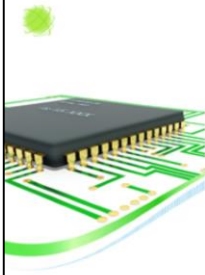


Definition

Back end is the processing phase that maps the IR to the target representation that is ready to perform the intended computation.

Extends the meaning,
regards the target architectures.

An intermediate representation uses the results of the preceding phases (lexical analysis, syntax analysis, semantic analysis) and performs more processing to make it easy generation of targets. While the structures generated by the semantic analysis phase describes what is in the input, the IR generation phase regards the target architecture so that target generation is possible.



Architecture

- Programmers see the computer through the window provided by the designers at the level of its functional architecture.
- This window is provided by you, the designer. (1)

❖The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation. (2)

1. From Prof. Dr. Bozşahin's CENG444 course lecture.
2. Amdahl, Gene & Blaauw, Gerrit & Brooks, Jr, Frederick. (2000). Architecture of the IBM System/360. IBM Journal of Research and Development. 44. 21-36. 10.1147/rd.82.0087.



Architecture

```
int f(int p)
{
    if (p>1)
        return p*f(p-1);

    return 1;
}
```

Identical semantics mapped different
Instruction Set Architectures (ISA)

Translation requires sound knowledge on
the target ISA

<pre>000077ff79ef52320 09 4c 24 00 mov dword ptr [rsp+8],ecx 000077ff79ef52324 55 push rbp 000077ff79ef52325 57 push rdi 000077ff79ef52326 48 b1 cc e8 00 00 00 sub rbp,0f0h 000077ff79ef52320 48 b0 6c 24 20 lea rbp,[rsp+20h] 000077ff79ef52332 48 bd 00 f0 0c 01 00 lea rcx,[_f791f1de_lecture01@cpp (07ff79ef63029h)] 000077ff79ef52339 e8 aa f0 ff ff call ___CheckForDebuggerJustlyCode (07ff79ef5136h) 000077ff79ef5233e 5f (pop) 000077ff79ef5233e 83 bd e8 00 00 00 01 cmp dword ptr [p],1 000077ff79ef52345 7e 1c jle f443h (07ff79ef52363h) 000077ff79ef52347 05 05 e8 00 00 00 mov eax,dword ptr [p] 000077ff79ef5234d ff c8 dec eax 000077ff79ef5234f 08 c8 mov ecx,eax 000077ff79ef52351 e8 75 ff ff ff call f (07ff79ef512cbh) 000077ff79ef52356 b8 bd e8 00 00 00 mov ecx,dword ptr [p] 000077ff79ef5235c 0f af c8 imul ecx,eax 000077ff79ef5235f 08 c1 mov eax,ecx 000077ff79ef52361 e8 05 jmp f440h (07ff79ef52363h) 000077ff79ef52363 05 01 00 00 00 mov eax,1 000077ff79ef52368 48 b0 a5 c8 00 00 00 lea rdi,[rbp+0cbh] 000077ff79ef5236f 5f pop rdi 000077ff79ef52370 5d pop rbp 000077ff79ef52371 c3 ret</pre>	<pre>0x10000309c <40>: sub sp, sp, #0x20 0x1000030a0 <44>: stp x29, x30, [sp, #0x10] 0x1000030a4 <48>: add x29, sp, #0x10 0x1000030a8 <12>: str w0, [sp, #0x8] 0x1000030ac <16>: ldr w0, [sp, #0x8] 0x1000030b0 <20>: sub w0, w0, #0x1 0x1000030b4 <24>: cset w0, le 0x1000030b8 <28>: tbz w0, #0x0, 0x1000030e4 ; <72> at main.cpp 0x1000030bc <32>: b 0x1000030c0 ; <36> at main.cpp:13:14 0x1000030c0 <36>: ldr w0, [sp, #0x8] 0x1000030c4 <40>: str w0, [sp, #0x4] 0x1000030c8 <44>: ldr w0, [sp, #0x8] 0x1000030cc <48>: sub w0, w0, #0x1 0x1000030d0 <52>: bl 0x10000309c ; <40> at main.cpp:11 0x1000030d4 <56>: ldr w0, [sp, #0x4] 0x1000030d8 <60>: mul w0, w0, w0 0x1000030dc <64>: stur w0, [x29, #0x4] 0x1000030e0 <68>: b 0x1000030f0 ; <84> at main.cpp:16:1 0x1000030e4 <72>: mov w0, #0x1 0x1000030e8 <76>: stur w0, [x29, #0x4] 0x1000030ec <80>: b 0x1000030f0 ; <84> at main.cpp:16:1 0x1000030f0 <84>: ldr w0, [x29, #0x4] 0x1000030f4 <88>: ldp x29, x30, [sp, #0x10] 0x1000030f8 <92>: add sp, sp, #0x20 0x1000030fc <96>: ret</pre>
---	--

