Characteristics of DSPs

- Multiply-accumulate units: multiplication and accumulation in a single clock cycle (vector products, digital filters, correlation, fourier transforms, etc)
- Multiple-access memory architectures for high bandwidth between processor and memory
  – Goal: throughput of one operation per clock cycle.
  – Required: several memory accesses per clock cycle.
  – Separate data and program memory space: harvard architecture.
  – Multiple memory banks
  – Arithmetic operations in parallel to memory accesses. But often irregular restrictions.

Characteristics of DSPs

- Specialized addressing modes:
  - linear addressing mode: addition/subtraction of offset to/from base address, often with auto-modify (pre-/post increment/decrement)
  - circular addressing: address results from incrementing an address register and taking the remainder of the division of this value by a constant.
  - bit-reverse addressing: carry bits are propagated in bit-reverse order (-> FFT)

Characteristics of DSPs

- Branches depending on control bits: early branch detection allowing for small interlock penalties in combination with typically short pipelines
- Residual control: execution behavior depends on specific control bits set by a preceding instruction.
- Predicated/guarded execution: instruction execution depends on the value of explicitly specified bit values or registers.
- Increasingly: SIMD instructions for packed arithmetic (multimedia applications).

Characteristics of DSPs

- Hardware loops / zero overhead loops: no explicit loop counter increment/decrement, no loop condition check, no branch back to top of loop
- Restricted interconnectivity between registers and functional units -> more severe phase coupling problems.
- Strongly encoded instruction formats: a throughput of one instruction per clock cycle requires one instruction to be fetched per cycle. Thus each instruction has to fit in one memory word -> reduction of bit width of the instruction.

Characteristics of DSPs

Techniques for reducing the instruction width:

- Reducing the number of addressing modes. Example: immediate memory accesses restricted to a small set of instructions.
- Restricting the set of source and destination operands: short addressing modes for elements of certain register groups, implied operands.
- Mode bits, for example a single shift instruction that performs arithmetic or logical shifts depending on a control bit in a mode register.
- Consequence: Increased irregularity of the instruction set – but reduction of processor and system cost and increased individual instruction performance.

