ByteFrost Development Log - Addressing Modes Proposal (v2)

June 7 - July 12, 2025

Overview

In this development log, we shall go over the new addressing modes proposal. We shall first go over the new instructions, their semantics, and example usage, with motivations for the new instructions and how they improve the ISA (with comparisons to how equivalent operations would be done in the current ISA). We shall then discuss any instructions that can be removed or deprecated. Lastly, we shall cover the required hardware changes in terms of updates to control signal assignments and new combinational logic circuits.

ISA Updates

New Instruction Semantics

There are `6` new instructions and `2` instructions with updated semantics; their names and semantics are shown below.

Instruction	Semantics	Example Usage	New or Modified?
`JSR`	`Push PC to the stack (SP--; *SP = PC[H]; SP--; *SP = PC[L]), jump to DP (PC = DP)`	`JSR`	Modified
`RTS`	`Pop return address from the stack (SP++; DHPC = *SP; SP--; PC[L] = *SP, PC[H] = DHPC; SP++; SP++;)`	`RTS`	Modified
`MAG`	`Rd = ARSrc[L/H]`	`MAG R1, %SP[H]`	New
`LDW`	`Rd = *(ARSrc + Imm)`	`LDW R2, 16(%BP)`	New
`SDW`	`*(ARSrc + Imm) = Rs`	`SDW R3, -4(%DP)`	New
`LDA`	`ARDest[L/H] = Imm`	`LDA %DP[L], #0x54`	New
`MGA`	`ARDest[L/H] = Rs1`	`MGA %BP[L], R2`	New
`MAA`	`ARDest = ARSrc + Imm`	`MAA %SP, %SP, #-1`	New

`JSR` Discussion

In the proposal, the `JSR` instruction's operands and semantics are modified.

`JSR`'s current semantics

In the current ISA, the `JSR` instruction has one immediate operand (`JSR Imm`) and the following semantics:

1. Push the low byte of the `PC` to the stack.
    1. `*SP = PC[L]` (Push `PC[L]` to the stack)
    2. `SP++`
2. Set the `PC` to the target address.
    1. `PC[L] = Imm` (Set `PC[L]` to the low byte of the target address)
    2. `PC[H] = DHPC` (Set `PC[H]` to `DHPC` which should contain the high byte of the target address)

This has the following issues:

1. `JSR` doesn't push `PC[H]` to the stack, meaning it must be done by a prior instruction.
2. `JSR` doesn't allow specifying the high byte of the target address, meaning it must be done by a prior instruction.
3. `JSR` sets `PC[L]` to an immediate (i.e., function pointers are not possible).
4. `JSR` is implemented for an empty-ascending stack instead of a full-descending stack.

Currently, the `CALL :label` assembly instruction is implemented in the following way:

1. Push `PC[H]` to the stack.
2. Set `DHPC` to the high byte of the target address.
3. `JSR low_byte_of_target_address`.

- This sets `PC[H]` to `DHPC` and `PC[L]` to `low_byte_of_target_address`, since the active low PC Load Hi signal is active (low) for the `JSR`, `RTS`, and Branch Absolute instructions.
    - **Note:** The PC Load Hi signal is deprecated in this proposal. It is replaced by the **AR Data Bus Load Enable** LUT's `PC[L] + PC[H] = DHPC Load Enable` signal.

```
//  1.  Push PC[H] to the stack.
LDR R0, #PC_HI_VALUE (known to assembler at compile time)
PUSH R0

//  2. Set DHPC to the high byte of the target address.
LSP %DHPC, #TARGET_ADDRESS_HI

//  3. JSR
JSR #TARGET_ADDRESS_LO
```

As we can see, the `CALL` assembly instruction is implemented using `4` ISA instructions and has additional unwanted semantics, such as overwriting `R0` (due to the lack of a `PUSH Imm` ISA instruction).

`JSR`'s new semantics

In the proposal, `JSR` has the following semantics:

1. Push the `PC` to the stack.
   1. `SP--`
   2. `*SP = PC[H]`
   3. `SP--`
   4. `*SP = PC[L]`
2. Jump to the address in `DP`.
   1. `PC = DP`

This implementation resolves issues `1`, `3`, and `4`, but doesn't fix issue `2`.

1. **Issue 1:** `JSR` doesn't push `PC[H]` to the stack, meaning it must be done by a prior instruction.

- **Resolution:** `JSR` now pushes the entire `PC` to the stack, including `PC[H]`, so that it doesn't need to be done by a prior instruction.

2. **Issue 2:** `JSR` doesn't allow specifying the high byte of the target address, meaning it must be done by a prior instruction.

- **Worse:** `JSR` now doesn't allow specifying the high nor low byte of the target address, meaning they must both be set by prior instructions. The target address is always assumed to be in the `DP` Address Register.

3. **Issue 3:** `JSR` sets `PC[L]` to an immediate (i.e., function pointers are not possible).

- **Resolution:** `JSR` sets the `PC` to `DP`, a register; hence, the target address may change during runtime (allowing for function pointer semantics).

4. **Issue 4:** `JSR` is implemented for an empty-ascending stack instead of a full-descending stack.

- **Resolution:** `JSR` is now implemented for a full-descending stack.

Regarding Issue 2: We can observe that `JSR` now doesn't take any operands for specifying the target address, and assumes that the target address is already loaded into `DP`. That means that calling an arbitrary address takes `3` instructions (`DP[L] = target_address_low`, `DP[H] = target_address_hi`, `JSR`). This is an improvement over the `4` instruction implementation of `CALL` and has no other unusual semantics. `JSR` can be modified to have an additional Immediate or GR operand to specify parts of the address, but the first option restricts the function pointer semantics (issue `3`) and the second requires loading the relevant target address byte to a GR. I think that the proposed implementation is a clean way to handle this with a smaller extra instruction cost than the current implementation.

Now, the `CALL :label` assembly instruction can be implemented as follows:

1. Set `DP = TARGET_ADDRESS`.
2. `JSR`.

```
//  1.  Set DP to the target address.
LDA %DP[L], #TARGET_ADDRESS_LOW
LDA %DP[H], #TARGET_ADDRESS_HI
```

```
//  2.  JSR.
JSR
```

Note: The `CALL :label` assembly instruction can be used when the function target is known at compile-time. For function pointer semantics, `JSR` should be used directly, i.e.:

```
//  1.  Set DP to dynamically-determined address, e.g. from GRs
MGA %DP[L], R0
MGA %DP[H], R1

//  2.  JSR.
JSR
```

`RTS` Discussion

In the proposal, the `RTS` instruction's operands and semantics are modified.

`RTS`' current semantics

In the current ISA, the `RTS` instruction has the following semantics:

1. Pop the return address from the stack and load it to the `PC`.
    1. `SP--` (`SP` now points at `ret[L]`)
    2. `SP--` (`SP` now points at `ret[H]`)
    3. `DHPC = *SP` (`DHPC` is now set to `ret[H]`)
    4. `SP++` (`SP` now points at `ret[L]`)
    5. `PC[L] = *SP, PC[H] = DHPC` (`PC[L]` is now set to `ret[L]`, `PC[H]` is now set to `DHPC`)
    6. `SP--` (`SP` now points at `ret[H]` (empty byte after end of stack))

This has the following issue:

1. `RTS` is implemented for an empty-ascending stack instead of a full-descending stack.

Note: This implementation of `RTS` uses the deprecated Special Register hardware and semantics; e.g., to specify that the Special Register to load using the *Load Special Register* control signal, bits `7:6` of `RTS`' instruction string MUST be set to `00` which is mapped to the `DHPC` by the Load Special Pointer hardware implementation.

`RTS`' new semantics

In the proposal, `RTS` has the following semantics:

1. Pop the return address from the stack and load it to the `PC`. (same)
    1. `SP++, DHPC = *SP`. (`SP` points at `ret[H]` and `DHPC` is set to `ret[H]`)
    2. `SP--, PC[L] = *SP, PC[H] = DHPC`. (`SP` points at `ret[L]` and `PC[L]` is set to `ret[L]` and `PC[H]` is set to `DHPC`, which has `ret[H]`)

3. `SP++, SP++`. (`SP` now points at the byte above `ret[H]` (new last byte of the stack)) **Note:** `DHPC` is loaded first since when writing to `PC[L]`, `PC[H]` reads from `DHPC`; hence, `DHPC` must have the correct value (`ret[H]`) BEFORE `PC[L]` reads `ret[L]`.

This implementation resolves issue `1`.

1. **Issue 1:** `RTS` is implemented for an empty-ascending stack instead of a full-descending stack.

- **Resolution:** `RTS` is now implemented for a full-descending stack.

`MAG` Discussion

The proposal introduces the `MAG` instruction, which has the following semantics:

1. `Rd = ARSrc[L/H]`.

This instruction provides the following benefits:

1. **Issue 1:** In the current ISA, there is no way to write the value of an Address Register high / low byte to a General Purpose Register.

- **Resolution:** This is now possible using the `MAG` instruction.

`LDW` Discussion

The proposal introduces the `LDW` instruction, which has the following semantics:

1. `Rd = *(ARSrc + Imm)`.

This instruction provides the following benefits:

1. **Issue 1:** Handling 16-bit addresses is complex; the current ISA mitigates some of the complexity by setting the high byte of an address in the Special Registers `DP[H]`, `SP[H]`, and `DHPC`. Then, instructions such as `LMA`, `SMA`, `LMR`, `SMR`, and Branch Absolute set the bottom byte and perform memory loads / stores / update the PC in one instruction. `LMA`, `SMA`, `LMR`, and `SMR` can only use the `DP` to store the address and overwrite the `DP[L]` byte, however.

- **Improvement:** Now handling addresses might be slightly simplified, since we can use dedicated Address Registers to hold addresses and avoid updating them when we have spatial (of +127/-128 byte offsets) or temporal locality in accessing memory (specifically in terms of loads / stores). `LDW` is better than `LMA` and `LMR` since it allows using additional Address Registers besides the `DP` and doesn't overwrite the `DP`'s low byte.
  - The spatial locality benefit is the same as for `LMA` and `LMR` (256 byte range), but it is now centered around the `DP` address instead of the page specified by `DP[H]`. This may be equivalent, better, or worse, depending on the situation. I think this is better for situations that require known offsets from a base address (e.g., accessing struct fields or offsets in a stack frame), since then the offset is the same every time, whereas the offset would change if the struct isn't stored at the beginning of a page (i.e., at some address `{DP[H], 00}`).

This instruction allows deprecating `LMA` and `LMR`.

`SDW` Discussion

The proposal introduces the `SDW` instruction, which has the following semantics:

1. `*(ARSrc + Imm) = Rs`.

This instruction provides the following benefits:

1. **Issue 1:** The current memory store instructions (`SMA` and `SMR`) can only use the `DP[H]` as the high byte of the address, and overwrite `DP[L]`.

