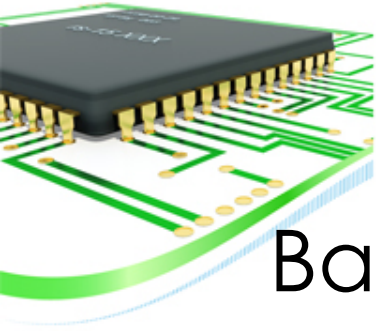


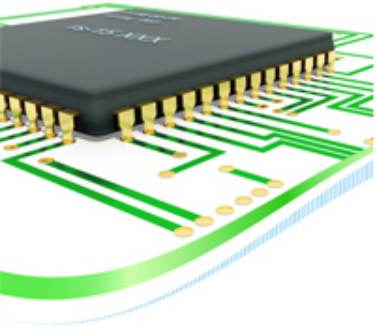
Back End



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.

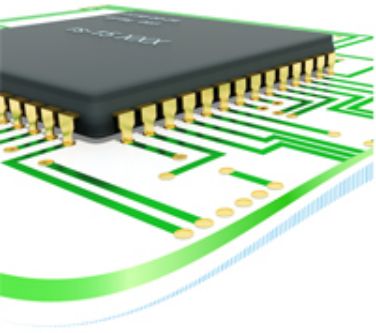


Architecture

- 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.



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

    return 1;
}
```

Architecture

Identical semantics mapped different
Instruction Set Architectures (ISA)

Translation requires sound knowledge on
the target ISA

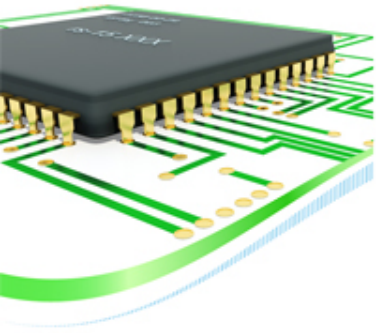
```
00007FF79EF52320 89 4C 24 08      mov     dword ptr [rsp+8],ecx
00007FF79EF52324 55           push    rbp
00007FF79EF52325 57           push    rdi
00007FF79EF52326 48 81 EC E8 00 00 sub     rsp,0E8h
00007FF79EF5232D 48 8D 6C 24 20  lea     rbp,[rsp+20h]
00007FF79EF52332 48 8D 0D F0 0C 01 lea     rcx,[_F791F1DE_Lecture01@cpp (07FF79EF63029h)]
00007FF79EF52339 E8 AA F0 FF FF   call   ___CheckForDebuggerJustMyCode (07FF79EF513E8h)
        if (p>1)
00007FF79EF5233E 83 BD E0 00 00 01 cmp     dword ptr [p],1
00007FF79EF52345 7E 1C       jle     f+43h (07FF79EF52363h)
        return p*f(p-1);
00007FF79EF52347 8B 85 E0 00 00 00 mov     eax,dword ptr [p]
00007FF79EF5234D FF C8       dec     eax
00007FF79EF5234F 8B C8       mov     ecx,eax
00007FF79EF52351 E8 75 EF FF FF   call   f (07FF79EF512C8h)
00007FF79EF52356 8B 8D E0 00 00 00 mov     ecx,dword ptr [p]
00007FF79EF5235C 0F AF C8     imul    ecx,eax
00007FF79EF5235F 8B C1       mov     eax,ecx
00007FF79EF52361 EB 05       jmp     f+48h (07FF79EF52368h)

    return 1;
00007FF79EF52363 B8 01 00 00 00   mov     eax,1
}
00007FF79EF52368 48 8D A5 C8 00 00 lea     rsp,[rbp+0C8h]
00007FF79EF5236F 5F           pop     rdi
00007FF79EF52370 5D           pop     rbp
00007FF79EF52371 C3           ret
```

x64 - CISC