x64 - CISC

ArmV8 - RISC

In most cases, a compiler's ultimate output is the executable code, which is the finite sequences of the instructions that will be fed to the CPU. Each CPU exposes its programmability through its ISA (Instruction Set Architecture), which defines the instructions by groups (such as those arithmetic and logic, status and flow control, data moving, and so on), addressing modes in conjunction with memory and IO management mechanisms, register file, modes of operation (word length, process space size, privilege levels, and so on), properties critical to concurrency control, and more. The ISA is the most critical, major determinant of the translation to be performed by the compiler.



Architecture

```
int f(int p)
{
    if (p>1)
        return p*f(p-1);

    return 1;
}
```

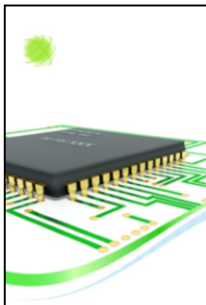
Identical semantics mapped different
Instruction Set Architectures (ISA)

ISA can also be defined at software level
that may be interpreted or translated further
targeting another lower level ISA.

```
15 ssr 1[1 0x00000001] ; Prologue f 50n(10n10n)
16 ssr 29[90 0x0000002a] ; Debug expression prologue p>1
17 pmw base pointer offset -3
18 psh int8 1 0x01
19 cvt_i_t -1 regs
20 gre_1
21 clf
22 ssr 3[1 0x00000001]
23 jz 37
24 ssr 29[109 0x0000000d] ; Debug expression prologue p*f(p-1);
25 pmw base pointer offset -3
26 psh signature entry point 1
27 pmw base pointer offset -4
28 pmw base pointer offset -3
29 psh int8 1 0x01
30 cvt_i_t -1 regs
31 sub 1
32 clf 1
33 ssr 6
34 mul 1
35 ssr 5[1 0x00000001]
36 jmp 41
37 ssr 29[138 0x0000000a] ; Debug expression prologue 1;
38 psh int8 1 0x01
39 cvt_i_t -1 regs
40 ssr 5[1 0x00000001]
41 ssr 2[1 0x00000001] ; Epilogue f 50n(10n10n)
42 rtf 1
```

A sample stack machine

Instruction Set Architecture can be defined at software level for the purposes of emulation, interpretation, or similar. The target generated for software defined ISA can also be translated further to processor level ISA. The just in time (JIT) compiler that is part of Java execution model is a good example of this two level. According to this model, Java source is translated to bytecodes by the java compiler, then the JIT compiler translates the bytecode to the processor native code to enable execution. You can see the Oak as a historical mark, Dalvik as a virtually modern approach on Android, ART as a target scheme on Android.



Interoperability over ABI

The Application Binary Interface

- Use of Register File
 - Shape of Activation Records
 - Implementation of Specific Calling Conventions
-
- Memory Organization (Stack,)
 - Roles specific to the Registers
 - Registers to Scratch
 - Registers to Preserve

An application binary interface is a set of standards / recommendations that governs the data flow between the code units. When respected by the back-ends the well established ABI enables integration of the code units regardless their source languages. Generally, an Application Binary Interface defines the conventions used in parameter passing, value returning, register utilization. Stack and memory organization may also be addressed to a certain extent. An ABI defines the conventions specific to a certain architecture. Ideally, an ABI responds to every possible calling convention and related formal structures of the activation records.



Architecture

Identical semantics mapped different Instruction Set Architectures (ISA)

