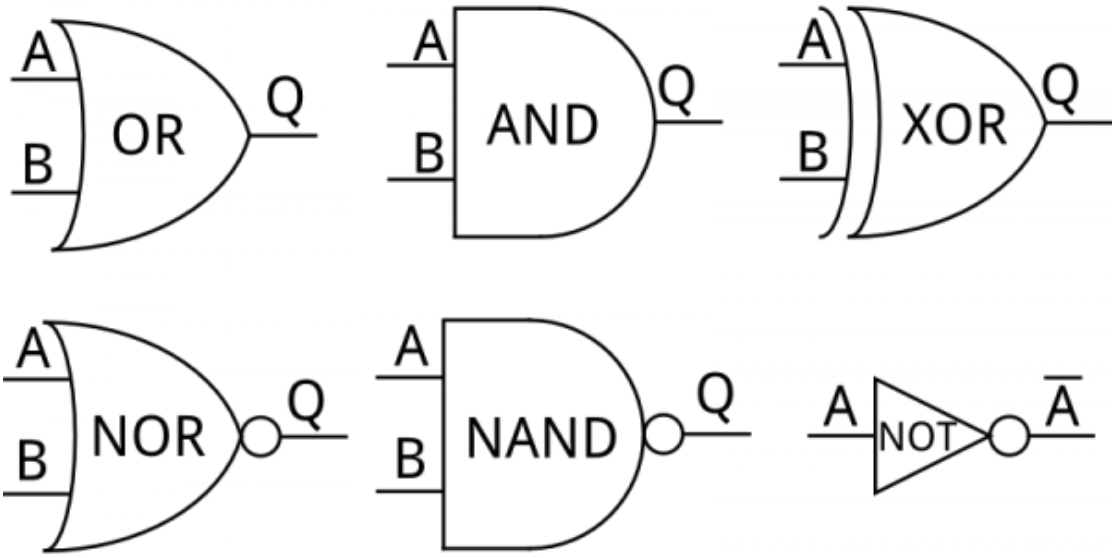
NarvAC

Autor: Carlos Manuel Rodríguez Martínez
Email: fis.carlosmanuel@gmail.com

El objetivo de este proyecto es crear una máquina capaz de realizar cómputo a partir de compuertas lógicas.



Auxiliar

- ### Ejecución

  Definiciones necesarias para imitar el comportamiento de ciertos circuitos. MemoryEvaluate funciona creando un símbolo especial por cada tag, permitiendo así evaluar la función con el argumento de la evaluación anterior. Esto realiza una anidación de la función definida en expr por cada evaluación sucesiva.

In[1]:=
```
HalfSplit[l_] := Block[{len},
   len = Length[l]/2;
   {Take[l, len], Drop[l, len]}
  ];

SetAttributes[MemoryEvaluate, HoldAll];
MemoryEvaluate[expr_, tag_] := Block[{},
   If[Head[SpecialSymbol[tag]] == SpecialSymbol,
    SpecialSymbol[tag] = 0;
   ];
   SpecialSymbol[tag] = ReleaseHold[expr[SpecialSymbol[tag]]];
   Return[SpecialSymbol[tag]];
  ];

Unprotect[BitAnd];
BitAnd[1, x_] := x;
BitAnd[x_, 1] := x;
Protect[BitAnd];

Unprotect[BitNot];
BitNot[0] := 1;
BitNot[1] := 0;
Protect[BitAnd];
```

Ejemplo

Ejemplo de uso de MemoryEvaluate

In[•]:=
```
MemoryEvaluate[f, "Example"]
```
Out[•]=
```
f[0]
```

In[•]:=
```
MemoryEvaluate[f, "Example"]
```
Out[•]=
```
f[f[0]]
```

In[•]:=
```
MemoryEvaluate[f, "Example"]
```
Out[•]=
```
f[f[f[0]]]
```

Visualización

Visualización de resultados de circuitos.

In[12]:=
```
VisualizeAdder[module_, inputSize_, flagName_] :=
  Block[{logicTable, labels, gridList},
   logicTable =  Map[Join[#, Apply[module, #]] &, Tuples[{0, 1}, inputSize]];
   labels =
    Join[Take[Map[ToUpperCase, Alphabet[]], inputSize], {flagName, "Result"}];
   gridList = Join[{labels}, logicTable];
   Grid[gridList, Frame → All, Background → {None, {LightGreen}}]
  ];

VisualizeMultiplexer[mux_, inputSize_] := Block[{logicTable, labels, gridList},
   logicTable =  Map[
     Join[#, {mux[#, Take[Alphabet[], 2^inputSize]]}] &, Tuples[{0, 1}, inputSize]];
   labels = Join[Take[Map[ToUpperCase, Alphabet[]], inputSize], {"Selection"}];
   gridList = Join[{labels}, logicTable];
   Grid[gridList, Frame → All, Background → {None, {LightGreen}}]
  ];
```

Visualización de la estructura de árbol

Evaluar sólo el encabezado

In[14]:=
```
EvalWithValues[expr_, values_] := expr /. values;
EvaluateHead[expr_, values_] :=
  ReplaceRepeated[expr, {f_Symbol[arg__] :→ EvalWithValues[f[arg], values][arg]}];
```

In[16]:=
```
DenestEvaluating[expr_, values_] :=
  ReplaceAll[
   Map[
    If[Head[#] === List,
      DenestEvaluating[#, values],
      EvaluateHead[#, values]
     ] &,
    Level[expr, 1]
   ],
   values
  ];
```

Ejemplos (solo relevantes para la visualización)

In[•]:=
```
expr = ALUAdder[{a1, a2}, {b1, b2}]
```

Out[•]=
```
{BitOr[BitAnd[b1, BitXor[a1, BitAnd[a2, b2]]], BitAnd[a1, a2, b2]],
 {BitXor[a1, b1, BitAnd[a2, b2]], BitXor[a2, b2]}}
```

In[•]:=
```
DenestEvaluating[expr, {a1 → 0, a2 → 1, b1 → 0, b2 → 1}]
```

Out[•]=
```
{0[0[0, 1[0, 1[1, 1]]], 0[0, 1, 1]], {1[0, 0, 1[1, 1]], 0[1, 1]}}
```

Graficar expresión

In[17]:=
```
SetOptions[$FrontEndSession,
  AutoStyleOptions → {"HighlightFormattingErrors" → False}];

SelectColor[coord_, edge_] :=
  If[Head[First[edge]] =!= Symbol,
   If[Last[edge] == 1,
    {ColorData["HTML"]["LawnGreen"], Line[coord]},
    {GrayLevel[0.3], Line[coord]}
   ]
   ,
   If[Head[Last[edge]] =!= Symbol,
    If[Last[edge] == 1,
     {ColorData["HTML"]["LawnGreen"], Line[coord]},
     {GrayLevel[0.3], Line[coord]}
    ]
    ,
    {ColorData["HTML"]["SteelBlue"], Line[coord]}
   ]
  ];

PlotExpression[expr_, values_] := Rasterize[
  TreeForm[DenestEvaluating[expr, values],
   VertexRenderingFunction → None,
   EdgeRenderingFunction → (
     ReleaseHold[SelectColor[#, #2]] &),
   AspectRatio -> 1,
   Background → Black
  ]
 ]
```

Forma alternativa, evaluar para cambiar de estilo

```
In[•]:= rules =
         Thread[{List, BitOr, BitAnd, BitXor, BitNot, 0, 1, Association} → RandomColor[8]]

In[•]:= rules = {List → ■, BitOr → ■, BitAnd → ■,
           BitXor → ■, BitNot → ■, 0 → ■, 1 → ■, Association → ■};
        GetColorForPlotExpression[f_, rules_] := ReplaceAll[f, rules];
        PlotExpression[expr_, values_] := Block[{plt1, plt2, symbols, symbolRules},
           plt1 = TreeForm[
             DenestEvaluating[expr, values],
             VertexRenderingFunction → None,
             EdgeRenderingFunction → (
               ReleaseHold[SelectColor[#, #2]] &),
             AspectRatio → 0.4,
             Background → Black,
             ImageSize → 2200
            ];

           symbols = GetSymbols[expr];
           symbolRules = Join[rules, Thread[symbols → White]];
           plt2 = TreeForm[
             expr,
             VertexRenderingFunction →
              ({ReleaseHold[GetColorForPlotExpression[#2, symbolRules]], Disk[#, .05]} &),
             EdgeRenderingFunction → None,
             AspectRatio → 0.4,
             Background → Black,
             ImageSize → 2200
            ];

           ImageAdd[Rasterize[plt1], 0.4 * Rasterize[plt2]]
          ];
```

Graficar sistema (sin necesidad de especificar el valor de las variables)

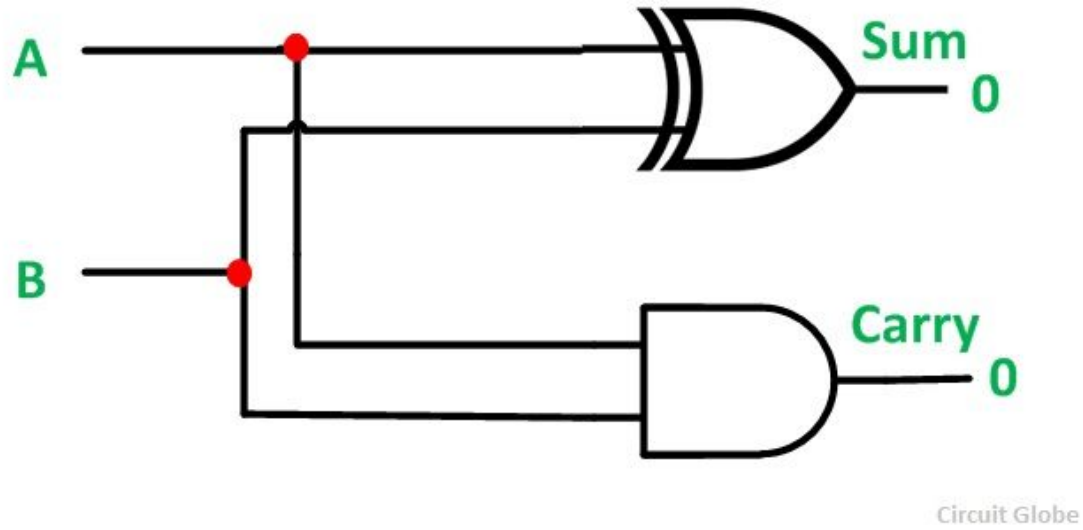
In[20]:=

```
GetSymbols[system_] := Complement[
    DeleteDuplicates[Level[system, {-1}, Heads → True]],
    {List, BitOr, BitAnd, BitXor, BitNot, 0, 1, Association}
  ];

SetOptions[$FrontEndSession, DynamicEvaluationTimeout → 20];
PlotInteractiveSystem[system_, plotVertex_: Automatic] :=
  DynamicModule[{symbols, controllerSymbols, boxes, controls, plot},
   Clear["*Ctrl"];
   symbols = GetSymbols[system];
   controllerSymbols =
    Map[Symbol[StringJoin[SymbolName[#], "Ctrl"]] &, symbols];
   boxes = Map[Checkbox[Dynamic[#], {0, 1}] &, controllerSymbols];

   controls = Panel[Multicolumn[Map[Row[{SymbolName[#[[1]]] <> ": ", #[[2]]}] &,
       Transpose[{symbols, boxes}]], 5], Style["Control", Bold]];

   plot = Dynamic[
     TreeForm[
      DenestEvaluating[system, Thread[symbols → controllerSymbols]],
      EdgeRenderingFunction → (
        ReleaseHold[SelectColor[#, #2]] &),
      VertexRenderingFunction → plotVertex,
      ImageSize → 400,
      AspectRatio -> 1 / 2,
      Background → Black
     ]
    ];

   Column[{controls, Framed[plot]}]
  ];
```

ALU

Half adder

El Half Adder, Full Adder, y Ripple Carry Adder son circuitos para implementar sumas



