Embedded Systems Development

Lecture 7
Compiler Backends &
Code Generation for Embedded Processors

Daniel Kästner
AbsInt Angewandte Informatik GmbH
kaestner@absint.com
Code Generation Phases

**Code selection**
- \( c = a + b \)
- load \( \text{adr}(a) \)
- load \( \text{adr}(b) \)
- add
- store \( \text{adr}(c) \)

**Register allocation**
- \( \text{a} \)
- \( \text{b} \)
- \( \text{c} \)

**Instruction scheduling**
- \( r_1 = \text{load} \, \text{adr}(a) \)
- \( \text{add} \, r_1, \, r_2 \)
- store \( \text{adr}(c), \, r_1 \)
The Code Generation Problem

- Code selection: NP-complete
- Register allocation: NP-complete
- Instruction scheduling: NP-complete
- All of them are interdependent, i.e. decisions made in one phase may impose restrictions to the other.
- Usually they are addressed
  - by heuristic methods
  - in separate phases.

- Thus: often suboptimal combination of suboptimal partial results.
The Phase Ordering Problem

d1 = s1 * s1;

\[ R_3 = R_1 \times R_1 \]

\[ d_2 = s_1 + s_2; \]

\[ R_3 = R_1 + R_2 \]

\[ R_3 = r_1 \times r_1, \quad R_4 = R_1 + R_2 \]

Optimize number of used registers

Optimize code speed and size
Code Generation: Integrated Methods

- Integration of register allocation with instruction selection for expression trees

- Restrictions:
  - **Simple** machine model:
    - $r$ general purpose registers $R_0, \ldots, R_{r-1}$
    - Two-address instructions:
      - $R_i := M[V]$ Load
      - $M[V] := R_i$ Store
      - $R_i := R_i \text{ op } M[V]$ Compute
      - $R_i := R_i \text{ op } R_j$
  - Very **restricted scope** (expression tree $<$ basic block $<$ procedure).

- Two phases:
  1. Computing register requirements
  2. Generating code, allocating register and temporaries
Example Tree

- Source:
  - $r := (a+b)-(c-(d+e))$

- Tree:
Generated Code

- Given two registers $R_0$ and $R_1$, two possible code sequences are:

$$
R0 := M[a] \\
R0 := R0 + M[b] \\
R1 := M[d] \\
R1 := R1 + M[e] \\
M[t1] := R1 \\
R1 := M[c] \\
R1 := R1 - M[t1] \\
R0 := R0 - R1 \\
M[f] := R0
$$

- Given two registers $R_0$ and $R_1$, two possible code sequences are:

$$
R0 := M[c] \\
R1 := M[d] \\
R1 := R1 + M[e] \\
R0 := R0 - R1 \\
R1 := M[a] \\
R1 := R1 + M[b] \\
R1 := R1 - R0 \\
M[f] := R1
$$
Generated Code

- **Left code:**
  - stores result for \( c - (d + e) \) in a temporary
  - no register available

- **Right code:**
  - evaluates \( c - (d + e) \) first (needs 2 registers)
  - saves one instruction
The Algorithm

- Principle: Given tree t for expression e1 op e2
- t1 needs r1 registers, t2 needs r2 registers
- Assume r ≥ r1 > r2:
  - After evaluation of t1:
    - r1-1 registers freed
    - one holds the result
  - t2 gets enough registers to evaluate, hence t can be evaluated in r1 registers
- Assume r1=r2:
  - t needs r1+1 registers to evaluate
- Assume one of t1 or t2 need more than r registers:
  - spill to temporary required
Labeling Phase

- Labels each node with its register needs

- **Bottom up** pass:
  - **Left leaves** labeled with '1' have to be loaded into registers
  - **Right leaves** labeled with '0' are used as operands
  - **Inner nodes**:

\[
\text{regneed}(\text{op}(t_1, t_2)) = \begin{cases} 
\max(r_1, r_2), & \text{if } r_1 \neq r_2 \\
 r_1 + 1, & \text{if } r_1 = r_2 
\end{cases}
\]

where \( r_1 = \text{regneed}(t_1), r_2 = \text{regneed}(t_2) \)
Example

\[ r := 2 \\
- 2 \\
+ 1 \\
| 1 \\
| a b c \\
+ 1 \\
| 1 \\
| d e \\
0 \]
Generation Phase

