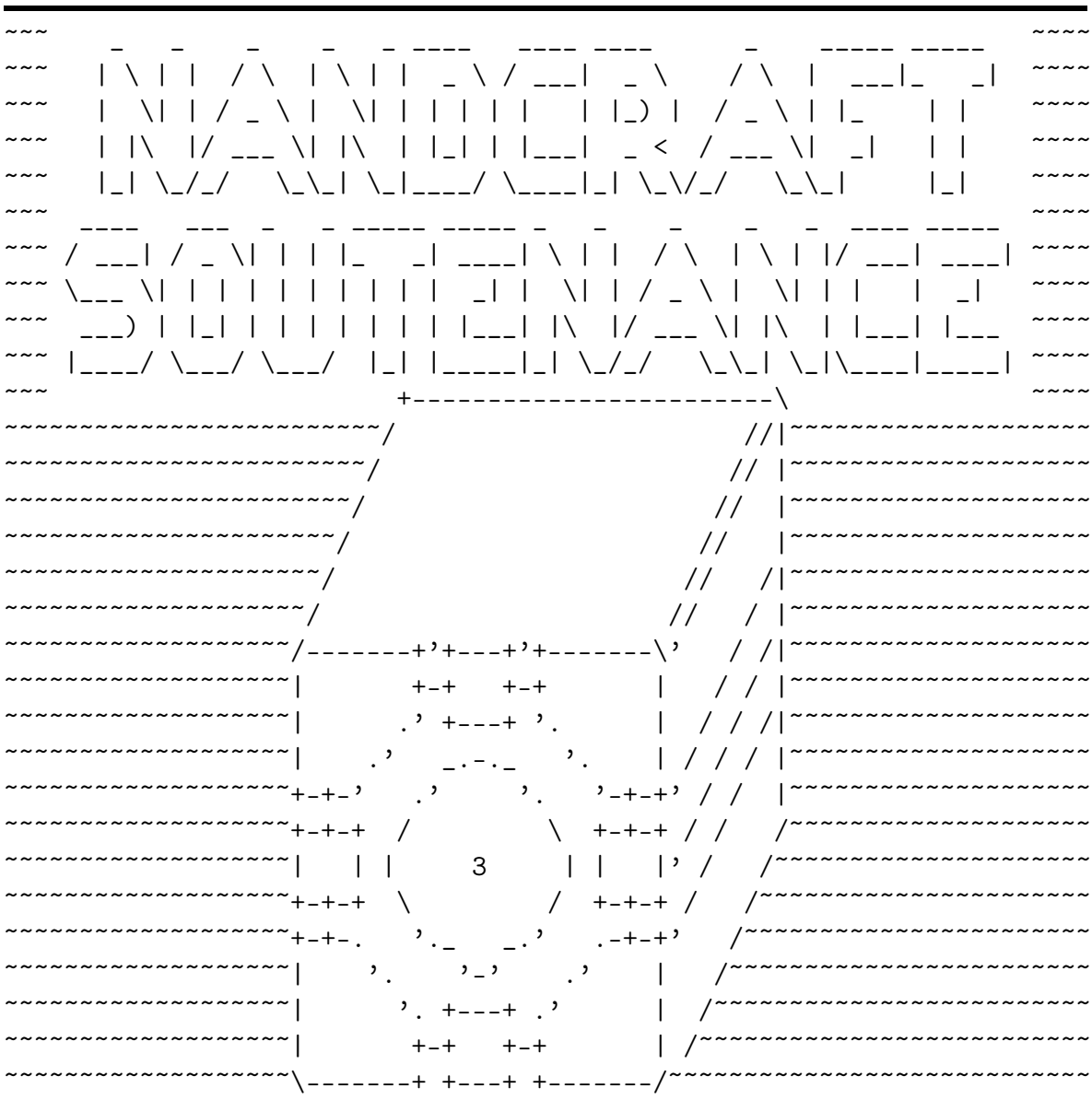
CPU - Cube Processing Unit

audebe_r - halfr
hervot_p - Dettorer
eddequ_n - nass
pruvot_a - Chaf



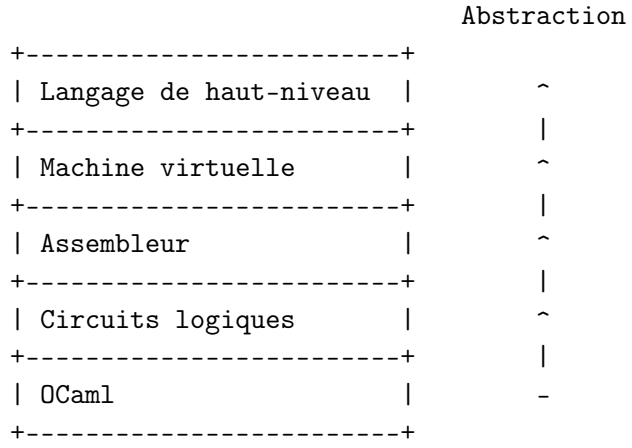
Sommaire

1 Ré-Introduction

1.1 NANDCRAFT

NANDCRAFT est un projet de création d'ordinateur en passant par toutes les abstractions, de la porte logique jusqu'au langage de haut niveau.