In[23]:=
```
HalfAdder[a_, b_] := {BitAnd[a, b], BitXor[a, b]};
```

Ejemplo

In[•]:=
```
VisualizeAdder[HalfAdder, 2, "Carry"]
```

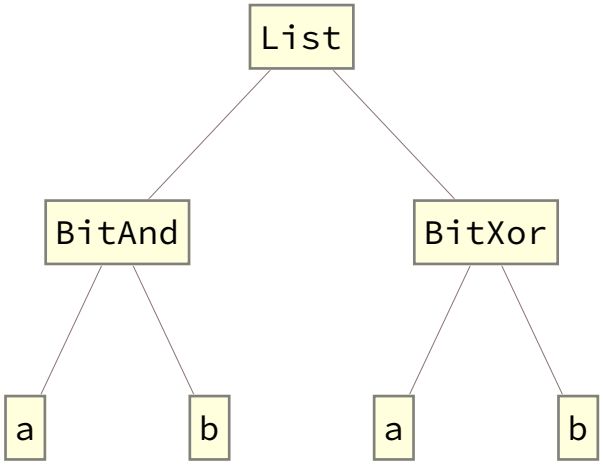
Out[•]=

A	B	Carry	Result
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

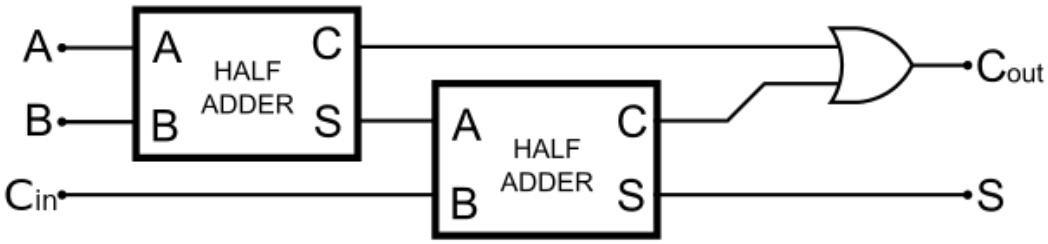
Visualización

In[•]:=
```
TreeForm[HalfAdder[a, b]]
```

Out[•]//TreeForm=



Full adder



In[24]:=
```
FullAdder[a_, b_, c_] := Block[{halfAdd1, halfAdd2},
   halfAdd1 = HalfAdder[a, b];
   halfAdd2 = HalfAdder[Last[halfAdd1], c];
   {BitOr[First[halfAdd1], First[halfAdd2]], Last[halfAdd2]}
  ];
```

Ejemplo

In[•]:=
```
VisualizeAdder[FullAdder, 3, "Carry"]
```

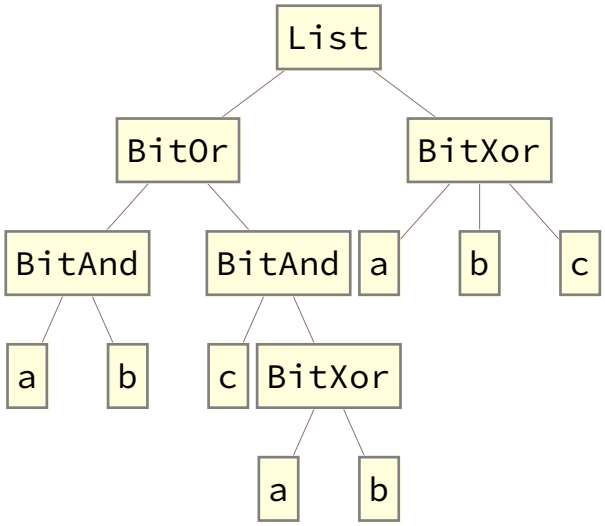
Out[•]=

A	B	C	Carry	Result
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

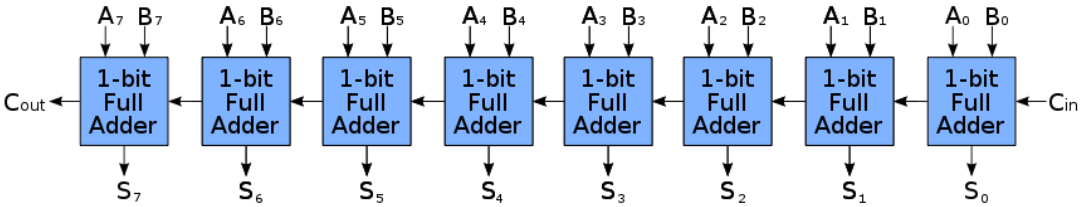
Visualización

In[•]:=
```
TreeForm[FullAdder[a, b, c]]
```

Out[•]//TreeForm=



- Ripple carry adder



In[25]:=
```
ChangeEndianness[l_] := Reverse[l];
ALUAdder[A_, B_, carry_: 0] :=
  Block[{littleEndianA, littleEndianB, pairs, sum, bigEndianSum},
   littleEndianA = ChangeEndianness[A];
   littleEndianB = ChangeEndianness[B];

   (* Se crea conjunto de pares a sumar *)
   pairs = Transpose[{littleEndianA, littleEndianB}];

   (* Se añaden dos pares extra para la última adición *)
   pairs = Join[pairs, {{0, 0}, {0, 0}}];
   sum =
    Drop[FoldPairList[{Last[#1], Apply[FullAdder, Join[{First[#1]}, #2]]} &,
      {carry, 0}, pairs], 1];
   bigEndianSum = ChangeEndianness[sum];
   Return[{First[bigEndianSum], Rest[bigEndianSum]}];
  ];
```

Ejemplo

In[•]:=
```
pairs = Tuples[{Tuples[{0, 1}, 2], Tuples[{0, 1}, 2]}];
result = MapThread[ALUAdder, Transpose[pairs]];
labels = {"A", "B", "Carry", "Result"};
logicTable = MapThread[Join[#1, #2] &, {pairs, result}];
Grid[Join[{labels}, logicTable], Frame -> All, Background → {None, {LightGreen}}]
```

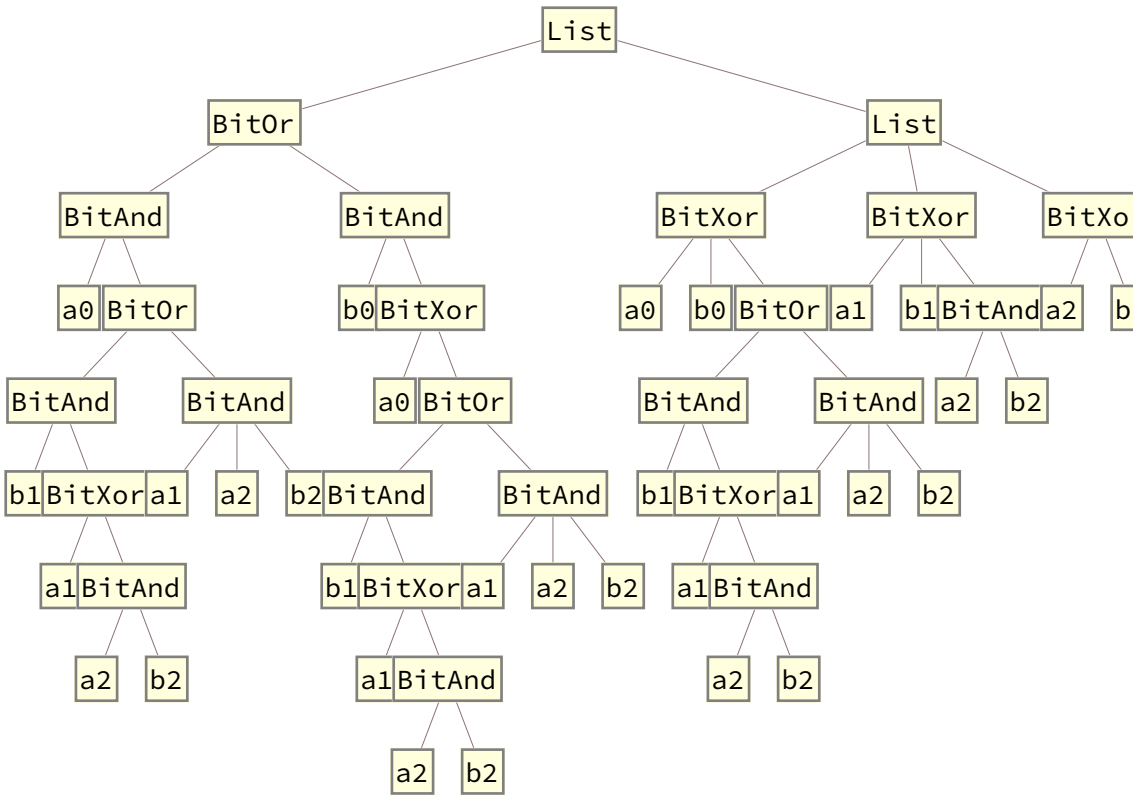
Out[•]=

A	B	Carry	Result
{0, 0}	{0, 0}	0	{0, 0}
{0, 0}	{0, 1}	0	{0, 1}
{0, 0}	{1, 0}	0	{1, 0}
{0, 0}	{1, 1}	0	{1, 1}
{0, 1}	{0, 0}	0	{0, 1}
{0, 1}	{0, 1}	0	{1, 0}
{0, 1}	{1, 0}	0	{1, 1}
{0, 1}	{1, 1}	1	{0, 0}
{1, 0}	{0, 0}	0	{1, 0}
{1, 0}	{0, 1}	0	{1, 1}
{1, 0}	{1, 0}	1	{0, 0}
{1, 0}	{1, 1}	1	{0, 1}
{1, 1}	{0, 0}	0	{1, 1}
{1, 1}	{0, 1}	1	{0, 0}
{1, 1}	{1, 0}	1	{0, 1}
{1, 1}	{1, 1}	1	{1, 0}

Visualización

In[•]:=
```
TreeForm[ALUAdder[{a0, a1, a2}, {b0, b1, b2}]]
```

Out[•]//TreeForm=



En esencia no deja de ser una composición de operaciones binarias

In[•]:=
```
ALUAdder[{a0, a1, a2}, {b0, b1, b2}]
```

Out[•]=
```
{BitOr[BitAnd[a0,
    BitOr[BitAnd[b1, BitXor[a1, BitAnd[a2, b2]]], BitAnd[a1, a2, b2]]], BitAnd[b0,
    BitXor[a0, BitOr[BitAnd[b1, BitXor[a1, BitAnd[a2, b2]]], BitAnd[a1, a2, b2]]]]],
 {BitXor[a0, b0, BitOr[BitAnd[b1, BitXor[a1, BitAnd[a2, b2]]], BitAnd[a1, a2, b2]]],
  BitXor[a1, b1, BitAnd[a2, b2]], BitXor[a2, b2]}}
```

Substractor

In[27]:=
```
ALUSubstractor[A_, B_, borrow_: 1] :=
  Block[{littleEndianA, littleEndianB, pairs, sum, bigEndianSum},
   littleEndianA = ChangeEndianness[A];
   littleEndianB = ChangeEndianness[BitNot[B]];

   (* Se crea conjunto de pares a sumar *)
   pairs = Transpose[{littleEndianA, littleEndianB}];

   (* Se añaden dos pares extra para la última adición *)
   pairs = Join[pairs, {{0, 0}, {0, 0}}];
   sum =
    Drop[FoldPairList[{Last[#1], Apply[FullAdder, Join[{First[#1]}, #2]]} &,
      {borrow, 0}, pairs], 1];
   bigEndianSum = ChangeEndianness[sum];
   Return[{BitNot[First[bigEndianSum]], Rest[bigEndianSum]}];
  ];
```

Ejemplo

In[•]:=
```
pairs = Tuples[{Tuples[{0, 1}, 2], Tuples[{0, 1}, 2]}];
result = MapThread[ALUSubstractor, Transpose[pairs]];
labels = {"A", "B", "Borrow", "Result"};
logicTable = MapThread[Join[#1, #2] &, {pairs, result}];
Grid[Join[{labels}, logicTable], Frame -> All, Background → {None, {LightGreen}}]
```