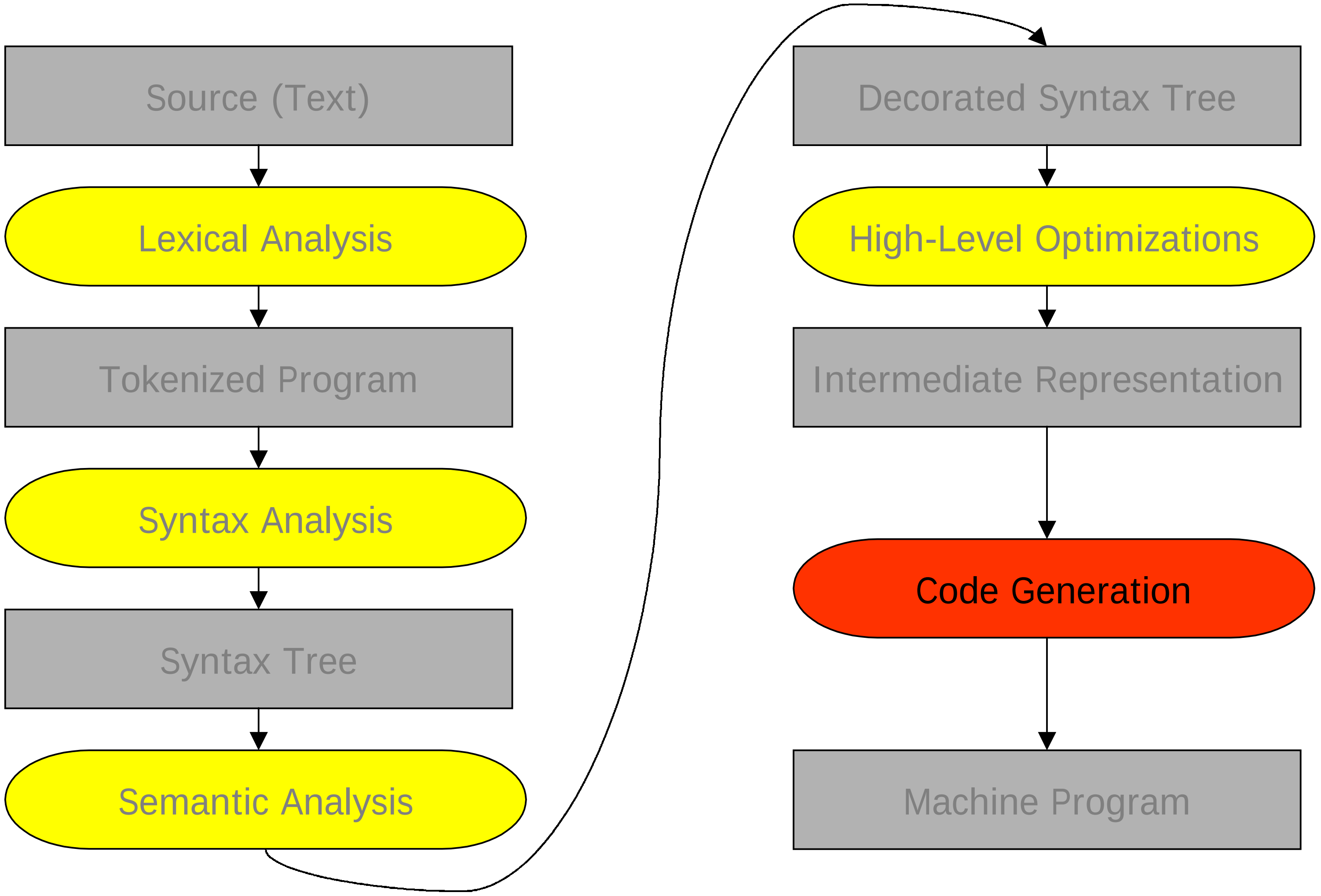
Characteristics of DSPs

- Overall consequence:
  - Irregularity
  - Severe phase coupling problems during code generation
  - Need for specialized algorithms

Next Lectures

Compiler Construction and Code Generation

Repetition: Compiler Structure



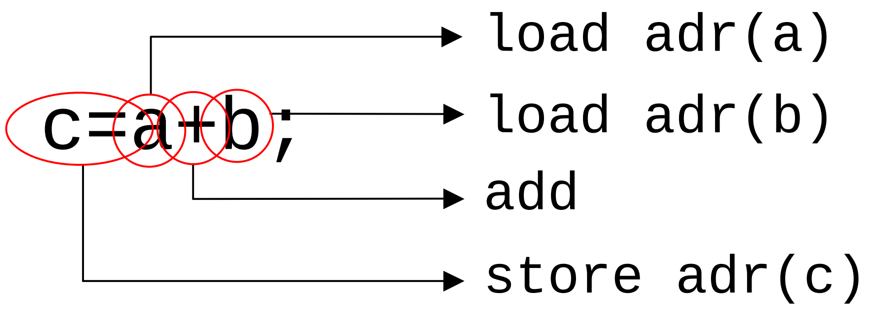
Middle End: High-Level Optimizations

- High-level optimizations are usually termed machine-independent optimizations. They comprise e.g. dead code elimination, constant propagation, constant folding, common subexpression elimination, loop unrolling, loop fusion, software pipelining,...