- **Improvement:** `SDW` allows using any ISA-addressable AR as a base address (i.e., `PC`, `DP`, `SP`, or `BP`), does not overwrite the used AR, and maintains the same amount of spatial locality now centered at the address in the used AR (better for known offsets when the address is arbitrary, such as in struct field accesses).

This instruction allows deprecating `SMA` and `SMR`.

`LDA` Discussion

The proposal introduces the `LDA` instruction, which has the following semantics:

1. `ARDest[L/H] = Imm`.

This instruction provides the following benefits:

1. **Issue 1:** The current ISA allows only setting the high byte of some ARs (`DHPC`, `DP[H]`, and `SP[H]`) to an immediate. There is no way to set the low byte of an AR to an immediate (except for the unusual semantics of `LMA` and `SMA` that set `DP[L]` to their immediate operand).

- **Resolution:** `LDA` allows setting any byte of an ISA-addressable AR (`DHPC`, `PC[L] (and PC[H] = DHPC)`, `DP[H/L]`, `SP[H/L]`, and `BP[H/L]`) to an immediate.

This instruction allows removing `LSP`.

`MGA` Discussion

The proposal introduces the `MGA` instruction, which has the following semantics:

1. `ARDest[L/H] = Rs1`.

This instruction provides the following benefits:

1. **Issue 1:** The current ISA does not allow setting any AR byte to the value of a GR.

- **Resolution:** `MGA` allows setting any byte of an ISA-addressable AR (`DHPC`, `PC[L] (and PC[H] = DHPC)`, `DP[H/L]`, `SP[H/L]`, and `BP[H/L]`) to a GR.

`MAA` Discussion

The proposal introduces the `MAA` instruction, which has the following semantics:

1. `ARDest = ARSrc + Imm`.

This instruction provides the following benefits:

1. **Issue 1:** The current ISA does not allow setting an AR to another AR's value.

- **Resolution:** `MAA` allows setting any ISA-addressable AR (`PC`, `DP`, `SP`, and `BP`) to another ISA-addressable AR (i.e., with `ARDest = ARSrc + 0`)

2. **Issue 2:** The current ISA does not provide an easy way for performing 16-bit pointer arithmetic.

- **Resolution:** `MAA` allows for limited 16-bit pointer arithmetic by allowing adding an immediate offset to an ISA-addressable AR's value (the limit being that the immediate is signed 8-bit (in the range `[-128, 127]`)).

2. **Issue 2:** The current ISA makes implementing an x86-style calling convention very difficult if not impossible. Some of the reasons for this are that an additional AR (`BP`) is needed, setting `SP` to `BP` and vice versa is needed, and accessing the stack from the `BP` with an offset is needed to access variables (taken care of by `LDW` and `SDW`).

- **Resolution:** The proposal introduces the new `BP` AR and `MAA` allows for easily setting one AR to another.

Example Code

Here are a few example programs / program snippets.

1. Stack Calling Convention

Let us attempt to implement the following calling convention (copied largely from the 32-bit x86 calling convention, see https://aaronbloomfield.github.io/pdr/book/x86-32bit-ccc-chapter.pdf):

1. Caller (Before Callee Runs)
    1. Push caller-saved registers (`R0` - `R3`).
    - **Note:** ARs are NOT saved. The `SP` and `BP` are always for the current function's stack frame, the `PC` points at the next instruction, and the `DP` is overridable; the caller should assume its value is not preserved after a call to a function.
        - In fact, the `DP` is used to store the address of the return value, so that the `DP` is always overwritten (for functions that return `void`, either the `DP` may have an arbitrary value (e.g., `NULL`), or it could also just not be changed; however, it's likely that the `DP` will be overwritten by the callee anyway for loads / stores and other pointer usage).
    2. Push callee arguments (last in the order first)
    - E.g., for a function call `f(3, 4, 5)`, first `5` will be pushed, then `4`, then `3`. This allows for more intuitive argument offsets from the `BP`, since the positive argument offset for `3` (the first argument) will be smaller than the positive argument offset for `4` (the second argument).
    3. Push return address and jump to callee.
2. Callee (Prolog)
    - The caller's return address is at the top of the stack.
    1. Push the current `BP` (save the caller's base pointer).
    2. Set the `BP` to the `SP` (defines the callee's base pointer).
    3. Make space for local variables by decrementing the `SP` by the total size of the callee's local variables in bytes.

3. Callee (Epilog) 0. (Return Value) Set the `DP` to the address of the return value.
    1. Free the stack frame by setting the `SP` to the `BP`.
    2. Pop the caller's `BP` from the stack (i.e., set the `BP` to `{*(SP + 1), *SP}`)
    3. Pop the return address and return to the caller.
4. Caller (After Callee Returns)
    1. Pop the callee arguments from the stack.
    2. Restore the caller-saved registers (`R0` - `R3`).
    3. (Return Value) Use return value whose address is in `DP`.

There's a question of how to pass the return value to the caller; I think a simple approach is to always return a pointer to it in the `DP`. (This may present complications / not be optimal, however, since the return value may be stored in the callee's stack frame, in which case the caller needs to make a local copy first, which may need to be implemented carefully if the return value is a type that's too large).

Let us attempt to do this for the following C code:

```
uint8_t add(uint8_t x, uint8_t y) {
    uint8_t z = x + y;
    return z;
}

int main() {
    //  ...
    uint8_t a = add(3, 5);
    //  ...
}
```

This is almost impossible to implement in the current ISA.

Let us attempt to implement it using the new instructions:

```
:main
    //  ...
    //  uint8_t a = add(3, 5);
    //  1.  Push caller-saved registers
    PUSH R0
    PUSH R1
    PUSH R2
    PUSH R3

    //  2.  Push callee arguments
    LDR R0, #5
    PUSH R0     //  Push argument for uint8_t y
    LDR R0, #3
    PUSH R0     //  Push argument for uint8_t x

    //  3.  Push return address and jump to callee.
    LDA %DP[H], TARGET_ADDRESS_HI (:add[H])
    LDA %DP[L], TARGET_ADDRESS_LO (:add[L])
```

```
    JSR

    //  4.  Pop the callee arguments from the stack
    POP R0      //  Pop argument for uint8_t x
    POP R0      //  Pop argument for uint8_t y

    //  5.  Restore the caller-saved registers
    POP R3
    POP R2
    POP R1
    POP R0

    //  6.  Use return value (uint8_t a = RV = *DP); assuming that a is in R1
    LDW R1, 0(%DP)

    //  ...

:add
    //  Prolog
    //  1.  Push the current BP. (Save the caller's BP)
    MAG R0, %BP[H]      //  Push BP[H]
    PUSH R0
    MAG R0, %BP[L]      //  Push BP[L]
    PUSH R0

    //  2.  Set the BP to the SP.
    MAA %BP, %SP, #0

    //  3.  Make space for local variables (1 byte for uint8_t z)
    MAA %SP, %SP, ADD_LOCAL_VARS_OFFSET (#-1)

    //  z = x + y.
    LDW R0, 4(BP)   //  R0 = *(BP + 4) = x
    LDW R1, 5(BP)   //  R1 = *(BP + 5) = y
    ADD R2, R0, R1  //  R2 = R0 + R1 = x + y
    SDW R2, -1(BP)  //  *(BP - 1) = x + y --> z = x + y

    //  Epilog
    //  0.  (Return Value) Set the DP to the address of the return value.
    MAA %DP, %BP, #-1   //  DP = BP - 1 = &z

    //  1.  Free the stack frame by setting the SP to the BP.
    MAA %SP, %BP, #0    //  SP = BP + 0 = BP

    //  2.  Pop the caller's BP from the stack
    POP R0              //  R0 = *SP = callerBP_LO
    MGA %BP[L], R0      //  BP[L] = R0 = callerBP_LO
    POP R0              //  R0 = *SP = callerBP_HI
    MGA %BP[H], R0      //  BP[H] = R0

    //  3.  Pop the return address and return to the caller.
    RTS
```

2. Iterating through an array of structs

Let us attempt to implement the following C code using the new proposal's instructions:

```
typedef struct {
    char * playerName;
    uint8_t health;
    uint8_t maxHealth;
    uint8_t isFighting;
} player_t Player;

int main() {
    //  Let Player * players point to an array of Player structs of length 125.
    Player * healthiestPlayer = players[0];
    for (uint8_t i = 1; i < 125; i++) {
        if (players[i]->health > healthiestPlayer->health) {
            healthiestPlayer = &(players[i]);
        }
    }

    return healthiestPlayer->health;
}
```

First, we can observe that the `Player` struct has size `2 + 1 + 1 + 1 = 5` bytes, with offsets `playerName_offset = 0`, `health_offset = 2`, `maxHealth_offset = 3`, and `isFighting_offset = 4`.