Application Binary Interface (ABI) as OS dependent architectural manifestation.

```
int f(int p)
{
    if (p>1)
        return p*f(p-1);

    return 1;
}
```

```
000077ff79ef52320 89 4c 24 00      mov     dword ptr [rsp+8],ecx
000077ff79ef52324 55              push   rbp
000077ff79ef52325 57              push   rdi
000077ff79ef52326 48 b1 ec 00 00 00 sub     rsp,000h
000077ff79ef52320 48 b0 0c 24 20    lea     rbp,[rsp+20h]
000077ff79ef52322 48 d0 00 f0 0c 01 00 lea     rcx,[_F793f1DE_lecture01@cpp (07ff79ef53029h)]
000077ff79ef52329 e8 aa f0 ff ff    call    _CheckForDebuggerJustlyCode (07ff79ef5338h)
000077ff79ef5233e if (p>1)
000077ff79ef5233e 83 80 e0 00 00 01 cmp     dword ptr [p],1
000077ff79ef52345 7e 1c           jle     f443h (07ff79ef52363h)
000077ff79ef52347 8b 05 e0 00 00 00 mov     eax,dword ptr [p]
000077ff79ef52347 8b 05 e0 00 00 00 mov     ecx,ecx
000077ff79ef5234f b8 c8           mov     ecx,ecx
000077ff79ef52351 e8 75 ff ff ff    call    f (07ff79ef512c0h)
000077ff79ef52356 48 d0 e0 00 00 00 mov     rcx,dword ptr [p]
000077ff79ef5235c 0f af c8        imul    ecx,ecx
000077ff79ef5235f 8b c1           mov     eax,ecx
000077ff79ef52361 e8 05           jmp     f440h (07ff79ef52360h)
000077ff79ef52363 8b 01 e0 00 00 00 mov     eax,1
000077ff79ef52368 48 bd a5 c8 00 00 00 lea     rsp,[rbp+0c0h]
000077ff79ef5236f 5f              pop     rdi
000077ff79ef52370 5d              pop     rbp
000077ff79ef52371 c3              ret
```

x64 – Windows ABI

<https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170>

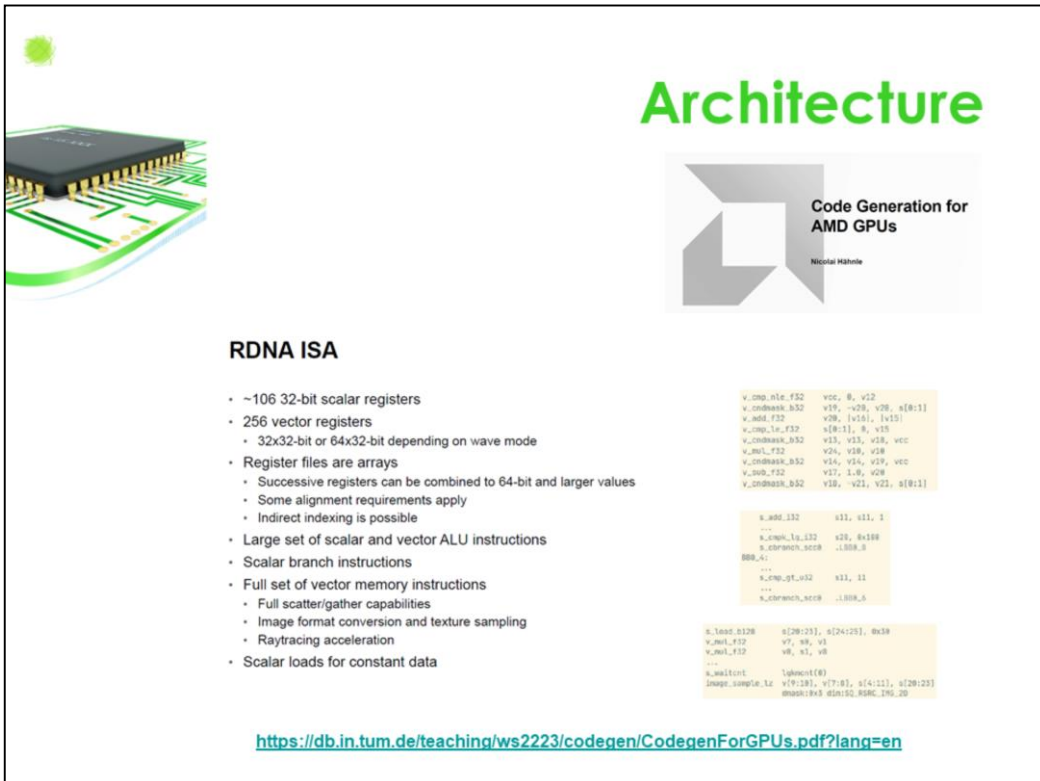
```
00005555555551cd: push    %rbp
00005555555551ce: mov     %rsp,%rbp
00005555555551d1: sub     $0x10,%rsp
00005555555551d5: mov     %edi,-0x4(%rbp)
00005555555551d8: if (p>1)
00005555555551d8: cmpl    $0x1,-0x4(%rbp)
00005555555551dc: jle     0x5555555551f1 <_Z1fi+40>
00005555555551de: return p*f(p-1);
00005555555551de: mov     -0x4(%rbp),%eax
00005555555551e1: sub     $0x1,%eax
00005555555551e4: mov     %eax,%edi
00005555555551e6: call    0x5555555551c9 <_Z1fi>
00005555555551eb: imul    -0x4(%rbp),%eax
00005555555551ef: jmp     0x5555555551f6 <_Z1fi+45>
00005555555551f1: return 1;
00005555555551f1: mov     $0x1,%eax
00005555555551f6: leave
00005555555551f7: ret
```

x64 – Linux ABI

<https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/linux-x64-calling-convention-stack-frame>