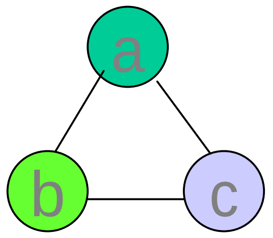
- BUT: Many machine-independent optimizitations are not machine-independent at all. For example:
  - constant folding may lead to large immediate constants resulting in code growth or preventing instruction-level parallelism
  - common subexpression elimination may increase the register pressure and cause problems if few registers are available
  - loop unrolling may cause the instruction cache to overflow

Back End: Code Selection, Register Allocation, and Instruction Scheduling

Code selection



Register allocation



$a \mapsto r1$
$b \mapsto r2$
$c \mapsto r3$

Instruction scheduling

```
r1=load adr(a) || r2=load adr(b)
r3=add r1, r2
store adr(c), r3
```

Main Tasks of Code Generation (1)

- Code selection: Map the intermediate representation to a semantically equivalent sequence of machine operations that is as efficient as possible.
- Register allocation: Map the values of the intermediate representation to physical registers in order to minimize the number of memory references during program execution.
  - Register allocation proper: Decide which variables and expressions of the IR are mapped to registers and which ones are kept in memory.
  - Register assignment: Determine the physical registers that are used to store the values that have been previously selected to reside in registers.

Main Tasks of Code Generation (2)

- Instruction scheduling:
  Reorder the produced operation stream in order to minimize pipeline stalls and exploit the available instruction-level parallelism.

- Resource allocation / functional unit binding:
  Bind operations to machine resources, e.g. functional units or buses.

The Code Generation Problem

- Instruction scheduling, register allocation and code selection are NP complete problems.
- In classical approaches they are addressed by heuristic methods in separate phases.
- Unfortunately, all the code generation phases are interdependent, i.e. decisions made in one phase may impose restrictions to the other phases.
- Thus: often suboptimal combination of suboptimal partial results.
- Moreover: specific/irregular hardware features not well covered by standard code generation methods.

Calling Conventions

- Calling conventions specify how the procedure stack is built and how to pass parameters between functions and procedures.

- Calling conventions have to be respected during code transformations. Optimization opportunities can only be exploited based on interprocedural analyses of the complete ICFG.

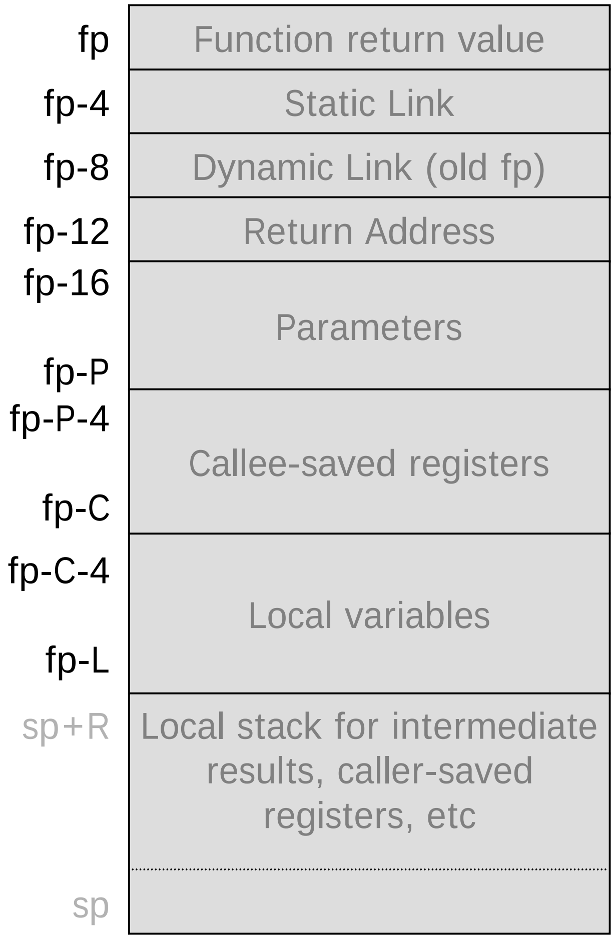
- Calling conventions of the GHS TriCore C compiler (excerpt):
  - Up to 4 32-bit data arguments are passed in registers D4-D7.
  - Up to 2 64-bit data arguments are passed in register pairs E4 and E6.
  - Up to 4 32-bit address arguments are passed in registers A4-A7.
  - Arguments that cannot be passed in registers are passed on the stack.
  - 32-bit data (address) return values are returned in D2 (A2); 64-bit data values in E2 (D2/D3).