```
0x10000309c <+0>: sub     sp, sp, #0x20
0x1000030a0 <+4>: stp     x29, x30, [sp, #0x10]
0x1000030a4 <+8>: add     x29, sp, #0x10
0x1000030a8 <+12>: str     w0, [sp, #0x8]
0x1000030ac <+16>: ldr     w8, [sp, #0x8]
0x1000030b0 <+20>: subs    w8, w8, #0x1
0x1000030b4 <+24>: cset    w8, le
0x1000030b8 <+28>: tbznz   w8, #0x0, 0x1000030e4 ; <+72> at main.cpp
0x1000030bc <+32>: b       0x1000030c0 ; <+36> at main.cpp:13:14
0x1000030c0 <+36>: ldr     w8, [sp, #0x8]
0x1000030c4 <+40>: str     w8, [sp, #0x4]
0x1000030c8 <+44>: ldr     w8, [sp, #0x8]
0x1000030cc <+48>: subs    w0, w8, #0x1
0x1000030d0 <+52>: bl      0x10000309c ; <+0> at main.cpp:11
0x1000030d4 <+56>: ldr     w8, [sp, #0x4]
0x1000030d8 <+60>: mul     w8, w8, w0
0x1000030dc <+64>: stur    w8, [x29, #-0x4]
0x1000030e0 <+68>: b       0x1000030f0 ; <+84> at main.cpp:16:1
0x1000030e4 <+72>: mov     w8, #0x1
0x1000030e8 <+76>: stur    w8, [x29, #-0x4]
0x1000030ec <+80>: b       0x1000030f0 ; <+84> at main.cpp:16:1
0x1000030f0 <+84>: ldur    w0, [x29, #-0x4]
0x1000030f4 <+88>: ldp     x29, x30, [sp, #0x10]
0x1000030f8 <+92>: add     sp, sp, #0x20
0x1000030fc <+96>: ret
```

ArmV8 - RISC



Architecture

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

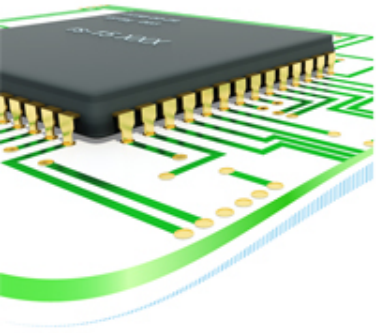
    return 1;
}
```

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 S0n(i0ni0n)
16 ssr 29[90 0x0000005a]              ; Debug expression prologue p>1
17 pmw base pointer offset -3
18 psh int8 1 0x01
19 cvt_i_t -1 regs
20 gre_i
21 czf
22 ssr 3[1 0x00000001]
23 jz 37
24 ssr 29[109 0x0000006d]              ; 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_i
32 clf 1
33 ssr 6
34 mul_i
35 ssr 5[1 0x00000001]
36 jmp 41
37 ssr 29[138 0x0000008a]              ; 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 S0n(i0ni0n)
42 rtf 1
```

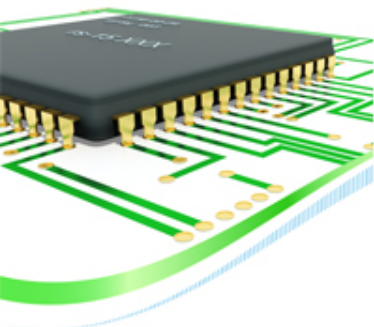
A sample stack machine



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



```

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

    return 1;
}

```

```

00007FF79EF52320 89 4C 24 08      mov     dword ptr [rsp+8],ecx
00007FF79EF52324 55          push    rbp
00007FF79EF52325 57          push    rdi
00007FF79EF52326 48 81 EC E8 00 00 sub     rsp,0E8h
00007FF79EF5232D 48 8D 6C 24 20 lea     rbp,[rsp+20h]
00007FF79EF52332 48 8D 0D F0 0C 01 lea     rcx,[_F791F1DE_Lecture01@cpp (07FF79EF63029h)]
00007FF79EF52339 E8 AA F0 FF FF call    __CheckForDebuggerJustMyCode (07FF79EF513E8h)
        if (p>1)
00007FF79EF5233E 83 BD E0 00 00 01 cmp     dword ptr [p],1
00007FF79EF52345 7E 1C      jle     f+43h (07FF79EF52363h)
        return p*f(p-1);
00007FF79EF52347 8B 85 E0 00 00 00 mov     eax,dword ptr [p]
00007FF79EF5234D FF C8      dec     eax
00007FF79EF5234F 8B C8      mov     ecx,eax
00007FF79EF52351 E8 75 EF FF FF call    f (07FF79EF512CBh)
00007FF79EF52356 8B 8D E0 00 00 00 mov     ecx,dword ptr [p]
00007FF79EF5235C 0F AF C8 imul    ecx,eax
00007FF79EF5235F 8B C1      mov     eax,ecx
00007FF79EF52361 EB 05      jmp     f+48h (07FF79EF52368h)

        return 1;
00007FF79EF52363 B8 01 00 00 00 mov     eax,1
}
00007FF79EF52368 48 8D A5 C8 00 00 lea     rsp,[rbp+0C8h]
00007FF79EF5236F 5F          pop     rdi
00007FF79EF52370 5D          pop     rbp
00007FF79EF52371 C3          ret

```

x64 – Windows ABI

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

Architecture

Identical semantics mapped different
Instruction Set Architectures (ISA)

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

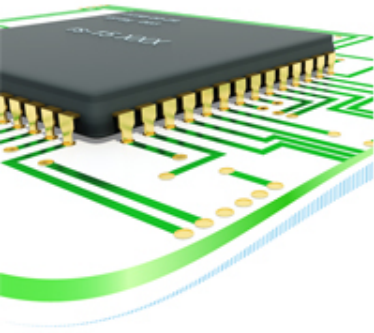
```