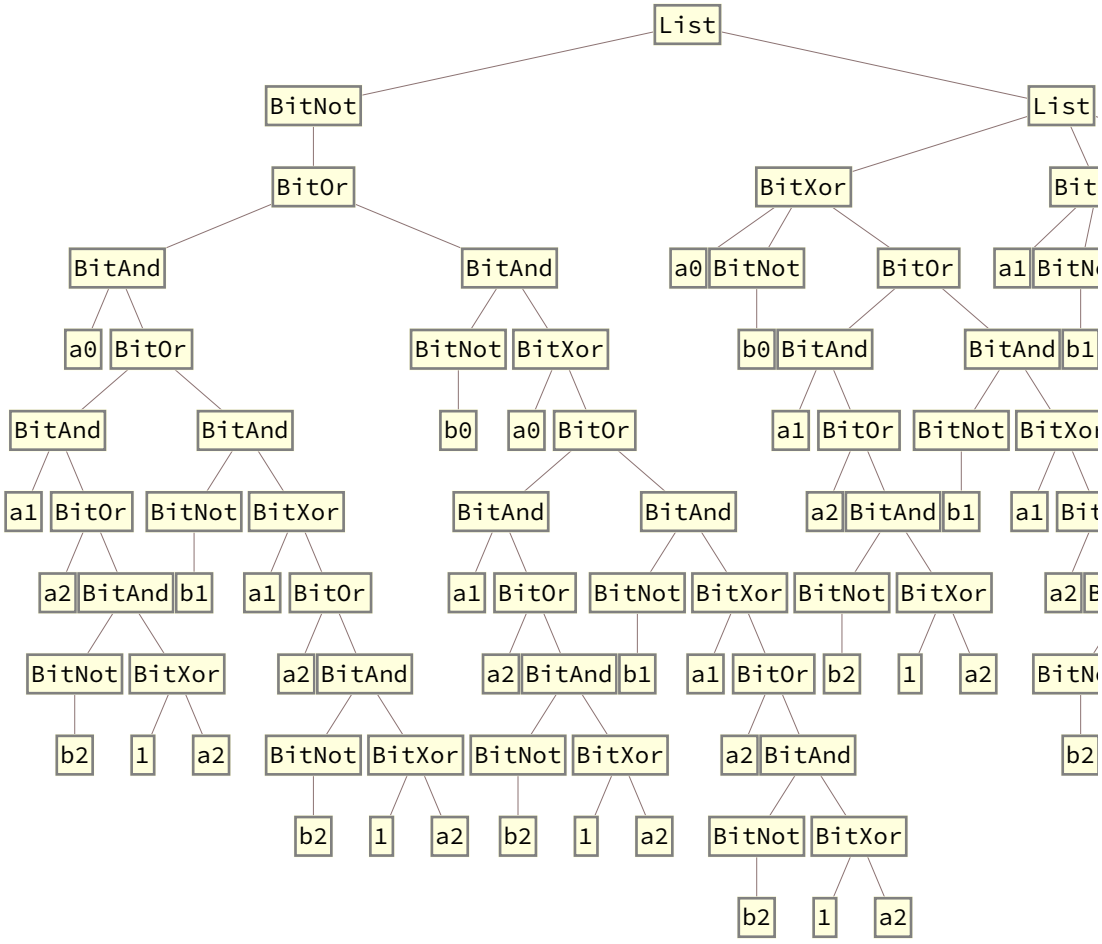
Out[•]=

A	B	Borrow	Result
{0, 0}	{0, 0}	0	{0, 0}
{0, 0}	{0, 1}	1	{1, 1}
{0, 0}	{1, 0}	1	{1, 0}
{0, 0}	{1, 1}	1	{0, 1}
{0, 1}	{0, 0}	0	{0, 1}
{0, 1}	{0, 1}	0	{0, 0}
{0, 1}	{1, 0}	1	{1, 1}
{0, 1}	{1, 1}	1	{1, 0}
{1, 0}	{0, 0}	0	{1, 0}
{1, 0}	{0, 1}	0	{0, 1}
{1, 0}	{1, 0}	0	{0, 0}
{1, 0}	{1, 1}	1	{1, 1}
{1, 1}	{0, 0}	0	{1, 1}
{1, 1}	{0, 1}	0	{1, 0}
{1, 1}	{1, 0}	0	{0, 1}
{1, 1}	{1, 1}	0	{0, 0}

Visualización

In[•]:=
```
TreeForm[ALUSubstractor[{a0, a1, a2}, {b0, b1, b2}]]
```

Out[•]//TreeForm=



Opcodes

In[28]:=
```
ALUOpcodeAdd[] = {0, 0, 0};
ALUOpcodeAddWithCarry[] = {0, 0, 1};
ALUOpcodeSubstract[] = {0, 1, 0};
ALUOpcodeSubstractWithBorrow[] = {0, 1, 1};
ALUOpcodeNegate[] = {1, 0, 0};
ALUOpcodeIncrement[] = {1, 0, 1};
ALUOpcodeDecrement[] = {1, 1, 0};
ALUOpcodePassThrough[] = {1, 1, 1};

aluMnemonics = {
   ALUOpcodeAdd[] → "ALU Add", ALUOpcodeAddWithCarry[] → "ALU Add with carry",
   ALUOpcodeSubstract[] → "ALU Substract",
   ALUOpcodeSubstractWithBorrow[] → "ALU Substract with borrow",
   ALUOpcodeNegate[] → "ALU Negate", ALUOpcodeIncrement[] → "ALU Increment",
   ALUOpcodeDecrement[] → "ALU Decrement", ALUOpcodePassThrough[] → "ALU Pass"
  };
```

Ejecución

Banderas

In[37]:=
```
ALUZero[a_] := BitNot[Apply[BitOr, a]];
ALUParity[a_] := Apply[BitXor, a];
```

Ejecución

In[39]:=
```
ALUExecute[opcode_, a_, b_] := Block[
   {
    add, addWithCarry, substract, substractWithBorrow,
    negate, increment, decrement, pass, output, overflowFlag,
    negativeFlag
   },

   add = ALUAdder[a, b];
   addWithCarry = ALUAdder[a, b, 1];
   substract = ALUSubstractor[a, b];
   substractWithBorrow = ALUSubstractor[a, b, 0];
   negate = ALUSubstractor[ConstantArray[0, Length[a]], a];
   increment = ALUAdder[PadLeft[{1}, Length[a]], a];
   decrement = ALUSubstractor[a, PadLeft[{1}, Length[a]]];
   pass = a;
```

```
output = Mux8to1[opcode,
  GenerateMuxOutputByOpcode[3,
   {
    {ALUOpcodeAdd[], Last[add]},
    {ALUOpcodeAddWithCarry[], Last[addWithCarry]},
    {ALUOpcodeSubstract[], Last[substract]},
    {ALUOpcodeSubstractWithBorrow[], Last[substractWithBorrow]},
    {ALUOpcodeNegate[], Last[negate]},
    {ALUOpcodeIncrement[], Last[increment]},
    {ALUOpcodeDecrement[], Last[decrement]},
    {ALUOpcodePassThrough[], pass}
   }
  ]
 ];

overflowFlag = Mux8to1[opcode,
  GenerateMuxOutputByOpcode[3,
   {
    {ALUOpcodeAdd[], First[add]},
    {ALUOpcodeAddWithCarry[], First[addWithCarry]},
    {ALUOpcodeSubstract[], 0},
    {ALUOpcodeSubstractWithBorrow[], 0},
    {ALUOpcodeNegate[], 0},
    {ALUOpcodeIncrement[], First[increment]},
    {ALUOpcodeDecrement[], 0},
    {ALUOpcodePassThrough[], 0}
   }
  ]
 ];

negativeFlag = Mux8to1[opcode,
  GenerateMuxOutputByOpcode[3,
   {
    {ALUOpcodeAdd[], 0},
    {ALUOpcodeAddWithCarry[], 0},
    {ALUOpcodeSubstract[], First[substract]},
    {ALUOpcodeSubstractWithBorrow[], First[substractWithBorrow]},
    {ALUOpcodeNegate[], First[negate]},
    {ALUOpcodeIncrement[], 0},
    {ALUOpcodeDecrement[], First[decrement]},
    {ALUOpcodePassThrough[], 0}
   }
  ]
```

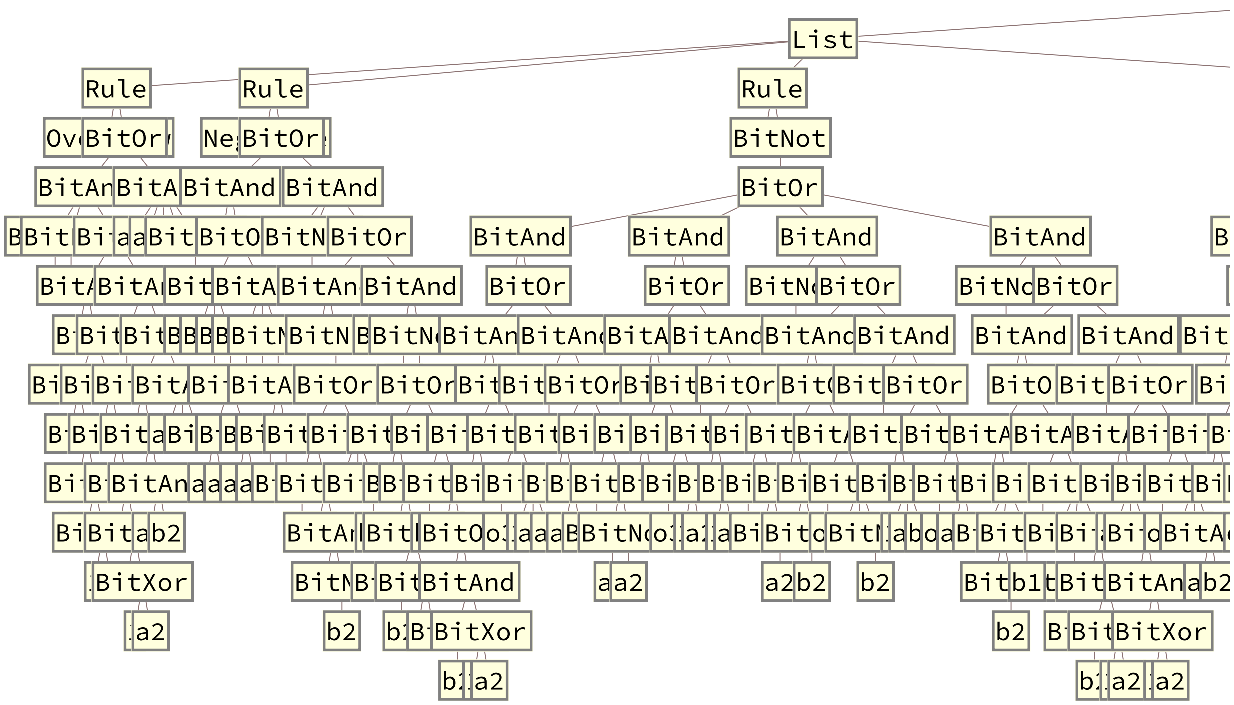
```
    ];

   Return[{<|"Overflow" → overflowFlag, "Negative" → negativeFlag,
      "Zero" → ALUZero[output], "Parity" → ALUParity[output]|>, output}];
  ];

ALUClear[] := Block[{},
   WriteFlag[0, "ALUFlagOverflow"];
   WriteFlag[0, "ALUFlagNegative"];
   WriteFlag[0, "ALUFlagZero"];
   WriteFlag[0, "ALUFlagParity"];
  ];
```

Visualización

Complejidad de una ALU de 2 bits

In[∘]:=
```
TreeForm[Normal[ALUExecute[{o1, o2, o3}, {a1, a2}, {b1, b2}]],
 PlotRangePadding -> Scaled[.002]]
```

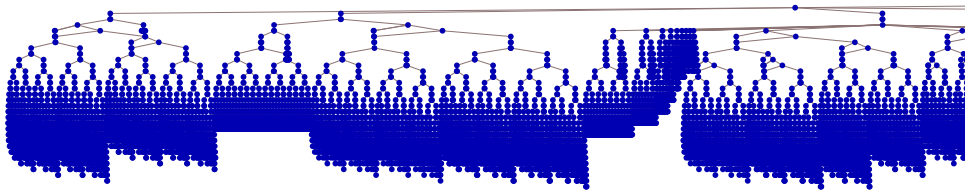
Out[∘]//TreeForm=



Complejidad de una ALU de 8 bits

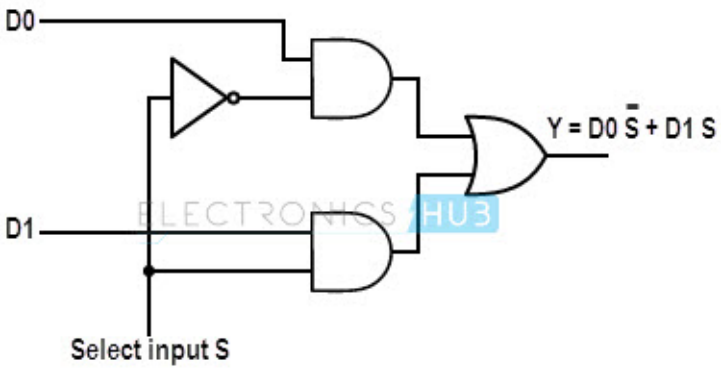
```
In[•]:= TreeForm[Normal[ALUExecute[{o1, o2, o3}, {a1, a2, a3, a4, a5, a6, a7, a8},
          {b1, b2, b3, b4, b5, b6, b7, b8}]], VertexLabeling → Automatic, ImageSize → 1500]
```