The assembly code for this could look as follows:

```
:main
    // Prolog
    //  1.  Save the caller's BP.
    MAG R0, %BP[H]      //  Push BP[H]
    PUSH R0
    MAG R0, %BP[L]      //  Push BP[L]
    PUSH R0

    //  2.  Set the BP to the SP.
    MAA %BP, %SP, #0

    //  3.  Make space for local variables
    //      sizeof(players) + sizeof(healthiestPlayer) + sizeof(i) =
    //          = (125 * 5) + 2 + 1 (+ 2) = 630
    //      BP -->  BP[L]
    //              players[124][4]
    //              players[124][3]
    //              ...
    //              players[0][0]
    //              healthiestPlayer[H]
    //              healthiestPlayer[L]
    //              i
```

```
    //              DPTemp[H]
    //              DPTemp[L]
    //      Therefore, SP = SP - 630

    LDR R0, #4     //  R0 = 4
    :sp_lv_space_for_loop   //  (SP local variable making space for loop)
        TST R0, #0                      //  while (R0 > 0)
        BEQ :sp_lv_space_for_loop_end   //  {
        MAA %SP, %SP, #-128             //      SP = SP - 128;
        DEC R0                          //      R0--;
        JMP :sp_lv_space_for_loop       //  }
    :sp_lv_space_for_loop_end
    MAA %SP, %SP, #-118                 //  SP = SP - 118

    //  Player * healthiestPlayer = players[0]
    MAA %DP, %SP, #5    //  DP = &(players[0])
    MAG R0, %DP[H]      //  R0 = (&(players[0]))[H]
    SDW R0, 4(%SP)      //  *(SP + 2) = healthiestPlayer[H] = &(players[0])[H]
    MAG R0, %DP[L]      //  R0 = (&(players[0]))[L]
    SDW R0, 3(%SP)      //  *(SP + 1) = healthiestPlayer[L] = &(players[0])[L]

    LDR R0, #1
    SDW R0, 2(%SP)      //  uint8_t i = 1;
    :for_loop_1
        TST R0, #125            //  for (; i < 125; i++)
        BEQ :for_loop_1_end     //  {

        //  if (players[i]->health > healthiestPlayer->health)
        MAA %DP, %DP, #11       //  DP = &(players[i - 1]) + sizeof(Player)
        LDW R1, health_offset(%DP)  //  R1 = player[i]->health

        //  Save DP in DPTemp (DPTemp = DP)
        MAG R2, %DP[H]          //  R2 = DP[H] = &players[i][H]
        SDW R2, 1(%SP)
        MAG R2, %DP[L]          //  R2 = DP[L] = &players[i][L]
        SDW R2, 0(%SP)

        //  DP = healthiestPlayer
        LDW R2, 4(%SP)          //  R2 = healthiestPlayer[H]
        MGA %DP[H], R2          //  DP[H] = healthiestPlayer[H]
        LDW R2, 3(%SP)          //  R2 = healthiestPlayer[L]
        MGA %DP[L], R2          //  DP[L] = healthiestPlayer[L]

        LDW R2, health_offset(%DP)  //  R2 = healthiestPlayer->health

        //  DP = DPTemp (&players[i])
        LDW R3, 1(%SP)          //  R3 = &players[i][H]
        MGA %DP[H], R3          //  DP[H] = &players[i][H]
        LDW R3, 0(%SP)          //  R3 = &players[i][L]
        MGA %DP[L], R3          //  DP[L] = &players[i][L]

        SUB R2, R2, R1          //  R2 = healthiestPlayer->health
                                //      - player[i]->health
        BPL :after_if_1
```

```
            //  healthiestPlayer = &(players[i]);
            MAG R1, %DP[H]      //  R1 = &players[i][H]
            SDW R1, 4(%SP)      //  healthiestPlayer[H] = &players[i][H]
            MAG R1, %DP[L]      //  R1 = &players[i][L]
            SDW R1, 3(%SP)      //  healthiestPlayer[L] = &players[i][L]
        :after_if_1

        //  i++
        INC R0                  //  i++
        SDW R0, 2(%SP)
        JMP :for_loop_1         //  }
    :for_loop_1_end

    //  Epilog
    //  0.  (Return Value) Set the DP to the address of the return value
    //  return healthiestPlayer->health;
    //  Set DP = &(healthiestPlayer->health)
    LDW R0, 4(%SP)              //  R0 = healthiestPlayer[H]
    MGA %DP[H], R0              //  DP[H] = healthiestPlayer[H]
    LDW R0, 3(%SP)              //  R0 = healthiestPlayer[L]
    MGA %DP[L], R0              //  DP[L] = healthiestPlayer[L]
    MAA %DP, %DP, health_offset //  DP = healthiestPlayer + health_offset

    //  1.  Free the stack frame by setting the SP to the BP.
    MAA %SP, %BP, #0    //  SP = BP + 0 = BP

    //  2.  Pop the caller's BP from the stack
    POP R0              //  R0 = *SP = callerBP_LO
    MGA %BP[L], R0      //  BP[L] = R0 = callerBP_LO
    POP R0              //  R0 = *SP = callerBP_HI
    MGA %BP[H], R0      //  BP[H] = R0

    //  3.  Pop the return address and return to the caller.
    RTS
```

3. Iterating through doubly linked list

Instruction String Operands Updates

There are `3` new instruction operands in this proposal. They are:

Operand	**Size (in bits)**	**Bit Locations**	**Semantics**
`(AR) L/H`	`1`	`5`	Whether the high or low byte of an AR operand is used (relevant for writing or reading a data word from / to the data bus)
`ARSrc`	`2`	**Special Case; See Below**	Specify which AR to read from

Operand	Size (in bits)	Bit Locations	Semantics
`ARDest`	`2`	`7:6`	Specify which AR to write to

Note: The `ARSrc` bit location depends on the opcode of the current instruction in the following way:

1. If the opcode is of the new `MAG` instruction, then `ARSrc` is in bits `9:8`.
2. Otherwise, `ARSrc` is in bits `{5, 0}` (where `0` is the lsb of the opcode).

- In both cases, the higher bit is the msb of `ARSrc` (`9` or `5`).

New Instruction Operand Values

1. `(AR) L/H`

Bit 0 (location: `5`)	Meaning
`0`	Low Byte of an AR
`1`	High Byte of an AR

2. `ARSrc` and `ARDest`

Bit 1 (location: `ARSrc`: `9` or `5`; `ARDest`: `7`)	Bit 0 (location: `ARSrc`: `8` or `0`; `ARDest`: `6`)	Meaning
`0`	`0`	`PC` (Note: When `ARSrc` is `PC`, then `PC` is `{PC[H], PC[L]}`; when `ARDest` is `PC`, then `PC` is `{DHPC, PC[L]}`!)
`0`	`1`	`DP`
`1`	`0`	`SP`
`1`	`1`	`BP`

New Instructions

There are `6` new instructions, which require a total of `9` opcodes. They are:

Note: The opcode assignment of the new instructions assumes that the `LSP` instruction is removed and that none of the deprecated instructions are removed. Different opcode assignments may be required / more optimal if these assumptions are false.

Instruction	Semantics	Example Usage	Number of Opcodes	Opcode Assignment
`MAG`	`Rd = ARSrc[L/H]`	`MAG R1, %SP[H]`	`1`	`0x1a`
`LDWL` and `LDWH`	`Rd = *(ARSrc+ Imm)`	`LDW R2, 16(%BP)`	`2`	`0x14` and `0x15`

Instruction	Semantics	Example Usage	Number of Opcodes	Opcode Assignment
`SDWL` and `SWDH`	`*(ARSrc + Imm) = Rs`	`SDW R3, -4(%DP)`	2	`0x16` and `0x17`
`LDA`	`ARDest[L/H] = Imm`	`LDA %DP[L], #0x54`	1	`0x1b`
`MGA`	`ARDest[L/H] = Rs1`	`MGA %BP[L], R2`	1	`0x1c`
`MAAL` and `MAAH`	`ARDest = ARSrc + Imm`	`MAA %SP, %SP, #-1`	2	`0x18` and `0x19`

Note: Instructions that require two opcodes (`LDW`, `SDW`, and `MAA`) will have each opcode named differently with a `L` or `H` suffix to distinguish the opcodes within the ISA (i.e., these will be implemented as two separate ISA instructions) but will be implemented as a single ByteFrost Assembly instruction (i.e., there will only be one `LDW` ByteFrost Assembly instruction that will be assembled into either `LDWL` or `LDWH` by the ByteFrost Assembler depending on the AR specified in the `ARSrc` operand).

The new instructions have the following instruction strings:

1. `MAG Rd, ARSrc[L/H]` (`Rd = ARSrc[L/H]`)

`15:10`	`9:8`	`7:6`	`5`	`4:0`
X	`ARSrc`	`Rd`	`(AR) L/H`	Opcode

2. `LDW Rd, Imm(ARSrc)` (`Rd = *(ARSrc + Imm)`)

`15:8`	`7:6`	`5`	`4:0`
`Imm_8`	`Rd`	`ARSrc`	Opcode (and `ARSrc` lsb)

3. `SDW Rs, Imm(ARSrc)` (`*(ARSrc + Imm = Rs)`)

`15:8`	`7:6`	`5`	`4:0`
`Imm_8`	`Rd`	`ARSrc`	Opcode (and `ARSrc` lsb)

4. `LDA ARDest[L/H], #Imm` (`ARDest[L/H] = Imm`)

`15:8`	`7:6`	`5`	`4:0`
`Imm_8`	`ARDest`	`(AR) L/H`	Opcode

5. `MGA ARDest[L/H], Rs1` (`ARDest[L/H] = Rs1`)

`15:14`	`13:12`	`11:8`	`7:6`	`5`	`4:0`
X	`Rs1`	X	`ARDest`	`(AR) L/H`	Opcode

6. `MAA ARDest, ARSrc, #Imm` (`ARDest = ARSrc + Imm`)

15:8	7:6	5	4:0
Imm_8	ARDest	ARSrc	Opcode (and ARSrc lsb)

Updated Instructions

There are 4 updated instructions. They are:

Instruction	Current Semantics	New Semantics	Example New Usage	Number of Opcodes	Opcode Assignment
PUSH	`*SP = Rs1, SP++`	`SP--, *SP = Rs1`	`PUSH R2`	1	0x0e
POP	`SP--, Rd = *SP`	`Rd = *SP, SP++`	`POP R2`	1	0x0f
JSR	`*SP = PC[L], SP++, PC[L] = Imm, PC[H] = DHPC`	`SP--, *SP = PC[H], SP--, *SP = PC[L], PC = DP`	`JSR`	1	0x10
RTS	`SP -= 2, DHPC = *SP, PC[H] = DHPC, SP++, PC[L] = *SP, SP--`	`SP++, DHPC = *SP, SP--, PC[L] = *SP and PC[H] = DHPC, SP++, SP++`	`RTS`	1	0x11

PUSH and POP are updated to support a full-descending stack instead of an empty-increasing one.

JSR and RTS are similarly updated, with JSR updated to push the full PC (return address) to the stack and jump to the address in the DP (allowing for function pointer semantics).

Note: JSR's new implementation requires hardware support to push the PC value to the stack; since AR values are written to the Data Bus via the address bus, this requires having the PC L/H byte on the Data Bus while the SP's value is on the Address Bus. To do this, writing a byte from the Address Bus to the Data Bus must involve writing to a temporary register first, which will hold the value and only write it to the Data Bus in the next cycle.

Removed Instructions

There is 1 instruction that should be removed, which frees up 1 opcode. It is:

Instruction	Semantics	Example Usage	Number of Opcodes	Opcode Assignment
LSP	`ARDest[H] = Imm` (where `ARDest` != `BP`)	`LSP %DP, #0x30`	1	0x14

The LSP instruction should be removed because it is eclipsed by the proposed LDA instruction, which is a generalization of it.

Deprecated Instructions and Instruction Operands

The following `5` instructions, in their present form, can be marked as deprecated, and eventually removed to free up `5` opcodes.

Instruction	Semantics	Example Usage	Number of Opcodes	Opcode Assignment
Branch Relative	If the condition bits `C2C1C0` match the ALU flag register bits, then `PC[L] = PC[L] + Immediate`	`BEQ +14`	`1`	`0x07`
`LMA`	`Rd = *({DP[H], Imm}), DP[L] = Imm`	`LMA R2, #0x05`	`1`	`0x09`
`SMA`	`*({DP[H], Imm}) = Rs, DP[L] = Imm`	`SMA R3, #0x30`	`1`	`0x0a`
`LMR`	`Rd = *({DP[H], Rs1}), DP[L] = Rs1`	`LMR R2, R1`	`1`	`0x0b`
`SMR`	`*({DP[H], Rs1}) = Rs, DP[L] = Rs1`	`SMR R1, R3`	`1`	`0x0c`

1. Branch Relative (opcode `0x07`) may be marked as deprecated since it only updates the low byte of the `PC`; i.e., it can only branch within an "instruction page" (i.e., the 512 byte region starting with 9 lsbs being `0` and ending with 9 lsbs being `1`, with the high 7 bits unchanged). Additionally, the ByteFrost Assembler v2 only uses absolute branches, so this instruction isn't used by any ByteFrost Assembly program assembled with the ByteFrost Assembler v2.
2. Instructions `LMA`, `SMA`, `LMR`, and `SMR` (opcodes `0x09` to `0x0c`) may be marked as deprecated since they have unexpected semantics (they overwrite the low byte of the `DP` address register) and are effectively deprecated by `LDW` and `SDW` (with `LDA` and `MGA` to set address register bytes with an immediate / GR).