- Generates instruction \( \text{OP} \) for operator \( \text{op} \) in \( \text{op}(t_1, t_2) \) after generating code for \( t_1 \) and \( t_2 \)
- Order of \( t_1 \) and \( t_2 \) depends on their register needs
- Upon execution of the generated Op-instruction: value of \( t_1 \) in register
- \( \text{RSTACK} \): available registers, initially all registers.
- Before processing \( t \), the result register for \( t \) is \( \text{top(RSTACK)} \)
- After processing \( t \): all registers are available, \( \text{top(RSTACK)} \) is result register for \( t \)
- \( \text{TSTACK} \): available temporaries
Algorithm GEN_OPT_CODE

Algorithm

var RSTACK: stack of register;
var TSTACK: stack of address;
proc Gen_Code (t : tree);
var R: register, T: address;
case t of
(leaf a, 1) : (*left leaf*)
    emit(top(RSTACK) := a);
    op((t_1, r_1), (leaf a, 0)) : (*right leaf*)
    Gen_Code(t_1);
    emit(top(RSTACK) := top(RSTACK) Op a);
    op((t_1, r_1), (t_2, r_2)) : cases
    r_1 < min(r_2, r):
    BEGIN
        exchange(RSTACK);
        Gen_Code(t_2);
        R := pop(RSTACK);
        Gen_Code(t_1);
        emit(top(RSTACK) := top(RSTACK) Op R);
        push(RSTACK, R);
        exchange(RSTACK);
    END;

RSTACK-Contents resp. result register
(R', R'', ...)
result in R'
(R', R'', ...)
result in R'
(R', R'', ...)
result in R''
(R', R'', ...)
result in R'
(R', R'', ...)
result in R''
(R', R'', ...)
result in R'
Algorithm GEN_OPT_CODE

\[ r_1 \geq r_2 \land r_2 < r: \]
BEGIN
\( \text{Gen\_Code}(t_1); \)
\( R := \text{pop}(R\text{STACK}); \)
\( \text{Gen\_Code}(t_2); \)
\( \text{emit}(R := R \text{ Op top}(R\text{STACK})); \)
\( \text{push}(R\text{STACK}, R); \)
END;
\[ r_1 \geq r \land r_2 \geq r: \]
BEGIN
\( \text{Gen\_Code}(t_2); \)
\( T := \text{pop}(T\text{STACK}); \)
\( \text{emit}(M[T] := \text{top}(R\text{STACK})); \)
\( \text{Gen\_Code}(t_1); \)
\( \text{emit}(\text{top}(R\text{STACK}) := \text{top}(R\text{STACK}) \text{ Op } M[T]); \)
\( \text{push}(T\text{STACK}, T); \)
END;
endcases
endcase
endproc

\( (R', R'', \ldots) \)
result in \( R' \)
\( (R'', \ldots) \)
result in \( R'' \)
result in \( R' \)
\( (R', R'', \ldots) \)
result in \( R' \)
result in \( M[T] \)
result in \( R' \)
result in \( R' \)
Register Allocation and Instruction Selection by Dynamic Programming

- **More complex** architecture:
  - \( r \) general purpose registers \( R_0, \ldots, R_{r-1} \)
  - Instruction formats:
    - \( R_i := e \) Compute
    - \( R_i := M[V] \) Load
    - \( M[V] := R_i \) Store,

where
- \( e \) term with registers and memory cells
- costs are associated with each instruction

- **Goal**: Generate cheapest instruction sequence using no more than \( r \) registers
- Assume **contiguous computation of subtrees** → only one register required to hold the result
- Use instruction selection techniques to compute **cheapest instruction sequence**.
Canonical Recursive Solution

- Assume $e$ of instruction $R1:=e$ matches tree $t$

- Subtrees of $t$ correspond to the memory operands of $e$: they are computed into memory and no registers are occupied after that

- Let $e$ have $k$ register operands: how to compute the corresponding subtrees $t_{i_1}, ..., t_{i_k}$ into these registers?

- Assume order $i_1, i_2, ..., i_k$ and $j$ available registers

- $t_{i_1}$ has $j$ registers available, $t_{i_2}$ has $j-1$, $t_k$ has $j-k$
Canonical Recursive Solution

- If this fits $(j-k-regneed(t_{ik}) >= 0)$, add the minimal costs for computing all subtrees in this way to the costs of $e$ to yield the minimal costs for this combination.