Un beau schéma vaut mieux qu'un long discours, résumant chacun de nos sous-projets :



Nous vous présentons dans ce rapport notre avancement sur chacune de ces parties.

2 Circuits logiques

2.1 Logique séquentielle

Toutes (ou presque) les portes que nous avons construites jusqu'ici pour les précédentes soutenances étaient des portes combinatoires. C'est à dire que les sorties dépendaient uniquement des combinanaisons des valeurs passées en entrée de la porte. Ces portes apportent la plupart des fonctions importantes du centre logique de la machine mais sont incapables de maintenir un état.

Or, un ordinateur doit être capable non seulement de calculer des valeurs mais aussi de les sauvegarder pour les réutiliser plus tard. C'est pourquoi nous avons aussi besoins d'élements de mémoire capables de préserver des données sur le temps. Ces élements sont construits à partir de porte séquentielles dont les sorties dépendent non seulement des combinaisons des valaurs passées en entréé de la porte mais aussi des sorties précédentes.

Nous allons donc cette fois construires des portes séquentielles variées (horloges, bascules, registres, ...) afin de compléter notre éventail de portes logiques.

2.2 Le VHDL

Pour se rapprocher de ce qui est utilisé ailleurs en Europe et donc de ce qui pourra nous servir en regardant après la Sup nous avons décidé d'utiliser un intermédiaire entre la description de la porte en CAML et son utilisation réelle. Pour celà nous avons choisi d'utiliser le langage VHDL. Ainsi notre implémentation de porte en CAML pourra génèrer un fichier VHDL qui pourra être utilisé plus simplement par la suite.

VHDL est un langage de description de matériel destiné à représenter le comportement ainsi que l'architecture d'un système électronique numérique. Son nom complet est VHSIC Hardware Description Language.

Le but d'un langage de description matériel tel que le VHDL est de faciliter le développement d'un circuit numérique en fournissant une méthode rigoureuse de description du fonctionnement et de l'architecture du circuit désirée. L'idée est de ne pas avoir à réaliser un composant réel, en utilisant à la place des outils de développement permettant de vérifier le fonctionnement attendu. Ce langage permet en effet d'utiliser des simulateurs, dont le rôle est de tester le fonctionnement décrit par le concepteur.

En VHDL, tout composant (dans le sens logiciel) est décrit sous deux aspects :

- L'interface avec le monde extérieur, décrite dans une section dénommée entity.
- L'implémentation elle-même, décrite dans une section dénommée architecture.

C'est donc la section architecture qui contient la description de la fonction matérielle désirée :

- soit sous forme de description structurelle précise de l'architecture matérielle (les portes logiques à utiliser et leurs interconnexions), ici c'est cette caractéristiques qui est cruiciale dans notre projet, qui je le rappelle, consiste à construire les différentes portes logiques existantes à l'aide de la seule porte NAND.
- soit sous forme de comportement attendu, c'est-à-dire orienté fonctionnel.

Voici le code CAML que nous avons déjà et le code VHDL qui devra être généré:

2.2.1 Le code CAML

```
let my_and =
{
  name = "and";
  inputs = [| "a" |> 1 ; "b" |> 1 |];
  outputs = [| "o" |> 1 |];
  parts = [| (my_nandio [|"a" <@ "a"; "b" <@ "b"|] [|"o" @> "o1"|]);
             (my_notio [|"a" <@ "o1"|] [|"o" @> "o"|]) |]
};;
```

2.2.2 Le code VHDL

```
entity My_Nand is
  port (
    a, b : in  std_logic;
    o  : out std_logic);
end My_Nand;

entity My_Not is
  port (
    a : in  std_logic;
    o  : out std_logic);
end My_Not;

-- Porte AND

entity My_And is
  port (
    a, b : in  std_logic;
    o  : out std_logic);
end My_And;

architecture Logique of My_And is

  component My_Nand
    port (
      a, b : in  std_logic;
      o  : out std_logic);
  end component;

  component My_Not
    port (
      a, b : in  std_logic;
      o    : out std_logic);
  end component;

  signal o1 : std_logic := '0';

begin
  n1 : My_Nand port map (
    a => a,
    b => b,
    o => o1);
  n2 : My_Not port map (
    a => o1,
    o => o);
end Logique;
```

Nous voyons ici que la porte AND est créée grâce à l'association des entrées et sorties de portes NOT et NAND dont on rapelle la définition à chaque fois que l'on en a besoin comme montre l'exemple ci-dessus. De plus nous voyons bien les relations entre le CAML et le VHDL, les liens entres les portes qui servent à définir le AND sont les mêmes grâce au port map ainsi que les entrées et les sorties.