00005555555551cd: push     %rbp
00005555555551ce: mov      %rsp,%rbp
00005555555551d1: sub      $0x10,%rsp
00005555555551d5: mov      %edi,-0x4(%rbp)
14          if (p>1)
00005555555551d8: cmpl     $0x1,-0x4(%rbp)
00005555555551dc: jle      0x5555555551f1 <_Z1fi+40>
15          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>
17          return 1;
00005555555551f1: mov      $0x1,%eax
00005555555551f6: leave
00005555555551f7: ret

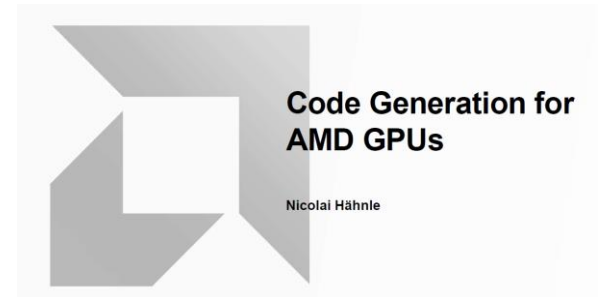
```

x64 – Linux ABI

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



Architecture



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, 0, v12
v_cndmask_b32    v19, -v20, v20, s[0:1]
v_add_f32        v20, |v16|, |v15|
v_cmp_le_f32     s[0:1], 0, v15
v_cndmask_b32    v13, v13, v18, vcc
v_mul_f32        v24, v10, v10
v_cndmask_b32    v14, v14, v19, vcc
v_sub_f32        v17, 1.0, v20
v_cndmask_b32    v10, -v21, v21, s[0:1]
```

```
s_add_i32        s11, s11, 1
...
s_cmpk_lg_i32    s28, 0x100
s_branch_scc0    .LBB0_8
BB0_4:
...
s_cmp_gt_u32     s11, 11
...
s_branch_scc0    .LBB0_6
```

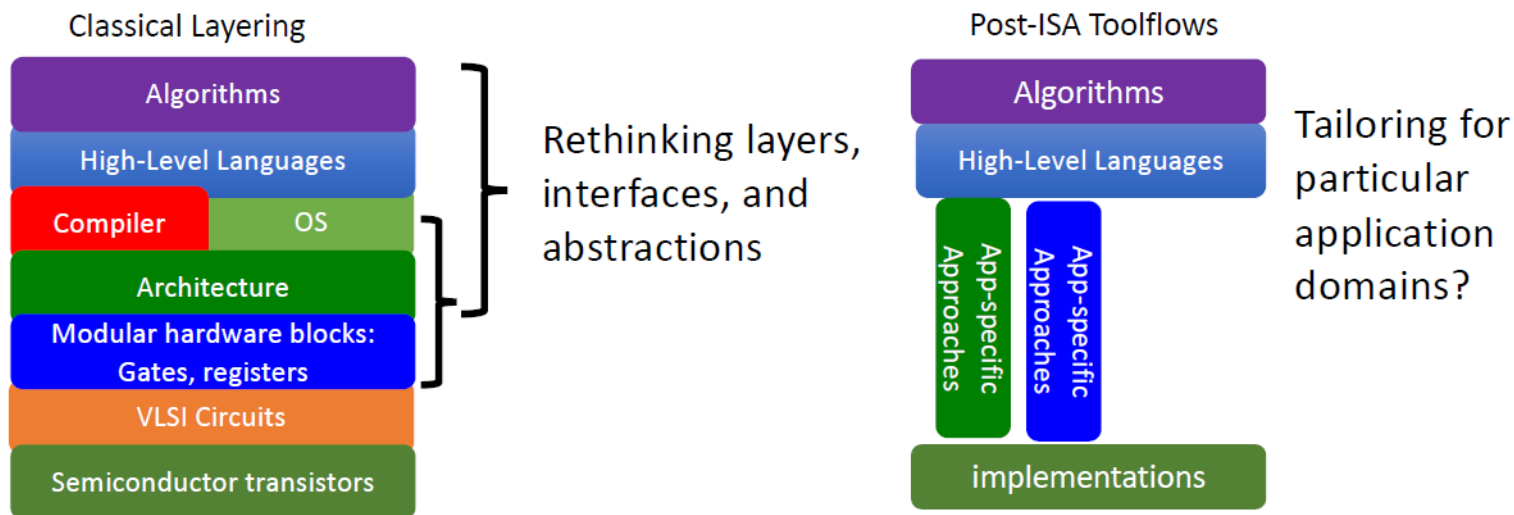
```
s_load_b128      s[20:23], s[24:25], 0x30
v_mul_f32        v7, s0, v1
v_mul_f32        v8, s1, v8
...
s_waitcnt        lgknmt(0)
image_sample_lz   v[9:10], v[7:8], s[4:11], s[20:23]
dmask:0x3 dlm:SQ_RSRC_IMG_2D
```