- If not enough registers are available, compute enough subtrees into memory, and sum up costs like above.

- Doing this for all potential combinations recomputes the costs for subtrees $\rightarrow$ exponential complexity.
Dynamic Programming

- Convert top-down algorithm into **bottom-up** algorithm tabulating partial solutions.
- Associate **cost vector** $C[0..r]$ with each node $n$.
  - $C[0]$: cheapest costs for computing $t/n$ into a temporary,
  - $C[i]$: cheapest costs computing $t/n$ into a register using $i$ registers.
- Compute cost vector at node $n$ minimizing over all legal combinations of
  - one applicable instruction
  - the cost vectors of the nodes under non-terminal nodes in the applied rule.
- What is a **legal combination** for $C[j]$, $j>0$?
  - Any combination of generated code for subtrees not needing more than $j$ registers.
- Extract **cheapest** instruction sequence in a **second** pass.
Impact of Code Selection

```c
struct {
    unsigned b1:1;
    unsigned b2:1;
    unsigned b3:1;
} s;

s.b3 = s.b1 && s.b2;
```

```
ld.bu d5, [sp]+48
extr.u d5, d5, #0, #1
ne d5, d5, #0
mov16 d4, d5
jeq d4, #0, _9
ld.bu d4, [sp]+48
extr.u d4, d4, #1, #1
ne d4, d4, #0
_9:
ld.b d3, [sp]+48
insert d3, d3, d4, #2, #1
st.b [sp]+48, d3
```

9 cycles, 42 bytes

```
ld.bu d5, [sp]+48
and.t d3, d5, #0, d5, #1
insert d5, d5, d3, #2, #1
st.b [sp]+48, d5
```

3 cycles, 16 bytes
Decorated Abstract Syntax Tree

Concrete syntax tree:

IR for code selection:

Abstract syntax tree:

add:

plus

reg reg

reg reg id id  iconst  id id  iconst
Generating Code Selectors

- Machine grammar: regular tree grammar;
  - terminals: operators from the program representation
  - non-terminals: represent storage resources
  - often ambiguous
  - each rule has associated costs
  - factorization, e.g. of addressing modes reduces size.

\[ \text{DREG} \rightarrow \text{m} \]
\[ \text{m} \rightarrow \text{plus} \]
\[ \text{plus} \rightarrow \text{bconst} \]
\[ \text{plus} \rightarrow \text{AREG} \]
\[ \text{plus} \rightarrow \text{IREG} \]
Generating Code Selectors

- A machine grammar enables IR trees for expressions to be derived. The \textit{derivation tree} for an IR tree represents one possibility of generating code for the IR tree.

- The generated code selector
  - parses intermediate representations of programs
  - computes derivations according to the machine grammar, each corresponding to a sequence of machine instructions
  - has to select a \textit{cheapest} derivation, corresponding to the (locally) cheapest code sequence
  - may compute costs in states or use dynamic programming
Tree Languages

- An alphabet with arity is a finite set $\Sigma$ of operators together with a function $\rho: \Sigma \rightarrow \mathbb{N}_0$ (arity).

$$\Sigma_k = \{a \in \Sigma \mid \rho(a) = k\}$$

- The homogeneous tree language over $\Sigma$ is the following inductively defined set $T(\Sigma)$:
  - $a \in T(\Sigma)$ for all $a \in \Sigma_0$
  - Are $b_1, b_2, ..., b_k \in T(\Sigma)$ and is $f \in \Sigma_k$, then $f(b_1, b_2, ..., b_k) \in T(\Sigma)$

- Example: $\Sigma=\{a,\text{cons, nil}\}; \rho(a)=\rho(\text{nil})=0, \rho(\text{cons})=2$
  Some trees over $\Sigma$: $a$, $\text{cons(\text{nil, nil})}$, $\text{cons(cons(a, nil), nil)}$
Tree Grammars