Calling Conventions (c'ed)

– Caller-saved registers: caller is responsible for saving these registers on the stac and for restoring them after the callee has returned.

– Callee-saved registers: callee is responsible for saving these registers upon procedure entry and for restoring them before returning.

Typical Stack Frame Layout



- Static link: fp of the static predecessor (variable accesses)
- Dynamic link: fp of dynamic predecessor, i.e. the fp-value that has to be restored when returning from the current function.
- Return address: address of the next instruction to be executed after return from the current function.

- Usually stack frames are arranged in memory so that the beginning is at a higher address than the end of it. This way offsets from the stack pointer are always non-negative.

Program Representations

- Abstract Syntax Tree (AST)
- Static Single Assignment (SSA)
- Control Flow Graph (CFG), Call Graph (CG) and Interprocedural Control Flow Graph (ICFG)
- Data Dependence Graph (DDG)
- Low-Level Intermediate Representation:
  Abstract Machine Code / Register Transfer Languages

High-Level and Low-Level IRs

- High-level intermediate representation: close to source level. Typically centered around source language constructs. Constructs: implicit memory addressing, expression trees, for- while-, switch-statements, etc.

- Low-level intermediate representation: close to machine level. Typically centered around basic entities that specify properties of machine operations.

- Most program representations can be defined at high-level and at low-level.

IR Levels

High-Level

```
t1 = a[i][j+3];
```

Medium-Level

```
t1 = addr(a);
t2 = i*20;
t3 = j+3;
t4 = t2+t3;
t5 = 4*t4;
t6 = t1+t5;
t7 = *t6;
```

Low-Level

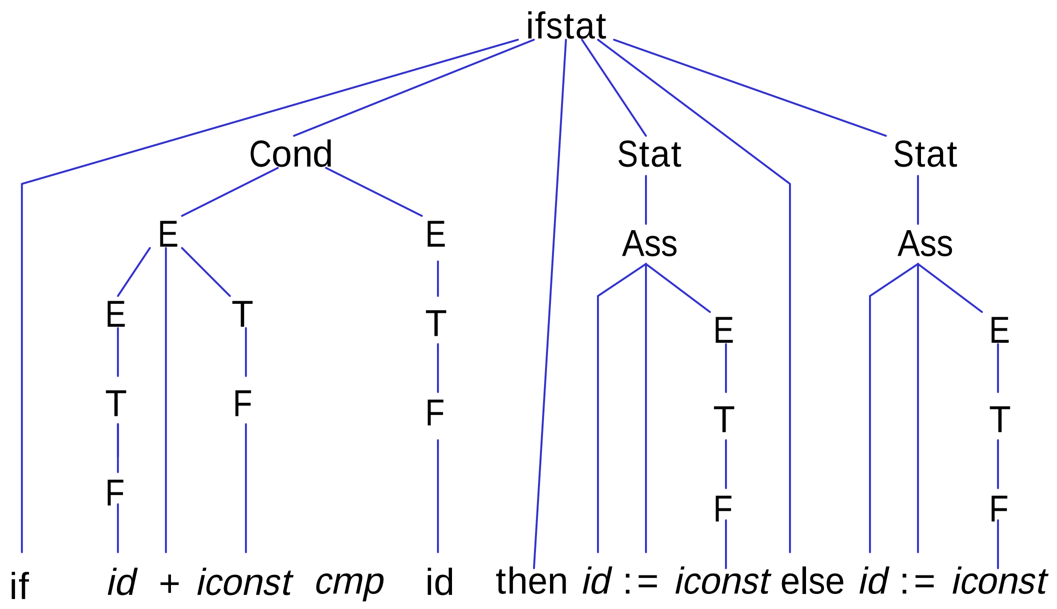
```
v1 = fp-216;
v2 = [fp-4];
v3 = v2*20;
v4 = [fp-8];
v5 = v4+3;
v6 = v3+v5;
v7 = 4*v6;
v8 = [v1+v7];
```

