hack.encode

```
Require Import hack.ast.
Require Import hack.ast.reg.
Require Import hack.ast.instruction.
Require Import hack.nbitstream.
```

Binary encoding.

This module encodes hack assembly language programs into binary. The structure of this source is as follows.

1. Each module below packages the types and functions relevant for a certain portion of the assembler instruction. For example, the module jump packages stuff relevant for encoding the jump portion of the instruction.

2. Inside the module for the component `Comp` of the instruction, the type `encoding` is the stream type relevant for `Comp`. In particular it fixes the number of bits relevant for that type.

3. The `encode` function then takes care of converting the actual `ast` element to the binary encoding.

Jumps

```
Module jump.
```

Jump is encoded using 3-bits

```
Definition encoding := stream 3.
```

Converts a `instruction.jump` to binary

```
Definition jump (j : instruction.jump) : encoding :=
  vector match j with
    | JGT => [ 0; 0; 1]
    | JEQ => [ 0; 1; 0]
    | JGE => [ 0; 1; 1]
    | JLT => [ 1; 0; 0]
    | JNE => [ 1; 0; 1]
    | JLE => [ 1; 1; 0]
    | JMP => [ 1; 1; 1]
    end%bit.
```

Encoding optional jump into `jump.encoding`

```
Definition encode (oj : option instruction.jump) : encoding :=
  match oj with
  | None => zeros
  | Some j => jump j
  end%bit.
```

```
End jump.
```

Destination

```
Module destination.
```

Encoded as 3bits one for each bit A, D, and M

```
  Definition encoding := stream 3.
  Definition reg (r : ast.reg.t) : encoding :=
    vector match r with
      | reg.M => [0; 0; 1]
      | reg.D => [0; 1; 0]
      | reg.A => [1; 0; 0]
      end%bit.
```

Encoding multiple destinations into `destination.encoding`

```
  Definition encode (rs : list ast.reg.t) : encoding :=
    let comb (s1 s2 : encoding) := match s1, s2 with
                                   | streamOf n1, streamOf n2 => streamOf (N.lor n1
n2)
                                   end in
    List.fold_left comb (List.map reg rs) zeros.

End destination.
```

Comptational output

```
Module computation.
```

A computation is encoding into 7 bits

```
  Definition encoding := stream 7.
```

Encoding a constant into `computation.encoding`

```
  Definition constant (c : instruction.const) : encoding :=
    vector match c with
      | Zero => [0;1;0;1;0;1;0]
      | One => [0;1;1;1;1;1;1]
      | MinusOne => [0;1;1;1;0;1;0]
      end%bit.
```

Encoding application of unary operation

```
  Module unary.
```

Almost all (except the succ instruction) unary instructions can actually be split into the encoding of the operator and the register separately

```
    Definition defaultRegEnc (r : ast.reg.t) : stream 5 :=
      vector match r with
        | reg.D => [0;0;0;1;1]
        | reg.A => [0;1;1;0;0]
        | reg.M => [1;1;1;0;0]
        end%bit.

    Definition defaultUnary (o : instruction.unary) : stream 2 :=
      vector match o with
        | ID => [0;0]
        | BNeg => [0;1]
        | UMinus => [1;1]
        | Pred => [1;0]
        | Succ => [1;1]
        end%bit.
```

The succ instruction does not follow the default encoding of registers so we have a function just for its encoding.

```
    Definition succ (r : ast.reg.t) : encoding :=
      vector match r with
        | reg.D => [0;0;1;1;1;1;1]
        | reg.A => [0;1;1;0;1;1;1]
        | reg.M => [1;1;1;0;1;1;1]
        end%bit.

    Definition encode (o : instruction.unary)(r : ast.reg.t) : encoding :=
      match o with
      | Succ => succ r
      | _ => defaultUnary o ++ defaultRegEnc r
      end%bit.
  End unary.
```

Encoding application of a binary operator

```
  Module binary.

    Definition encode (o : instruction.binary)(r : ast.reg.AOrM) : encoding :=
      let ambit := match r : ast.reg.t with
                   | reg.A => 0%bit
                   | _ => 1%bit
                   end in
      let opbits := vector match o with
                      | Add => [0;0;0;0;1;0]
                      | Sub => [0;1;0;0;1;1]
                      | SubFrom => [0;0;0;1;1;1]
                      | BAnd => [0;0;0;0;0;0]
                      | BOr => [0;1;0;1;0;1]
                      end%bit in
      (ambit :: opbits)%bit.

  End binary.
```

Encoding the output portion of the instruction

```
  Definition encode (out : instruction.output) : encoding :=
    match out with
    | instruction.constant c => constant c
    | instruction.uapply o r => unary.encode o r
    | instruction.bapply o r => binary.encode o r
    end.

End computation.
```

Encoding an instruction

```
Definition encoding := stream 16.
Definition address (a : N) := nbitstream.of_N (sz := 15) a.

Definition encode (i : ast.instruction.t N) : encoding :=
  match i with
  | ast.instruction.At a => 0 :: address a
  | ast.instruction.C regs out oj =>
      let des := destination.encode regs in
      let com := computation.encode out in
      let jmp := jump.encode oj in
      vector [1;1;1] ++ des ++ com ++ jmp
  end%bit.


Module assemble.

  Definition encodeList (enc : encoding -> string) : block.t N -> list string :=
List.map (enc ∘ encode).
  Definition base2 : block.t N -> string := String.concat "\n" ∘ encodeList
nbitstream.base2.
  Definition hex : block.t N -> string := String.concat "\n" ∘ encodeList
nbitstream.hex.
  Definition bytes : block.t N -> string := String.concat "" ∘ encodeList
nbitstream.bytes.

End assemble.
```

Index

This page has been generated by coqdoc