3 Émulateur

Le processeur, ou CPU (pour Central Processing Unit), est le composant dans un ordinateur qui exécute les programmes informatiques. Le processeur récupère les informations dans les différentes mémoires mises à sa disposition, effectue les calculs requis par le programme et peut dialoguer avec les appareils périphériques de l'ordinateur via la technique de `MMIO` (`Memory-maped Input/Output`) qui consiste en une association de la mémoire et des registres des périphériques à des adresses mémoires prédéfinies dans la `RAM`. L'ensemble de ces informations ajoutées à des méta-données transmises par le biais du language intermédiaire sous forme de commentaires puis de sections complètes plus tard renseignent le debuggueur dans la lutte contre les erreurs et surtout contre les utilisateurs incompétents. En effet, notre débuggueur crée dans un premier temps un fichier « .dbg » puis affiche différentes informations relatives à l'état du processeur et du code `haut niveau` à chaque cycle sur la sortie standard.

3.1 Améliorations

3.1.1 Assembleur

L'assembleur est le traducteur qui prend en entrée de l'assembleur pour sortir du binaire. Ce binaire est créé grâce à une table des symboles et était autrefois sous la forme symbolique de 0 et de 1. Notre assembleur sort désormais du vrai binaire. Malheureusement les options offertes par OCaml en terme de binaire sont restreintes, on peut donc sortir un entier soit en 32 bits non formaté soit octet par octet. Nous avons bien évidemment choisi d'écrire les octets deux par deux afin de récupérer du 16 bits que nous traduisons nous même au niveau de l'émulateur.

3.1.2 Émulateur

Notre processeur virtuel est composé d'une Unité arithmétique et logique (en anglais ALU) qui gère les calculs arithmétiques et les tests pour des comparaisons de valeur et de deux registres. Il lit les informations contenues dans la ROM et a un accès en lecture et écriture dans les autres mémoires c'est à dire le registre de pointeur d'instructions, de destination qui récupère les informations à la sortie de l'ALU et permet un stockage « intermédiaire » de valeurs si nécessaire et la RAM représentée par un tableau dont l'accès se fait via le registre de pointeur d'instruction.

La gestion du clavier et de l'écran se fait toujours par MMIO. L'optimisation de l'écran et autres modifications ont été nécessaires pour le bon fonctionnement de l'émulateur du fait de l'architecture sur du 16 bits signées. Une refonte du système d'exploitation a été nécessaire via des astuces dans notre language assembleur mais la limite de 32767 instructions assembleur reste pesante, nous allons travailler dur ce point-ci afin de pouvoir permettre le développement de plus d'applications sur notre machine.

3.2 Le débuggeur

3.2.1 Rappels: Langague assembleur

Nous avons dû perfectionner à la dernière soutenance notre assembleur qui traduit des programmes en langage assembleur vers du langage binaire avec une table des symboles. La syntaxe de notre langage assembleur est la suivante:

- Commentaires : **//commentaire**
- Pseudo-commande : **(label)**
- A-instruction : **@label** ou **@1123**
- C-instruction : **destination=calcul;saut**

Pour plus de facilités, les commentaires peuvent désormais être sur la même ligne qu'une instruction. Tous ces commentaires sont stockés dans un « .dbg » avec le pointeur d'instruction qui leur correspond.

Nous avons également ajouté un nouvel élément dans le language assembleur:

- Opérations sur les éléments principaux marquée par le

Nous pouvons par exemple avoir :

ASM	comentaire
D=M	// D=R13
@3	
A=D-A	// A=R13-3
~A=(*R13)-3	
D=M	// D=M(R13-3)
@ARG	
M=D	// ARG <- M(R13-3)

Le débuggeur récupère les instructions unes à unes dans l'ordre puis analyse le type de l'instruction en cours.

Si le premier bit est à « 0 » c'est une instruction d'adressage, la valeur des 15 bits suivant est donc stockée sous forme d'entier dans le registre A.

Sinon, sont analysés les bits de 4 à 10 pour savoir quoi calculer (les opérations élémentaires effectuées par l'ALU) puis les bits de 11 à 13 pour savoir où placer le resultat de l'opération et enfin les bits 14 à 16 pour connaître le besoin de saut conditionnel (et si oui à quelle condition).