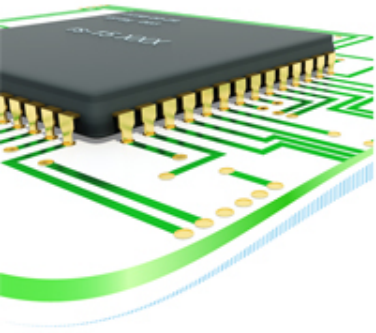



Architecture

Application specific architectures have deep impact

End-of-Moore Systems: Rethinking Full-Stack Approaches



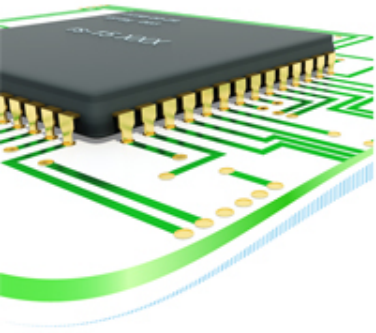


Code Generation

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



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`

...

$4*i + 5$

...

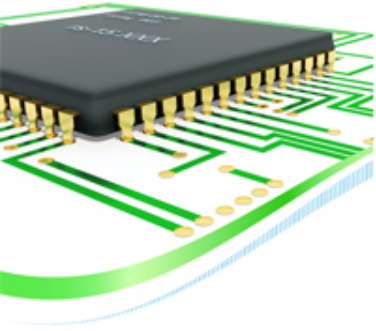
`mov rcx, [rbp + 16]`

`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”



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

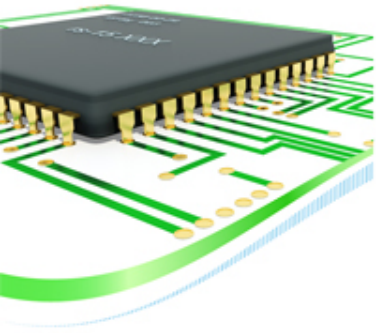
Instruction level parallelism

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”