Out[•]//TreeForm=



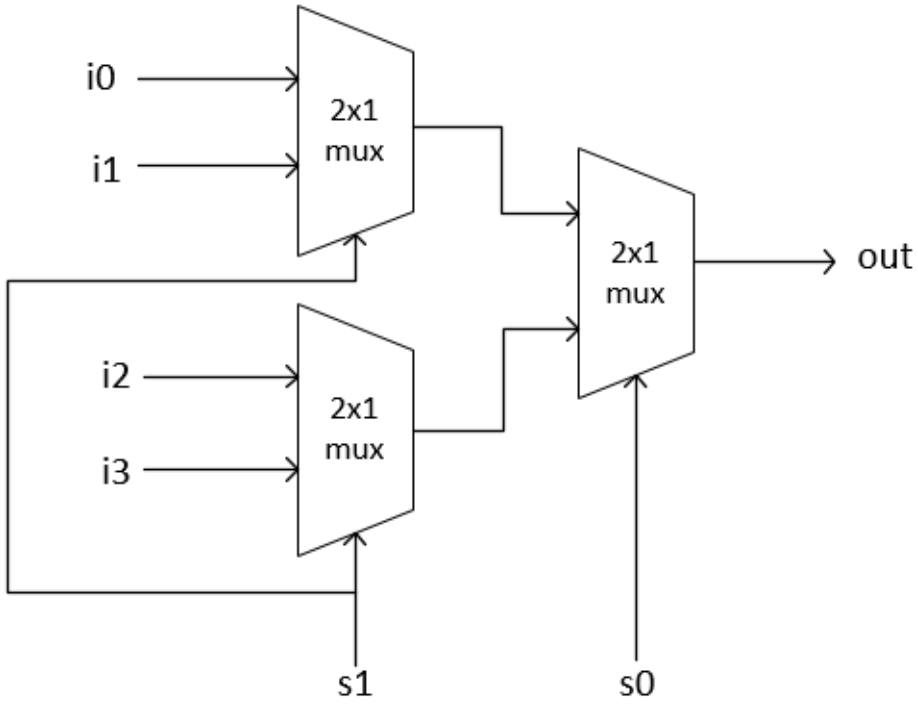
Multiplexor

- 2 a 1



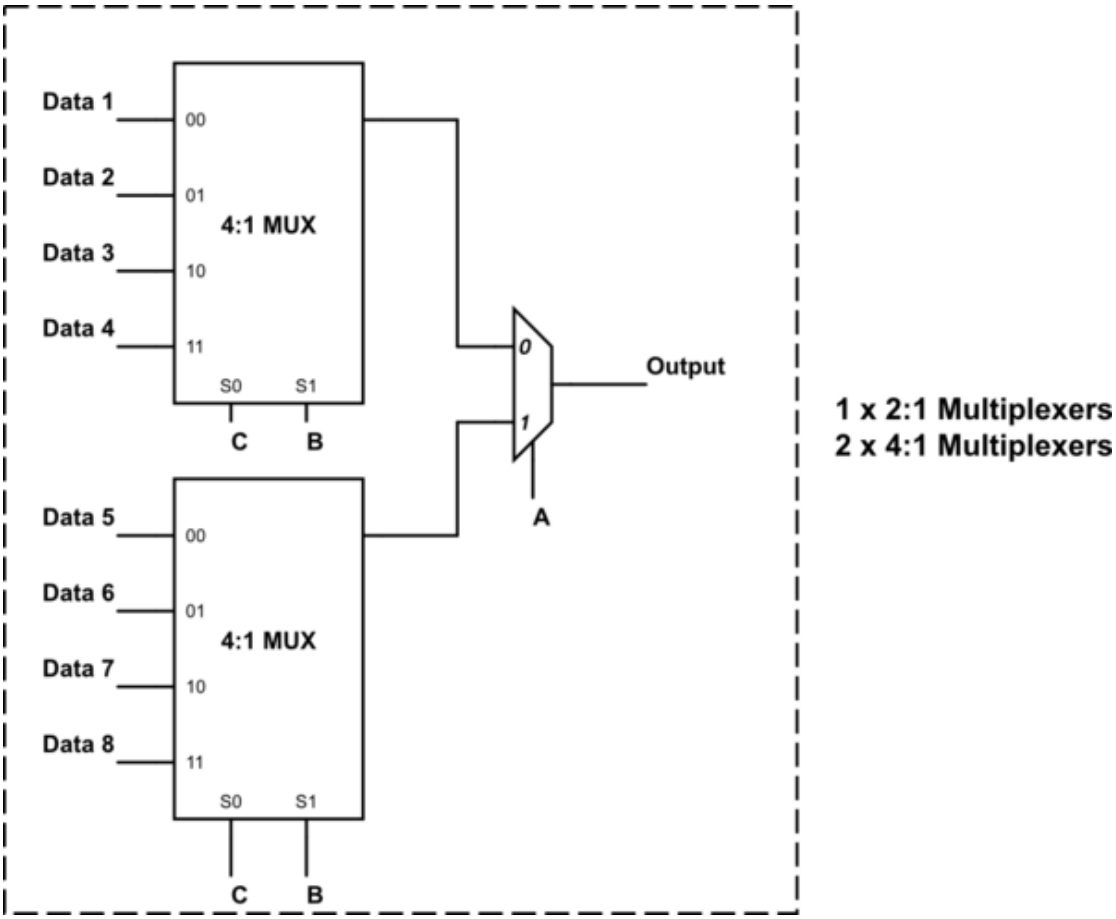

```
In[41]:= Mux2to1[{select_}, {input1_, input2_}] :=
           BitOr[BitAnd[input1, BitNot[select]], BitAnd[input2, select]];
```

- 4 a 1



In[42]:=
```
Mux4to1[{select1_, select2_}, {input1_, input2_, input3_, input4_}] := Mux2to1[
   {select1},
   {
    Mux2to1[{select2}, {input1, input2}],
    Mux2to1[{select2}, {input3, input4}]
   }
  ];
```

8 a 1



In[43]:=
```
Mux8to1[{select1_, select2_, select3_},
   {input1_, input2_, input3_, input4_, input5_, input6_, input7_, input8_}] :=
  Mux2to1[
   {select1},
   {
    Mux4to1[{select2, select3}, {input1, input2, input3, input4}],
    Mux4to1[{select2, select3}, {input5, input6, input7, input8}]
   }
  ];
```

Visualizaciones

Resultado

```
In[•]:= {VisualizeMultiplexer[Mux2to1, 1],
         VisualizeMultiplexer[Mux4to1, 2], VisualizeMultiplexer[Mux8to1, 3]}
```

Out[•]=

A	Selection
0	a
1	b

,

A	B	Selection
0	0	a
0	1	b
1	0	c
1	1	d

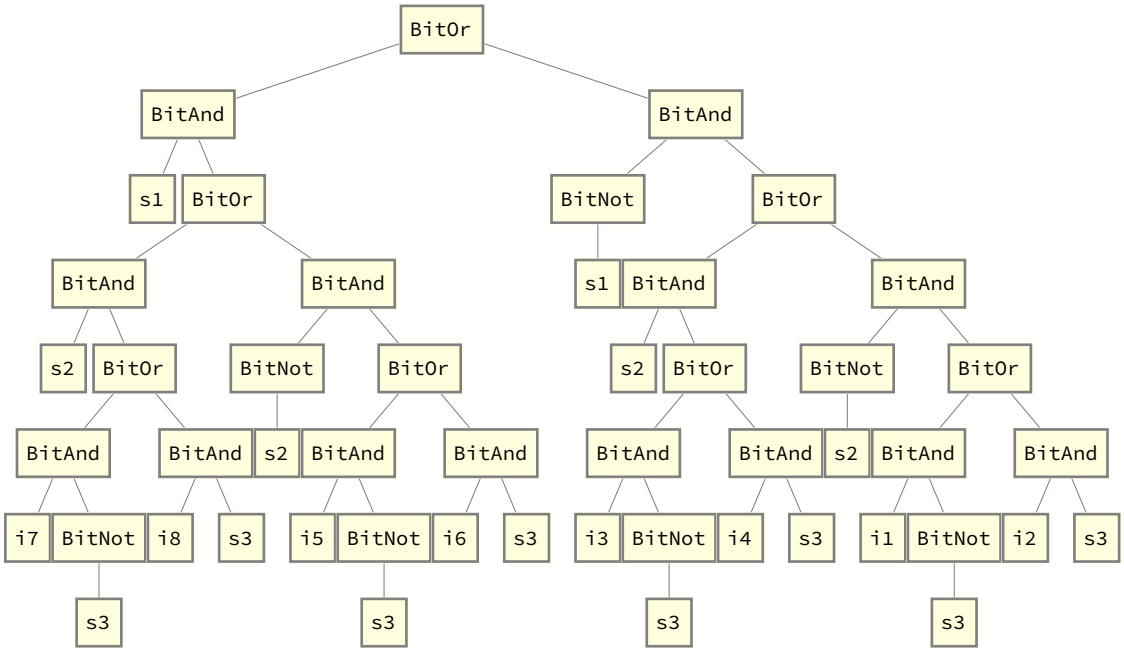
,

A	B	C	Selection
0	0	0	a
0	0	1	b
0	1	0	c
0	1	1	d
1	0	0	e
1	0	1	f
1	1	0	g
1	1	1	h

Árbol de complejidad

```
In[•]:= TreeForm[Mux8to1[{s1, s2, s3}, {i1, i2, i3, i4, i5, i6, i7, i8}]]
```

Out[•]//TreeForm=



Grid cuadrado

In[44]:=
```
MultiplexerNTo1[2, {select_}, input__] :=
  BitOr[BitAnd[First[input], BitNot[select]], BitAnd[Last[input], select]];
MultiplexerNTo1[n_, select_, input__] := Block[{partInput},
   partInput = HalfSplit[input];
   MultiplexerNTo1[
    2,
    {First[select]},
    {
     MultiplexerNTo1[n/2, Rest[select], First[partInput]],
     MultiplexerNTo1[n/2, Rest[select], Last[partInput]]
    }
   ]
  ];
```

Ejemplo

In[•]:=
```
VisualizeMultiplexer[MultiplexerNTo1[16, #1, #2] &, 4]
```

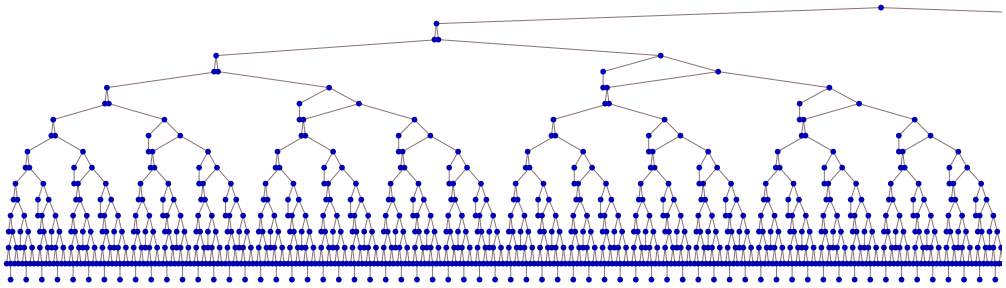
Out[•]=

A	B	C	D	Selection
0	0	0	0	a
0	0	0	1	b
0	0	1	0	c
0	0	1	1	d
0	1	0	0	e
0	1	0	1	f
0	1	1	0	g
0	1	1	1	h
1	0	0	0	i
1	0	0	1	j
1	0	1	0	k
1	0	1	1	l
1	1	0	0	m
1	1	0	1	n
1	1	1	0	o
1	1	1	1	p

Visualización

In[•]:=
```
selectSymbols = Table[Symbol["s" <> ToString[i]], {i, 1, 8}];
inputSymbols = Table[Symbol["i" <> ToString[i]], {i, 1, 256}];
TreeForm[MultiplexerNTo1[256, selectSymbols, inputSymbols],
 VertexLabeling → Automatic, ImageSize → 1300]
```

Out[•]//TreeForm=



Selección de Opcode por Multiplexor

Función auxiliar para ordenar los pares entrada-salida en los multiplexores

In[46]:=
```
GenerateAddresses[bitsize_] :=
  Map[PadLeft[IntegerDigits[#, 2], bitsize] &, Range[0, 2^bitsize - 1]];
GenerateMuxOutputByOpcode[n_, opcodeAndOutput_] := Block[{addresses, missing},
   addresses = GenerateAddresses[n];
   missing = Thread[{Complement[addresses, opcodeAndOutput[[All, 1]]], 0}];
   Return[SortBy[Join[missing, opcodeAndOutput], First][[All, 2]]];
  ];
```

Ejemplo

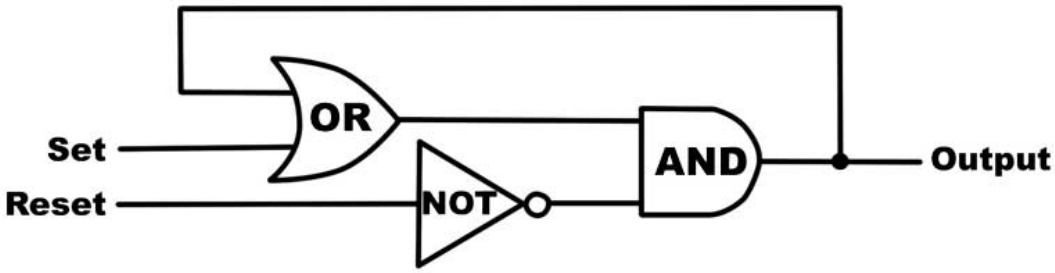
In[•]:=
```
GenerateMuxOutputByOpcode[3,
 {
  {ALUOpcodeAdd[], add},
  {ALUOpcodeSubstract[], substract},
  {ALUOpcodeIncrement[], increment}
 }
]
```

Out[•]= {add, 0, substract, 0, 0, increment, 0, 0}

Memoria

And-Or Latch

De la Wikipedia: In electronics, a flip-flop or latch is a circuit that has two stable states and can be used to store state information. A flip-flop is a bistable multivibrator. The circuit can be made to change state by signals applied to one or more control inputs and will have one or two outputs. It is the basic storage element in sequential logic. Flip-flops and latches are fundamental building blocks of digital electronics systems used in computers, communications, and many other types of systems.