Instructions that could be deprecated in the future

The following `2` instructions can be marked as deprecated in the future once a set of conditions is cleared.

Instruction	Semantics	Example Usage	Number of Opcodes	Opcode Assignment
`OUT`	`Print Rs1 as an ASCII character or hex integer`	`OUT R2, A`	`1`	`0x08`
`OUT Immediate`	`Print Immediate as an ASCII character of hex integer`	`OUT #0x54, I`	`1`	`0x0d`

1. `OUT` and `OUT Immediate` (opcodes `0x08` and `0x0d`, respectively) may eventually be deprecated, along with the instruction operand `A/I (Display)`, when the following condition is cleared:
    1. Replace the current display with a display whose interface to the ByteFrost is part of the MMIO section of the address space.

- This will allow for the following benefits:
    1. Free `2` opcodes.

2. Allow for a more general display interface, and potentially for a larger display (e.g., hopefully a `600x800` resolution?)

Control Signals Updates

New Control Signals

The proposal introduces `5` new control signals. They are:

Control Signal	Signal ID Assignment	Instructions that use it
loadARHorL	`18`	`?`
TmpARRead	`19`	`?`
TmpARWrite	`20`	`?`
AddressBusToDataBus	`21`	`?`
AddressHorL	`22`	`?`

Note: *loadARHorL* and *AddressHorL* match up in all the instructions that use them (or can; i.e., one might be set to a particular value and the other may be `X`; it is never the case that one is `1` and the other is `0` in the same cycle, for instance, and since they're both "guarded" by *loadAR* and *AddressBusToDataBus*, respectively, they're `X` when not used instead of `0`).

Updated Control Signals

The proposal updates (renames) `2` control signals. They are:

Control Signal	Old Name	Signal ID Assignment	Instructions that use it
loadAR	*Load Special Register*	`17`	`?`
SP Out	*RAM Address Select*	`14`	`?`

Removed Control Signals

`?`

Combinational Logic Updates

Implementing this proposal requires updating `5` of the combinational logic units of the ByteFrost:

1. Decode - `ARSrc` Operand
2. PC Loading (When `PC[L]` is loaded, set `PC[H]` to load from `DHPC`).
3. AR Data Bus Load Enable (Replaces the Load Special Pointer unit)
4. ARSelect (Replaces the Address Bus Arbiter)
5. AddressByteSelect

1. Decode - `ARSrc` Operand

Inputs:

1. `Opcode_MAG` (active low signal from opcode decoder)
2. `INSTR[5]`
3. `INSTR[9:8]`

Outputs: `ARSrc` (2 bits).

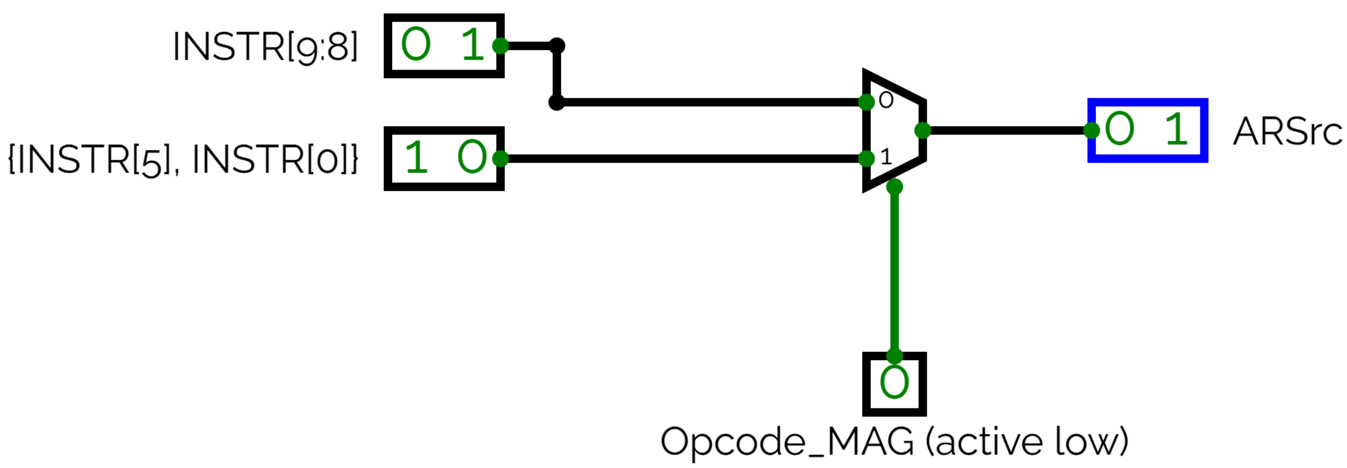
Logic:

1. If Opcode == `MAG_OPCODE`:
    1. `ARSrc = INSTR[9:8]`.
2. Else:
    1. `ARSrc = {INSTR[5], INSTR[0]}`.

Truth Table:

`Opcode_MAG` (active low)	Output (2 bits)
`0` (active)	`INSTR[9:8]`
`1` (inactive)	`{INSTR[5], INSTR[0]}`

Implementation Diagram:



2. PC Loading / Out Revision

Remove the `PC Ld Hi` active low signal, and rename the `PC Ld Lo` active low signal to `PC Ld Branch` (the signal generated in the Branch (v2.0) slide 26 in CPU v2).

The `PC Ld Branch` signal is now sent as an input to the **AR Data Bus Load Enable** unit described in the next section.

The DHPC load enable signal comes from the AR Data Bus Load Enable unit, named `DHPC Load Enable`.

The `PC[L]` and `PC[H]` load enable signals come from the AR Data Bus Load Enable unit, named `PC[L] + PC[H] = DHPC Load Enable`.

The `PC Out` control signal is now an input of the **ARSelect** unit described in a further section. The `74HC245` "Read PC Low" tristate IC can be removed, as the `PC` may only write to the Address Bus (writing the `PC` to the

Data Bus involves first writing the `PC` to the Address Bus).

Implementation Diagram: TODO

3. AR Data Bus Load Enable

Inputs:

1. `PC Ld Branch` signal
2. *loadAR* control signal
3. *loadARHorL* control signal
4. Opcode (`INSTR[4:0]`)
5. `ARDest` instruction operand
6. `(AR) L/H` instruction operand

Outputs: {`None (No AR reads from Data Bus)`, `DHPC Load Enable`, `PC[L] + PC[H] = DHPC Load Enable`, `DP[H] Load Enable`, `DP[L] Load Enable`, `SP[H] Load Enable`, `SP[L] Load Enable`, `BP[H] Load Enable`, `BP[L] Load Enable`}

Note: There are two possible semantics when writing to the `PC`: Writing to the `DHPC` and writing to `PC[L]` in which case `PC[L]` loads from `DHPC`. (i.e., `DHPC = Data Bus` OR `PC[L] = Data Bus AND PC[H] = DHPC`).

Logic:

1. If `PC Ld Branch` is active (low):

- A Data Bus write to the `PC` is requested by a branch instruction (Branch Absolute or Branch Relative (deprecated)).
    1. Return `PC[L] + PC[H] = DHPC Load Enable`.

2. Else if the *loadAR* control signal is active (high):

- This means that the current instruction being executed is `LDA`, `MGA`, `MAA`, `JSR`, or `RTS`.
    1. If the *PC Load* control signal is active (high):
    - A Data Bus write to the `PC` is requested, but to which byte?
    - The request is made either by `JSR` or `RTS`; the byte is specified by the *loadARHorL* control signal.
        1. If *loadARHorL* (control signal) is `0` (L):
            1. Return `PC[L] + PC[H] = DHPC Load Enable`.
        2. Else:
            1. Return `DHPC Load Enable`.
    2. Else:
    - The request is made by the `LDA`, `MGA`, or `MAA` instruction, meaning that the particular AR to write to is specified by the `ARDest` instruction operand. Now we must identify whether the high or low byte is specified.
    - If it's made by `MAA` (i.e., by an instruction that involves writing to both bytes of an AR), then the byte MUST be specified by the *loadARHorL* control signal.
    - Otherwise, the byte is specified by the `(AR) L/H` instruction operand.
        1. If Opcode == `MAA_OPCODE`:
            1. Return `{ARDest} {loadARHorL control signal} Load Enable`.
        2. Else:

            1. Return `{ARDest} {(AR) L/H instruction operand} Load Enable`.

   3. Else:
      1. Return `None`.

Truth Table:

`PC Load Branch` (active low)	*loadAR*	*PC Load*	*loadARHorL*	`Opcode_MAA` (active low)	AR byte that loads from the Data Bus
`0`	`X`	`X`	`X`	`X`	`PC[L] + PC[H] = DHPC Load Enable`
`1`	`0`	`X`	`X`	`X`	`None`
`1`	`1`	`1`	`0`	`X`	`PC[L] + PC[H] = DHPC Load Enable`
`1`	`1`	`1`	`1`	`X`	`DHPC Load Enable`
`1`	`1`	`0`	`0`	`0`	`ARDest[L] Load Enable`
`1`	`1`	`0`	`1`	`0`	`ARDest[H] Load Enable`
`1`	`1`	`0`	`X`	`1`	`ARDest[(AR) L/H instruction operand] Load Enable`

Implementation Using Look Up Table EEPROM

For **AR Data Bus Load Enable**, there are `8` 1-bit inputs:

1. `PC Load Branch` (active low)
2. *loadAR*
3. *PC Load*
4. *loadARHorL*
5. `Opcode_MAA` (active low)
6. `ARDest[1]`
7. `ARDest[0]`
8. `(AR) L/H instruction operand`

There are `8` outputs:

1. `PC[L] + PC[H] = DHPC Load Enable` (active low)
2. `DHPC Load Enable` (active low)
3. `DP[L] Load Enable` (active low)
4. `DP[H] Load Enable` (active low)
5. `SP[L] Load Enable` (active low)
6. `SP[H] Load Enable` (active low)
7. `BP[L] Load Enable` (active low)
8. `BP[H] Load Enable` (active low)

As there are `8` 1-bit inputs, there are `2^8 = 256` possible input states, each requiring `8` bits to store the corresponding output state.

Hence, this EEPROM would need to have at least `256` addresses (8 address inputs) each of which maps to `8` bits.

With an `8` bit output, the EEPROM would need to be `256 * 8 = 2 Kbit (256 bytes)` in size.