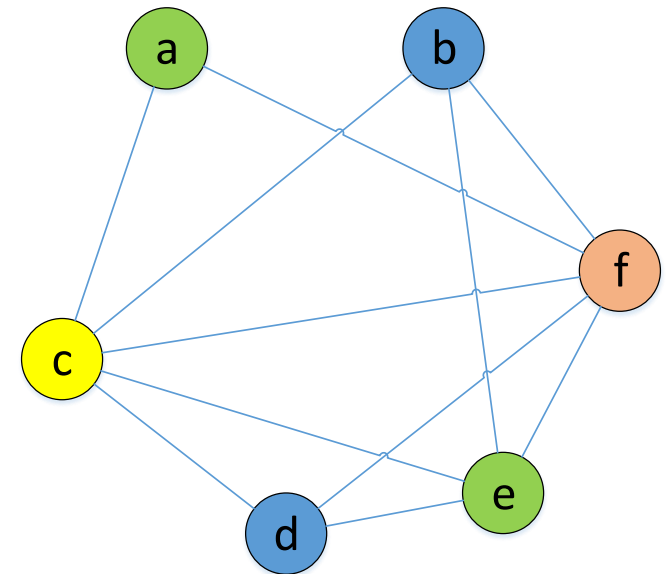
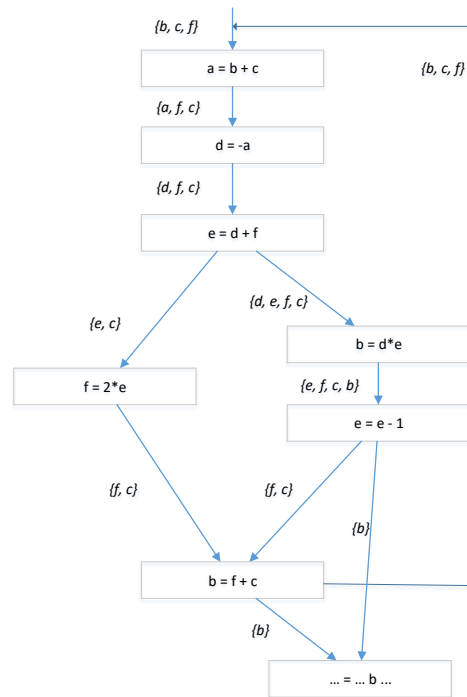
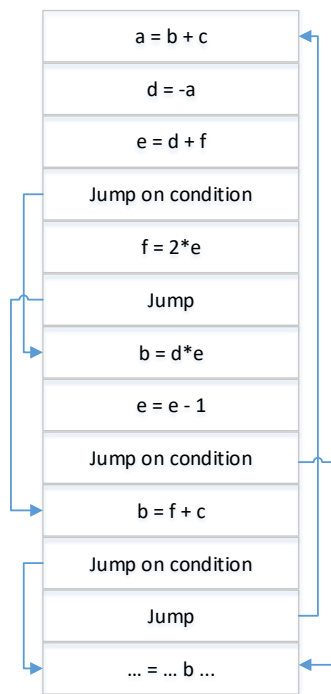
Code Generation

Register Allocation

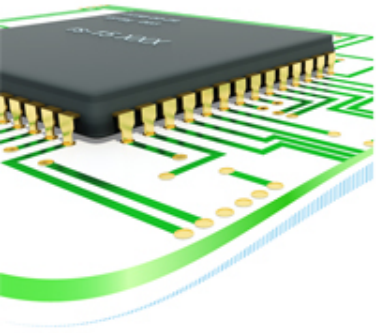
Local / Global Register Allocation

Graph Coloring Problem