Le débuggeur écrit sur la sortie standard à chaque début de cycle le type de saut, les valeurs des différents registres ainsi que la valeur contenue dans le pointeur de stack et la fonction dans laquelle nous nous trouvons lors de l'instruction en cours.

Récapitulatifs des opcodes A-instruction: forme « @Xxx » :

- premier bit: 0
- 15-bits restants: valeur en système binaire du nombre décimal représenté
- par la A-instruction.

C-instruction: forme « destination=computation;jump » :

- bit 1: 1
- bits 2,3: 1 et 1 (inutilisés)
- bits 4,5,6,7,8,9,10: fonction de l'opération logique à effectuer par l'`ALU`
- bits 11,12,13: fonction des registres dans lesquels placer la valeur calculée
- bits 14,15,16: fonction de l'absence de saut, de saut conditionnel ou inconditionnel

Des améliorations possibles L'idéal serait de générer de l'ELF (Executable and Linked Format) avec un en-tête fixe, des segments contenant des informations portant sur l'exécution et des sections contenant des informations relatives à la résolution des liens entre les fonctions en vue de l'implémentation d'un loader et d'un linker. De nombreuses bibliothèques permettant la manipulation de fichiers au format ELF existent, nous pourrions alors coder la nôtre en OCaml. Avec un peu d'acharnement, nous pourrions même espérer faire tourner GDB sur nos fichiers exécutables afin d'optimiser le debuggage.

3.3 Pour la soutenance 4 : Entrées/Sorties

La gestion de l'écran initialement prévue pour la quatrième soutenance est finie ainsi que celle du clavier. D'autres entrées/sorties telles que le son sont en cours de réalisation. Le pong en assembleur de la précédente soutenance était un exemple. Maintenant l'implémentation des évenements du clavier laisse encore un peu à désirer et sera totalement fonctionnel pour la dernière soutenance.

La création et implémentation d'un loader et d'un linker passera par l'aboutissement de conception de notre format d'exécutable. Elle est donc plus ou moins en cours de réalisation. D'autre part, le désassembleur est également en cours de réalisation et ne sera pas réellement difficile à terminer du fait de la connaissance préalable des opcodes. Néanmoins il faut que la table des symboles soit implémentée à l'intérieur du format exécutable afin de pouvoir espérer retrouver une vraie version assembleur à partir du code machine.

4 Machine virtuelle

La machine virtuelle est un outil puissant qui intervient pendant la compilation d'un programme. Elle travail sur un langage intermédiaire situé entre le haut niveau et l'assembleur, ce qui permet beaucoup d'optimisations ainsi qu'un debogage plus efficace.

4.1 Rappels

4.1.1 Le Bytecode

Le langage intermédiaire utilisé par la machine virtuelle fonctionne sur le principe de pile, c'est à dire que les instructions qui forment ce langage manipulent la mémoire sous forme d'une pile maitresse et de segments de mémoire.

Les fonctions *PUSH* et *POP* déplacent des valeurs entre la pile et les segments de mémoire. Il existe huit segments de mémoire : Static, Argument, Local, This, That, Pointer, Temp et Constant.

Les fonctions arithmétiques telles que l'addition, la soustraction, les comparaisons ou les opérations logiques manipulent uniquement les valeurs en haut de la pile. *ADD* supprimera les deux valeurs les plus hautes de la pile et y mettra le résultat de leur addition, *EQ* y placera un booléen représentant si oui ou non les deux valeurs sont égales, *NOT* remplacera le booléen en haut de la pile par son complément, etc...

Enfin, des fonctions de contrôle de flux permettent de définire des fonctions, de les appeler et de créer des branchements conditionnels.

4.1.2 Le traducteur

La fonction principale de la machine virtuelle est donc de traduire un programme écrit dans ce langage intermédiaire en langage assembleur, elle se base pour cela sur des protocoles précis permettant de traduire chaque instructions en son équivalent assembleur.

4.2 L'émulateur

Vient avec la machine virtuelle un interprêteur, dont le but est de simuler directement l'exécution du Bytecode. Il représente pour cela la mémoire sous forme d'un tableau Ocaml et simule l'exécution des instructions du Bytecode sur cette représentation.

C'était donc la totalité des fonctions déjà prises en charge par le traducteur qu'il fallait réimplémenter en Ocaml. À la dernière soutenance, les fonctions arithmétiques et de transfert de mémoire (*PUSH* et *POP*) étaient fonctionnelles. Les fonctions de contrôle de flux d'execution sont maintenant implémentées, et l'émulateur peut servir à vérifier que le code a bien été traduit du haut niveau au Bytecode et fait bien ce qu'il est censé faire.