Even if the set instruction set architectures are the most dominant determinant in generation of the back-ends, the software layers that underpin the execution of the programs must also be considered as part of the architecture which the compiler must be conformant to. The application binary interface (ABI) requirements that define the parameter passing conventions can be different between the operating systems even if they run on the same hardware. The 64 bit versions of Windows and Linux use different ABIs so the code generators must be developed keeping the differences in mind even if they run on the same Intel based PCs for example. On top of these, it is quite possible to develop a code generator that uses totally different, custom ABI to run code in an isolated fashion for some application specific reasons. Integer parameters are passed using 4 register fast call on Windows (RCX, RDX, R8, and R9), 6 register fast call on Linux (RDI, RSI, RDX, RCX, R8, R9). There are more conventional differences that have to be comprehended and applied in machine code synthesis.

For more, see <https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170> for windows, <https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/linux-x64-calling-convention-stack-frame> for Linux.



The slide is titled "Architecture" in large green letters. On the left, there is a 3D illustration of a GPU chip on a circuit board. On the right, there is a logo for "Code Generation for AMD GPUs" by "Nicolai Hähle". The main content is the "RDNA ISA" section, which lists features of the architecture and provides examples of assembly code.

RDNA ISA

- ~106 32-bit scalar registers
- 256 vector registers
 - 32x32-bit or 64x32-bit depending on wave mode
- Register files are arrays
 - Successive registers can be combined to 64-bit and larger values
 - Some alignment requirements apply
 - Indirect indexing is possible
- Large set of scalar and vector ALU instructions
- Scalar branch instructions
- Full set of vector memory instructions
 - Full scatter/gather capabilities
 - Image format conversion and texture sampling
 - Raytracing acceleration
- Scalar loads for constant data

```

v_cmp_nle_f32 vcc, R, v12
v_cdbash_b32 v19, v28, v29, s[R:1]
v_add_f32 v29, [v16], [v15]
v_cmp_le_f32 s[R:1], R, v15
v_cdbash_b32 v13, v13, v18, vcc
v_mul_f32 v24, v18, v18
v_cdbash_b32 v14, v14, v19, vcc
v_sub_f32 v17, v16, v08
v_cdbash_b32 v18, v21, v21, s[R:1]

s_mov_i32 s11, s11, 1
...
s_cdbash_i32 s28, s188
s_branch_scc8 .LBBR_5
BBR_4:
...
s_cmp_gt_v32 s11, 11
...
s_branch_scc8 .LBBR_5

s_load_b16 s[20:23], s[24:25], s38
v_mul_f32 v7, s9, v1
v_mul_f32 v8, s1, v8
...
s_waitcnt lgprcnt(1)
smgp_sample_t2 v[7:8], v[7:8], s[4:11], s[20:23]
smgp_s3x3_ssm(s0, s0MC, 16, 20)

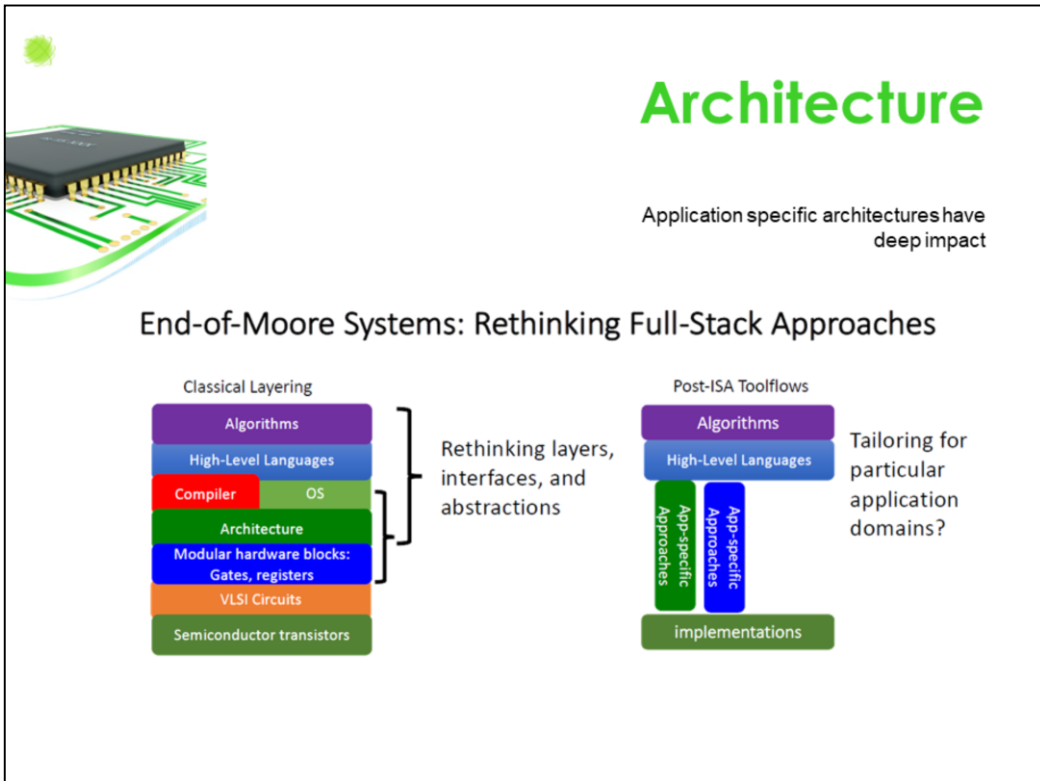
```

