The Flow Component

Karl Fant
April 2015

The flow component is the unit of composition of a flow network.

It consists of:

a boundary of input and output token names each representing a half oscillation with an associated closure path.

a flow relation that specifies the structure of token flow fom input to output.

a body that is a completeness interaction behavior mapping the input tokens to the output tokens.

a link generating completeness closure for each input token and accepting closure from each output token linking the input oscillations to the output oscillations.

The Parts of a Flow Component

function table

	X/0	X/1	X/2
Y/0	Z/0	Z/1	Z/2
Y/1	Z/1	Z/2	Z/3

The binary-trinary-quaternary adder

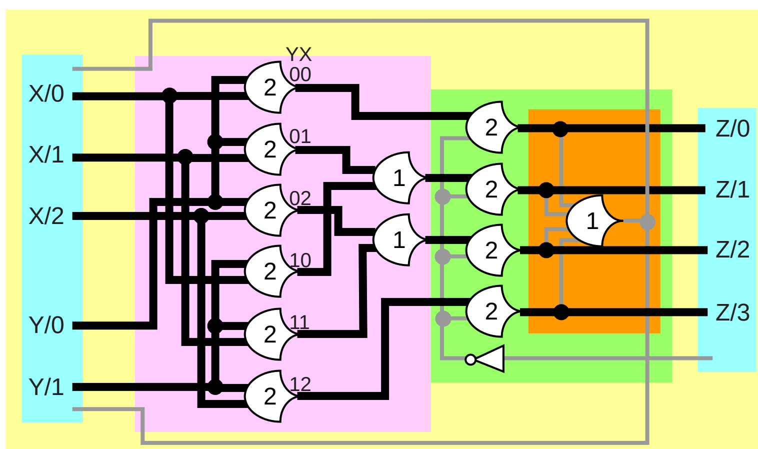
```
token Y{1,0}, X{2, 1, 0}, Z{3, 2, 1, 0};

(X, Y -> Z){
flow [X, Y] -> Z;

Z/0 = X/0 & Y/0;
Z/1 = X/1 & Y/0 | X/0 & Y/1;
Z/2 = X/2 & Y/0 | X/1 & Y/1;
Z/3 = X/2 & Y/1;
}
close X = #Z;
close Y = #Z;
close Z = #?;
```

The **boundary specification** specifies the input tokens and the output tokens. Each token represents a half oscillation and assumes a closure rail.

The **flow relation** specifies the AND-OR relations among boundary tokens imposed by the component. While the declarations specify the AND-OR relation structure for individual tokens, the flow relation specifies the AND-OR relations among flowing tokens. The relations imposed by the component project beyond the component and is a key factor in analyzing the integrity of composition.

If a flow relation is not present an AND relationship among the input tokens and among the output tokens is implied.

The **body** is a shared completeness behavior that specifies the transfer function of the flowing wavefronts.



The **closure equations** generate the closure flows for the input tokens. They are generally a direct derivation of the flow relation and the token declarations.

The inversion is included in close name = #?.

The **oscillation link** linking 3 oscillations. There is a link for each output flow. It is implied by the output name list but can be made explicit with the reference Z/closure which assumes the inversion.

Change of Course

We have specified AND-OR relations in
token declarations and in flow relations with [ ] and { }.

```
flow PCcontrol{
        /next[curPC] -> [newPC, nextinst],
        /branch[curPC, cond, PCimm] -> [newPC, nextinst],
        /AUIPC[curPC , PCimm] -> [newPC, nextinst, jumpreturn],
        /JAL[curPC, PCimm] -> [newPC, nextinst, jumpreturn],
        /JALR[curPC, PCimm, PCrs1] -> [newPC, nextinst, jumpreturn]};
```

We notice that the transfer function equations in the body of a component are AND-OR relations and it occurs that representing all AND-OR relations in the language the same way would be coherent, uniform and intuitive.