- A Regular Tree Grammar is a grammar $G = (N, \Sigma, P, S)$ where
  
  - $N$ is a finite set of non-terminals
  - $\Sigma$ is a finite alphabet with arity of terminals (operators labeling nodes)
  - $P$ is a finite set of rules $X \rightarrow s$ where $X \in N$ and $s \in T(\Sigma \cup N)$
  - $S \in N$ is the start symbol
Example: Machine Grammar

\[ G_m = (N_m, \Sigma, P_m, \text{REG}) \]

\[ \Sigma = \{ \text{const, } m, \text{plus, } \text{REG} \} \]
where \( \rho(\text{const}) = 0 \); \( \rho(m) = 1 \); \( \rho(\text{plus}) = 2 \),

\[ N_m = \{ \text{REG} \} \]

\[ P_m = \{ \text{addmc} : \text{REG} \rightarrow \text{plus}(m(\text{const}), \text{REG}), \text{addm} : \text{REG} \rightarrow \text{plus}(m(\text{REG}), \text{REG}), \text{add} : \text{REG} \rightarrow \text{plus}(\text{REG}, \text{REG}), \text{ldmc} : \text{REG} \rightarrow m(\text{const}), \text{ldc} : \text{REG} \rightarrow \text{const}, \text{ld} : \text{REG} \rightarrow \text{REG} \} \]
Generating Code Selectors

- Let $G = (N, \Sigma, P, S)$ be a regular tree grammar. From $G$ a non-deterministic finite tree automaton (NTFA) is generated whose computations correspond to the derivation trees wrt $G$. In a second step, the non-deterministic finite tree automaton is transformed into a DFTA.

- Starting with the leaves of the input tree, the NFTA guesses the correct right-hand sides of grammar rules and checks that the right-hand sides really fit. This way, the input tree is covered by the right-hand side of the grammar rules.

- Code selector generators are, e.g., BEG [Emm89], Iburg [FraHan95], ...
Example: Generating Code by Computing the Derivation Tree
Intermediate Representations

- Call Graph
- Control Flow Graph
- Basic Block Graph
- Data Dependence Graph
Call Graph

- There is a node for the main procedure – being the entry node of the program – and a node for each procedure declared in the program.
- The nodes are marked with the procedure names.
- There is an edge between the node for a procedure \( p \) to the node of procedure \( q \), if there is a call to \( q \) inside of \( p \).
Control Flow Graph

\[ y := x \] \(^1\)
\[ z := 1 \] \(^2\)

while \([y>1]\) \(^3\)

\[ y := x \] \;
\[ z := 1 \] \;

\[ y := y - 1 \] \(^5\)

\[ y := 0 \] \(^6\)
Control Flow Graph

- The control flow graph of a procedure is a directed graph $G_C = (N_C, E_C, n_A, n_\Omega)$ with node and edge labels. For each instruction $i$ of the procedure there is a node $n_i$ that is marked by $i$. The edges $(n, m, \lambda)$ denote the control flow of the procedure: $\lambda \in \{T, F, \epsilon\}$ is the edge label. The nodes for composed statements are shown on the next slide. Edges belonging to unconditional branches lead from the node of the branch to the branch destination. The node $n_A$ is the uniquely determined entry point in the procedure; it belongs to the first instruction to be executed. $n_\Omega$ denotes the end node that is reached by any path through the control flow graph.

- Nodes with more than one predecessor are called joins and nodes with more than one successor are called forks.
Control Flow Graph – Composed Statements

\( \text{cfg (while B do S od)} = \text{cfg (S)} \)

\( \text{cfg (if B then S}_1 \text{ else S}_2 \text{ fi)} = \text{cfg (S}_1 \text{) + cfg (S}_2 \text{)} \)

\( \text{cfg (S}_1 ; \text{S}_2 \text{)} = \text{cfg (S}_1 \text{)} + \text{cfg (S}_2 \text{)} \)
Basic Block Graph

- A basic block in a control flow graph is a path of maximal length which has no joins except at the beginning and no forks except possibly at the end.

- The basic block graph $G_B=(N_B, E_B, b_A, b_\Omega)$ of a control flow graph $G_C=(N_C, E_C, n_A, n_\Omega)$ is formed from $G_C$ by combining each basic block into a node. Edges of $G_C$ leading into the first node of a basic block lead to the node of that basic block in $G_B$. Edges of $G_C$ leaving the last node of a basic block lead out of the node of that basic block in $G_B$. The node $b_A$ denotes the uniquely determined entry block of the procedure; $b_\Omega$ denotes the exit block that is reached at the end of any path through the procedure.
Interprocedural Control Flow Graph

The interprocedural control flow graph consists of three parts:

1. Call graph whose nodes are meta-nodes containing basic block graphs.

2. Basic block graph for each procedure in the program.

3. Ordered list of instructions for each block in the basic block graph of each procedure.

The ICFG describes the control flow of a program completely.
Data Dependence Graph

- Let $G_C$ be a control flow graph. It data dependence graph is a directed graph $G_D=(N_D,E_D)$ with node and edge labels whose nodes are labeled by the operations of the procedure. An edge runs from the node of an operation $i$ to the node of an operation $j$, if $i$ has to be executed before $j$, i.e. if there is a path from $i$ to $j$ in the control flow graph and if
  - $i$ defines a resource $r$, $j$ uses it and the path from $i$ to $j$ does not contain other definitions of $r$ (true dependence, RAW): $(i,j,r,t) \in E_D$
  - $i$ uses a resource, $j$ defines it and the path from $i$ to $j$ does not contain any definitions of $r$ (anti dependence, WAR): $(i,j,r,a) \in E_D$
  - $i$ and $j$ define the same resource and the path from $i$ to $j$ does not contain any uses nor definitions of $r$ (output dependence, WAW): $(i,j,r,o) \in E_D$

(1) $r_1 = r_2 \cdot r_3$;  (1) $r_1 = r_2 \cdot r_3$;  (1) $r_1 = r_2 \cdot r_3$
  (1, 2, r1, t)  (1, 2, r2, a)  (1, 2, r1, o)

(2) $r_5 = r_1 + r_1$;  (2) $r_2 = r_5 + r_6$;  (2) $r_1 = r_5 + r_6$
Instruction Scheduling

- Definition: Reordering an operation sequence in order to exploit instruction-level parallelism and to minimize pipeline stalls.
  - Complexity: NP-complete.

- Terminology:
  - An operation is a basic machine operation like add, sub, etc.
  - An instruction is a set of machine operations that are issued simultaneously (cf. VLIW).

Example:

```
r3=r1+r4, r3=r10+r14, r11=dm(i6, m6), r12=pm(i15, m15);
```
Instruction Scheduling

- Scope of instruction scheduling:
  - **local acyclic** instruction scheduling: reordering operations inside basic blocks. Standard technique: list scheduling.

  - **global acyclic** instruction scheduling: reordering operations across basic block boundaries but not across loop boundaries. Standard technique: trace scheduling.

  - **cyclic** instruction scheduling: reordering operations across loop boundaries. Standard technique: software pipelining.
List Scheduling

```c
SET data_ready;
int cycle=0;

Insert operations without predecessors in the data
dependence graph into the data_ready set.

while (data_ready ≠ ∅) do {
  cycle = cycle+1;

  Choose operations from data_ready in priority order and
  insert them into the current cycle, until data_ready
  is empty or the insertion leads to a resource conflict.

  Insert all operations into data_ready that can be
  scheduled in the next cycle without violating data
  dependences.
}
```
List Scheduling

- The priority in which operations from the data ready set are chosen is determined by heuristics.

- Common heuristics: highest-level-first heuristics.
  - The priority of each operation is the length of the longest path in the data dependence graph starting from this operation.

- Code quality: often within 10% from local optimum (inside basic blocks)
Life Range and Register Interference

- A symbolic register (a variable) \( r \) is live at a program point \( p \), if there is a program path from the entry node of the procedure to \( p \) that contains a definition of \( r \) and there is a path from \( p \) to a use of \( r \) on which \( r \) is not defined. The life range of a symbolic register \( r \) is the set of program points at which \( r \) is live.

- Two life ranges of symbolic registers interfere, if one of them is defined during the life range of the other. The register interference graph is an undirected graph whose nodes are life ranges of symbolic registers and whose edges connect the nodes of interfering life ranges.

\[
\begin{align*}
v_1 &= \text{Mem}[0xAFA0] \\
v_2 &= \text{Mem}[0xAF0C] \\
v_3 &= v_1 + v_2 \\
v_4 &= v_1 \times v_2 \\
v_5 &= v_3 + v_4 \\
\text{return } v_5
\end{align*}
\]
Register Allocation by Graph Coloring

- If $k$ physical registers are available, the $k$-coloring problem must be solved.
- NP-complete for $k>2$ -> Use heuristics

Algorithm:
- If $G$ contains a node $n$ with degree $< k$:
  - $n$ and its neighbors can be colored with different colors
  - Remove $n$ from $G$, decreasing the size of $G$.
  - $G$ is $k$-colorable, if we arrive at the empty graph.
- If $G$ is not empty and there exists no node with degree $< k$:
  - use heuristics to select one node to remove (spilling)
  - modify program inserting spills at definitions and loads at uses
  - reflect changes in graph.
Register Allocation by Graph Coloring

- **Heuristics for node removal**
  - Degree of the node: high degree of node causes many deletions of edges.
  - Costs of spilling.
  - Register constraints of operands.