<https://db.in.tum.de/teaching/ws2223/codegen/CodegenForGPUs.pdf?lang=en>


However, the lower end architecture has more to do with from the perspective of a compiler. There may be cases where the hardware elements from external to the CPU must be considered. As a contemporary hot topic, GPU code generation can be given as an example. NVidia has a custom C++ compiler (CUDA C - nvcc) to generate and run GPU kernels. Industry leader companies are in continuous research and development phases aiming at better GPU code generation.

See, <https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/> to have a rough idea on a customized language processor.

See the presentation from AMD as a very informative resource, “Nicolai Hähle, Code Generation for AMD GPUs, 2023, AMD”. Source: <https://db.in.tum.de/teaching/ws2223/codegen/CodegenForGPUs.pdf?lang=en> Retrieved on Feb 2nd, 2024.



As a final remark on the architecture, we must consider the architectural properties that may span multiple layers of a system. Hardware and lower-level software components may become largely variable and application-specific in a way to force the developers to update multiple the layers of their language processors. The impact of alternatives may be so deep that changes in the language definition and the semantics of the whole translation becomes inevitable, most probably in an enriching fashion.



Code Generation

Mapping IC to Machine Code

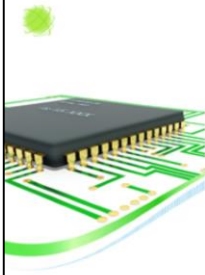
Constraints, conditions, problems

- ISA (Load - Store, Register - Memory)
- ABI
- Code and Data Organization
 - Generated code / data
 - RTTI
 - Object / target code standards
 - Intrinsic code / objects
 - More ...
- Instruction Selection
- Instruction Scheduling
- Register Allocation

In a way, the target architecture is a generalization that defines the limits of the computations that can be performed. In this sense, it defines the set of the target elements that can be composed up to meet the purpose of the source code.

Instruction Set Architectures (Register Memory, Register Register etc.), compiler runtimes (new operator in C++, software abstractions such as VMs, memory models defining a range of abstractions from allocation strategies to the variables, structures, words, etc.

Instruction selection, scheduling, register allocation. Ideally, the IR (and hence the IC) is isolated from the target architectures. But, it may be more feasible to derive hints for the code generators at the IR processing phase.



Code Generation

Instruction Selection

The complexity of instruction selection derives from the large number of alternative implementations that a typical ISA provides for even simple operations (*).

$6*i + 5$

...

`mov rax, [rbp + 16]`

`imul rax, 6`

`add rax, 5`

...

$4*i + 5$

...

`mov rcx, [rbp + 16]`

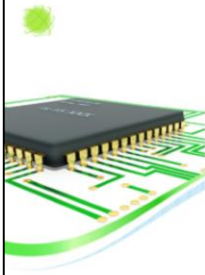
`lea rax, [5 + 4*rcx]`

...

Techniques may rely on pattern matching on both Graphical and Linear IR.

(*) Excerpt from "Cooper, K.D., Torczon, L.; Engineering A Compiler, 2nd Edition"

Instruction selection problem arises from the abundance of the alternatives. The code generator may be forced to make a choice between the shorter and the faster. This is highly dependent on the ISA architecture. This is a high complexity job because the execution and the memory costs must be analyzed separately for each possible combination.