Infix assignment notation

```
Z/0 = X/0 & Y/0;
Z/1 = X/1 & Y/0 | X/0 & Y/1;
Z/2 = X/2 & Y/0 | X/1 & Y/1;
Z/3 = X/2 & Y/1;
```

Function flow notation

```
[X/0, Y/0] -> Z/0;
{[X/1, Y/0], [X/0, Y/1]} -> Z/1;
{[X/2, Y/0], [X/1, Y/1]} -> Z/2;
[X/2, Y/1] -> Z/3;
```

The expressivity is identical. the mapping is direct. Only the form of the notation changes.

The functional flow notation better represents the flow structure nature of the language whereas the infix assignment notation reflects more a nature of ordered events

A language of computation flow that is constructive in contrast to procedural specifying a a directed network with a protocol of flow.

Steer

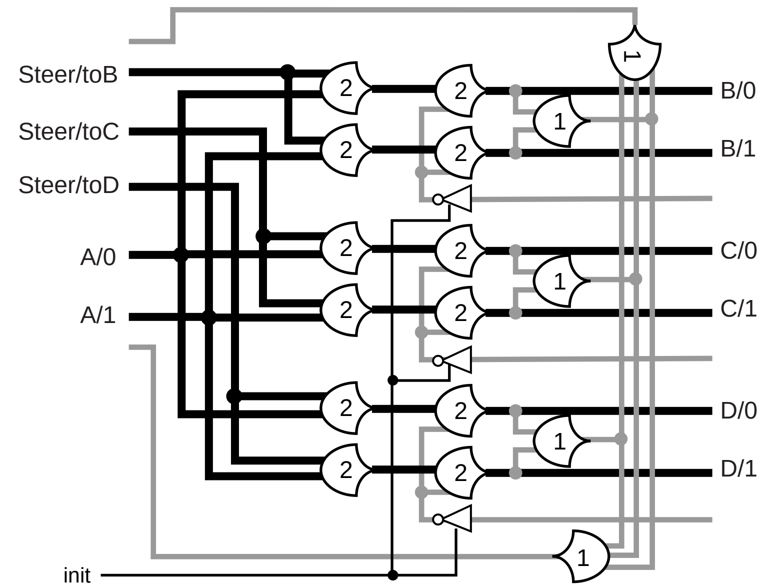
```
token bit{0, 1};
token (A, B, C, D)bit
token Steer{toB, toC, toD};

(A, Steer -> B, C, D){
flow [A, Steer] -> {B, C, D};

[steer/toB & A/0] -> B/0;
[steer/toB & A/1] -> B/1;
[steer/toC & A/0] -> C/0;
[steer/toC & A/1] -> C/1;
[steer/toD & A/0] -> D/0;
[steer/toD & A/1] -> D/1;
}
close A <- {#B, #C, #D};
close Steer <- {#B, #C, #D};
close B <- #?;
close C <- #?;
close D <- #?;
```