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



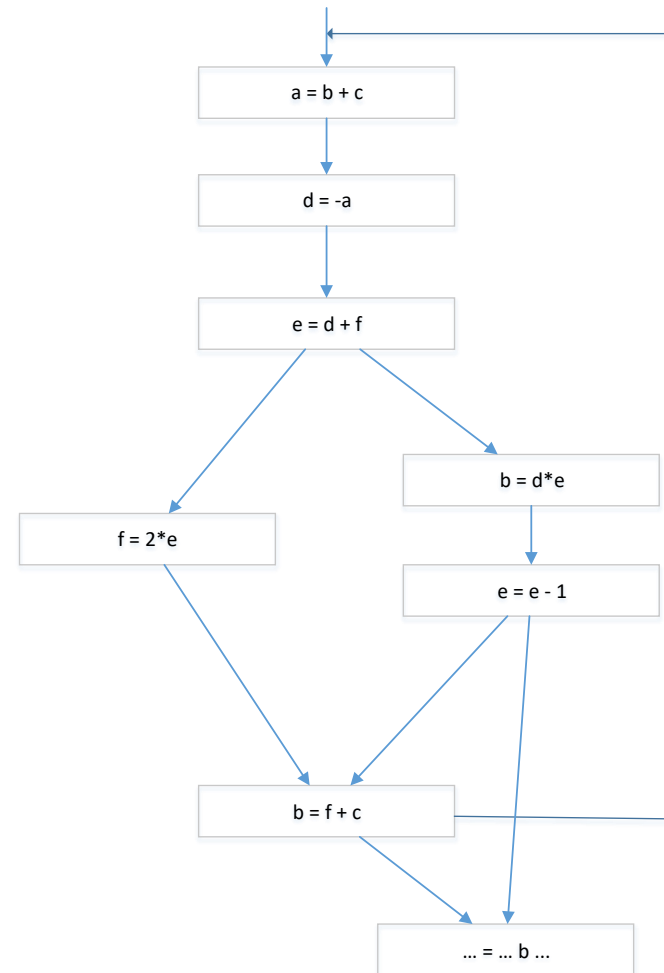
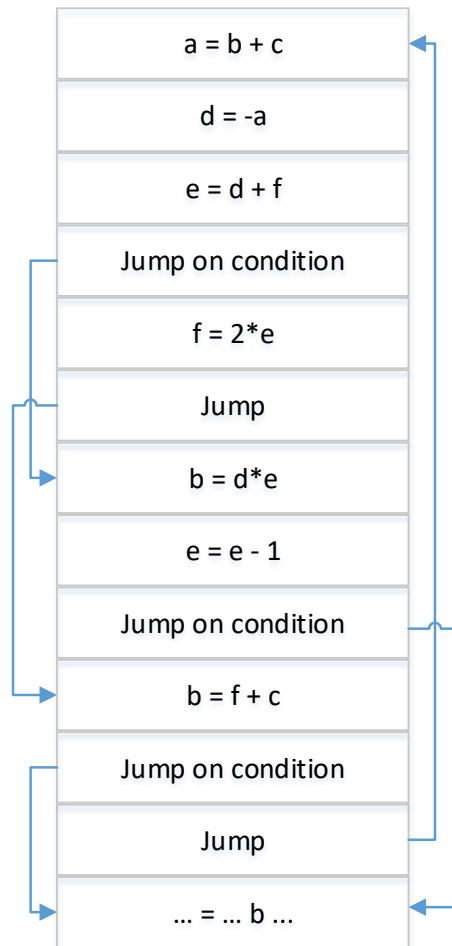
(*) CFG example from Portland State University CS536 Spring 2001 notes,
“<http://web.cecs.pdx.edu/~mperkows/temp/register-allocation.pdf>”