- **Enhancements:**
  - Coalescing.
  - Life range splitting.
  - Pre-colored nodes.
  - Support for multiple register sets.
  - Preserve topological information (loop nesting level).
Example: Optimistic Graph Coloring

\[
v1 = \text{Mem}[0xafa0] \\
v2 = \text{Mem}[0xafc0] \\
v3 = v1 + v2 \\
v4 = v1 \times v2 \\
v5 = v3 + v4 \\
\]

return v5
Embedded Processors: Application Areas

- Automotive
- Avionics & Space
- Telecommunication
- Consumer electronics
- Healthcare technology
Design of Embedded Systems: Goal Conflict

- High performance requirements
- Increasing software complexity
- High dependability

- Low cost
- Low power consumption
- Short product and development cycles
Types of Microprocessors

- **Complex Instruction Set Computer (CISC)**
  - large number of complex addressing modes
  - many versions of instructions for different operands
  - different execution times for instructions
  - few processor registers
  - microprogrammed control logic

- **Reduced Instruction Set Computer (RISC)**
  - one instruction per clock cycle
  - memory accesses by dedicated load/store instructions
  - few addressing modes
  - hard-wired control logic
# Example Instruction (IA-32)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>04 ib</td>
<td>ADD AL,imm8</td>
<td>Add imm8 to AL</td>
</tr>
<tr>
<td>05 iw</td>
<td>ADD AX,imm16</td>
<td>Add imm16 to AX</td>
</tr>
<tr>
<td>05 id</td>
<td>ADD EAX,imm32</td>
<td>Add imm32 to EAX</td>
</tr>
<tr>
<td>80 /0 ib</td>
<td>ADD r/m8,imm8</td>
<td>Add imm8 to r/m8</td>
</tr>
<tr>
<td>81 /0 iw</td>
<td>ADD r/m16,imm16</td>
<td>Add imm16 to r/m16</td>
</tr>
<tr>
<td>81 /0 id</td>
<td>ADD r/m32,imm32</td>
<td>Add imm32 to r/m32</td>
</tr>
<tr>
<td>83 /0 ib</td>
<td>ADD r/m16,imm8</td>
<td>Add sign-extended imm8 to r/m16</td>
</tr>
<tr>
<td>83 /0 ib</td>
<td>ADD r/m32,imm8</td>
<td>Add sign-extended imm8 to r/m32</td>
</tr>
<tr>
<td>00 /r</td>
<td>ADD r/m8,r8</td>
<td>Add r8 to r/m8</td>
</tr>
<tr>
<td>01 /r</td>
<td>ADD r/m16,r16</td>
<td>Add r16 to r/m16</td>
</tr>
<tr>
<td>01 /r</td>
<td>ADD r/m32,r32</td>
<td>Add r32 to r/m32</td>
</tr>
<tr>
<td>02 /r</td>
<td>ADD r8,r/m8</td>
<td>Add r/m8 to r8</td>
</tr>
<tr>
<td>03 /r</td>
<td>ADD r16,r/m16</td>
<td>Add r/m16 to r16</td>
</tr>
<tr>
<td>03 /r</td>
<td>ADD r32,r/m32</td>
<td>Add r/m32 to r32</td>
</tr>
</tbody>
</table>

**Execution time:**
- 1 cycle: ADD EAX, EBX
- 2 cycles: ADD EAX, memvar32
- 3 cycles: ADD memvar32, EAX
- 4 cycles: ADD memvar16, AX

**Instruction Width:**
- between 1 byte (NOP) and 16 bytes
Types of Microprocessors

- **Superscalar Processors**
  - subclass of RISCs or CISCs
  - multiple instruction pipelines for overlapping execution of instructions
  - parallelism not necessarily exposed to the compiler