4.3 Les optimisations

Jusqu'à maintenant, l'utilité de la machine virtuelle restait tout de même limité, mais les outils déployés autour de celle-ci permettent maintenant d'exploiter toute sa puissance, à savoir optimiser le code.

4.3.1 Push et Pop

Parfois, une séquence d'instruction Bytecode fait transiter des valeurs sur la pile alors que l'opération attendu est un simple transfert d'un segment de mémoire à un autre. En bytecode, c'est le cas quand une instruction *POP* suit immédiatement un *PUSH*, la valeurs que l'on manipule quitte son segment d'origine, s'arrête sur la pile et est transférée de nouveau vers un autre segment. Une optimisation est donc de détecter cette suite d'instruction et de la traduire d'un bloc, sans passer par la pile. Cette optimisation nous fait passer de vingt-deux instructions assembleur (*PUSH* et *POP* séparés) à quatorze, ce qui, sur un programme entier, est une réduction considérable car ce cas de figure se répète assez souvent.

4.3.2 Factorisation du code assembleur

Les fonctions de comparaison *GT*, *LT* et *EQ* prennent environ dix-huit instructions assembleurs, car elle doivent se faire via les instructions de saut de code de l'assembleur, seul outil permettant la comparaison de valeurs. Ce code n'a en fait pas besoin d'être réécris à chaque fois que l'on souhaite traduire une comparaison. En effet une optimisation possible est, à la rencontre d'une fonction de comparaison, de sauvegarder le numéro de l'instruction courrante dans un registre temporaire, puis de sauter à un endroit du code écrit au préalable (lors de la phase d'initialisation) qui va alors récupérer les valeurs en haut de la pile, les comparer, mettre le bon résultat en haut de la pile, puis retourner à l'instruction dont le numéro se trouve dans le registre. Nous gagnons ainsi encore une dizaine d'instruction assembleur à chaque comparaison, ce qui est considérable.

Plus compliqué maitenant, les fonctions d'appel et de retour de fonction généraient une quantitée colossale d'instruction assembleur. Dans le cas de *return*, la procédure est la même dans tous les cas, il n'était donc pas excessivement compliqué de garder le bloc de code au début du programme et d'y faire appel à chaque fois que l'on en avait besoin. Le vrai tour de force était d'optimiser l'instruction *call*. Celle-ci a besoin de trois informations n'étant quasiment jamais les mêmes : le nom de la fonction appelée, le nombre d'arguments, et une addresse de retour. Le problème étant que nous disposons de trois registres temporaires pour y mettre ces valeurs, mais que l'un d'eux est déjà utilisé lors de la procédure d'appel. Il a donc fallu ruser et garder une des informations dans le registre D, un registre processeur à très court terme car constament utilisé. L'adresse de retour se trouve être la seule information qui, après quelques ajustements, peut être utilisée avec une telle contrainte.

Grâce à ces optimisation, nous avons par exemple, sur un programme à la base de 31 000 instructions assembleur, économisé plus de 6 000 instructions.

4.4 Le debogage

Étendons-nous un peu sur le debogage présentée dans la partie assembleur. L'assembleur peut à tout moment connaitre la valeur contenue dans ses registres, seulement quand on cherche une erreur, ces valeurs peuvent se révéler complêtement obscure. C'est pourquoi nous avons ajouté au traducteur une fonctionnalité lui permettant de donner à l'assembleur la signification de ces valeurs.

4.4.1 Informations sur les fonctions

Tout d'abord, chaque portion de code correspondant à une instruction en bytecode est précédée d'un commentaire renseignant le nom de cette instruction, afin qu'à tout endroit du code assembleur l'on puisse savoir dans quelle fonction nous somme. L'information sera précédée d'un croisillon. Exemple : la traduction en assembleur de l'instruction *function Main.fibo 2* sera précédée de la ligne **#function Main.fibo 2**

4.4.2 Informations sur les registres

Ensuite, une information encore plus utile consiste à renseigner régulièrement la signification du contenu des registres du processeur, c'est à dire par exemple pendant un séquence correspondant à un *PUSH*, de préciser que la valeur actuellement contenue dans le registre *D* est en fait la valeur que l'on veut push. C'est cette fois-ci un tilde qui introduira la ligne donnant l'information. Exemple : la ligne correspondant à l'exemple ci-dessus sera **∼D=push_value**

4.5 Pour la soutenance 4

4.5.1 Bonus : la précompilation

Au finale, la machine virtuelle saura réaliser une optimisation beaucoup plus complexe, mais aussi beaucoup plus intéressante : la précompilation. Le principe est ici d'utiliser l'émulateur de machine virtuelle afin de simuler l'execution du code j'usqu'à ateindre un point qui requiert une information indisponible pour le moment, par exemple une zone du clavier. L'émulateur s'arrêtera alors à cet endroit et génèrera une séquence en assembleur permettant de charger rapidement l'état actuel de la mémoire, le traducteur n'aura alors qu'à traduire les instructions restantes et le programme s'initialisera extrêmement plus vite.