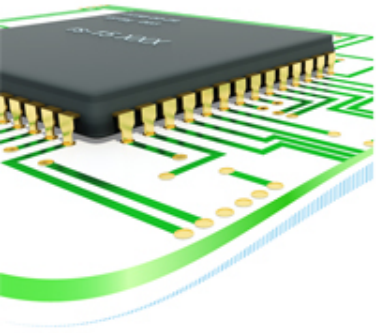


Code Generation

Control Flow Graph

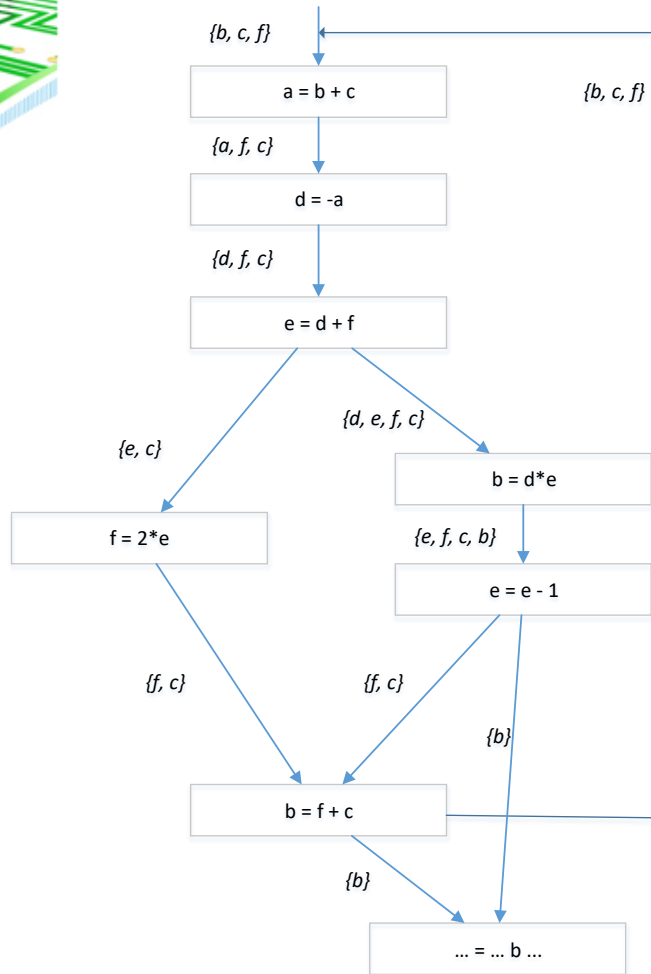
Linear IR



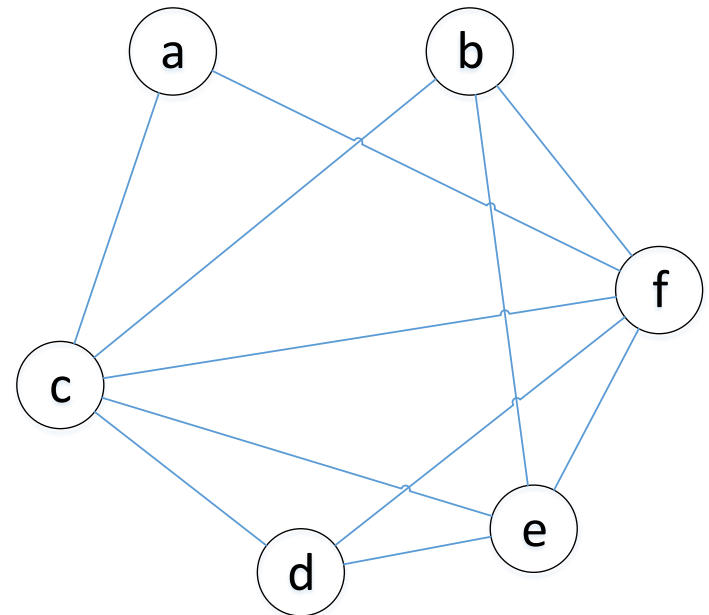


Code Generation

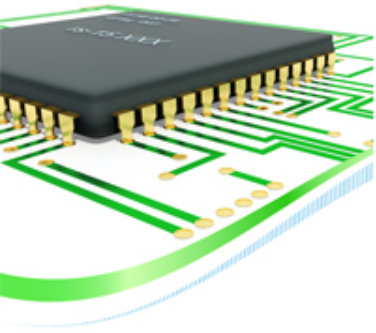
Register Interference Graph (RIG)



Edge Labeling



Graph Construction

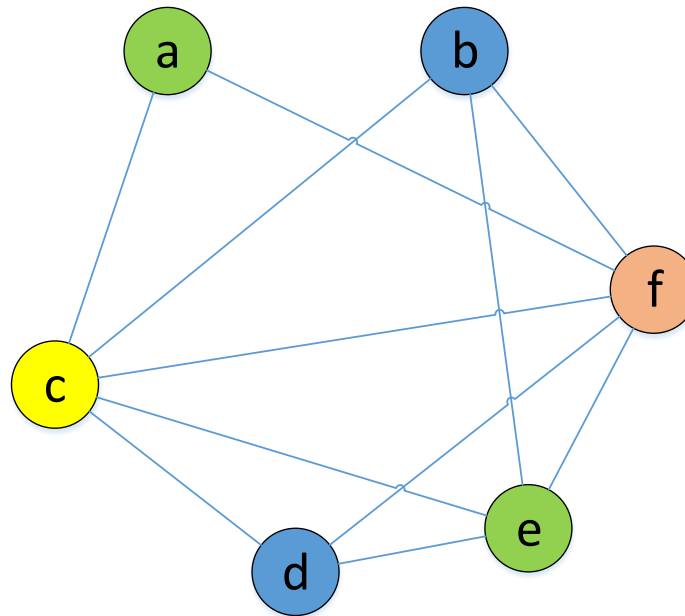


Code Generation

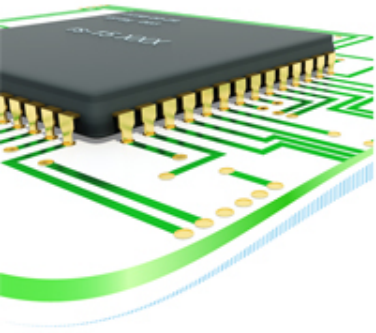
Register Coloring

Register Pressure

“Do we have four registers?”



Graph Coloring ($k=4$)