Assumption: Input language C, `a` *declared as* `int a[10][20]`*;*

IR Levels

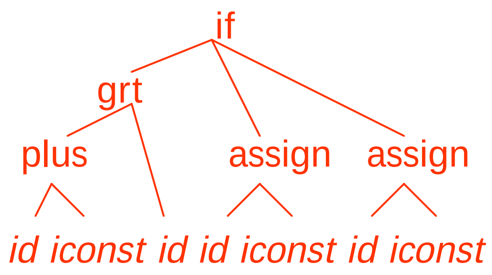
- High-level IRs:
  - abstract syntax tree
  - ➢ control flow graph and data dependence graph used for array dependence analysis and high-level code transformations
- Low-level IRs:
  - abstract machine code (medium-level)
  - direct representation of target machine instructions
  - register transfer language (machine-independent representation for machine-specific instructions)
  - ➢ control flow graph and data dependence graph used for machine-level dependence analysis and low-level code transformations

Decorated Abstract Syntax Tree


ifstat
Cond
Stat
Stat
E
E
Ass
Ass
E
T
T
E
E
T
F
F
T
T
F
F
F
if
id + *iconst* *cmp* id then *id* := *iconst* else *id* := *iconst*

concrete syntax tree

abstract syntax tree



Call Graph

- There is a node for the main procedure – being the entry node of the program – and a node for each procedure or function 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*.

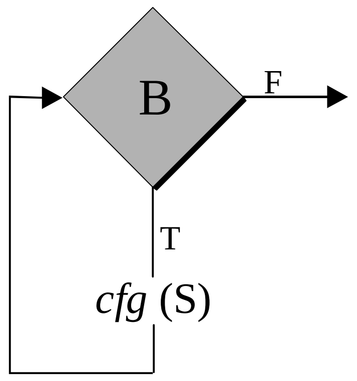
Control Flow Graph

- 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,λ) denote the control flow of the procedure: $\lambda \in \{T,F,\varepsilon\}$ 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_Ω denotes the end node that is reached by any path through the control flow graph.

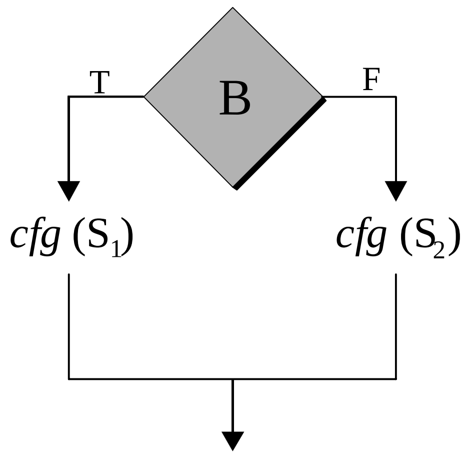
- Nodes with more than one predecessor are called joins and nodes with more than one successor are called forks.

Control Flow Graph – Composed Statements

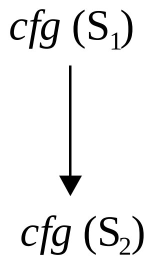
cfg (while B do S od) =



cfg (if B then S_1 else S_2 fi) =



cfg (S_1;S_2) =



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_Ω 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:

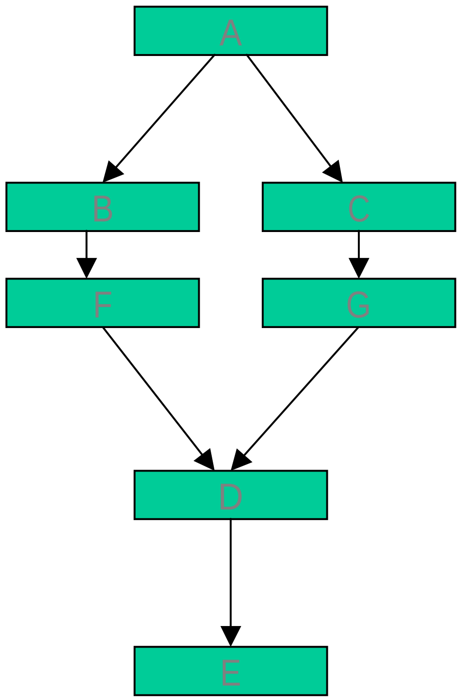
1. Call graph whose nodes are meta-nodes containing basic block graphs.
2. Basic block graph for each procedure in the program.
3. Ordered list of instructions for each block in the basic block graph of each procedure.

The ICFG describes the control flow of a program completely.

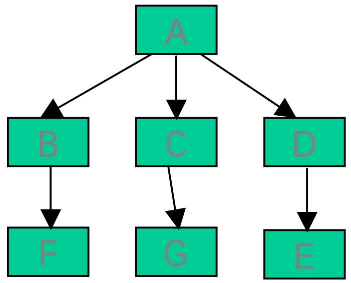
Control Dependence Graph

- Operation i dominates an operation j, if i appears on every path from the entry node of the procedure to j. Each operation dominates itself.
- Operation j postdominates i, if j appears on every path from i to the exit node of the procedure.
- Given a control flow graph $G_C=(N_C,E_C,n_A,n_\Omega)$. Node m $\in N_C$ is control dependent on n $\in N_C$ if
  - (n,a) is an edge of the control flow graph
  - m does not postdominate n
  - there is a path from n, a, ..., m so that m postdominates all nodes between n and m.
- The dominance frontier of a node x of the CFG (or BBG) is the set of all nodes y so that x dominates an immediate predecessor of y, but not y itself.

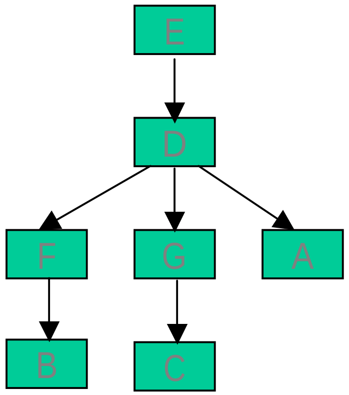
Control Dependence



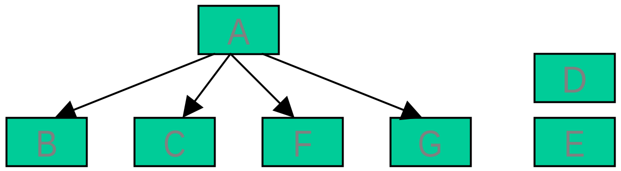
Control Flow Graph



Dominator Tree



Postdominator Tree



Control Dependence Graph

{D} is dominance frontier of B, C, F, G

Data Dependence Graph Low Level View

- 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) r1 = r2*r3;
             (1, 2, r1, t)
(2) r5 = r1+r1;
```

Data Dependence Graph High Level View

- Iteration distance: Number of loop iterations between two dependent instruction instances (0 for intra-iteration dependences).
- Delay: Minimal number of clock cycles between the issuing of two dependent operation instances.
- Edges of the DDG are labeled with (*itDist*, *delay*,*type*).
- The delay for a dependence $a \rightarrow b$ depends on the latencies of *a* and *b* and the type of the dependence:
  - true dependence (def-use): *latency(a)*
  - anti dependence (use-def): *1 - latency(b)*
  - output dependence (def-def): *1 + latency(a) - latency(b)*

```
for (i=2; i<100; i++) {
  (a) A[i]=B[i]+C[i];
                (a, b, 2, 1, t)
  (b) D[i]=A[i-2];
}
```