Un exemple pratique : en soutenance 2, nous avions généré avec succès un programme affichant la phrase « Hello World! » du langage de haut niveau jusqu'à l'assembleur, cependant ce programme necessitait de charger des bibliothèque de police d'écriture complexes, et le code assembleur généré était bien trop long pour être exécuté correctement. Ce code peut être précompilé tant qu'il est déterministe, ce qui est le cas du chargement des bibliothèques, la machine virtuelle va donc simuler ce chargement pendant la compilation et restituer un code beaucoup plus court initialisant la mémoire à l'état où les bibliothèques sont chargées. Notre machine n'aura alors qu'à afficher la phrase à l'écran, les bibliothèques étant déjà à sa disposition.

5 Système d'exploitation

Nous avons un compilateur complet, un assembleur et une machine virtuelle. Il ne nous reste plus qu'une seule brique pour pouvoir commencer à écire de vraies applications : un **système d'exploitation**.

Le système d'exploitation doit faire le lien entre la couche matérielle et logicielle de l'ordinateur. Il doit donner au programmeur un moyen simple de développer des applications sans se soucier du fonctionnement de la machine.

Dans le cadre de notre projet il fournit surtout un ensemble de bibliothèque qui permette de s'abstraire du fonctionnement concret de la machine. Par exemple on multiplie deux entiers `x` et `y` en appelant `Math.multiply(x, y)` car la multiplication n'est pas gérée au niveau matériel. Si dans le futur on utilisait un composant dédié on pourrait juste remplacer l'implémentation de cette méthode par un appel à lui. Ainsi le code utilisant cette biliothèque n'aurait pas à être changé mais pourrait quand même bénéficier du gain de vitesse.

5.1 Les fonctionnalités du système d'exploitation CraftOS

- Primitives de dessin sur l'écran
- Affichage de texte à l'écran
- Gestion de l'allocation et de la libération de la mémoire

5.2 Algorithmes utilisés

Dans cette section on va présenter quelques algorithmes imlémentés dans le CraftOS.

5.2.1 Multiplication

L'ordinateur nandcraft ne dispose pas d'un circuit dédié à la multiplication, on doit donc implémenter un algorithme utilisant les opérations élémentaires dont nous disposons pour accomplier ma même tâche.

On utilise la méthode de la multiplication égyptienne pour calculer le produit de deux nombres.

(Pour rappel le code donné ici est écrit en Craft, le langage pour lequel on a déjà écrit un compilateur et un assembleur)

```
1  function int multiply(int x, int y) {
2    var int result;
3
4    let result = Math.multiplyAbs(x, y);
5    if(x < 0) {
6      let result = -result;
7    }
8    if(y < 0) {
9      let result = -result;
10   }
11
12   return result;
13 }
```

```
14
15 function int multiplyAbs(int x, int y) {
16   var int i, shiftedX, result;
17
18   let x = Math.abs(x);
19   let y = Math.abs(y);
20
21   let i = 0;
22   let shiftedX = x;
23   let result = 0;
24   while(i < 16) {
25     if(Math.bit(y, i) = 1) {
26       let result = result + shiftedX;
27     }
28     let shiftedX = shiftedX + shiftedX;
29     let i = i + 1;
30   }
31
32   return result;
33 }
```

Elle s'effectue en $O(n)$ additions pour des nombres de n bits.

5.2.2 Division euclidienne

La méthode naïve pour diviser qui consiste à retrancher successivement le diviseur au dividende tant quand que le résultat est supérieur à zéro. Cependant on voit bien que la complexité de cet algorithme croit avec la taille du quotient.

On l'optimise en retranchant le plus grand multiple de deux du diviseur à chaque fois. Il a été implémenté ici de manière récursive, mais on aurait aussi pu le faire de manière itérative.

```
1  function int divideRec(int x, int y) {
2    var int q;
3
4    if(y < 0) {
5      return 0;
6    }
7    if(y > x) {
8      return 0;
9    }
10   let q = Math.divideRec(x, y + y);
11   if((x - ((q + q) * y)) < y) {
12     return q + q;
13   }
14   else {
15     return q + q + 1;
16   }
17 }
```

On ne parlera pas des opérations avec des nombres à virgule flottante car on ne souhaite pas les gérer dans l'OS de base. Ils pourront cependant être utilisés *via* une bibliothèque supplémentaire.