- **Very Long Instruction Word (VLIW)**
  - statically determined instruction-level parallelism (under compiler control)
  - instructions are composed of different machine operations whose execution is started in parallel
  - many parallel functional units
  - large register sets
VLIW Architectures
Analog Devices SHARC

- 32-bit **DSP** for **speech, sound, graphics, and imaging** applications. ADSP-21060 used in Dolby Surround AV/R-Receivers.
- **Cycle time:** 25ns, **clock frequency:** 40 MHz
- **Memory:**
  - 2/4 MBit on-chip dual-ported SRAM
  - 48-bit PM (program memory) bus (DAG1)
  - 40-bit DM (data memory) bus (DAG2)
  - PM and DM can be accessed simultaneously
  - word addressable (default)
- **General purpose register file:** 16 40-bit registers used for fixed-point and floating-point data
Analog Devices SHARC

- Four types of **address registers** each consisting of 8 registers in DAG1 (32 bit) and 8 registers in DAG2 (24 bit):
  - index registers (I): base address of memory accesses
  - modify registers (M): offset/increment value
  - base registers (B): base address of circular buffer
  - length registers (L): length of circular buffer

- **Addressing modes**:
  - direct addressing
  - indirect addressing with pre- and post-modify
  - circular addressing
  - bit-reverse addressing
Analog Devices SHARC

- Three computation units:
  - ALU
  - multiplier with fixed-point accumulator
  - shifter
  - restricted parallelism of ALU and multiplier
- Load/store architecture
- Harvard architecture allows simultaneous accesses to DM and PM; however the parallelism is restricted.
- Instruction width: 48 bit
- Instruction set covers min/max, clipping, multiply/accumulate, bit extraction and insertion, shift and rotate operations
- Predicated execution; guards are flags of control and status registers
Analog Devices SHARC

- The ADSP-2106x processes instructions in three clock cycles (best case assumption).

- 3-stage execution pipeline:
  - fetch: instruction is read from cache or program memory
  - decode: instruction is decoded
  - execute: execution of the microoperations

- Pipelining allows for a throughput of one instruction per clock cycle.
Analog Devices SHARC

- Restricted **parallelism** between ALU and multiplier.

\[
\begin{align*}
\text{R1} &= \text{R1} \times \text{R4} \\
\text{R2} &= \text{R8} + \text{R12}
\end{align*}
\]

\[
\begin{align*}
\text{R1} &= \text{R1} \times \text{R3} \\
\text{R2} &= \text{R8} + \text{R12}
\end{align*}
\]
TriMedia TM1000

- Media processor for real-time processing of audio, video, graphics and communication.
- **DSPCPU**: 100 MHz VLIW core CPU; described in the following.
- 32-bit VLIW processor with 128 general purpose registers and 27 functional units.
- 16 KB dual-ported data cache and 32 KB instruction cache, both 8-way set-associative with 64 KB block size.
- Load/store architecture.
TriMedia TM1000

- Instruction set is composed of **RISC-like operations, multimedia and DSP operations** (SIMD; e.g. ifir)
- Each **instruction word** is composed of 5 microoperations.
TriMedia TM1000

- **Execution time** of operations is between 1 and 17 clock cycles.

- The **packing** of operations into instructions (assignment of operations to issue slots of the instruction word) is restricted:
  - E.g. no more than two load/store operations can be packed into a single instruction.
  - No more than 5 results can be written during any one cycle.

- **Predicated execution**: each operation can optionally be guarded. The operation is executed depending on the value of the general-purpose register specified as guard operand.
TriMedia TM1000

- Feasible issue slot combinations:

<table>
<thead>
<tr>
<th>Type</th>
<th>Slot1</th>
<th>Slot2</th>
<th>Slot3</th>
<th>Slot4</th>
<th>Slot5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Example code sequence:

(* cycle 0 *)
IF r1  iaddi(0x2) r0 -> r38,  IF r1  isubi(0x4) r0 -> r7,
IF r1  isubi(0x3) r0 -> r8,  IF r1  isubi(0x2) r0 -> r36,
IF r1  isubi(0x1) r0 -> r37;

(* cycle 1 *)
IF r1  iaddi(0x3) r0 -> r39,  IF r1  uimm(0x190) -> r34,
IF r1  iaddi(0x8) r0 -> r35,  IF r1  h_st8d(0) r7 r6,
IF r1  h_st8d(4) r0 r6;