4. ARSelect

Inputs:

1. *SP Out* (control signal)
2. *TmpARWrite* (control signal)
3. *PC Out* (control signal)
4. Bus Grant signal
5. FetchCycle signal
6. Opcode (`INSTR[4:0]`),
7. `ARSrc` instruction operand

Outputs: {`None` (Address Bus floats), `PC Output Enable`, `DP Output Enable`, `SP Output Enable`, `BP Output Enable`, `TmpAR Output Enable`}

Logic:

1. If the Bus Grant signal is active:
    1. Return `None`.
2. Else if the FetchCycle signal is active OR *PC Out* is active:
    1. Return `PC Output Enable`.
3. Else if the *SP Out* control signal is active:
    1. Return `SP Output Enable`.
4. Else if the *TmpARWrite* control signal is active:
    1. Return `TmpAR Output Enable`.
5. Else if the instruction has an `ARSrc` instruction operand:
    - This means that the instruction has the opcode of any of the following instructions: (`MAG`, `LDW`, `SDW`, `MAA`)
    1. Return `ARSrc Output Enable`.
6. Else:
    - Default behavior: write `DP` to the bus.
    1. Return `DP Output Enable`.

Truth Table:

`Bus Grant`	`FetchCycle`	*PC Out*	*SP Out*	*TmpARWrite*	`Opcode_MAG_LDW_SDW_MAA` (active low)	ARSelect
`1`	`X`	`X`	`X`	`X`	`X`	`None`
`0`	`1`	`X`	`X`	`X`	`X`	`PC Output Enable`

Bus Grant	FetchCycle	*PC Out*	*SP Out*	*TmpARWrite*	Opcode_MAG_LDW_SDW_MAA (active low)	**ARSelect**
0	0	1	X	X	X	PC Output Enable
0	0	0	1	X	X	SP Output Enable
0	0	0	0	1	X	TmpAR Output Enable
0	0	0	0	0	0	ARSrc Output Enable
0	0	0	0	0	1	DP Output Enable

Implementation Approach Using Look-up Table (EEPROM)

Implementing the combinational logic of this table directly (using a decoder or even without) requires a high number of logic gates, which in turn requires a higher number of ICs, which is costly area-wise. A different approach could be to use an EEPROM as a LUT; the input signals would be used as part of the EEPROM address and the outputs stored as part of the 8-bit data words.

This would take overall less area and be easier to fix (simply overwrite the EEPROM contents).

For **ARSelect**, there are `8` 1-bit inputs:

1. `Bus Grant`
2. `FetchCycle`
3. *PC Out*
4. *SP Out*
5. *TmpARWrite*
6. `Opcode_MAG_LDW_SDW_MAA` (active low)
7. `ARSrc[1]`
8. `ARSrc[0]`

There are `5` outputs:

1. `PC Output Enable` (active low)
2. `DP Output Enable` (active low)
3. `SP Output Enable` (active low)
4. `BP Output Enable` (active low)
5. `TmpAR Output Enable` (active low)

Since there are `8` 1-bit inputs, there are `2^8 = 256` possible input states, which can be stored in a 256-byte EEPROM.

With a `5` bit output, the EEPROM would need to be `256 * 5 = 1280 bits (160 bytes)` in size.

5. AddressByteSelect

Inputs:

1. *AddressHorL* (control signal)
2. `(AR) L/H` instruction operand
3. Opcode (`INSTR[4:0]`)

Outputs: {`Address Bus High Register`, `Address Bus Low Register`}

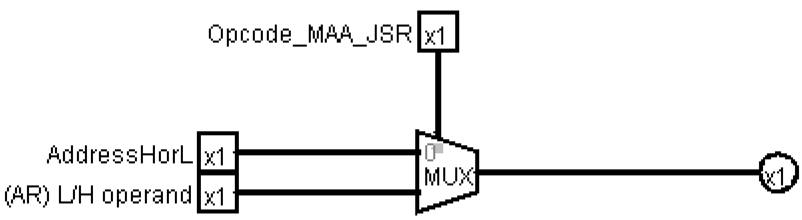
Logic:

1. If Opcode == `MAA_OPCODE` OR Opcode == `JSR_OPCODE`:
   1. If *AddressHorL* is active (High):
      1. Return `Address Bus High Register`.
   2. Else:
      1. Return `Address Bus Low Register`.
2. Else:
   1. Return `Address Bus {(AR) L/H instruction operand} Register`.

Truth Table:

`Opcode_MAA_JSR` (active low)	AddressByteSelect
`0`	*AddressHorL*
`1`	`(AR) L/H instruction operand`

Implementation Diagram:



Updated Instruction Microcode

The microcode of `10` instructions will need to be updated to implement this proposal. The instructions and their new microcode follows:

Note: In the last microinstruction of every instruction, the *PC Advance* control signal is set so that the Fetch State Machine will fetch the next instruction in the next cycle. As a result of *PC Advance* being active, the *FetchCycle* signal is high, meaning that the AR Select unit will set the `PC` to write to the Address Bus AND that the Data Bus is in use (as memory is writing the low instruction byte of the next instruction to the Data Bus). As a result, in the last microinstruction of any instruction, no AR beside the `PC` may write to the Address Bus and no entity may write to the Data Bus beside memory (this may cause some instructions to require an additional cycle just to fetch the next instruction).

Control Signal List:

Bit	Control Signal
23	Not Used.
22	*AddressHorL*
21	*AddressBusToDataBus*
20	*TmpARWrite*
19	*TmpARRead*
18	*loadARHorL*
17	*loadAR* (formerly: Load Special Pointer)
16	*Stack Pointer Increment / Decrement* (`0`: decrement / `1`: increment)
15	*Stack Pointer Count*
14	*SP Out* (formerly: RAM Address Select)
13	*Use Rd as Source*
12	*Lower Address Register Load*
11	*Mem Write*
10	*Mem Read*
9	*PC Out*
8	*PC Load*
7	*PC Advance*
6	*Program Register H Write to Bus*
5	*Register File Input Enable*
4	*Register File Output Enable*
3	*Register File Output Select* (`0`: Rs1 / `1` Rs2)
2	*ALU output enable*
1	*ALU load register A*
0	*ALU load register B*

`MAG` Instruction Microcode

Opcode: `0x1a`

`MAG` **Semantics:** `Rd = ARSrc[L/H]`

Cycle	`23:20`	`19:16`	`15:12`	`11:8`	`7:4`	`3:0`	Result
1	`0`	`0`	`0`	`0`	`0`	`0`	`0x00_00_00`

Cycle	23:20	19:16	15:12	11:8	7:4	3:0	Result
2	2	0	0	0	2	0	0x20_00_20
3	0	0	0	0	1000	0	0x00_00_80

1. *Cycle 1*. 0. No control signal is set; hence, microcode is `0x00_00_00`.
    1. Write `ARSrc` to the Address Bus.
        1. **ARSelect** = `ARSrc`.
        - Guaranteed behavior since Bus Grant is `0`, FetchCycle is `0`, *SP Out* is `0`, *TmpARWrite* is `0`, and opcode is of the `MAG` instruction.
2. *Cycle 2*. 0. Set control signals: 1. *AddressBusToDataBus* (signal `21`) to `1`. 2. *Register File Input Enable* (signal `5`) to `1`. * Microcode: `0010_0000_0000_0000_0010_0000` = `0x20_00_20`.
    1. Write Address Bus L/H Register to the Data Bus.
        1. **AddressByteSelect** = `(AR) L/H` instruction operand.
        - Guaranteed behavior since the instruction isn't `MAA`.
        2. *AddressBusToDataBus* = `1`.
    2. Set `Rd` to read from the Data Bus.
        1. *Register File Input Enable* = `1`.
3. *Cycle 3*. 0. Set control signals: 1. *PC Advance* (signal `7`) to `1`.
    1. Advance `PC`.
        1. *PC Advance* = `1`.

`LDW` Instruction Microcode

Opcodes: `LDWL` (`0x14`) and `LDWH` (`0x15`).

`LDW` **Semantics:** `Rd = *(ARSrc + Imm)`

Cycle	23:20	19:16	15:12	11:8	7:4	3:0	Result
1	0	1000	0	0	0100	0	0x08_00_40
2	0001	0	0	0100	0010	0	0x10_04_20
3	0	0	0	0	1000	0	0x00_00_80

1. *Cycle 1*. 0. Set control signals: 1. *Program Register H Write to Bus* (signal `6`) to `1`. 2. *TmpARRead* (signal `19`) to `1`. * Microcode: `0000_1000_0000_0000_0100_0000 = 0x08_00_40`.
    1. Write `Imm` to the Data Bus.
        1. *Program Register H Write to Bus* = `1`.
    2. Write `ARSrc` to the Address Bus.
        1. **ARSelect** = `ARSrc`.
        - Guaranteed behavior since opcode is of the `LDW` instruction.
    3. Set `TmpAR` to read (from the 16-bit adder).
        1. *TmpARRead* = `1`.
2. *Cycle 2*. 0. Set control signals: 1. *TmpARWrite* (signal `20`) to `1`. 2. *Mem Read* (signal `10`) to `1`. 3. *Register File Input Enable* (signal `5`) to `1`. * Microcode: `0001_0000_0000_0100_0010_0000 = 0x10_04_20`.
    1. Write `TmpAR` to the Address Bus.
        1. **ARSelect** = `TmpAR`.

            1. *TmpARWrite* = `1`.
    2. Write Memory data word to the Data Bus.
        1. *Mem Read* = `1`.
    3. Set `Rd` to read from the Data Bus.
        1. *Register File Input Enable* = `1`.
3. *Cycle 3*. 0. Set control signals: 1. *PC Advance* (signal `7`) to `1`. * Microcode: `0x00_00_80`.
    1. Advance `PC`.
        1. *PC Advance* = `1`.

`SDW` Instruction Microcode

Opcodes: `SDWL` (`0x16`) and `SDWH` (`0x17`).

`SDW` Semantics: `*(ARSrc + Imm) = Rs (Rd as source)`

Cycle	`23:20`	`19:16`	`15:12`	`11:8`	`7:4`	`3:0`	Result
`1`	`0`	`1000`	`0`	`0`	`0100`	`0`	`0x08_00_40`
`2`	`0001`	`0000`	`0010`	`1000`	`0001`	`0`	`0x10_28_10`
`3`	`0`	`0`	`0`	`0`	`1000`	`0`	`0x00_00_80`

1. *Cycle 1*. 0. Set control signals: 1. *Program Register H Write to Bus* (signal `6`) to `1`. 2. *TmpARRead* (signal `19`) to `1`.
    1. Write `Imm` to the Data Bus.
        1. *Program Register H Write to Bus* = `1`.
    2. Write `ARSrc` to the Address Bus.
        1. **ARSelect** = `ARSrc`.
            - Guaranteed behavior since opcode is of the `SDW` instruction.
    3. Set `TmpAR` to read.
        1. *TmpARRead* = `1`.
2. *Cycle 2*. 0. Set control signals: 1. *TmpARWrite* (signal `20`) to `1`. 2. *Register File Output Enable* (signal `4`) to `1`. 3. *Use Rd as a Source* (signal `13`) to `1`. 4. *Mem Write* (signal `11`) to `1`.
    1. Write `TmpAR` to the Address Bus.
        1. **ARSelect** = `TmpAR`.
            1. *TmpARWrite* = `1`.
    2. Write `Rs (Rd)` to the Data Bus.
        1. *Register File Output Enable* = `1`.
        2. *Use Rd as a Source* = `1`.
    3. Set Memory to read from the Data Bus.
        1. *Mem Write* = `1`.
3. *Cycle 3*. 0. Set control signals: 1. *PC Advance* (signal `7`) to `1`.
    1. Advance `PC`.
        1. *PC Advance* = `1`.