5.2.3 Racine carrée

Il existe plusiseurs moyens pour calculer efficacement la racine carré d'un nombre, par exemple en utilisant la méthode de Newton ou un développement limité. Pour le CraftOS on utilisera un algorithme simple qui un se base sur les proriétés de la racine carré $y = \sqrt{x}$:

- c'est une fonction croissante
- son inverse $x = y^2$ se base sur la multiplication (ce dont nous disposons déjà)

```
function int sqrt(int x) {
  var int left, right, mid, midSq, result;

  if(x < 0) {
    do Sys.error(4);
  }

  let left = 0;
  let right = Math.min(x, 181);
  let result = 0;
  while(left < (right + 1)) {
    let mid = (left + right) / 2;
    let midSq = mid * mid;
    if(midSq > x) {
      let right = mid - 1;
    }
    else {
      let left = mid + 1;
      let result = mid;
    }
  }

  return result;
}
```

Cet algorithme s'exécute en un temps au pire de $n/2$, soit une complexité de $O(n)$ opérations arithmétiques.

5.2.4 Traduction/Interprétation ASCII

Les applications doivent souvent manipuler des chaines de caractères, que ce soit pour les afficher ou car l'utilisateur lui en donne *via* le clavier.

Elles doivent également gérer la traduction d'un caractère ASCII vers un entier (par exemple dans un jeu quand l'utilisateur entre le nombre d'ennemis) et inversement (quand il faut afficher le score du joueur).

Pour cela certains processeurs possèdent des instructions dédiées au calcul arithmétiques sous forme BCD[1] ou ASCII[2]. Le processeur `nandcraft` n'en dispose pas, il faut donc recoder des fonctions pour passer d'un format à l'autre.

Caractères	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
Code ASCII	48	49	50	51	52	53	54	55	56	57

[1] Binary Coded Decimal
[2] American Standart for Information Interchange

```
method void setInt(int j) {
  var int d;
  var boolean sign;

  let sLength = 0;
  let sign = false;
  if(j < 0) {
    let sign = true;
    let j = -j;
  }
  if(j = 0) {
    do appendChar(48);
  }

  while(j > 0) {
    let d = j - ((j / 10) * 10);
    do appendChar(d + 48);
    let j = j / 10;
  }

  if(sign) {
    do appendChar(45);
  }

  return reverse();
}
```

```
method int intValue() {
  var int value, i;
  var boolean sign;

  let i = 0;
  let value = 0;
  let sign = false;
  if((sLength > 0) & (string[0] = 45)) {
    let sign = true;
    let i = i + 1;
  }
  while(true) {
    if(i = sLength) {
      if(sign) {
        let value = -value;
      }
      return value;
    }
    if((string[i] > 47) & (string[i] < 58)) {
      let value = (10 * value) + (string[i] - 48);
    }
    else {
      return value;
    }
    let i = i + 1;
  }
}
```

5.2.5 Gestion de la mémoire

Les programmes déclarent et allouent des variables, la plupart sont scalaires (un entier, un caractère) ou bien leur taille est connue. Ils peuvent également utiliser des types de données

dont la taille n'est pas connue à la compilation ou bien créer des objets à volée pendant l'exécution.

Les systèmes d'exploitation utilisent différentes techniques pour gérer la mémoire allouée (`Memory.alloc`) et libérée (`Memory.free`) dynamiquement.

Un bon algorithme d'allocation mémoire doit avoir les caractéristiques suivantes :

- minimise la fragmentation mémoire, c'est à dire qu'on doit perdre le moins de place possible ;
- pouvoir trouver le plus rapidement un bloc de mémoire libre disponible

```
function Array alloc(int size) {
  var Array currentPointer;
  var Array result;
  var Array temp;

  if(size < 1) {
    do Sys.error(5);
  }

  let currentPointer = freeListHead[1];
  while(true) {
    if(currentPointer = freeListTail) {
      do Sys.error(6);
    }
    if(currentPointer[0] > (size + 2)) {
      let temp = (currentPointer[0] - (size + 1)) + currentPointer;
      let temp[0] = size + 1;
      let result = temp + 1;
      let currentPointer[0] = currentPointer[0] - (size + 1);
      return result;
    }
    let currentPointer = currentPointer[1];
  }
}
```

```
function void deAlloc(Array o) {
  var Array newNode;
  var Array previous;
  var Array current;

  let newNode = o - 1;
  let previous = freeListHead;
  let current = previous[1];

  while(true) {
    if(current > newNode) {
      let newNode[1] = current;
      let previous[1] = newNode;
      do Memory.mergeBlocks(newNode, current);
      do Memory.mergeBlocks(previous, newNode);
      return;
    }
    let previous = previous[1];
    let current = current[1];
  }
}
```