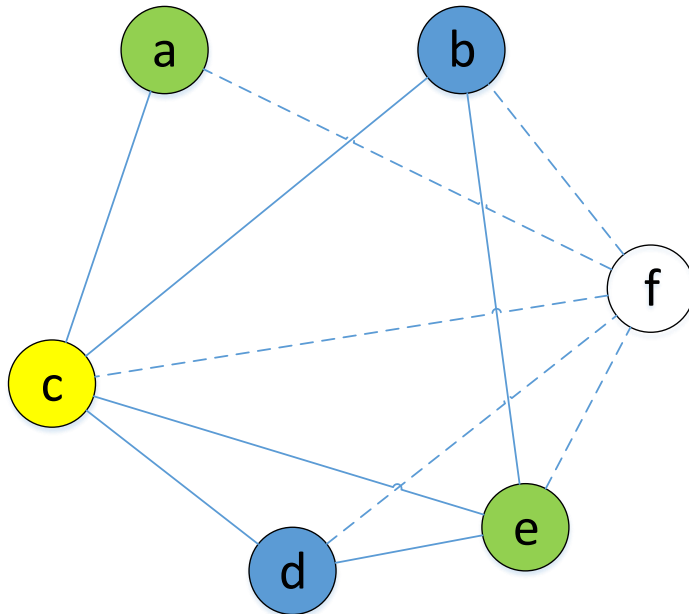


Code Generation

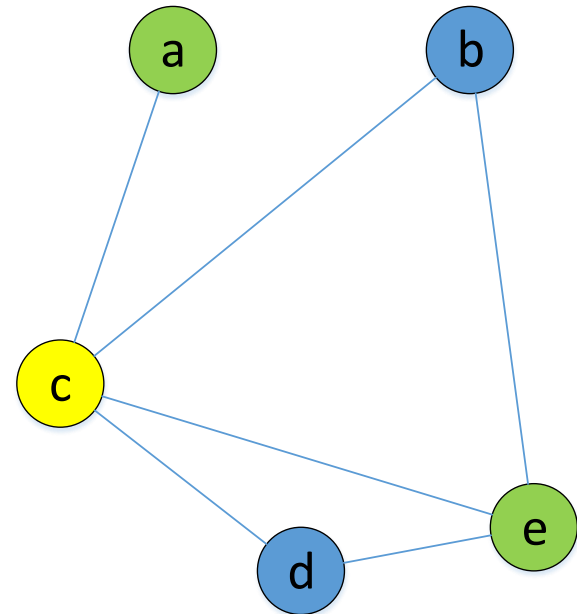
Register Coloring

Register Pressure

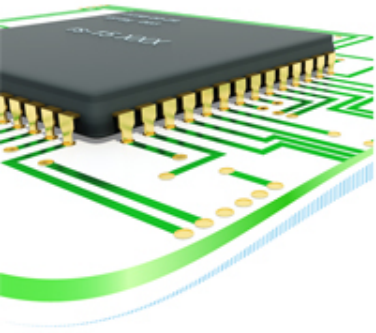
“What if we have three registers only?”



Spill f



Graph Coloring ($k=3$)

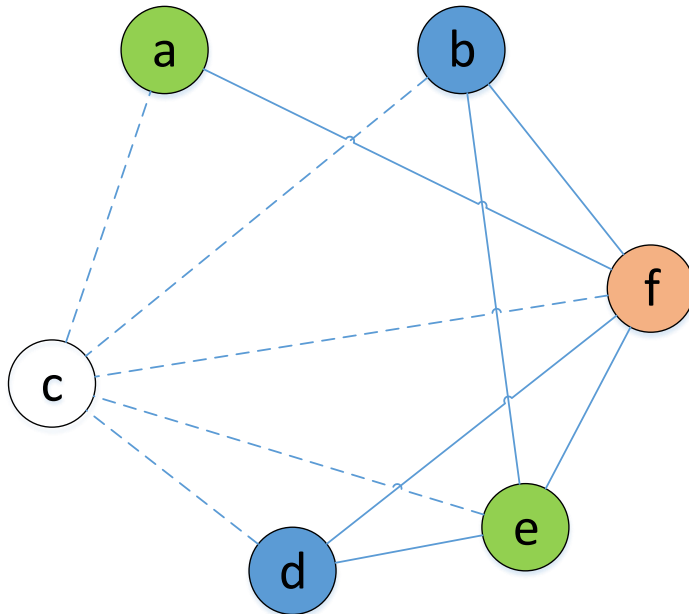


Code Generation

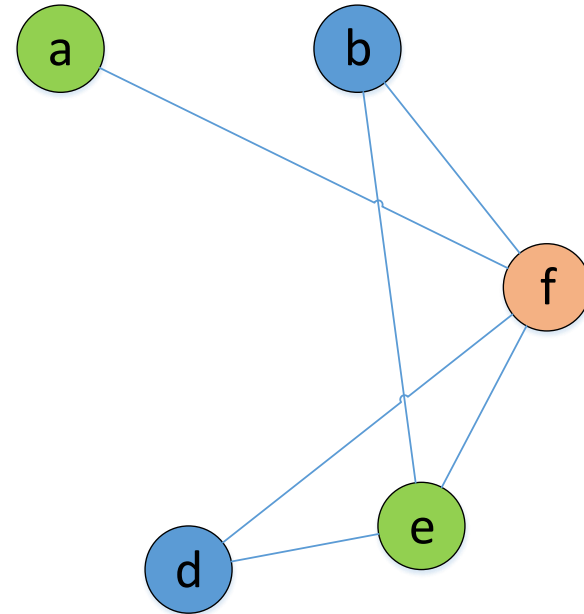
Register Coloring

Register Pressure

“What if we have three registers only?”



Spill c



Graph Coloring ($k=3$)