`LDA` Instruction Microcode

Opcode: `0x1b`

`LDA` **Semantics:** `ARDest[L/H] = Imm`.

Cycle	`23:20`	`19:16`	`15:12`	`11:8`	`7:4`	`3:0`	Result
`1`	`0`	`0010`	`0`	`0`	`0100`	`0`	`0x02_00_40`
`2`	`0`	`0`	`0`	`0`	`1000`	`0`	`0x00_00_80`

1. *Cycle 1*. 0. Set control signals: 1. *Program Register H Write to Bus* (signal `6`) to `1`. 2. *loadAR* (signal `17`) to `1`.
    1. Write `Imm` to the Data Bus.
        1. *Program Register H Write to Bus* = `1`.
    2. Set `ARDest[L/H]` to read from the Data Bus.
        1. **AR Data Bus Load Enable** = `ARDest[(AR) L/H operand]`.
            1. *loadAR* = `1`.
2. *Cycle 2*. 0. Set control signals: 1. *PC Advance* (signal `7`) to `1`.
    1. Advance `PC`.
        1. *PC Advance* = `1`.

`MGA` Instruction Microcode

Opcode: `0x1c`

`MGA` **Semantics:** `ARDest[L/H] = Rs1`.

Cycle	`23:20`	`19:16`	`15:12`	`11:8`	`7:4`	`3:0`	Result
`1`	`0`	`0010`	`0`	`0`	`0001`	`0`	`0x02_00_10`
`2`	`0`	`0`	`0`	`0`	`1000`	`0`	`0x00_00_80`

1. *Cycle 1*. 0. Set control signals: 1. *Register File Output Enable* (signal `4`) to `1`. 2. *Register File Output Select* (signal `3`) to `0`. 3. *loadAR* (signal `17`) to `1`.
    1. Write `Rs1` to the Data Bus.
        1. *Register File Output Enable* = `1`.
        2. *Register File Output Select* = `0` (Rs1).
    2. Set `ARDest[L/H]` to read from the Data Bus.
        1. **AR Data Bus Load Enable** = `ARDest[(AR) L/H operand]`.
            1. *loadAR* = `1`.
2. *Cycle 2*. 0. Set control signals: 1. *PC Advance* (signal `7`) to `1`.
    1. Advance `PC`.
        1. *PC Advance* = `1`.

`MAA` Instruction Microcode

Opcodes: `MAAL` (`0x18`) and `MAAH` (`0x19`).

`MAA` **Semantics:** `ARDest = ARSrc + Imm`.

Cycle	`23:20`	`19:16`	`15:12`	`11:8`	`7:4`	`3:0`	Result

Cycle	`23:20`	`19:16`	`15:12`	`11:8`	`7:4`	`3:0`	Result
`1`	`0`	`1000`	`0`	`0`	`0100`	`0`	`0x08_00_40`
`2`	`0001`	`0`	`0`	`0`	`0`	`0`	`0x10_00_00`
`3`	`0011`	`0010`	`0`	`0`	`0`	`0`	`0x32_00_00`
`4`	`0110`	`0110`	`0`	`0`	`0`	`0`	`0x66_00_00`
`5`	`0`	`0`	`0`	`0`	`1000`	`0`	`0x00_00_80`

1. *Cycle 1*. 0. Set control signals: 1. *Program Register H Write to Bus* (signal `6`) to `1`. 2. *TmpARRead* (signal `19`) to `1`.
    1. Write `Imm` to the Data Bus.
        1. *Program Register H Write to Bus* = `1`.
    2. Write `ARSrc` to the Address Bus.
        1. **ARSelect** = `ARSrc`.
        - Guaranteed behavior since opcode is of the `MAA` instruction.
    3. Set `TmpAR` to read.
        1. *TmpARRead* = `1`.
2. *Cycle 2*. 0. Set control signals: 1. *TmpARWrite* (signal `20`) to `1`.
    1. Write `TmpAR` to the Address Bus.
        1. **ARSelect** = `TmpAR`.
            1. *TmpARWrite* = `1`.
3. *Cycle 3*. 0. Set control signals: 1. *TmpARWrite* (signal `20`) to `1`. 2. *AddressHorL* (signal `22`) to `0`. 3. *AddressBusToDataBus* (signal `21`) to `1`. 4. *loadAR* (signal `17`) to `1`. 5. *loadARHorL* (signal `18`) to `0`.
    1. Write `TmpAR` to the Address Bus.
        1. **ARSelect** = `TmpAR`.
            1. *TmpARWrite* = `1`.
    2. Write Address Bus Low register to the Data Bus.
        1. **AddressByteSelect** = `L` (`0`).
            1. *AddressHorL* = `L` (`0`).
            - Guaranteed behavior since opcode is of the `MAA` instruction.
        2. *AddressBusToDataBus* = `1`.
    3. Set `ARDest[L]` to read from the Data Bus.
        1. **AR Data Bus Load Enable** = `ARDest[L]`.
            1. *loadAR* = `1`.
            2. *loadARHorL* = `L` (`0`).
            - Guaranteed behavior since opcode is of the `MAA` instruction.
4. *Cycle 4*. 0. Set control signals: 1. *AddressHorL* (signal `22`) to `1`. 2. *AddressBusToDataBus* (signal `21`) to `1`. 3. *loadAR* (signal `17`) to `1`. 4. *loadARHorL* (signal `18`) to `1`.
    1. Write Address Bus High register to the Data Bus.
        1. **AddressByteSelect** = `H` (`1`).
            1. *AddressHorL* = `H` (`1`).
        2. *AddressBusToDataBus* = `1`.
    2. Set `ARDest[H]` to read from the Data Bus.
        1. **AR Data Bus Load Enable** = `ARDest[H]`.
            1. *loadAR* = `1`.

2. *loadARHorL* = `H` (`1`).
   - Guaranteed behavior since opcode is of the `MAA` instruction.

5. *Cycle 5*. 0. Set control signals: 1. *PC Advance* (signal `7`) to `1`.
   1. Advance `PC`.
      1. *PC Advance* = `1`.

`PUSH` Instruction Microcode

Opcode: `0x0e`

`PUSH` Semantics: `SP--; *SP = Rs1`.

Cycle	`23:20`	`19:16`	`15:12`	`11:8`	`7:4`	`3:0`	Result
`1`	`0`	`0`	`1000`	`0`	`0`	`0`	`0x00_80_00`
`2`	`0`	`0`	`0100`	`1000`	`0001`	`0`	`0x00_48_10`
`3`	`0`	`0`	`0`	`0`	`1000`	`0`	`0x00_00_80`

1. *Cycle 1*. `SP--;` 0. Set control signals: 1. *Stack Pointer Count* (signal `15`) to `1`. 2. *Stack Pointer Increment / Decrement* (signal `16`) to `0`.
   1. Decrement `SP`.
      1. *Stack Pointer Count* = `1`.
      2. *Stack Pointer Increment / Decrement* = `decrement` (`0`).
2. *Cycle 2*. `*SP = Rs1;` 0. Set control signals: 1. *Register File Output Enable* (signal `4`) to `1`. 2. *Register File Output Select* (signal `3`) to `0`. 3. *SP Out* (signal `14`) to `1`. 4. *Mem Write* (signal `11`) to `1`.
   1. Write `Rs1` to the Data Bus.
      1. *Register File Output Enable* = `1`.
      2. *Register File Output Select* = `Rs1` (`0`).
   2. Write `SP` to the Address Bus.
      1. **ARSelect** = `SP`.
         1. *SP Out* = `1`.
   3. Set Memory to read from the Data Bus.
      1. *Mem Write* = `1`.
3. *Cycle 3*. 0. Set control signals: 1. *PC Advance* (signal `7`) to `1`.
   1. Advance `PC`.
      1. *PC Advance* = `1`.

`POP` Instruction Microcode

Opcode: `0x0f`

`POP` Semantics: `Rd = *SP; SP++`.

Cycle	`23:20`	`19:16`	`15:12`	`11:8`	`7:4`	`3:0`	Result
`1`	`0`	`0001`	`1100`	`0100`	`0010`	`0`	`0x01_c4_20`
`2`	`0`	`0`	`0`	`0`	`1000`	`0`	`0x00_00_80`

1. *Cycle 1*. `Rd = *SP; SP++;` 0. Set control signals: 1. *SP Out* (control signal `14`) to `1`. 2. *Mem Read* (control signal `10`) to `1`. 3. *Register File Input Enable* (control signal `5`) to `1`. 4. *Stack Pointer Count* (control signal `15`) to `1`. 5. *Stack Pointer Increment / Decrement* (control signal `16`) to `1`.
    1. Write `SP` to the Address Bus.
        1. **ARSelect** = `SP`.
            1. *SP Out* = `1`.
    2. Set Memory to write to the Data Bus.
        1. *Mem Read* = `1`.
    3. Set `Rd` to read from the Data Bus.
        1. *Register File Input Enable* = `1`.
    4. Increment `SP`.
        1. *Stack Pointer Count* = `1`.
        2. *Stack Pointer Increment / Decrement* = `increment` (`1`).
2. *Cycle 2*. 0. Set control signals: 1. *PC Advance* (signal `7`) to `1`.
    1. Advance `PC`.
        1. *PC Advance* = `1`.

`JSR` Instruction Microcode

Opcode: `0x10`

`JSR` **Semantics:** `SP--; *SP = PC[H]; SP--; *SP = PC[L]; PC = DP`.