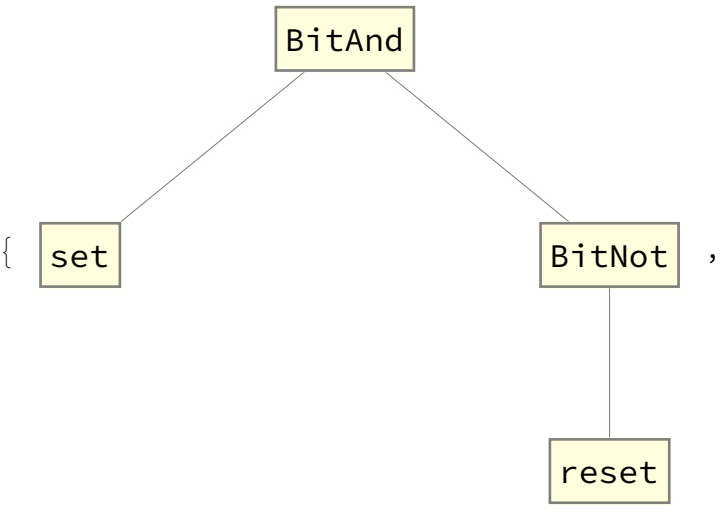
In[48]:=
```
AndOrLatch[set_, reset_, tag_] :=
  MemoryEvaluate[BitAnd[BitOr[#, set], BitNot[reset]] &, tag];
```

Este es el circuito base para todos los tipos de memoria que se utiliza a continuación

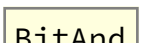
Visualización

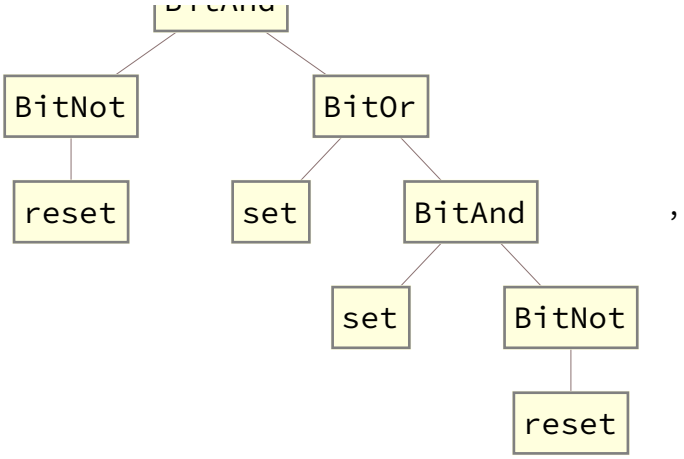
In[ ]:=
```
Table[TreeForm[AndOrLatch[set, reset, tag], ImageSize → Medium], 3]
```

Out[ ]=

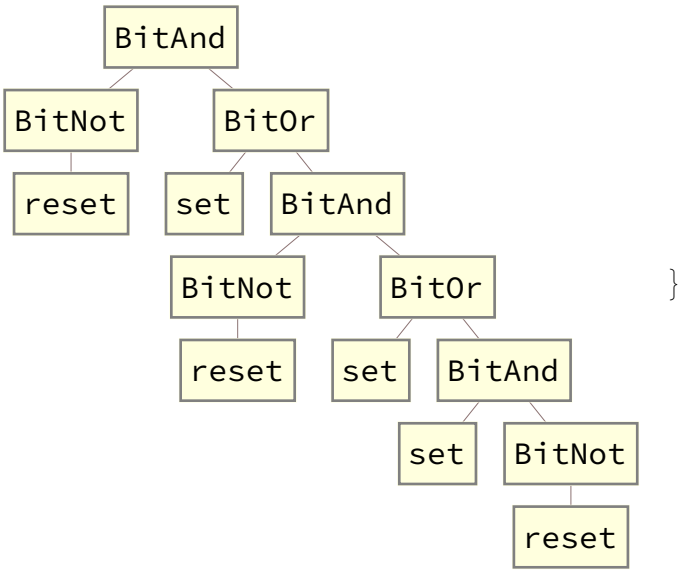


,



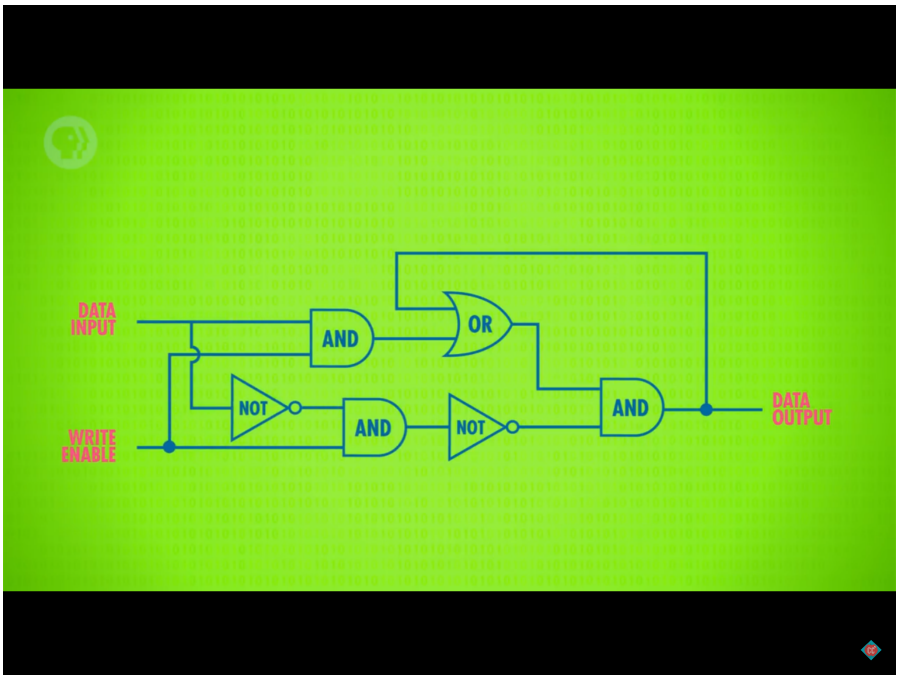

BitNot
BitOr
reset
set
BitAnd
set
BitNot
reset

,


BitAnd
BitNot
BitOr
reset
set
BitAnd
BitNot
BitOr
reset
set
BitAnd
set
BitNot
reset

}

- Gated Latch



Fuente: Crash Course Computer Science

In[49]:=
```
GatedLatch[dataInput_, writeEnable_, tag_] := AndOrLatch[
   BitAnd[dataInput, writeEnable], BitAnd[BitNot[dataInput], writeEnable], tag];
```

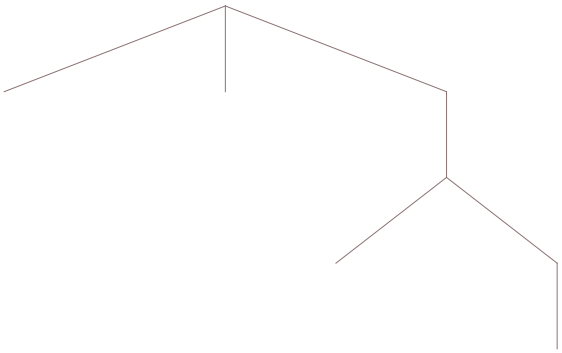
Estos circuitos nos sirven para guardar banderas de 1 bit

In[50]:=
```
ReadFlag[tag_] := GatedLatch[0, 0, tag];
WriteFlag[dataInput_, tag_] := GatedLatch[dataInput, 1, tag];
```

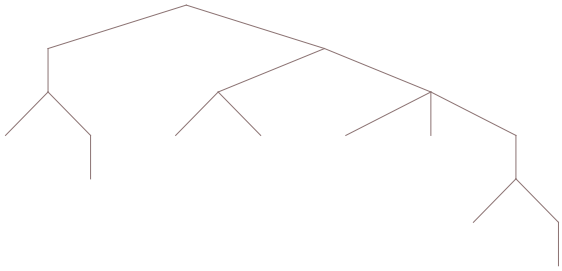
Visualización

In[ ]:=
```
Table[TreeForm[GatedLatch[dataInput, writeEnable, tag],
  ImageSize → Medium, VertexRenderingFunction → None], 3]
```