Steer

	/toB	/toC	/toD
A/0	B/0	C/0	D/0
A/1	B/1	C/1	D/1

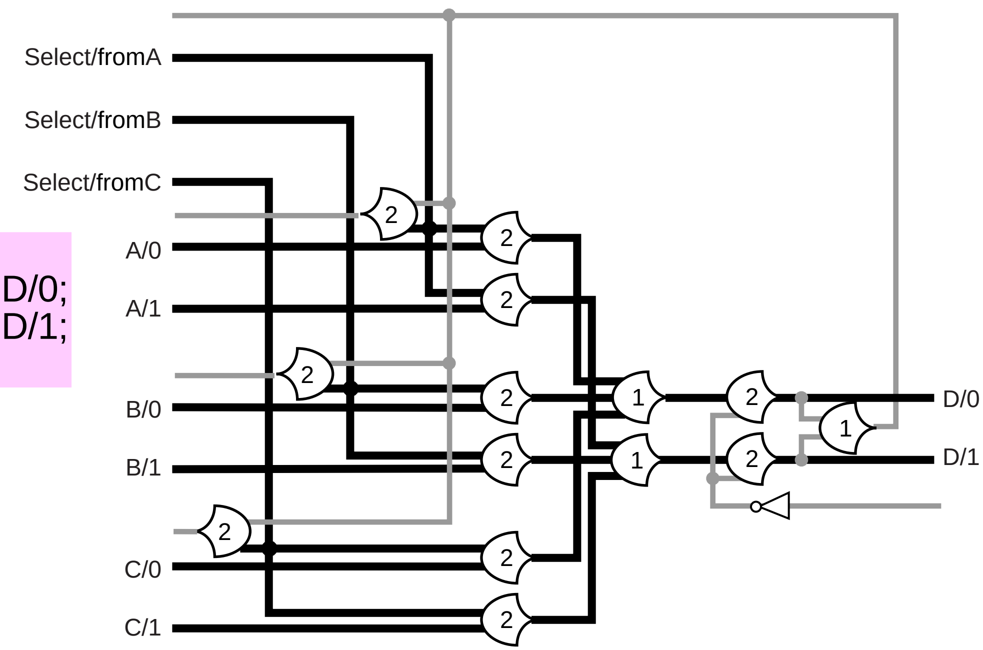
Select

	/fromA	/fromB	/fromC
A/0	D/0	–	–
A/1	D/1	–	–
B/0	–	D/0	–
B/1	–	D/1	–
C/0	–	–	D/0
C/1	–	–	D/1



Select

```
token bit{0, 1};
token (A, B, C, D)bit
token Select{fromA, fromB, fromC};

(A, B, C, Select -> D){
flow [{A, B, C}, Select] -> D;

{[Select/fromA, A/0], [Select/fromB, B/0], [Select/fromC, C/0]} -> D/0;
{[Select/fromA, A/1], [Select/fromB, B/1], [Select/fromC, C/1]} -> D/1;
}
close A <- [#D, Select/fromA];
close B <- [#D, Select/fromB];
close C <- [#D, Select/fromC];
close Select <- #D;
close D <- #?;
```

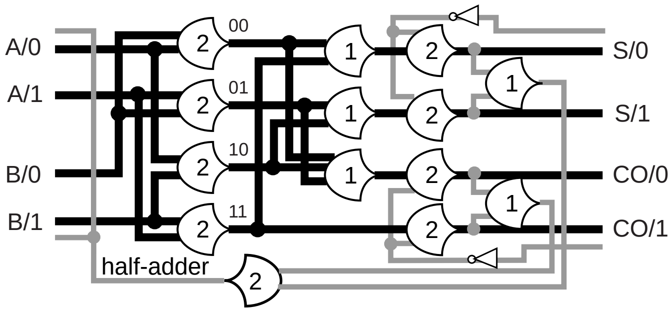


A component can be named and referenced.

The boundary becomes formal boundary names with internal token declarations which can be matched with actual boundary names with external token declarations.

halfadder
A component with two outputs

```
token (A, B, S, CO){0, 1}

(A, B -> S, CO){
flow [A, B] -> [S, CO];

{[A/0, B/0], [A/1, B/1]} -> S/0;
{[A/0, B/1], [A/1, B/0]} -> S/1;
{[A/0, B/0], [A/0, B/1], [A/1, B/0]} -> CO/0;
[A/1,B/1] -> CO/1;
}
close A <- [#S, #CO];
close B <- [#S, #CO];
close S <- #?;
close CO <- #?;
```




```
halfadd(A, B -> S, CO){
token (A, B, S, CO){0, 1}
`
flow [A, B] -> [S, CO];

{[A/0, B/0], [A/1, B/1]} -> S/0;
{[A/0, B/1], [A/1, B/0]} -> S/1;
{[A/0, B/0], [A/0, B/1], [A/1, B/0]} -> CO/0;
[A/1,B/1] -> CO/1;
}
close A <- [#S, #CO];
close B <- [#S, #CO];
close S <- #?;
close CO <- #?;
```

An 8 bit incrementer

```
token(M, N, O)[0,7]{0,1}
tokenP[1,8]{0,1}
halfadd(M/0, 1 -> O/0, P/1)
halfadd(M/1, P/1 -> O/1, P/2)
halfadd(M/2, P/2 -> O/2, P/3)
halfadd(M/3, P/3 -> O/3, P/4)
halfadd(M/4, P/4 -> O/4, P/5)
halfadd(M/5, P/5 -> O/5, P/6)
halfadd(M/6, P/6 -> O/6, P/7)
halfadd(M/7, P/7 -> O/7, P/8)
```