Cycle	`23:20`	`19:16`	`15:12`	`11:8`	`7:4`	`3:0`	Result
`1`	`0`	`0`	`1000`	`0010`	`0`	`0`	`0x00_82_00`
`2`	`0110`	`0000`	`1100`	`1000`	`0`	`0`	`0x60_c8_00`
`3`	`0`	`0`	`0`	`0010`	`0`	`0`	`0x00_02_00`
`4`	`0010`	`0`	`0100`	`1000`	`0`	`0`	`0x20_48_00`
`5`	`0`	`0`	`0`	`0`	`0`	`0`	`0x00_00_00`
`6`	`0110`	`0110`	`0`	`0001`	`0`	`0`	`0x66_01_00`
`7`	`0010`	`0010`	`0`	`0001`	`0`	`0`	`0x22_01_00`
`8`	`0`	`0`	`0`	`0`	`1000`	`0`	`0x00_00_80`

1. *Cycle 1*. `SP--; Address Bus Register High = PC[H]`. 0. Set control signals: 1. *Stack Pointer Count* (signal `15`) to `1`. 2. *Stack Pointer Increment / Decrement* (signal `16`) to `0`. 3. *PC Out* (signal `9`) to `1`.
    1. Decrement `SP`.
        1. *Stack Pointer Count* = `1`.
        2. *Stack Pointer Increment / Decrement* = `decrement` (`0`).
    2. Write `PC` to the Address Bus.
        1. **ARSelect** = `PC`.
            1. *PC Out* = `1`.
2. *Cycle 2*. `*SP = PC[H]; SP--;` 0. Set control signals: 1. *SP Out* (signal `14`) to `1`. 2. *AddressHorL* (signal `22`) to `1`. 3. *AddressBusToDataBus* (signal `21`) to `1`. 4. *Mem Write* (signal `11`) to `1`. 5. *Stack Pointer Count*

(signal `15`) to `1`. 6. *Stack Pointer Increment / Decrement* (signal `16`) to `0`.
    1. Write `SP` to the Address Bus.
        1. **ARSelect** = `SP`.
            1. *SP Out* = `1`.
    2. Write Address Bus Register High to the Data Bus (`PC[H]`).
        1. **AddressByteSelect** = `H` (`1`).
            1. *AddressHorL* = `H` (`1`).
        2. *AddressBusToDataBus* = `1`.
    3. Set Memory to read from the Data Bus.
        1. *Mem Write* = `1`.
    4. Decrement `SP`.
        1. *Stack Pointer Count* = `1`.
        2. *Stack Pointer Increment / Decrement* = `decrement` (`0`).
3. *Cycle 3*. `Address Bus Register Low = PC[L]` 0. Set control signals: 1. *PC Out* (signal `9`) to `1`.
    1. Write `PC` to the Address Bus.
        1. **ARSelect** = `PC`.
            1. *PC Out* = `1`.
4. *Cycle 4*. `*SP = PC[L]`. 0. Set control signals: 1. *SP Out* (signal `14`) to `1`. 2. *AddressHorL* (signal `22`) to `0`. 3. *AddressBusToDataBus* (signal `21`) to `1`. 4. *Mem Write* (signal `11`) to `1`.
    1. Write `SP` to the Address Bus.
        1. **ARSelect** = `SP`.
            1. *SP Out* = `1`.
    2. Write Address Bus Register Low to the Data Bus (`PC[L]`).
        1. **AddressByteSelect** = `L` (`0`).
            1. *AddressHorL* = `L` (`0`).
        2. *AddressBusToDataBus* = `1`.
    3. Set Memory to read from the Data Bus.
        1. *Mem Write* = `1`.
5. *Cycle 5*. `Address Bus Register High = DP[H]`. 0. Set control signals: none set.
    1. Write `DP` to the Address Bus.
        1. **ARSelect** = `DP`.
        - Default behavior of **ARSelect**.
6. *Cycle 6*. `DHPC = DP[H]; Address Bus Register Low = DP[L]`. 0. Set control signals: 1. *AddressHorL* (signal `22`) to `1`. 2. *AddressBusToDataBus* (signal `21`) to `1`. 3. *loadAR* (signal `17`) to `1`. 4. *PC Load* (signal `8`) to `1`. 5. *loadARHorL* (signal `18`) to `1`.
    1. Write `DP` to the Address Bus.
        1. **ARSelect** = `DP`.
        - Default behavior of **ARSelect**.
    2. Write Address Bus Register High to the Data Bus (`DP[H]`).
        1. **AddressByteSelect** = `H` (`1`).
            1. *AddressHorL* = `H` (`1`).
        2. *AddressBusToDataBus* = `1`.
    3. Set `DHPC` to load from the Data Bus.
        1. **AR Data Bus Load Enable** = `DHPC`.
            1. *loadAR* = `1`.
            2. *PC Load* = `1`.

3. *loadARHorL* = `H` (`1`).
7. *Cycle 7*. `PC[L] = DP[L] (and PC[H] = DHPC)`. 0. Set control signals: 1. *AddressHorL* (signal `22`) to `0`. 2. *AddressBusToDataBus* (signal `21`) to `1`. 3. *loadAR* (signal `17`) to `1`. 4. *PC Load* (signal `8`) to `1`. 5. *loadARHorL* (signal `18`) to `0`.
    1. Write Address Bus Register Low to the Data Bus (`DP[L]`).
        1. **AddressByteSelect** = `L` (`0`).
            1. *AddressHorL* = `L` (`0`).
        2. *AddressBusToDataBus* = `1`.
    2. Set `PC[L]` to load from the Data Bus.
        1. **AR Data Bus Load Enable** = `PC[L] + PC[H] = DHPC`.
            1. *loadAR* = `1`.
            2. *PC Load* = `1`.
            3. *loadARHorL* = `L` (`0`).
8. *Cycle 8*. Advance `PC`. 0. Set control signals: 1. *PC Advance* (signal `7`) to `1`.
    1. Advance `PC`.
        1. *PC Advance* = `1`.

`RTS` Instruction Microcode

Opcode: `0x11`

`RTS` **Semantics:** `SP++; DHPC = *SP; SP--; PC[L] = *SP, PC[H] = DHPC; SP++; SP++;`

Cycle	`23:20`	`19:16`	`15:12`	`11:8`	`7:4`	`3:0`	Result
`1`	`0`	`0001`	`1000`	`0`	`0`	`0`	`0x01_80_00`
`2`	`0`	`0110`	`1100`	`0101`	`0`	`0`	`0x06_c5_00`
`3`	`0`	`0011`	`1100`	`0101`	`0`	`0`	`0x03_c5_00`
`4`	`0`	`0001`	`1000`	`0000`	`1000`	`0`	`0x01_80_80`

1. *Cycle 1*. `SP++;` 0. Set control signals: 1. *Stack Pointer Count* (signal `15`) to `1`. 2. *Stack Pointer Increment / Decrement* (signal `16`) to `1`.
    1. Increment `SP`.
        1. *Stack Pointer Count* = `1`.
        2. *Stack Pointer Increment / Decrement* = `increment` (`1`).
2. *Cycle 2*. `DHPC = *SP; SP--;` 0. Set control signals: 1. *SP Out* (signal `14`) to `1`. 2. *Mem Read* (signal `10`) to `1`. 3. *loadAR* (signal `17`) to `1`. 4. *PC Load* (signal `8`) to `1`. 5. *loadARHorL* (signal `18`) to `1`. 6. *Stack Pointer Count* (signal `15`) to `1`. 7. *Stack Pointer Increment / Decrement* (signal `16`) to `0`.
    1. Write `SP` to the Address Bus.
        1. **ARSelect** = `SP`.
            1. *SP Out* = `1`.
    2. Set Memory to write to the Data Bus.
        1. *Mem Read* = `1`.
    3. Set `DHPC` to read from the Data Bus.
        1. **AR Data Bus Load Enable** = `DHPC`.
            1. *loadAR* = `1`.

            2. *PC Load* = `1`.
            3. *loadARHorL* = `H` (`1`).
        4. Decrement `SP`.
            1. *Stack Pointer Count* = `1`.
            2. *Stack Pointer Increment / Decrement* = `decrement` (`0`).
3. *Cycle 3*. `PC[L] = *SP; SP++`. 0. Set control signals: 1. *SP Out* (signal `14`) to `1`. 2. *Mem Read* (signal `10`) to `1`. 3. *loadAR* (signal `17`) to `1`. 4. *PC Load* (signal `8`) to `1`. 5. *loadARHorL* (signal `18`) to `0`. 6. *Stack Pointer Count* (signal `15`) to `1`. 7. *Stack Pointer Increment / Decrement* (signal `16`) to `1`.
    1. Write `SP` to the Address Bus.
        1. **ARSelect** = `SP`.
            1. *SP Out* = `1`.
    2. Set Memory to write to the Data Bus.
        1. *Mem Read* = `1`.
    3. Set `PC[L]` to read from the Data Bus (and `PC[H] = DHPC`).
        1. **AR Data Bus Load Enable** = `PC[L] + PC[H] = DHPC`.
            1. *loadAR* = `1`.
            2. *PC Load* = `1`.
            3. *loadARHorL* = `L` (`0`).
    4. Increment `SP`.
        1. *Stack Pointer Count* = `1`.
        2. *Stack Pointer Increment / Decrement* = `increment` (`1`).
4. *Cycle 4*. `SP++;` 0. Set control signals: 1. *Stack Pointer Count* (signal `15`) to `1`. 2. *Stack Pointer Increment / Decrement* (signal `16`) to `1`. 3. *PC Advance* (signal `7`) to `1`.
    1. Increment `SP`.
        1. *Stack Pointer Count* = `1`.
        2. *Stack Pointer Increment / Decrement* = `increment` (`1`).
    2. Advance `PC`.
        1. *PC Advance* = `1`.

Roadmap

This proposal requires many hardware changes to implement; it is advisable therefore to not attempt to implement everything at once, but in an incremental approach (implement, test, fix / repeat with next piece).

To recap, here is a list of everything that must be done to fully implement this proposal:

1. Hardware
    1. `ARSrc` Operand Decode Logic
    2. PC Loading / Out Revision
    3. AR Data Bus Load Enable (Look Up Table - EEPROM)
        1. Replace current Data Bus -> AR Loading hardware (`LSP` hardware)
    4. ARSelect (Look Up Table - EEPROM)
        1. Replace current Address Bus Arbitrator with ARSelect LUT
    5. AddressByteSelect Logic
2. Microcode
    1. Update microcode for `PUSH`, `POP`, `JSR`, and `RTS`

    2. Overwrite microcode for `LSP` and add microcode for `MAG`, `LDWL`, `LDWH`, `SDWL`, `SDWH`, `LDA`, `MGA`, `MAAL`, and `MAAH`
3. Software
    1. ByteFrost Assembler v2
        1. Ensure all instructions in the ISA have a corresponding representation in the assembler code.
            1. Remove `LSP`.
            2. Update `JSR`'s instruction string to take no operands.
            3. Add `MAG`, `LDWL`, `LDWH`, `SDWL`, `SDWH`, `LDA`, `MGA`, `MAAL`, and `MAAH`.
        2. Update the set of ByteFrost Assembly instructions:
            1. Remove `LSP`.
            2. Add `MAG`, `LDW`, `SDW`, `LDA`, `MGA`, and `MAA`.
            3. Update `CALL` to work with the new `JSR` ISA instruction.
        3. Update the parser to recognize new syntax
            1. AR Offset token (`Imm(AR)`) used in `LDW` and `SDW`.
    2. **AR Data Bus Load Enable** and **ARSelect** Look Up Table Generators
        1. Write a program that generates the look up tables for these EEPROMs. (**DONE**)

Dependency Graph

The implementation of each instruction requires the following hardware changes:

Implementing the **ARSelect** LUT

Implementing the **ARSelect** LUT requires the following:

1. Removals
    1. Removing the Address Bus Arbiter.
    2. Removing the `Read PC Low` `74HC245` chip that allows writing `PC[L]` to the Data Bus. (The *PC Out* control signal becomes an input of the **ARSelect** LUT).
    - This would break `JSR` and Branch Relative (`JSR` until its new microcode is implemented).
2. Additions
    1. Creating the active low `OPCODE_MAG_LDW_SDW_MAA` signal by ANDing together the active low signals for opcodes `0x14` through `0x1a`.
    2. Adding the *TmpARWrite* control signal.
    3. Fully implementing the `ARSrc` operand decode logic.
    4. Adding an EEPROM that contains the **ARSelect** LUT and connecting all inputs and outputs to it.
3. Replacements
    1. Replace `Stack Pointer OE` (output of the Address Bus Arbiter) with the **ARSelect** LUT's `SP Output Enable` signal.