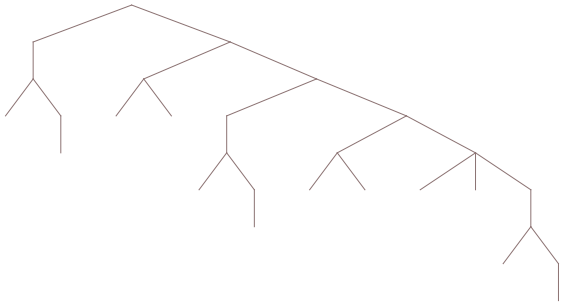
Out[ ]=



,



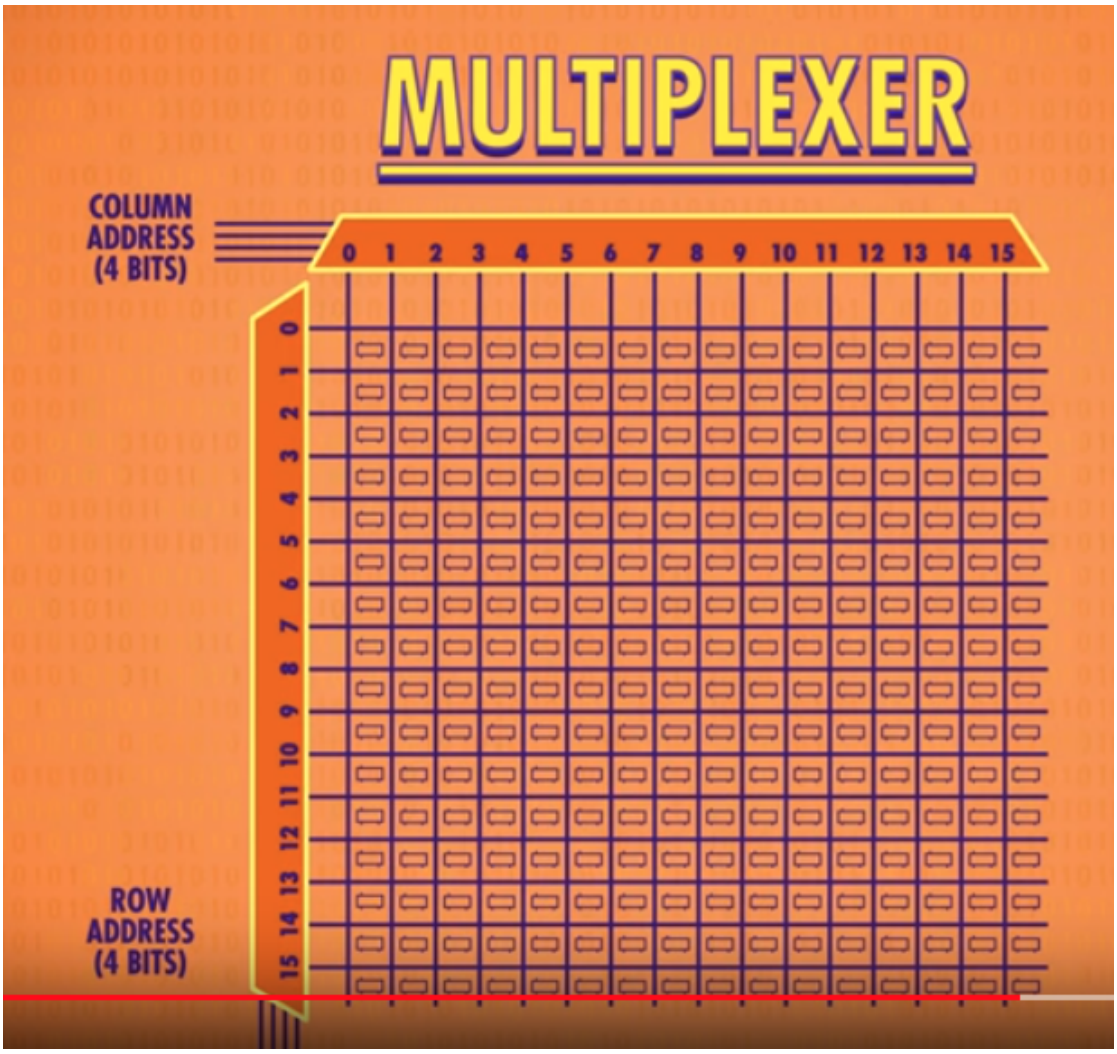
,



}

Celda

In[ ]:=

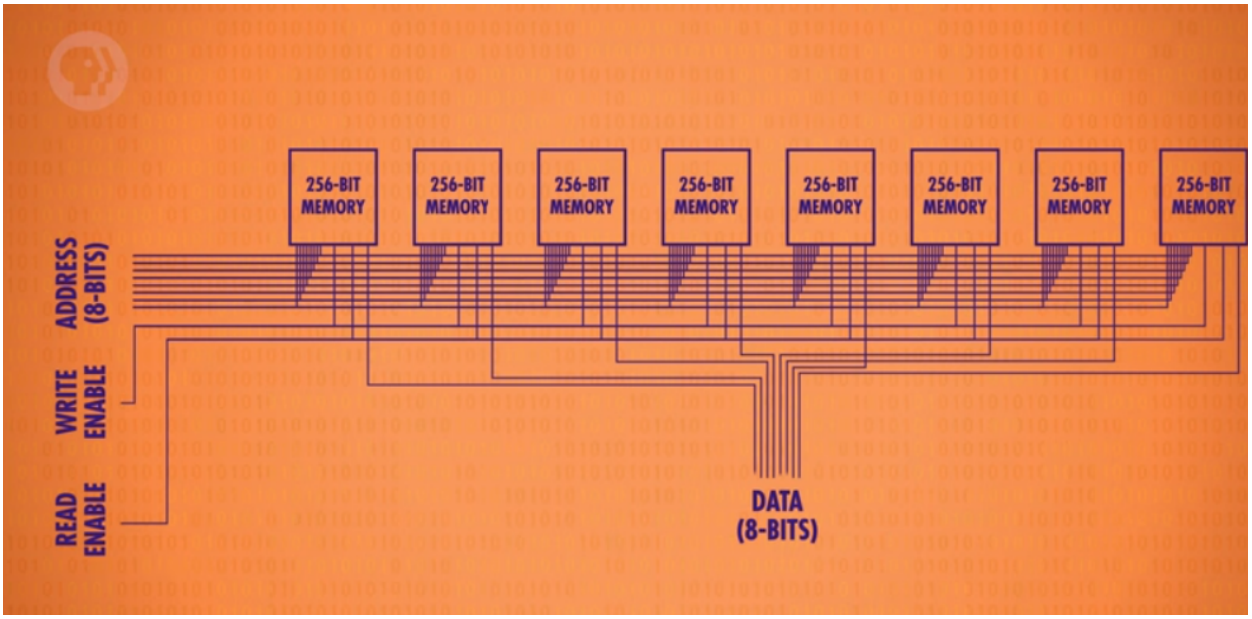


Fuente: Crash Course Computer Science

In[52]:=
```
Bit4AddressCell[dataInput_, writeEnable_, address__, tag_] :=
  GatedLatch[
   dataInput,
   writeEnable,
   StringJoin[
    tag,
    "bit_",
    ToString[
     MultiplexerNTo1[
      16,
      address,
      Range[16]
     ]
    ]
   ]
  ];
```

Bloque

In[•]:=



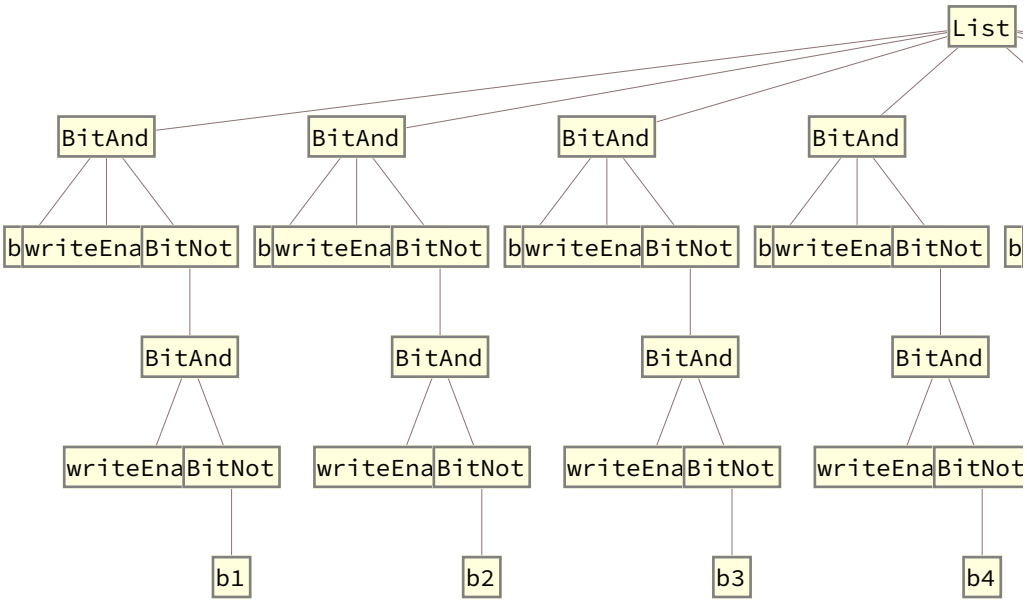
Fuente: Crash Course Computer Science

In[53]:=
```
Bit4AddressBlock[dataInput__ /; Length[dataInput] == 8, writeEnable_,
    address__, tag_] := Table[Bit4AddressCell[Part[dataInput, i],
     writeEnable, address, StringJoin[tag, "bit_", ToString[i]]], {i, 8}];

ReadBit4AddressBlock[address__, tag_] :=
  Bit4AddressBlock[{0, 0, 0, 0, 0, 0, 0, 0}, 0, address, tag];
WriteBit4AddressBlock[dataInput__ /; Length[dataInput] == 8, address__, tag_] :=
  Bit4AddressBlock[dataInput, 1, address, tag];
```

Visualización

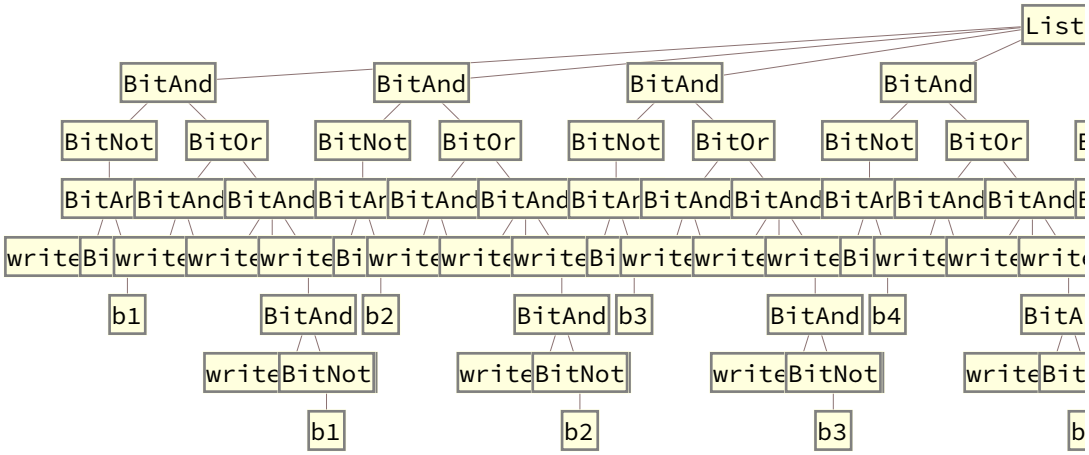
In[ ]:=
```
TreeForm[Bit4AddressBlock[{b1, b2, b3, b4, b5, b6, b7, b8},
  writeEnable, {a1, a2, a3, a4}, "exampletag"]]
```

Out[ ]//TreeForm=



```
In[•]:= TreeForm[Bit4AddressBlock[{b1, b2, b3, b4, b5, b6, b7, b8},
          writeEnable, {a1, a2, a3, a4}, "exampletag"]]
```

Out[•]//TreeForm=

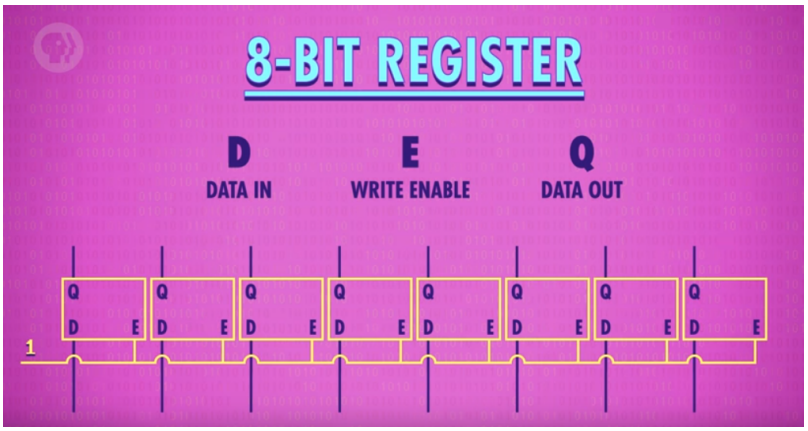


Visualización recursiva interactiva

```
In[•]:= trees = Table[TreeForm[Bit4AddressBlock[{b1, b2, b3, b4, b5, b6, b7, b8}, writeEnable,
          {a1, a2, a3, a4}, "manip"], VertexLabeling → Automatic, ImageSize → 1300], 15];
        Manipulate[trees[[i]], {i, 1, Length[trees], 1}, SaveDefinitions → True]
```

Registros

In[•]:=



Fuente: Crash Course Computer Science

In[56]:=
```
Bit3Register[dataInput__, writeEnable_, tag_] := Table[
   GatedLatch[
    Part[dataInput, index],
    writeEnable,
    StringJoin[tag, "_bit3register", ToString[index]]
   ],
   {index, 1, 3}
  ];
ReadBit3Register[tag_] := Bit3Register[{0, 0, 0, 0}, 0, tag];
WriteBit3Register[dataInput__, tag_] := Bit3Register[dataInput, 1, tag];


Bit4Register[dataInput__, writeEnable_, tag_] := Table[
   GatedLatch[
    Part[dataInput, index],
    writeEnable,
    StringJoin[tag, "_bit4register", ToString[index]]
   ],
   {index, 1, 4}
  ];
ReadBit4Register[tag_] := Bit4Register[{0, 0, 0, 0}, 0, tag];
WriteBit4Register[dataInput__, tag_] := Bit4Register[dataInput, 1, tag];

Bit8Register[dataInput__, writeEnable_, tag_] := Table[
   GatedLatch[
    Part[dataInput, index],
    writeEnable,
    StringJoin[tag, "_bit8register", ToString[index]]
   ],
   {index, 1, 8}
  ];
ReadBit8Register[tag_] := Bit8Register[{0, 0, 0, 0, 0, 0, 0, 0}, 0, tag];
WriteBit8Register[dataInput__, tag_] := Bit8Register[dataInput, 1, tag];
```

