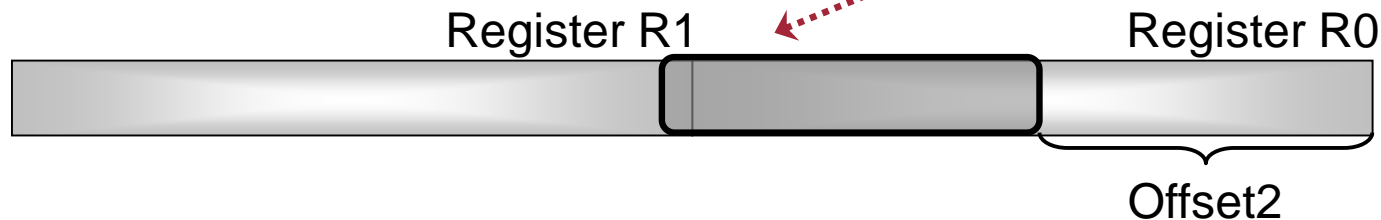
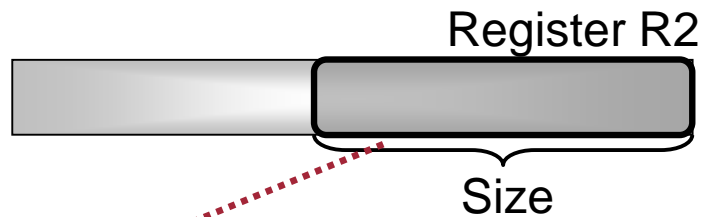


```
extr R2, R0, <Offset1>, <size>;
```



Insertion of Bit Packets

```
insert R0, R2, <Offset2>, <size>;
```



Chapter Contents

2. Compilers for Embedded Systems – Requirements & Dependencies

- Code Generation Tools
 - Compiler, Assembler, Linker
 - Source Code, Assembly Code, Object Code, Binary Code
- Source Languages for Compilers for Embedded Systems
 - C, C++, Java
- Embedded Processors
 - Digital Signal Processors
 - Multimedia Processors
 - *Very Long Instruction Word* Machines
 - Network Processors
- Requirements on Compilers for Embedded Systems
 - Code Quality vs. Compilation Times

Most Important Requirements on Compilers for ES

Maximal Code Quality

- Run-time efficiency
- Low energy consumption
- Low code size
- Maximal parallelization
- Real-time ability
- ...

Meaningful Measures

- Best-possible mapping of source language to machine language
- Exploitation of strong compiler optimizations
- Reuse of code fragments
- Maximal exploitation of fast and small memories
- Consideration of the WCET (Worst-Case Execution Time)
- ...

Less Important Requirements

Speed of the Compiler Itself

- Situation for “standard” desktop computers:
 - ✓ Large amount of available resources
 - ✓ Code quality of less interest
 - ✓ Compilers shall quickly generate correct code

- Situation for Embedded Systems:
 - ❑ Code quality of utmost importance
 - ❑ Compiler shall generate highly-optimized code
 - ❑ In the design process of Embedded Systems, compilers are called less frequently than when programming for desktop computers

👉 High Run-Times of Optimizing Compilers are acceptable!

References

Code Generation Tools

- John R. Levine. *Linkers & Loaders*.
Morgan Kaufmann, 2000.
ISBN 1-55860-496-0

Programming Languages

- Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language*.
Prentice Hall, 1988.
ISBN 0-13-110362-8
- *Embedded C++ Home Page*.
<http://www.caravan.net/ec2plus>, 2002.
- *The Real-Time Specification for Java*.
<http://www.rtsj.org>, 2007.

References

Processors & Instruction Sets

- Peter Marwedel. *Embedded System Design*. Springer, 2011.
ISBN 978-94-007-0256-1
- Rainer Leupers. *Code Optimization Techniques for Embedded Processors*. Kluwer Academic Publishers, 2000.
ISBN 0-7923-7989-6
- Jens Wagner. *Retargierbare Ausnutzung von Spezialoperationen für Eingebettete Systeme mit Hilfe bitgenauer Wertflussanalyse*. Dissertation University of Dortmund, Computer Science 12, 2006.

Summary (1)

Code Generation Tools

- Assembler and linker as further important tools besides compilers
- Address resolution, memory layout and libraries

Source Languages for Compilers for Embedded Systems

- ANSI-C: widespread imperative programming language; allows machine-oriented low-level programming; error-prone memory management; unspecified language details
- C++: similar to C; high overhead due to some object-oriented language constructs
- Java: very high overhead due to byte code interpretation; no real-time capabilities due to dynamic garbage collection during run-time
- ANSI-C most common programming language for Embedded Systems

Summary (2)

Embedded Processors

- Partially highly specialized instruction sets (multiply-accumulate, insert/extract, SIMD)
- Special registers and address generation units
- High degree of parallelism (several pipelines, multitude of functional units)

Requirements on Compilers for Embedded Systems

- Code quality primary concern
- Compile-times only secondary, in contrast to compilers for desktop computers