Code Generation

Instruction Scheduling

Instruction scheduling attempts to reorder the operations in a procedure to improve its running time. In essence, it tries to execute as many operations per cycle as possible. (*)

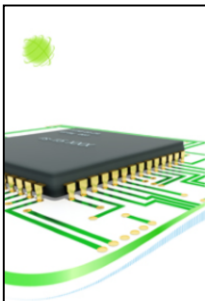
Superscalar architectures
Instruction level parallelism
Forcing architectures (Itanium)

Dependency analysis and out of order execution

Requires detailed analysis of processor specific parallelism

(*) Excerpt from "Cooper, K.D., Torczon, L.; Engineering A Compiler, 2nd Edition"

Reordering tries to meet optimization targets by moving the code around without affecting the correctness of the computation. Some architectures requires explicit instruction bundles which makes the scheduling problem more visible.



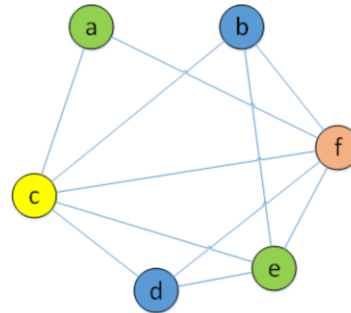
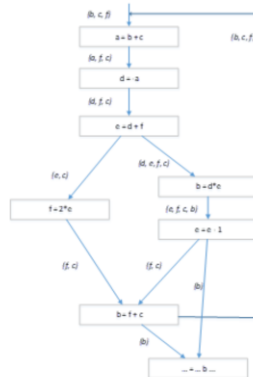
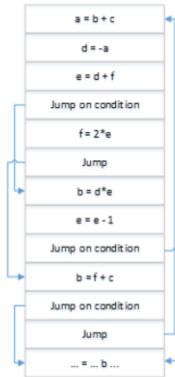
Code Generation

Register Allocation

Local / Global Register Allocation

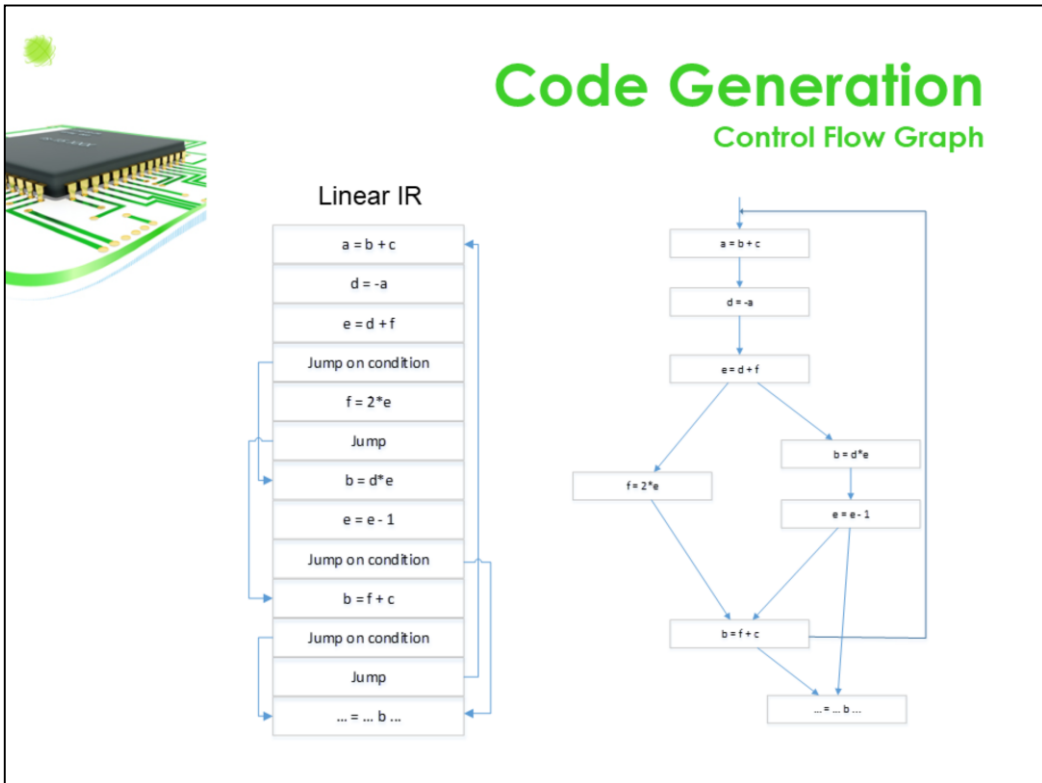
Graph Coloring Problem

Register Spilling, Heuristics, Interference Graphs, Live Ranges, ...