5.3 Spécification

On a implémenté les bibliothèques suivantes :

- Math
  - `function void init()` (utilisée en interne)
  - `function int abs(int x)` retourne la valeur absolue de `x`
  - `function int multiply(int x, int y)` retourne le prduit de `x` et `y`
  - `function int divide(int x, int y)` retourne le quotient de la division euclidienne de `x` par `y`
  - `function int min(int x, int y)` retourne le minimum de `x` et `y`
  - `function int max(int x, int y)` retourne le maximum de `x` et `y`
  - `function int sqrt(int x)` retourne la partie entière de la racine carrée de `x`
- String
  - `constructor String new(int maxLength)` construit une chaine vide (de taille 0) qui peut contenir jusqu'à `maxLength` caractèrs
  - `method void dispose()` libère cette chaine
  - `method int length()` retourne la taille de cette chaine
  - `method char charAt(int i)` retourne le caractère à la position `j` de la chaine
  - `method void setCharAt(int i, char c)` remplce le caractère à la position `i` de la chaine par `c`
  - `method String appendChar(char c)` ajoute `c` à la chaine et la renvoit
  - `method void eraseLastChar()` efface le dernier caractère de la chaine
  - `method int intValue()` retourne la valeur entière contenue dans la chaine
  - `method setInt(int i)` place dans la chaine la représentation `ASCII` de l'entier `j`
  - `method char backSpace()` retourne le caractère `backSpace`
  - `method char doubleQuote()` retourne le caractère = »=
  - `method char newLine()` retourne le caractère `newline`
- Array
  - `constructor Array new(int size)` construit un tableau de la taille `size`
  - `method void dispose()` libère la mémoire occupée par l'objet
- Output
  - `function void init()` (utilisée en interne)
  - `function void moveCursor(int i, int j)` déplace le curseur à la position `(i, j)`
  - `function void printChar(char c)` affiche la caractère `c`

  - `function void printString(String s)` affiche la chaine `s`
  - `function void printInt(int i)` affiche l'entier `i`
  - `function void printLn()` effectue un retour à la ligne
  - `function void backSpace()` déplace le curseur d'un caractère vers l'arrière

- Screen
  - `function void init()` (utilisée en interne)
  - `function void clearScreen()` efface l'écran
  - `function void setColor(boolean b)` spécifie la couleur des opérations `drawXXX` (`false` = blanc, `true` noir)
  - `function void drawPixel(int x, int y) dessigne le pixel =(x, y)`
  - =function void drawLine(int x1, int y1, int x2, int y2) dessine une ligne du pixel (x1, y1) au pixel (x2, y2)
  - `function void drawRectangle(int x1, int y1, int x2, int y2) dessine un rectangle plein de =(x1, y1)` à `(x2, y2)`
  - `function void drawCircle(int x, int y, int r) dessine un cercle de rayon =r` (`r` < 181) autour de `(x, y)`

- Keyboard
  - `function void init()` (utilisée en interne)
  - `function char keyPressed()` retourne le caractère pressé sur le clavier, 0 si aucun ne l'est actuellement
  - `function char readChar()` attend qu'une touche soit pressée et relachée, affiche le caractère à l'écran puis retourne sa valeur
  - `function String readLine(String message)` affiche le message `message` à l'écran puis attends que l'utilisateur entre une chaine de caractère au clavier terminée par la touche `Entrée`, affiche cette chaine à l'écran puis la retourne.
  - `function int readInt(String message)` affiche le message `message` à l'écran puis attends que l'utilisateur entre un entier au clavier terminé par la touche `Entrée`, l'affiche à l'écran puis le retourne

- Memory
  - `function void init()` (utilisée en interne)
  - `function int peek(int address)` retourne la valeur stockée à l'emplacement mémoire `address`
  - `function void poke(int address, int value)` met la valuer `value` à l'adresse `address`
  - `function Array alloc(int size)` trouve et alloue un bloc de mémoire libre du tas de la taille `size`, retourne son adresse de base

  - `function void deAlloc(Array o)` libère la mémoire occupée par l'object `o` et la rend disponible

- Sys
  - `function void init()` (utilisée en interne, appelle toutes les autres fonctions `init()`)
  - `function void halt()` arrete le programme
  - `function void error(int errorCode)` affiche le message d'erreur à l'écran puis quitte
  - `function void wait(int duration)` attends approximativment `duration` millisecondes et retourne

5.4 Ce qu'il reste à faire - Soutenance 4

L'objectif pour la dernière soutenance est de coder un jeu complet utilisant toutes les briques que nous avons créées.

6 Le mot de la fin

```
tl;dr On est toujours très très heureux et détendus
```