Unidad de control

Auxiliar

Reseteo de la unidad de control

In[65]:=
```
CUClear[] := Block[{},
   WriteBit4Register[{0, 0, 0, 0}, "InstructionAddress"];
   WriteBit8Register[{0, 0, 0, 0, 0, 0, 0, 0}, "InstructionRegister"];
   WriteBit8Register[{0, 0, 0, 0, 0, 0, 0, 0}, "RegisterA"];
   WriteBit8Register[{0, 0, 0, 0, 0, 0, 0, 0}, "RegisterB"];
   WriteFlag[0, "Ram Read-Enable"];
   WriteFlag[0, "Ram Write-Enable"];
   WriteFlag[0, "RegisterA Read-Enable"];
   WriteFlag[0, "RegisterB Read-Enable"];
   WriteFlag[0, "RegisterA Write-Enable"];
   WriteFlag[0, "RegisterB Write-Enable"];
   WriteFlag[0, "ALU Opcode"];
   WriteFlag[0, "Halt"];
  ];
```

Opcodes

In[66]:=
```
OpcodeLoadA[] = {0, 0, 1, 0};
OpcodeLoadB[] = {0, 0, 0, 1};
OpcodeStoreA[] = {0, 1, 0, 0};
OpcodeStoreB[] = {0, 1, 1, 0};
OpcodeAdd[] = {1, 0, 0, 0};
OpcodeSubstract[] = {1, 0, 0, 1};
OpcodeIncrement[] = {1, 0, 1, 0};
OpcodeJump[] = {1, 1, 0, 0};
OpcodeJumpNeg[] = {1, 1, 0, 1};
OpcodeJumpZero[] = {0, 1, 0, 1};
OpcodePrint[] = {1, 0, 1, 1};
OpcodeHalt[] = {0, 0, 0, 0};

cpuMnemonics = {
   OpcodeLoadA[] → "LoadA", OpcodeLoadB[] → "LoadB",
   OpcodeStoreA[] → "StoreA", OpcodeStoreB[] → "StoreB", OpcodeAdd[] → "Add",
   OpcodeSubstract[] → "Substract", OpcodeIncrement[] → "Increment",
   OpcodeJump[] → "Jump", OpcodeJumpNeg[] → "JumpNeg",
   OpcodeJumpZero[] → "JumpZero",
   OpcodePrint[] → "Print", OpcodeHalt[] → "Halt"
  };
```

Fetch

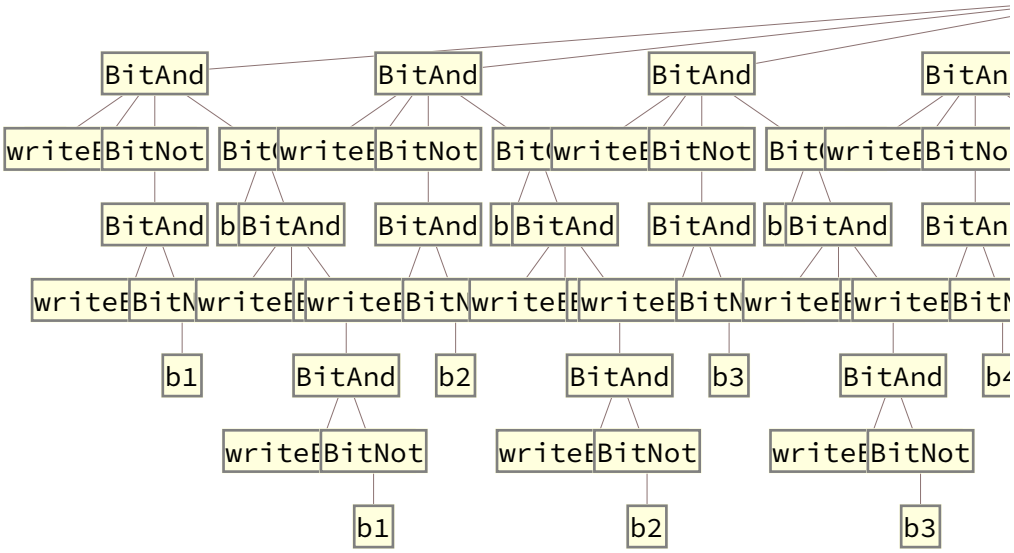
In[79]:=
```
CUFetch[] := Block[{instructionAddress, instruction},
   instructionAddress = ReadBit4Register["InstructionAddress"];
   instruction = ReadBit4AddressBlock[instructionAddress, "MainMemory"];
   WriteBit8Register[instruction, "InstructionRegister"];
  ];
```

Visualización

In[•]:=
```
op = WriteBit8Register[Bit4AddressBlock[{b1, b2, b3, b4, b5, b6, b7, b8}, writeEnable,
     ReadBit4Register["InstructionAddress"], "MainMemory"], "InstructionRegister"];
```

In[•]:=
```
TreeForm[op]
```

Out[•]//TreeForm=



Decode

En decode se escriben las banderas adecuadas que permitirán el paso del resultado a los registros o sectores de memoria adecuados

In[80]:=
```
CUDecode[] := Block[
   {
    opcode, ramReadEnable, ramWriteEnable, registerAReadEnable,
    registerBReadEnable, registerAWriteEnable, registerBWriteEnable,
    aluOpcode, aluInstruction, halt, jumpEnable, jumpNegEnable,
```

```
 jumpZeroEnable, printEnable,
 return = {}
},

opcode = First[HalfSplit[ReadBit8Register["InstructionRegister"]]];
ramReadEnable = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodeLoadA[], 1},
    {OpcodeLoadB[], 1}
   }
  ]
 ];
AppendTo[return, WriteFlag[ramReadEnable, "Ram Read-Enable"]];

ramWriteEnable = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodeStoreA[], 1}
   }
  ]
 ];
AppendTo[return, WriteFlag[ramWriteEnable, "Ram Write-Enable"]];


registerAReadEnable = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodeStoreA[], 1}
   }
  ]
 ];
AppendTo[return, WriteFlag[registerAReadEnable, "RegisterA Read-Enable"]];


registerBReadEnable = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodeStoreB[], 1}
   }
  ]
 ];
AppendTo[return, WriteFlag[registerBReadEnable, "RegisterB Read-Enable"]];
```

```
registerAWriteEnable = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodeLoadA[], 1},
    {OpcodeAdd[], 1},
    {OpcodeSubstract[], 1},
    {OpcodeIncrement[], 1}
   }
  ]
 ];
AppendTo[return, WriteFlag[registerAWriteEnable, "RegisterA Write-Enable"]];

registerBWriteEnable = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodeLoadB[], 1}
   }
  ]
 ];
AppendTo[return, WriteFlag[registerBWriteEnable, "RegisterB Write-Enable"]];

jumpEnable = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodeJump[], 1}
   }
  ]
 ];
AppendTo[return, WriteFlag[jumpEnable, "Jump"]];

jumpNegEnable = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodeJumpNeg[], 1}
   }
  ]
 ];
AppendTo[return, WriteFlag[jumpNegEnable, "JumpNeg"]];

jumpZeroEnable = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodeJumpZero[], 1}
   }
```

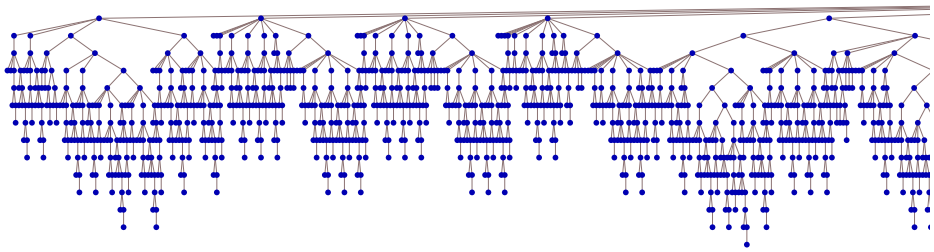
```
  ]
 ];
AppendTo[return, WriteFlag[jumpZeroEnable, "JumpZero"]];

aluInstruction = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodeAdd[], 1},
    {OpcodeSubstract[], 1},
    {OpcodeIncrement[], 1}
   }
  ]
 ];
AppendTo[return, WriteFlag[aluInstruction, "ALU Instruction"]];

aluOpcode = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodeAdd[], ALUOpcodeAdd[]},
    {OpcodeSubstract[], ALUOpcodeSubstract[]},
    {OpcodeIncrement[], ALUOpcodeIncrement[]}
   }
  ]
 ];
AppendTo[return, WriteBit3Register[aluOpcode, "ALU Opcode"]];

printEnable = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodePrint[], 1}
   }
  ]
 ];
AppendTo[return, WriteFlag[printEnable, "Print Enable"]];

halt = MultiplexerNTo1[16, opcode,
  GenerateMuxOutputByOpcode[4,
   {
    {OpcodeHalt[], 1}
   }
  ]
 ];
AppendTo[return, WriteFlag[halt, "Halt"]];
Return[return];
```

Visualización

In[ ]:=
```
op1 = WriteBit8Register[Bit4AddressBlock[{b1, b2, b3, b4, b5, b6, b7, b8},
    writeEnable, {a1, a2, a3, a4}, "MainMemory"], "InstructionRegister"];
op2 = CUDecode[];
```

In[ ]:=
```
TreeForm[op2, VertexLabeling → Automatic]
```

Out[ ]//TreeForm=



Execute

In[103]:=
```
CUExecute[] :=
  Block[{param, readFromRam, aluResult, nextInstructionAddress, return = {}},
   param = Last[HalfSplit[ReadBit8Register["InstructionRegister"]]];
   readFromRam = ReadBit4AddressBlock[param, "MainMemory"];

   (* Read operations *)
   AppendTo[return,
    Bit8Register[readFromRam, BitAnd[ReadFlag["RegisterA Write-Enable"],
      BitNot[ReadFlag["ALU Instruction"]]], "RegisterA"]];
   AppendTo[return, Bit8Register[readFromRam,
     BitAnd[ReadFlag["RegisterB Write-Enable"],
      BitNot[ReadFlag["ALU Instruction"]]], "RegisterB"]];

   (* ALU operations *)
   aluResult = ALUExecute[ReadBit3Register["ALU Opcode"],
     ReadBit8Register["RegisterA"], ReadBit8Register["RegisterB"]];
   AppendTo[return, Bit8Register[Last[aluResult], BitAnd[ReadFlag[
       "RegisterA Write-Enable"], ReadFlag["ALU Instruction"]], "RegisterA"]];
   AppendTo[return, GatedLatch[First[aluResult]["Overflow"],
     ReadFlag["ALU Instruction"], "ALUFlagOverflow"]];
   AppendTo[return, GatedLatch[First[aluResult]["Negative"],
     ReadFlag["ALU Instruction"], "ALUFlagNegative"]];
   AppendTo[return, GatedLatch[First[aluResult]["Zero"],
     ReadFlag["ALU Instruction"], "ALUFlagZero"]];
```

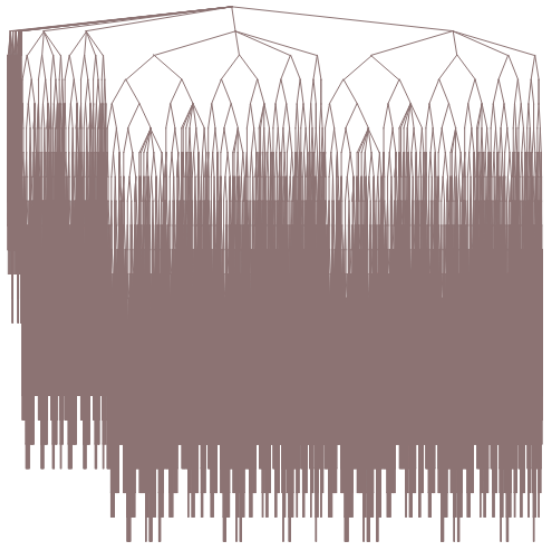
```
  AppendTo[return, GatedLatch[First[aluResult]["Parity"],
    ReadFlag["ALU Instruction"], "ALUFlagParity"]];

  (* Write operations *)
  AppendTo[return, Bit4AddressBlock[PadLeft[ReadBit8Register["RegisterA"], 8],
    ReadFlag["Ram Write-Enable"], param, "MainMemory"]];

  (* Instruction address operations *)
  AppendTo[return,
   Bit4Register[param, ReadFlag["Jump"], "InstructionAddress"]];
  AppendTo[return, Bit4Register[param, BitAnd[ReadFlag["JumpNeg"],
      ReadFlag["ALUFlagNegative"]], "InstructionAddress"]];

  nextInstructionAddress =
   Last[ALUAdder[ReadBit4Register["InstructionAddress"], {0, 0, 0, 1}]];
  AppendTo[return, Bit4Register[nextInstructionAddress,
    BitAnd[BitNot[ReadFlag["Jump"]], BitNot[ReadFlag["JumpNeg"]]],
    "InstructionAddress"]];
  AppendTo[return, Bit4Register[nextInstructionAddress,
    BitAnd[ReadFlag["JumpNeg"], BitNot[ReadFlag["ALUFlagNegative"]]],
    "InstructionAddress"]];

  If[ReadFlag["JumpNeg"], Print[ReadBit8Register["RegisterA"]]];

  Return[return];
 ];
```