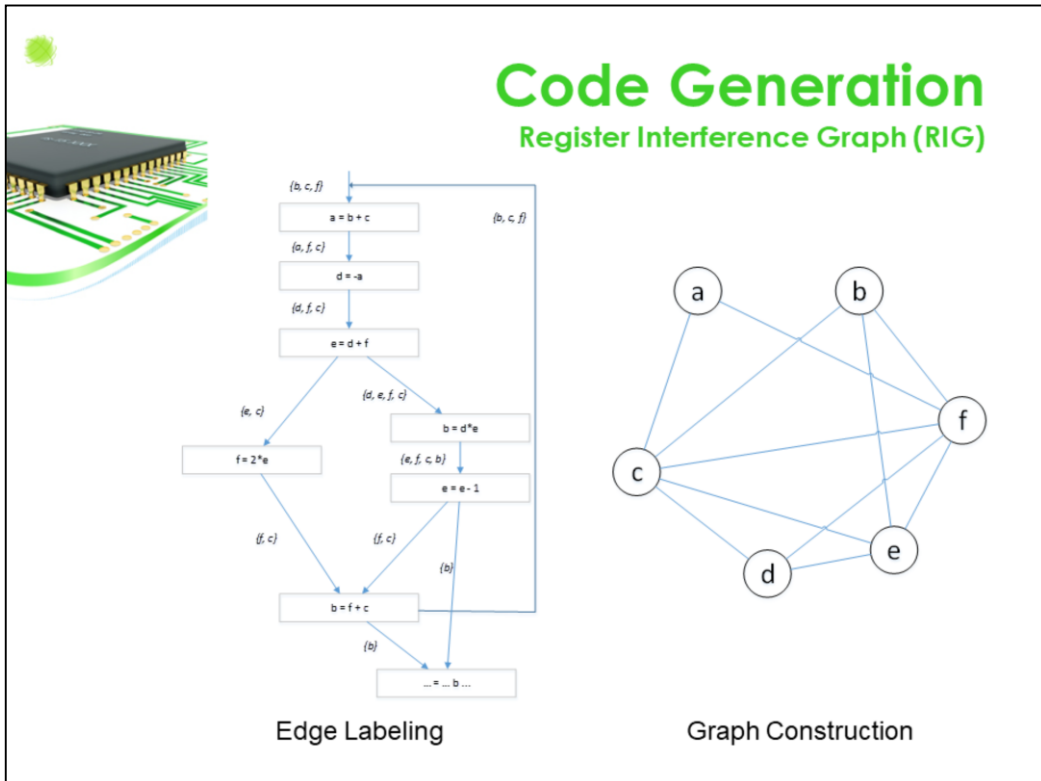


(*) CFG example from Portland State University CS536 Spring 2001 notes,
["http://web.cecs.pdx.edu/~mperkows/temp/register-allocation.pdf"](http://web.cecs.pdx.edu/~mperkows/temp/register-allocation.pdf)

Instruction selection, scheduling, register allocation. Ideally, the IR (and hence the IC) is isolated from the target architectures. But, it may be more feasible to derive hints for the code generators at the IR processing phase.



The method of generating control may be based linear IR or a completely new graph representation may be preferred. In the former case, an edge container refers to the liner IR nodes. The latter method constructs the graph from the linear IR and uses the derived representation at its own right. The CFG is helpful for identifying dead code, moveable code, register allocation, and more.

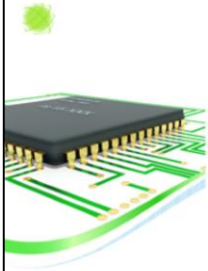


The edges on the control flow graph is labeled to identify the lifetime of the virtual variables.

The nodes on the graph visited in a loop and following two steps must be performed as long as a label change occurs in the whole graph:

- (1) If a node requires a variable to complete an operation, that variable must appear on all of the incoming edges.
- (2) Each of the variables on the outgoing edges must be copied into the label of the incoming edges except the one that is assigned on that node.

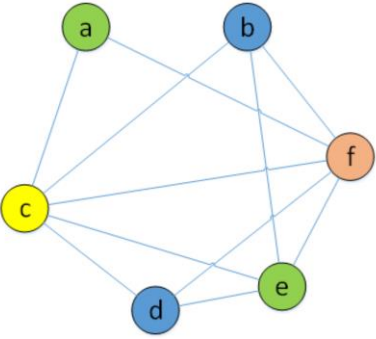
Note that the operations on the nodes of the example graph are compatible with three address notation.