Implementing the **AR Data Bus Load Enable** LUT

Implementing the **AR Data Bus Load Enable** LUT requires the following:

1. Removals
    1. Remove the Load Special Pointer combinational logic. (can leave the NOR gate chips, since 6 of the 8 **AR Data Bus Load Enable** LUT outputs will need them (6 NORs)).

    2. Remove the `PC Load Hi` active low signal.
        1. Remove the `PC Load Hi` combinational logic.
    3. Remove the `Ld Prog Ctr Dummy` signal (generated by the Load Special Pointer combinational logic).
        - Replace with `DHPC Load Enable` LUT output (active low) NOR'd with the Clock signal.
    4. Free control signal 12 (*Load Data Address*).
    5. Remove the `Ld Addr Ptr Hi` signal (generated by the Load Special Pointer combinational logic).
        - Replace with `DP[H] Load Enable` LUT output (active low) NOR'd with the Clock signal.
    6. Remove `DP[L]`'s current clock gating.
        - Replace with `DP[L] Load Enable` LUT output (active low) NOR'd with the Clock signal.
    7. Remove the `Ld Stack Pointer Hi` signal (generated by the Load Special Pointer combinational logic).
        - Replace with `SP[H] Load Enable` LUT output (active low) NOR'd with the Clock signal.
    8. Remove the `SP[L]`'s load 0 at Power On combinational logic. It makes more sense, now that the stack will be full-descending, to make the `SP` a normal register in the sense that its starting point must be explicitly set by the program (i.e., before using the stack, set the `SP` to the address that will be the start (bottom) of the stack).
2. Additions
    1. Add an EEPROM that contains the **AR Data Bus Load Enable** LUT and connecting all inputs and outputs to it.
    2. Add NOR gates so that the Load Enable output signals of the LUT are NOR'd with the clock signal; the output of these NORs are sent as the Load Enable (clock) signals to the AR byte registers.
        - **Exception:** `PC[L] + PC[H] = DHPC Load Enable` should be sent directly WITHOUT NORing it with the clock signal! This is because it acts as an actual load enable, not a clock input to the PC's 4-bit counter registers.
        - **Exception:** `SP[L] Load Enable` should be sent directly WITHOUT NORing it with the clock signal! This is because it acts as an actual load enable, not a clock input to the SP Low's 4-bit up/down counter registers.
    3. Connect `SP[L] Load Enable` (active low) DIRECTLY (without NOR'ing with the Clock signal) to the load enable (`PE` active low input) of the `SP[L]` 4-bit up/down counter registers. Connect their data inputs to the Data Bus.
3. Replacements
    1. Replace `PC Load Lo` and `PC Load Hi` with the **AR Data Bus Load Enable** LUT's `PC[L] + PC[H] = DHPC Load Enable` signal.
    2. Rename `PC Load Lo` to `PC Load Branch` (i.e., **DO NOT REMOVE THE COMBINATIONAL LOGIC OF `PC Load Lo`!** It is an input of the LUT!)
    3. Replace the `Ld Prog Ctr Dummy` with the **AR Data Bus Load Enable** LUT's `DHPC Load Enable`.
        - **Note:** We need to NOR the `DPHC Load Enable` active low signal (and all of the Load Enable active low signals generated by the LUT) with the clock to force the AR byte registers to read on the falling edge of the clock. This is because the Load Enable signals are actually the clock signals sent to the AR byte registers. In effect, this means that the data bus value must be ready by the falling edge of the clock (i.e., the clock speed is essentially halved as the critical path must fit in half a clock cycle for the Data Bus' value to be ready by the falling edge of the clock). Ideally, the Load Enable signals would cause the

AR byte registers to read at the next rising edge, but this would require normal load enables that don't exist and are too expensive to implement externally (i.e., each register would need a mux input that chooses between the new input and the current output, with the mux selector being the load enable).

`MAG` Hardware Dependencies

In order for the `MAG` microcode to work on the ByteFrost, the following things must be done:

1. Instruction Operands
   1. The `ARSrc` operand decode logic must be implemented.
   - Required for **ARSelect** implementation.
   2. The `(AR) L/H` operand decode logic (single wire) must be implemented.
2. Control Signals
   1. Adding *TmpARWrite*.
   - Required for **ARSelect** implementation.
   2. Adding *AddressBusToDataBus*.
3. Other Hardware Changes
   1. Fully implementing the **ARSelect** LUT.
   2. Fully implementing the **AddressByteSelect** combinational logic.

`LDW` Hardware Dependencies

In order for the `LDW` (`LDWL` and `LDWH`) microcode to work on the ByteFrost, the following things must be done:

1. Instruction Operands
   1. The `ARSrc` operand decode logic must be implemented.
   - Required for **ARSelect** implementation.
2. Control Signals
   1. Adding *TmpARWrite*.
   - Required for **ARSelect** implementation.
   1. Adding *TmpARRead*.
3. Other Hardware Changes
   1. Fully implementing the **ARSelect** LUT.

`SDW` Hardware Dependencies

In order for the `SDW` (`SDWL` and `SDWH`) microcode to work on the ByteFrost, the following things must be done:

1. Instruction Operands
   1. The `ARSrc` operand decode logic must be implemented.
   - Required for **ARSelect** implementation.
2. Control Signals
   1. Adding *TmpARWrite*.
   - Required for **ARSelect** implementation.
   2. Adding *TmpARRead*.
3. Other Hardware Changes
   1. Fully implementing the **ARSelect** LUT.

`LDA` Hardware Dependencies

In order for the `LDA` microcode to work on the ByteFrost, the following things must be done:

1. Instruction Operands
    1. The `ARDest` operand decode logic (two wires) must be implemented.
    - Required for **AR Data Bus Load Enable** implementation.
    2. The `(AR) L/H` operand decode logic (single wire) must be implemented.
2. Control Signals
    1. Adding *loadAR*.
    - Required for **AR Data Bus Load Enable** implementation.
    2. Adding *loadARHorL*.
    - Required for **AR Data Bus Load Enable** implementation.
3. Other Hardware Changes
    1. Fully implementing the **AR Data Bus Load Enable** LUT.

`MGA` Hardware Dependencies

In order for the `MGA` microcode to work on the ByteFrost, the following things must be done:

1. Instruction Operands
    1. The `ARDest` operand decode logic (two wires) must be implemented.
    - Required for **AR Data Bus Load Enable** implementation.
    2. The `(AR) L/H` operand decode logic (single wire) must be implemented.
2. Control Signals
    1. Add *loadAR*.
    - Required for **AR Data Bus Load Enable** implementation.
    2. Add *loadARHorL*.
    - Required for **AR Data Bus Load Enable** implementation.
3. Other Hardware Changes
    1. Fully implementing the **AR Data Bus Load Enable** LUT.

`MAA` Hardware Dependencies

In order for the `MAA` (`MAAL` and `MAAH`) microcode to work on the ByteFrost, the following things must be done:

1. Instruction Operands
    1. The `ARSrc` operand decode logic must be implemented.
    - Required for **ARSelect** implementation.
    2. The `ARDest` operand decode logic (two wires) must be implemented.
    - Required for **AR Data Bus Load Enable** implementation.
2. Control Signals
    1. Add *loadAR* and *loadARHorL*.
    - Required for **AR Data Bus Load Enable** implementation.
    2. Add *TmpARWrite*.
    - Required for **ARSelect** implementation.
    3. Add *TmpARRead*.
    4. Add *AddressHorL*.
    5. Add *AddressBusToDataBus*.

3. Other Hardware Changes.
    1. Fully implement the **ARSelect** LUT.
    2. Fully implement the **AddressByteSelect** module.
    3. Fully implement the **AR Data Bus Load Enable** LUT.

`PUSH` Hardware Dependencies

In order for the `PUSH` microcode to work on the ByteFrost, the following things must be done:

1. Instruction Operands
    1. The `ARSrc` operand decode logic must be implemented.
    - Required for **ARSelect** implementation.
2. Control Signals
    1. Add *TmpARWrite*.
    - Required for **ARSelect** implementation.
3. Other Hardware Changes.
    1. Fully implement the **ARSelect** LUT.

`POP` Hardware Dependencies

In order for the `POP` microcode to work on the ByteFrost, the following things must be done:

1. Instruction Operands
    1. The `ARSrc` operand decode logic must be implemented.
    - Required for **ARSelect** implementation.
2. Control Signals
    1. Add *TmpARWrite*.
    - Required for **ARSelect** implementation.
3. Other Hardware Changes.
    1. Fully implement the **ARSelect** LUT.

`JSR` Hardware Dependencies

In order for the `JSR` microcode to work on the ByteFrost, the following things must be done:

1. Instruction Operands
    1. The `ARSrc` operand decode logic must be implemented.
    - Required for **ARSelect** implementation.
    2. The `ARDest` operand decode logic (two wires) must be implemented.
    - Required for **AR Data Bus Load Enable** implementation.
2. Control Signals
    1. Add *TmpARWrite*.
    - Required for **ARSelect** implementation.
    2. Add *AddressHorL*.
    3. Add *AddressBusToDataBus*.
    4. Add *loadAR* and *loadARHorL*.
    - Required for **AR Data Bus Load Enable** implementation.
3. Other Hardware Changes
    1. Fully implement the **ARSelect** LUT.

2. Fully implement the **AddressByteSelect** module.
3. Fully implement the **AR Data Bus Load Enable** LUT.

`RTS` Hardware Dependencies

In order for the `RTS` microcode to work on the ByteFrost, the following things must be done:

1. Instruction Operands
   1. The `ARSrc` operand decode logic must be implemented.
   - Required for **ARSelect** implementation.
   2. The `ARDest` operand decode logic (two wires) must be implemented.
   - Required for **AR Data Bus Load Enable** implementation.
2. Control Signals
   1. Add *TmpARWrite*.
   - Required for **ARSelect** implementation.
   2. Add *loadAR* and *loadARHorL*.
   - Required for **AR Data Bus Load Enable** implementation.
3. Other Hardware Changes
   1. Fully implement the **ARSelect** LUT.
   2. Fully implement the **AR Data Bus Load Enable** LUT.