Visualización

In[104]:=
```
op1 = CUFetch[];
op2 = CUDecode[];
op3 = CUExecute[];
```

In[111]:=
```
Rasterize[
 TreeForm[op3, VertexRenderingFunction → None, AspectRatio → 1, ImageSize → 600]]
```

Out[111]=



In[•]:=
```
PlotInteractiveSystem[op3, None]
```

CPU

Atajos de ejecución

In[82]:=
```
CPUCycle[] := Block[{},
   CUFetch[];
   CUDecode[];

   If[ReadFlag["Halt"] ≠ 1, CUExecute[]];
  ];
Execute[program_] := Block[{steps},
   ClearProgram[];
   ALUClear[];
   CUClear[];
   LoadProgram[program];

   steps = Reap[
      CUFetch[];
      CUDecode[];
      Sow[CPUPlot[]];
```

```
      While[ReadFlag["Halt"] ≠ 1,
       CUExecute[];
       CUFetch[];
       CUDecode[];
       Sow[CPUPlot[]];
      ]
     ][[2, 1]];
   Return[steps];
  ];

Execute[program_, cycles_] := Block[{steps, iter = 0},
   ClearProgram[];
   ALUClear[];
   CUClear[];
   LoadProgram[program];

   steps = Reap[
      CUFetch[];
      CUDecode[];
      Sow[CPUPlot[]];

      While[ReadFlag["Halt"] ≠ 1 && iter < cycles,
       CUExecute[];
       CUFetch[];
       CUDecode[];
       Sow[CPUPlot[]];
       iter++;
      ]
     ][[2, 1]];
   Return[steps];
  ];
```

Utilidades

- Carga de programa a la memoria

In[85]:=
```
LoadProgram[program_] := Block[{addresses},
   addresses = Map[PadLeft[IntegerDigits[#, 2], 4] &, Range[0, 15]];
   MapThread[
    WriteBit4AddressBlock[#2, #1, "MainMemory"] &, {addresses, program}];
  ];
ClearProgram[] := Block[{addresses},
   addresses = Map[PadLeft[IntegerDigits[#, 2], 4] &, Range[0, 15]];
   Map[
    WriteBit4AddressBlock[{0, 0, 0, 0, 0, 0, 0, 0}, #1, "MainMemory"] &, addresses];
  ];
```

Compilar desde ensamblador

In[87]:=
```
assemblyOpcodes = Map[Reverse, cpuMnemonics];
HexToBinary[n_, pad_: 4] := PadLeft[IntegerDigits[FromDigits[n, 16], 2], pad];
DecToBinary[n_, pad_: 4] := PadLeft[IntegerDigits[FromDigits[n, 10], 2], pad];
BinaryToHex[n_] := BaseForm[FromDigits[n, 2], 16];
FromInstructionToBitcode[instruction_] := Switch[Length[Rest[instruction]],
   0, Join[First[instruction], {0, 0, 0, 0}],
   1, Join[First[instruction], DecToBinary[Last[instruction]]],
   2, Join[First[instruction],
    DecToBinary[Part[instruction, 2], 2], DecToBinary[Part[instruction, 3], 2]]
  ];
FromInstructionTableToBitcode[code_] := Map[FromInstructionToBitcode,
   ReplaceAll[Map[StringSplit, StringSplit[code, "\n"]], assemblyOpcodes]];
AssemblyCompile[code_] := PadRight[FromInstructionTableToBitcode[code],
   16, {{0, 0, 0, 0, 0, 0, 0, 0}}];
```

Gráficas

In[94]:=
```
PlotRAM[] := Block[{addresses, ram, currentAddress},
   addresses = Map[PadLeft[IntegerDigits[#, 2], 4] &, Range[0, 15]];
   ram = Map[ReadBit4AddressBlock[#, "MainMemory"] &, addresses];
   currentAddress = FromDigits[ReadBit4Register["InstructionAddress"], 2];
   Panel[Grid[Join[{{"Address", "Value"}}, Thread[{Range[0, 15], ram}]],
     Background → {None, {1 → LightGreen, (currentAddress + 2) → Yellow}},
     Frame → All], Style["RAM", Bold]]
  ];
ALUStatusPlot[] := Panel[
   Grid[{
```

```
      {"Overflow: ", ReadFlag["ALUFlagOverflow"]},
      {"Negative: ", ReadFlag["ALUFlagNegative"]},
      {"Zero: ", ReadFlag["ALUFlagZero"]},
      {"Parity: ", ReadFlag["ALUFlagParity"]}
     },
     Frame → All,
     Background → {{LightRed}, None}
    ],
    Style["ALU Status", Bold]
   ];

CUStatusPlot[] := Block[{mainRegisters, instructionRegisters, flagsStatus},
    mainRegisters = Grid[
      {
       {"Register A", "Register B"},
       {Row[ReadBit8Register["RegisterA"]], Row[ReadBit8Register["RegisterB"]]}
      },
      Frame → All,
      Background → {None, {LightGreen}}
     ];

    instructionRegisters = Grid[
      {
       {"Instruction Register", "Address Register"},
       {
        Grid[
         {
          {"Opcode", "Parameter"},
          Map[Row, HalfSplit[ReadBit8Register["InstructionRegister"]]],
          {First[HalfSplit[ReadBit8Register["InstructionRegister"]]] /.
            cpuMnemonics, BinaryToHex[
            Last[HalfSplit[ReadBit8Register["InstructionRegister"]]]]}
         },
         Frame → All, Background → {None, {LightCyan}}
        ],
        Row[ReadBit4Register["InstructionAddress"]]
       }
      },
      Frame → All,
      Background → {None, {LightBlue}}
     ];

    flagsStatus = Grid[{
       {"Ram Read-Enable: ", ReadFlag["Ram Read-Enable"]},
```

```
       {"Ram Write-Enable: ", ReadFlag["Ram Write-Enable"]},
       {"RegisterA Read-Enable: ", ReadFlag["RegisterA Read-Enable"]},
       {"RegisterB Read-Enable: ", ReadFlag["RegisterB Read-Enable"]},
       {"RegisterA Write-Enable: ", ReadFlag["RegisterA Write-Enable"]},
       {"RegisterB Write-Enable: ", ReadFlag["RegisterB Write-Enable"]},
       {"Jump: ", ReadFlag["Jump"]},
       {"JumpNeg: ", ReadFlag["JumpNeg"]},
       {"JumpZero: ", ReadFlag["JumpZero"]},
       {"ALU Instruction: ", ReadFlag["ALU Instruction"]},
       {"ALU Opcode: ", ReadBit3Register["ALU Opcode"] /. aluMnemonics},
       {"Halt: ", ReadFlag["Halt"]}
      },
      Frame → All,
      Background → {{LightRed}, None}
     ];

    Panel[Column[{mainRegisters, instructionRegisters, flagsStatus}],
     Style["CU Status", Bold], ImageSize → 250]
  ];

CPUPlot[] :=
  Panel[Grid[{{ALUStatusPlot[], CUStatusPlot[], PlotRAM[], CPUModelPlot[model]}},
    Alignment → Top], Style["CPU", Bold]];
ExecutionPlot[program_] := DynamicModule[{result = Execute[program]},
   Manipulate[Part[result, i], {i, 1, Length[result], 1}]
  ];
ExecutionPlot[program_, cycles_] :=
  DynamicModule[{result = Execute[program, cycles]},
   Manipulate[Part[result, i], {i, 1, Length[result], 1}]
  ];

CreateCPUModel[] :=
  Block[{cpuStep1, cpuStep2, cpuStep3, b1, b2, b3, b4, b5, b6, b7, b8},
   ClearProgram[];
   ALUClear[];
   CUClear[];
   cpuStep1 =
    WriteBit8Register[{b1, b2, b3, b4, b5, b6, b7, b8}, "InstructionRegister"];
   cpuStep2 = CUDecode[];
   cpuStep3 = CUExecute[];
   Return[cpuStep3];
  ];
model = CreateCPUModel[];
```

```
CPUModelPlot[model_] := Block[{argument, b1, b2, b3, b4, b5, b6, b7, b8},
   argument = Thread[{b1, b2, b3, b4, b5, b6, b7, b8} -> ReadBit4AddressBlock[
       ReadBit4Register["InstructionAddress"], "MainMemory"]];
   Panel[PlotExpression[model, argument], Style["Model plot", Bold]]
  ];
```

Ejecución

In[120]:=
```
code = "
LoadA 14
LoadB 15
Substract 1 0
JumpNeg 5
Jump 2
Add 1 0
StoreA 13
Halt
";

program = AssemblyCompile[code];
setValues = {{15, DecToBinary["110", 8]}, {16, DecToBinary["5", 8]}};
MapThread[(Part[program, #1] = #2) &, Transpose[setValues]];
```

In[127]:= `ExecutionPlot[program, 2]`

Out[127]=

i

2

CPU

ALU Status

Overflow:	0
Negative:	0
Zero:	0
Parity:	0

CU Status

Register A	Register B
01101110	00000000

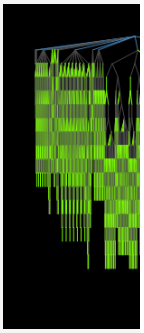
Instruction Register		Address Register
Opcode	Parameter	0001
0001	1111	
LoadB	f_{16}	

Ram Read–Enable:	1
Ram Write–Enable:	0
RegisterA Read–Enable:	0
RegisterB Read–Enable:	0
RegisterA Write–Enable:	0
RegisterB Write–Enable:	1
Jump:	0
JumpNeg:	0
JumpZero:	0
ALU Instruction:	0
ALU Opcode:	ALU Add
Halt:	0

RAM

Address	Value
0	{0, 0, 1, 0, 1, 1, 1, 0}
1	{0, 0, 0, 1, 1, 1, 1, 1}
2	{1, 0, 0, 1, 0, 1, 0, 0}
3	{1, 1, 0, 1, 0, 1, 0, 1}
4	{1, 1, 0, 0, 0, 0, 1, 0}
5	{1, 0, 0, 0, 0, 1, 0, 0}
6	{0, 1, 0, 0, 1, 1, 0, 1}
7	{0, 0, 0, 0, 0, 0, 0, 0}
8	{0, 0, 0, 0, 0, 0, 0, 0}
9	{0, 0, 0, 0, 0, 0, 0, 0}
10	{0, 0, 0, 0, 0, 0, 0, 0}
11	{0, 0, 0, 0, 0, 0, 0, 0}
12	{0, 0, 0, 0, 0, 0, 0, 0}
13	{0, 0, 0, 0, 0, 0, 0, 0}
14	{0, 1, 1, 0, 1, 1, 1, 0}
15	{0, 0, 0, 0, 0, 1, 0, 1}

Model plot



Visualización de ejecución de modelo del sistema

In[ ]:=
```
CPUModelPlotToExport[model_] := Block[{argument, b1, b2, b3, b4, b5, b6, b7, b8},
   argument = Thread[{b1, b2, b3, b4, b5, b6, b7, b8} -> ReadBit4AddressBlock[
       ReadBit4Register["InstructionAddress"], "MainMemory"]];
   PlotExpression[model, argument]
  ];

ExecuteModel[program_] := Block[{steps},
   ClearProgram[];
   ALUClear[];
   CUClear[];
   LoadProgram[program];

   steps = Reap[
      CUFetch[];
      CUDecode[];
      Sow[CPUModelPlotToExport[Last[model]]];

      While[ReadFlag["Halt"] ≠ 1,
       CUExecute[];
       CUFetch[];
       CUDecode[];
       Sow[CPUModelPlotToExport[Last[model]]];
      ]
     ][[2, 1]];
   Return[steps];
  ];
```

```
In[•]:= code = "
        LoadA 14
        LoadB 15
        Substract 1 0
        JumpNeg 10
        StoreA 12
        LoadA 13
        Increment
        StoreA 13
        LoadA 12
        Jump 2
        Halt
        ";
        program = AssemblyCompile[code];
        setValues = {{13 + 1, DecToBinary["0", 8]},
            {14 + 1, DecToBinary["110", 8]}, {15 + 1, DecToBinary["5", 8]}};
        MapThread[(Part[program, #1] = #2) &, Transpose[setValues]];

In[•]:= imgs = ExecuteModel[program];

        exportPath = FileNameJoin[{NotebookDirectory[], "plots", "image"}];
        Table[Export[StringJoin[exportPath, IntegerString[i, 10, 4], ".png"], imgs[[i]]],
          {i, 1, Length[imgs]}];
```