Code Generation

Register Coloring

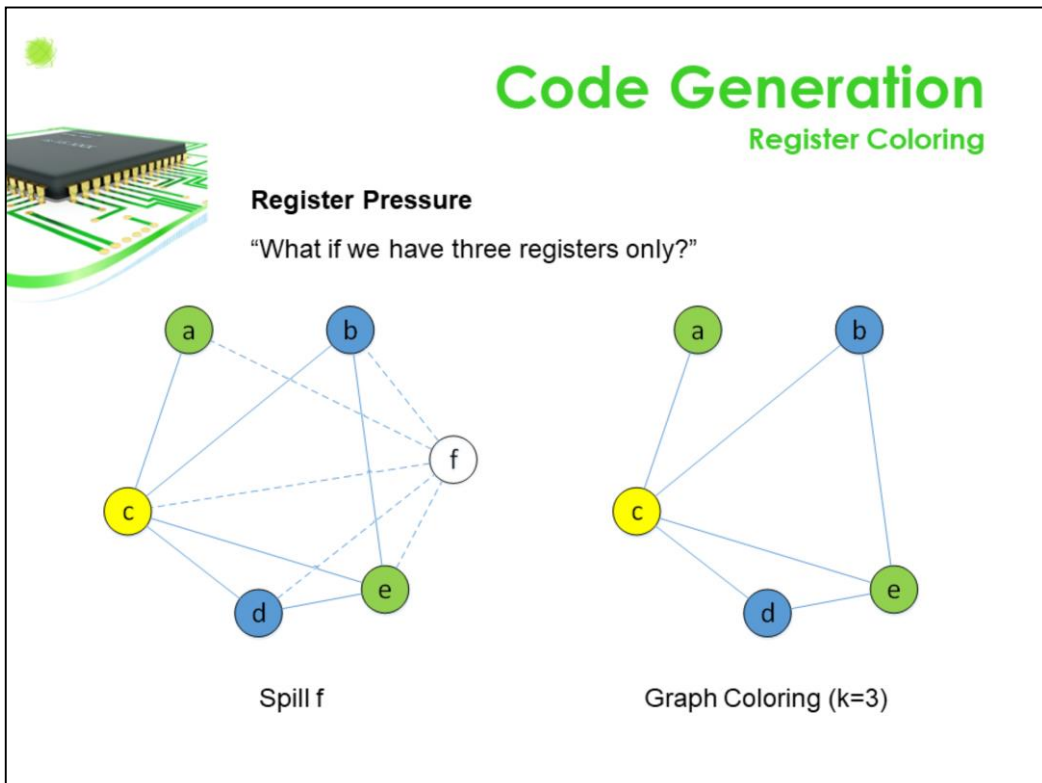
Register Pressure
“Do we have four registers?”




Graph Coloring ($k=4$)

You can see "Chaitin, G.J.; Register Allocation & Spilling via Graph Coloring; IBM Research; NY, 1982"

There are different coloring strategies based on heuristics. In 1982, Chaitin proposed a method that views the register allocation problem as a graph coloring problem.



When graph is colored after spilling, the whole liveness analysis and subsequent steps are repeated. Now, access to f is memory based! Stack is the usual destination for storing spilled variables. These are compiler generated temporaries materialized in memory. Register pressure is a term to address the mismatch between the live variables and the number of the available registers at some execution point.

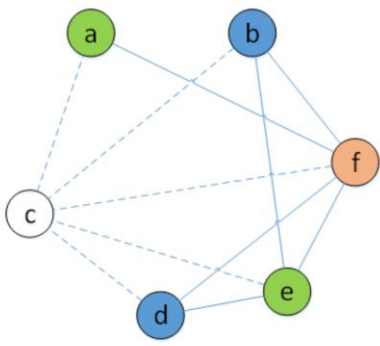


Code Generation

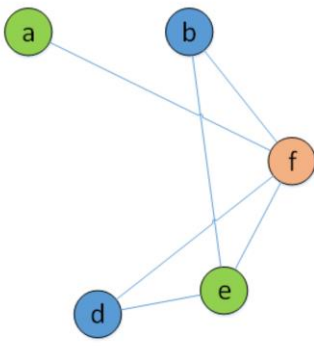
Register Coloring

Register Pressure

“What if we have three registers only?”



Spill c



Graph Coloring ($k=3$)

There are more than one spilling options!