Program Counter 1

```
token (cond, dual){1:0};
token (curPC, PCrs1, PCimm, nextinst, newPC, jumpreturn)[31:0][dual];
token PCcontrol{next, branch, JAL, JALR, AUIPC};

(PCcontrol, curPC, cond, PCrs1, PCimm -> nextinst, newPC, jumpreturn){
flow PCcontrol{
  [/next curPC] -> [newPC, nextinst],
  [/branch, curPC, cond, PCimm] -> [newPC, nextinst],
  [/AUIPC, curPC , PCimm] -> [newPC, nextinst, jumpreturn],
  [/JAL, curPC, PCimm] -> [newPC, nextinst, jumpreturn],
  [/JALR, curPC, PCimm, PCrs1] -> [newPC, nextinst, jumpreturn]};

PCcontrol{[[{/next, /AUIPC, [/branch, cond/F]}, next], [{[/branch, cond/T], /JAL, /JALR}, branch]} -> newPC;
NewPC -> nextinst;
PCcontrol{[[{/JAL, /JALR}, next], [/AUIPC, branch]} -> jumpreturn;

//  What if the PC overflows???????
//////////////////////////////////////////////////////////
////////// increment by 4 //////////////////////////
/////////////////////////////////////////////////////////
token dual{0,1};
token next[31:0][dual];
token (oldPC, C)[31:0][dual];

   [curPC, PCcontrol{/next, /AUIPC, [/branch, cond/F]}] -> oldPC;
   oldPC/0 -> next/0;
   oldPC/1 -> next/1;
   halfaddone(oldPC/2 -> next/2, C/3);
   for i=3:30(
     zeroadd(oldPC/i, C/i -> next/i, C/i+1)
     )
   zerosum(oldPC/31, C/31 -> next/31);
```

Program Counter 2

```
////////////////////////////////////////////////////////////
////////// 32 bit adder ///////////////////////////////
////////////////////////////////////////////////////////////
token branch[31:0][dual];
token (alpha], beta, c)[31:0][dual];

// steer inputs to adder
   PCcontrol{[/JALR, PCrs1/1:31], [{[/branch, cond/T], /JAL, /AUIPC}, curPC/1:31]} -> alpha/1:31;
   PCcontrol{[/branch, cond/T], /JALR, /AUIPC, /JAL}, imm/1:31] -> beta/1:31;
   PCcontrol[{[/JALR, 0], [/branch, cond/T], /JAL, /AUIPC}, curPC/0] -> alpha/0;
   PCcontrol[{[/JALR, 0], [/branch, cond/T], /JAL, /AUIPC}, imm/0] -> beta/0;
  halfadd( alpha/0, beta/0 -> branch/0, c/1);
  for i=1:30(
       fulladd( c/i, alpha/i, beta/i -> branch/i, c/i+1 );
     )
   sum( alpha/31, beta/31 -> branch/31);
}
////////////////////////////////////////////////////////////
// closure equations
////////////////////////////////////////////////////////////
close curPC <- {[#newPC, #nextPC, {PCcontrol/next, PC control/branch}], [#newPC, #nextPC,
#jumpreturn, {PCcontrol/JAL, PCcontrol/JALR, PCcontrol/AUIPC}]};

close opcode/PCcontrol <- {[#newPC, #nextPC, {PCcontrol/next, PC control/branch}], [#newPC,
#nextPC, #jumpreturn, {PCcontrol/JAL, PCcontrol/JALR, PCcontrol/AUIPC}]};

close cond <- [#newPC, #nextPC, PCcontrol/branch];
close PCrs1 <- [#newPC, #nextPC, #jumpreturn, PCcontrol/JALR];
close PCimm <- {[#newPC, #nextPC, {PCcontrol/branch, #newPC}], [#nextPC, #jumpreturn,
{PCcontrol/JAL, Ccontrol/JALR, PCcontrol/AUIPC}]};
```

```
close newPC <- #?;
close nextinst <- #?;
close jumpreturn <- #?;
```